A Logic Database System with Extended Functionality

Sang-goo Lee
Dept. of Computer Science
Seoul National University
Seoul, Korea

Dong-Hoon Choi
Dept. of Computer Science
Dongduk Women's University
Seoul, Korea

Sang-Ho Lee
School of Computing
Soongsil University
Seoul, Korea

Abstract

We present the architecture and design details of a logic database system that extends functionality in a number of ways. The system supports indefinite (non-Horn) facts in extensional database (EDB) and existential quantification in intensional database (IDB) rules. Semantic query optimization is integrated as part of the query processing module. The system also deals with efficient rule management for IDB and integrity constraints (IC). The proposed system supports the following database: The EDB consists of a set of positive ground formula, not necessarily Horn; The IDB includes rules that are Horn and may have existential quantifiers. The IC includes range-restricted Horn clauses with no existential quantification.

1 Introduction

One of the objectives of logic databases is to develop a database system that supports a highly expressive language and also allows rapid development of complex applications commonly required in expert systems, advanced sciences, and applied engineering. Over the last decade, there have been very active research efforts in the logic databases (or deductive databases) field and consequently a number of prototype systems have been reported in the literature.

A deductive database consists of three components: the extensional database (EDB) which is a set of facts stored explicitly, the intensional database (IDB) which is the set of facts that can be derived from the EDB by use of a set of rules called IDB rules, and integrity constraints (IC), a set of rules that the database (IDB as well as EDB) is supposed to follow. IDB rules and IC rules are represented by first order predicate logic formula while facts in EDB are generally considered to be positive unit ground clauses.

Much of the efforts in this field has been focused on defining the correct interpretation of the rules and methodologies that implement the interpretation so as to efficiently compute the derived facts in a sound and complete manner. Other areas of major development include recursive query processing and integrity constraint enforcement.

Despite the accomplishments in the field, there are areas which deserve further enhancements. First, non-Horn data are not allowed in most of the previous works. The ability to represent and operate on non-Horn data is important if logic database systems are to model and reason on uncertainty.

Second, existential quantification in IDB rules is not supported directly. Inability to provide such form of rules resulted in introducing extra-logical components to the language or requiring the use of host programming languages, both of which depart from the basis (or merits) of purely declarative logic databases.

Third, semantic knowledge such as integrity constraints are not fully utilized in the inference and query processing part of existing systems. Knowledge-based query processing needs have been found in applications such as digital libraries and multimedia databases.

In addition, logic is not a very friendly user language. The user should be able to write rules and queries without having to worry about range-restriction, stratification, and the likes, yet express those with the full power of the underlying logic language. Also, issues such as efficient rule management (IC and IDB) and integration with the object oriented model are areas demanding further studies.

We present, in this paper, the architecture and design details of a logic database system that extends functionality in most of these areas. The system supports indefinite (non-Horn) facts in the EDB and existential quantification in IDB rules. Semantic query optimization is integrated as part of the query processing module. We have developed as a

---

1 This work was supported in part by the Korea Science and Engineering Foundation (KOSEF) under grant number 93-0100-08-01-3.
formal user language whose syntax mimics Korean. The system also deals with efficient rule management for IDB and IC.

Figure 1.1 shows a graphical depiction of the scope of our system. The domain of a typical logic database system can be viewed as the intersection of the three sets; Horn clauses, no existential quantification, and syntactical query processing. Our system encompasses the domain including the intersections of any two of these sets (hatched area in the figure). The proposed system supports the following database: The EDB consists of a set of positive ground formula, not necessarily Horn; The IDB includes rules that are Horn and may have existential quantifiers. The IC includes range-restricted Horn clauses with no existential quantification.

Figure 1.1: Functional Extensions

The remainder of the paper is organized as follows. In section 2, we review some of the relevant previous works. Section 3 presents the overview of our system as a whole. Sections 4, 5, and 6, respectively, discuss the details of the three main subsystems; Inference and Translation, Query Optimization, and EDB and IDB Manipulation. The paper is concluded in section 7.

2 Previous Works

In this section we summarize the major characteristics of some of the current deductive database systems.

LDL [2] supports recursion, negation, functions, and sets in the language which is based on Horn clause logic. LDL evaluates queries in a bottom-up manner and uses several optimization techniques such as magic sets [1].

CORAL [12] provides a declarative language and an interface to C++. The declarative language enables the users to define complex queries and view definitions on databases. The C++ interface can be used to write applications in a procedural language for efficiency or for performing actions such as updates or user interaction. The C++ interface also allows the users to create new classes, which makes CORAL extensible. CORAL supports programs with recursive queries, negation, set and multisets, and non-ground facts.

Glue-Nail [7] has two languages, Nail and Glue. Nail is a declarative query language based on Horn clauses to express complex recursive queries and negative queries. Glue is a procedural language which provides non-query features such as control structures, update operations and I/Os. Glue also supports sets and aggregations. The declarative Nail and the procedural Glue combine to form a complete programming environment.

Aditi [15] supports recursive queries, function symbols, negation, and aggregates. Aditi mixes bottom-up with top-down evaluation. Some predicates can be declared to be evaluated in a top-down approach. In this case, user must ensure termination of evaluation. Queries in Aditi can be embedded in Nu-Prolog which supports procedural features.

Current deductive database systems described above emphasize recursion, negation, and restricted form of logic language. However, next-generation applications require more flexible language features, enforcement of more general form of integrity constraints, and storage and retrieval of incomplete information.

3 System Architecture

Figure 3.1 shows the overall architecture of our system. The user interacts with the system through the User Interface Module. The core of the system is the Query Processor which consists of the Inference Module, Translator Module, EDB Access Module, and the Query Optimizer Module. It is responsible for processing the user query against the database (EDB and IDB). IDB rules and ICs are managed by the Rule Management Module. Each of these modules, with the exception of the User Interface Module, is explained in detail in subsequent sections.

The user interface is text based. Because first order logic makes a poor user language, we have developed a formal user language based on Korean syntax. A legal sentence can be translated into only one first order logic formula (or its logical equivalent).

The Inference Module is responsible for compiling the IDB rules and queries. When a new rule is added to the IDB, it is compiled into a formula which defines the IDB predicate (head) in terms of EDB literals only. Because existential quantification in an IDB rule (or equivalently, universal quantification in the body of an IDB rule) is allowed, we use a new inference mechanism that can cope with existential quantification. The Inference Module, together with the Translator Module, makes possible this extension to the IDB.
When a query is given to the system, the Inference Module compiles it into a formula with only EDB predicates using the precompiled IDB rules. The compiled query is then given to the Query Optimizer for semantic query optimization, which produces a reformulated query formula. The Translator then converts this formula into a relational algebra formula which is ready for evaluation. A distinguishing feature of this translator is in its processing of existential quantification that occur in our IDB. Once this translation is done, the system need not worry about existential quantification.

Figure 3.1: System Architecture

Before actually evaluating the query, the relational algebra formula is given to the Query Optimizer once again for further optimization. The reason the same query is passed through the optimizer twice is that certain optimization opportunities may arise due to the translation process. The final access plan is then sent to the EDB Access Module.

The EDB Access Module evaluates the access plan against the database. Because non-Horn ground facts are allowed, this process is not trivial. However, because non-Horn IDB rules are not permitted, intrinsic complications introduced by non-Horn semantics can be contained within the EDB Access Module. In addition to evaluating queries against the EDB, the EDB Access Module is responsible for storage of the EDB.

Rule Manager Module manages the insertion, deletion, updates, and access to IDB and IC. The rest of the system may simply assume any IDB or IC rule needed is already in main memory in an internal representation format.

4 Inference and Translation Module

If only universal quantifiers are allowed in IDB, there are inherent expressive limitations in constructing various view definitions. In order to extend the expressiveness of IDB, an extended IDB is defined.

Definition An extended intensional database (EIDB) is defined to be of the form:

\[(\forall x_1)...(\forall x_n)(\forall y_1)...(\forall y_m)[P(x_1, ..., x_n) \leftarrow \alpha]\]

(or its logical equivalent), where Q denotes \(\exists\) or \(\forall\) and \(\alpha\) denotes a conjunction of positive literals. All variables in \(\alpha\) are \(x_1, x_2, y_1, ..., y_m\) and must be free. \(P\) is a positive literal.

A common way to handle existential quantification in IDB is to eliminate them by use of logical equivalence, \((\exists x)\alpha \equiv (\forall y)\neg \alpha\). This introduces new negative literals, which cannot be evaluated without a meta rule such as negation as failure and closed world assumption.

In our deduction scheme, an existential quantification is not transformed but is allowed to occur during deduction. This provides an opportunity for an efficient and natural evaluation. By doing so, we are able to evaluate existential quantification directly according to their intended semantics in evaluation phase. We use a new inference mechanism called substitution rule [4, 9] in deduction phase.

Definition [6] If \(P\) is an \(n\)-ary predicate and \(x_1, ..., x_n\) are distinct variables, the notation \(S^{a_1, ..., a_n}\) \(\alpha\) shall stand for \(\alpha\) unless the two conditions are satisfied that:

1. no wff part of \(\alpha\) of the form \((Qv)\gamma\) where \(v\) is a free variable of \(\beta\) other than \(x_1, ..., x_n\), contains an occurrence of \(P\); and
2. for each ordered \(n\)-tuple \(a_1, ..., a_n\) of individual variables or individual constants (or both, not necessarily all distinct) for which \(P(a_1, ..., a_n)\) occurs in \(\alpha\), the wff parts of \(\beta\), if any, that have the form \((Q a_1)\gamma, (Q a_2)\gamma, ..., (Q a_n)\gamma\) contain no free occurrences of \(x_1, ..., x_n\) respectively.

And, if these two conditions are satisfied, the notation shall stand for the result of replacing \(P(a_1, ..., a_n)\) at all of its occurrences in \(\alpha\) by \(S^{a_1, ..., a_n}\beta\). This replacement carried out simultaneously for all ordered \(n\)-tuples \(a_1, ..., a_n\).
of individual variables or individual constants (or both, not necessarily all distinct) such that \( P(a_1, \ldots, a_n) \) has occurrence in \( \alpha \).

**Definition** Let \( \alpha \) and \( \beta \) be extended IDB rules.

\[
(\forall x_1) \ldots (\forall x_n) \left( R(x_1, \ldots, x_n) \leftarrow \delta \right) \quad (\alpha)
\]

\[
(\forall u_1) \ldots (\forall u_n) \left( P(u_1, \ldots, u_n) \leftarrow \gamma \right) \quad (\beta)
\]

If \( P(u_1, \ldots, u_n) \) is unifiable with a predicate in \( \delta \), the substitution rule generates \( S^{(\alpha, \ldots, \beta)} \), called substituent.

**Example 4.1**

\[
(\forall u) \left[ P(u) \leftarrow (\exists v)(\exists w) \left( [Q(u,w) \land R(w,v)] \right) \right] \quad (\alpha)
\]

\[
(\forall x) \left[ A(x) \leftarrow (\exists y) \left( [P(x) \land B(x,y)] \right) \right] \quad (\beta)
\]

Since \( P \) in \( \alpha \) is unifiable with \( P \) in \( \beta \), the substitution rule generates the following substituent:

\[
S^{(\alpha, \beta)} = (\forall x) \left[ A(x) \leftarrow (\exists y) \left( [S^\alpha(A(x)) \land B(x,y)] \right) \right]
\]

It should be noted that the substitution rule does not remove existential quantifiers during deduction. The effect of the substitution rule is the same as resolution if all quantifiers are universal. The soundness and completeness of the substitution rule are shown in [4, 9].

The inference module receives a set of EIDB and performs a deduction process in which all IDB literals occurring in the bodies of EIDB are replaced by corresponding definitions by use of the substitution rule. A query is compiled in a similar manner.

The translator accepts the compiled query and translates it into an equivalent relational algebra expression. The semantics of existential quantifiers in EIDB is always evaluated directly via division operation in this framework. We believe that our approach not only is a more natural way for processing existential quantification but also outperforms existing approaches.

Recursive EIDB rules are translated into a set of relational algebra expressions as in [8]. However, when existential quantification and negation are combined in the body, a special treatment takes place:

1. Check whether an upper bound of recursive iterations required can be determined independently of the contents of database.
2. If so (in fact it is true in almost all cases [10]), transform a recursive EIDB set into an equivalent set of non-recursive EIDB.
3. Transform the result into algebra expressions as before.

The above procedure is intended mainly to take advantage of the property that occurrence of existentially quantified variables in the recursive body predicate tends to limit the level of recursive search dramatically [10].

### 5 Query Optimization

The query optimizer focuses mainly on semantic query optimization although some degree of syntax directed conventional optimization techniques are employed. Semantic Query Optimization (SQO) is the process of utilizing information implied by integrity constraints (IC) to minimize the search space for query processing [3, 11]. An IC becomes relevant to the given query \( Q \) if the body of the IC subsumes a subquery of \( Q \). Previous studies have focused mainly on queries with joins of base relations. When the query contains union operations, identifying relevant IC for a given query becomes a complex problem (combinatorial in some methods).

We have adopted the idea of [11] and extended on it to deal with formulas with unrestricted quantification. Because the IDB can have universal quantification in the body, the compiled query is no longer of the simple project-select-join type typically assumed in previous works. An And/Or graph is constructed for a given query as in [11]. However, a subquery that starts with a universal quantification is treated as a single node. This abstraction allows the process to apply the SQO method of [11] but at the same time eliminates the opportunity to optimize the particular subquery. To remedy this problem a second optimization effort is made after the query has been translated into relational algebra formula. At this stage, subqueries with roots of division are treated as single nodes.

The overall query optimization process consists of the following:

1. Get the compiled query in logical form from the Inference module. Do what optimization (SQO) can be done
2. Pass the (optimized) logic form to the Translation module. Get translated query in algebra form from the Translation module. Do what optimization (SQO) can be done here
3. Pass the resulting access plan to the EDB Access module. Exchange estimates for possible re-ordering of operations

The optimization process (SQO) of 1 and 2 above are basically identical. This process, shown in Figure 5.1, consists of three phases. In the first phase, the query tree (And/Or tree) is traversed and arcs are assigned unique labels. Given two database predicates (nodes) in the query tree, we can decide on the relative positions of the nodes just by looking at the labels of arcs leading to the nodes (called traces of the nodes). Edge labeling and trace computation can be done by a single traverse of the query node labels.
tree. In the second phase, the set of potentially relevant constraints (PRC) are identified. A PRC is an IC whose relational predicates in the body occur as join pairs of certain kinds in the query. This can be identified using the traces obtained in the first phase. The third phase deals with computable predicates of PRCs. If all computable predicates of a PRC are satisfied by the query, the PRC then becomes a relevant constraint (RC). The head of a RC is taken as a valid fact (called restriction) which can be used to optimize the query.

In the whole process, the system requires only one visit for each node of the query tree, two visits for ICs that turn out to be PRCs, and one or no visits to other ICs.

6 EDB and IDB Manipulation

6.1 EDB Access Module

We use a relational database system to manage our disjunctive EDB. Disjunctive information introduces two new problems.
1. We must extend the storage structure of the EDB access module for the disjunctive information because the relational model cannot support disjunctive facts. The access module in our system implements the extended relational model defined in [5].
2. We should pay special attention to checking the definiteness of intermediate tuples because definite tuples may be derived from disjunctive EDB, as shown in the following example.

Example 6.1

Given DB = \{ P(x) \leftarrow A(x) \land B(x), P(x) \leftarrow C(x), A(a) \lor C(a), B(a) \lor C(a) \} and a query P(a), accessing the EDB does not give the answer P(a). However, P(a) can be derived from DB through resolution. Thus, P(a) is true.

The extended relational model for disjunctive deductive databases in [5] maintains the uncertainty of disjunctive information and derives definite information from disjunctive one during the execution of database operation. The basic idea of the model is that disjunctive information is represented by a disjunctive clause which consists of multiple number of literals and that a literal is stored as a tuple. Extended relational algebra maintains the disjunctive relationship among the literals during its operations.

The subcomponents of the EDB Access Module are presented in Figure 6.1. The Extended Relational Storage Manager provides the storage structures to hold definite and disjunctive data. The Extended DB Extractor implements the extended DB operators defined in the extended relational algebra. The result of the Extended Relational Subsystem is an extended relation with source clause sets.

The Check-Definiteness Subsystem derives all definite tuples from the IDB rules for a query and the result of the Extended Relational Subsystem. The CNF converter transforms the IDB rules into a single clause of conjunctive normal form. The validity tester executes subsumption test for each disjunctive tuples. A successful test results in converting the tuple into a definite one.

6.2 Rule Manager (IDB and IC)

The Rule Manager retrieves the IDB rules and ICs required to process a given query from the IDB and IC. It also inserts, deletes and updates the IDB and IC. One of
the major issues in implementing the Rule Manager is the storage structure for the IDB and IC clauses. Several IDB storage structures based on graph model have been proposed [8, 14].

While the IDB storage structures developed so far are very efficient for the representation and compilation of the IDB, they are not suitable for the definiteness check operation of disjunctive tuples nor for efficient insertion, deletion and update of the IDB. We use an IDB storage structure that consists of an IDB table and a hash index table. The IDB table stores the IDB rules themselves. A tuple in the hash index table contains the rule numbers that have as head predicate with corresponding hash value.

For ICs, the predicates in the body are indexed instead of the head because ICs are accessed by body predicates (see section 5). The hash function is applied to pairs of predicates as well as single predicates since more than one predicates can occur in the body of an IC. It can be shown that this is sufficient for searching an IC with any number of predicates in the body [13].

7 Conclusions and Further Research

We have the architecture of a logic database system with extended functionality significant for next-generation applications but not covered in most of previous works.

Uncertainty of disjunctive information is supported in the EDB which implements the extended relational model. Existential quantification in IDB rules extends the representational flexibility in writing IDB rules. This virtually extends the expressive power of logic database language. As a result, users can compose their application in a more declarative logic language.

ICs are fully utilized to enhance the efficiency of query optimization. Also, we developed a user language which provides a more natural and easy-to-use interface to the users who are not familiar to first order logic.

We are currently extending the IDB to include negations in the body of a rule. When coupled with the non-Horn EDB, there are cases where the exact semantics are not quite clear. We are working on a meta-rule that can effectively define/confine the semantics of such situations. Once this meta-rule is defined a query translation method based on this rule should be developed. We are also working on ways to optimize the part of a query tree that has universal quantification.

References


