Loading Data into Description Reasoners

Alex Borgida*
Ronald J. Brachman
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

Knowledge-base management systems (KBMS) based on description logics are being used in a variety of situations where access is needed to large amounts of data stored in existing relational databases. We present the architecture and algorithms of a system that converts most of the inferences made by the KBMS into a collection of SQL queries, thereby relying on the optimization facilities of existing DBMS to gain efficiency, while maintaining an object-centered view of the world with a substantive semantics and significantly different reasoning facilities than those provided by Relational DBMS and their deductive extensions. We address a number of optimization issues that arise in the translation process due to the fact that SQL queries with different syntax (but identical semantics) are not treated uniformly by current database management systems.

1 Introduction

Description logics (DLs) are formal object-centered representation languages based on complex terms and their inter-relationships. They provide an alternative approach to knowledge base management, compared to traditional Relational DBMS (RDBMS) and their deductive extensions, which are squarely based on First Order Predicate Calculus. DLs have in fact been used in data management in a variety of roles (see [8] for a review): as a data model (CLASSIC [9], CANDIDE [3, 7]); as a language for expressing a conceptual model of some domain, including federated schemas [5, 6, 10, 19]; and as a query language [2, 3, 11].

In some of these applications (detailed later) we are faced with the problem of accessing data which is already stored in an existing DBMS. Because of the distinctive nature of DLs and their implementations (e.g., they tend to make inferences "eagerly" rather than "on demand"), we have sought new ways to access the data stored in DBMS. This paper presents our solutions to the problems of database loading in this context. Our approach has been embodied in an implemented system for data exploration [11], currently being tested.

The following section presents a brief summary of the main features of DLs and of KBMS based on them, focusing on the CLASSIC system developed and used at AT&T [9]. We then present the problems raised by coupling description logic reasoners to RDBMS, and outline our novel approach to solving these problems. We present successively better techniques for generating efficient SQL queries from descriptions, and for interfacing the conceptual domain model presented in CLASSIC with that offered in the relational database.

2 Description logics and their inferences

Suppose we start with a database populated by *individual objects* (e.g., `JoeCool`, `toaster#74`), initially grouped into (primitive) *classes* (e.g., `CUSTOMER`, `ITEM`) and related to each other by binary relations (e.g., `bought` relates customers to the things that they buy). The binary relations will be called *roles*, and they can be multi-valued, or restricted to be functions, in which case we call them *features*.[1]

2.1 The syntax and semantics of CLASSIC

Descriptions are first-order terms without variables that are used to denote classes (also known as *concepts*) or roles. For readers familiar with databases, they can be thought of as (unnamed) view definitions.

For example, the CLASSIC description

```
and(CUSTOMER,
    all(bought,EXPENSIVE-ITEM),
    at-least(3,children),
    all(creditRating,one-of(hi,very-hi)))
```

*On sabbatical leave from Rutgers University.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0217...$1.50

[1] The term "attribute", sometimes used for features in the literature, is reserved in this paper for columns in relational tables.

denotes objects in the intersection of the following sets:

- the set of objects of type CUSTOMER, probably a primitive specified elsewhere;
- the set of objects all of whose fillers for the role bought are instances of the concept EXPENSIVE-ITEM;
- the set of objects with at least three fillers for the children role;
- the set of objects whose creditRating values are in the set {hi, very-hi}.

Descriptions are therefore built up recursively as terms from subterms using *concept constructors.*

The concept constructor **fills** specifies a role filler, as in **fills**(children,Calvin), which denotes objects with Calvin as one of the fillers for children; **same-as** specifies equality of function composition, as in **same-as**((mother age)(father age)), which denotes objects whose father and mother have the same age value. Note that the arguments of **same-as** must all be features. CLASSIC also supports descriptors denoting sets of programming language values (called "host individuals"), integer ranges, and a "procedural escape hatch": a **test** concept can be defined by providing a LISP or C function.

The significant aspect of descriptions, in contrast to regular class definitions in object-oriented models, is that they can be reasoned with—they form a logic. The fundamental logical relationship between them is *subsumption* (containment): C is subsumed by B iff every possible individual instance of C would also be an instance of B. Thus **at-least**(5,bought) is subsumed by **at-least**(3,bought), and such simple rules can be combined to yield subsumption for complex concepts because of the compositional semantics of the DL.

Descriptions can be used for a variety of tasks in information management, including (i) conceptual schema design; (ii) stating queries, views, and integrity constraints; and (iii) expressing active triggers in the form of *rules*, where the antecedent and consequent are descriptions (e.g., IF *someone is a* GOOD-CUSTOMER THEN *they are a* MAILING-LIST-CUSTOMER).

2.2 Reasoning about individuals

Of course, a KBMS manages more than just class definitions. We want to be able to create new individual objects, and assert (and possibly retract) facts about them. There are two kinds of facts: membership in concepts, and role-filler information. For example, we might want to say that Calvin is a CUSTOMER, and has bought a particular television set, TV#4279 (i.e., TV#4279 is a filler for the bought role of Calvin). DLs *do not* make the closed world-assumption about role fillers: if we want to say that Calvin has not bought anything other than the television set, we need to explicitly assert that the role **bought** is *closed* for Calvin. These facilities are all available in CLASSIC.

A KBMS such as CLASSIC uses descriptions to perform a number of different kinds of inferences about individual objects, which may add new facts to the KB:

1. *Recognition (individual classification)*: for every individual, find the classes to which it is known to belong. Since classes are organized in an IS-A hierarchy by the subsumption relation, what is usually computed is the *most specific* subsuming concept(s).
2. *Consistency checking*: see if the individual violates any of the necessary conditions associated with the classes in which it has been asserted to be an instance (e.g., check that Calvin doesn't have more than one name).
3. *Propagation*: when an individual is asserted to be in a concept, if it is not currently known to satisfy some of the necessary conditions of that concept, then these are asserted of the individual. This can cause other objects to take on new properties. For example, if Adam, whose children include Cain and Abel, is asserted to satisfy **all**(children,TALL), then both Cain and Abel become instances of TALL. Note the difference from semantic and object-oriented data models, where a class definition imposes constraints that must already be known to hold of an individual.
4. *Rule/trigger application*: if an individual is recognized by the antecedent of a rule, the consequent description is asserted to hold true of it. This may lead to another cycle of recognition, consistency-checking, and rule-firing.

Each object can be assigned its own individual description in the full language used for concept definitions (e.g., "Bill has age 45 and has bought between 12 and 15 things"), which is then fully used in query processing (e.g., Bill is recognized as a GOOD-CUSTOMER). Although this aspect of DLs—essentially the ability to have undeclared views and perform view updates—will not be considered in this paper, it is a prime distinguishing feature of DL-based KBMS from other deductive DBMS.

3 Getting data into DL reasoners

3.1 Motivation

Many successful applications of description logic reasoners support *exploration and browsing* through a conceptual space consisting of general concepts, individuals, and their properties. For example, Software Information Systems such as LASSIE [14] and COMET [23]

help programmers familiarize themselves with the architecture and policies of large software systems, with thousands of component modules, in order to support orderly evolution and reuse. Such systems must contain data about existing code, data which can often be extracted by code analysis tools, such as CIA [13]; such tools frequently create relational databases containing information about procedures, variables, their uses, etc. Software Information Systems may also have access to project-management data, also stored in standard DBMS.

Other systems, such as IMACS [11] and CANDIDE [2, 4], support analysts who try to identify correlations and trends in large databases accumulated for other reasons—what has been termed *data mining/exploration*. In this case again, the data is usually available in some RDBMS.

In such situations, interfaces built on DLs provide several advantages:

- *A conceptual (or semantic) model of the domain*: rather than just representing the (flat) schema of the database, DLs can express an object-based model that is much more natural, richer in semantics, and supports redundancy and multiple viewpoints. DLs are particularly suitable for this task given their origin in research on natural language understanding. A conceptual model is especially valuable in the case of heterogeneous data sources, and applications of DLs for such tasks are cited in several places [5, 11, 19].

- *Data validation*: a familiar problem encountered in data exploration is the fact that the data in the original database is often "corrupt"; this is usually not detected until constraints stated in the much more expressive language of the DL model are evaluated.

- *Validation and organization of queries and views*: the subsumption relationship can be used to organize queries and views in an IS-A hierarchy. This supports query development by iterative refinement [3, 26, 27], and query reformulation by generalization in the case of queries that return empty answers [2]. It is also possible to determine if queries are inconsistent, and thus alert the user to problems.

- *Query and result reification*: by making it easy to add new concepts, relationships and individuals, the analyst can organize the intermediate results of the analysis (for example, by creating individuals corresponding to queries) in much more flexible forms than the original tabular databases [11].

3.2 The problem

Previous research in data management has considered the problem of connecting so-called expert systems to databases, particularly providing Prolog programs with access to relational databases [12, 17, 20, 28]. An influential early paper [28] distinguished two ways for a reasoning system to obtain data from the database:

- Loose coupling: data is to be retrieved *en masse* from the database as a snapshot, and thereafter all reasoning will continue in the KB.
- Tight coupling: data is obtained from the database on demand, as the reasoning proceeds.

The first approach appears quite simple on the face of it, especially for systems based on logic programming, for which there is a very natural correlation between predicates in the KB and tables in a relational DB. Even for frame-based representation systems such as KEE [1], there are mechanisms for specifying relatively simple mappings between relations and slots in frames. One drawback of such an approach is that the KB must usually develop its own efficient data access mechanisms if it is to handle reasoning with large amounts of data.

The difficulties of tightly coupled systems, on the other hand, revolve around the fact that they access the data "one tuple at a time," which is a very ineffective way of interacting with a DBMS. Therefore most research has tended to focus on ways to avoid this difficulty.

The standard approach to coupling databases to reasoners has been to add to the knowledge base classes/concepts that model the relations appearing in the database, and then use the reasoning facilities of the KBMS to connect these to the conceptual model of the domain already present. To see why even this might not be a trivial task consider the case when the relational database has a table `PEOPLE(name,father,mother)`, while the conceptual domain model has a concept `PERSON`, with role `children`. In a system such as CLASSIC, which has traded expressive power to gain efficiency in reasoning, such mappings cannot be expressed.

Our initial approach was therefore different. We wrote procedures that would traverse each relation in turn, and create new individuals or add information about old ones by invoking operations on our system. To our dismay, we discovered that when trying to add even only a few thousand individuals, the CLASSIC system—which is the fastest of the currently available description logic reasoners [18]—took an unacceptably long time (more than 24 hours of processing). Our diagnosis of the problem was the following:

- Description logic reasoners, like many previous knowledge representation systems in AI, assume that

there will be relatively many concepts (schema elements), and relatively few, but complex, individuals. Therefore their processing is oriented to dealing with one object at a time: for example, each individual is classified in turn with respect to the hierarchy of defined concepts.

- Knowledge representation and reasoning systems, such as description logic reasoners, are set up to make inferences, and spend much of their time looking to make such inferences. However, data from a database is mostly *simple* and *complete*, so that inferences almost never add new information (though inferences may be useful for checking consistency).

3.3 A new approach: delegating inferences

Although our long-term goal is to have description logic reasoners support persistence and optimization directly, at this stage we wished to exploit rather than reproduce the considerable amount of work that goes into data organization and query optimization in DBMS. We therefore turned to the following two-phase approach, where the inferences to be applied by the KBMS to the individuals to be loaded from the databases are mostly done by the DBMS:

- Replace, whenever possible, the process of making inferences about database individuals by a collection of queries directed to the relational database, which performs them in "bulk," so to speak, and then returns results that can be added to the KB through a "back door," without further processing.[2] For example, for each class description we will issue a query that retrieves exactly the instances in that class, thereby avoiding the classification inference.
- For those kinds of inferences that cannot be so computed (usually because they require an indeterminate number of new intermediate results), use queries to eliminate from consideration those individuals for which inferences will not add new information; this leads to a (usually much shorter) list of individuals that later need to be explicitly considered by the KBMS. For example, use queries to look for individuals that satisfy only the left hand, but not the right hand side of rules.

Therefore, our general approach to coupling a description logic reasoner with a database will require three components:

- *Query generation*: since queries, views, necessary conditions, and rules are all specified using descriptions, we will need general ways to obtain queries corresponding to descriptions. This subject will be treated in Sections 4 and 5.
- *Creating structured objects with fillers for their roles*: in response to a query, an RDBMS is only capable of generating flat tables, relating integers and strings. From these tuples, we need ways of constructing individual objects and roles relating them, so that they can be stored in the KB. This topic will be considered in Section 6.
- *A processing strategy*: the above algorithms need to be connected so that the database snapshot being "loaded" results in a coherent and consistent KB, identical to the one we would have obtained if the facts were added one by one to the DL reasoner. This and related issues are discussed in Section 7.

4 Translating descriptions to queries

As intimated above, we need to have tables in the database corresponding to concepts and roles in the KB, e.g., a `PERSON_Table` corresponding to `PERSON`, and a `bought_Table` corresponding to `bought`. The cleanest way to establish a correspondence between values in these tables and the facts in the KB is to have each tuple in a "concept table" correspond to an individual in the KB, and each tuple in a "role table" correspond to a role-filler relationship. Therefore, a concept table should have as an attribute the key necessary to distinguish the corresponding individuals, while a role table will have as attributes the keys of both the domain and range of the binary relation. For simplicity, we shall assume in this paper that all keys are single values; initially, we will have a sole (key) attribute in a concept table and designate it **id**[3]; the two attributes of a role table will be the *source* **id** and the *destination* **val**.

4.1 Primitive concepts and roles

Primitive concepts are intended to model so-called "natural kinds"—concepts that do not have necessary and sufficient conditions as definitions—and thus need to have their instances explicitly designated. Hence we must rely on an external agent to list them. We do so by requiring for every primitive concept `C` to be found in the database, a query defining a view `C_Table`, with key `id`, such that each tuple `t` in `C_Table` corresponds to a distinct individual (identifiable, somehow, through `t`) in the concept `C`.

Similarly, for every role and feature `r` to be loaded, we require a query defining a view `r_Table` such that every tuple `(s,d)` in `r_Table` represents an instance of a relationship `r` between the individuals identified by

[2] Deductive DBMS such as Nail! [24] originally took a similar approach in the sense that they generated SQL queries from non-recursive Horn clauses.

[3] This simple structure for a concept table will be expanded a bit later.

s and d. (Note that later on we will end up treating features in a different, special way.)

For example, in the case of the PEOPLE(name,father, mother) table, introduced earlier, the view children_Table, for the children role, would be defined by the SQL query

```
(SELECT father, name FROM PEOPLE)
UNION
(SELECT mother, name FROM PEOPLE).
```

4.2 Composite descriptions—a naive approach

In contrast to primitive concepts, it is relatively easy to obtain an SQL query that computes the instances of a composite description based on the denotation of its components, due to the precise and compositional semantics of DL-type languages.

For example, if instances of the concepts C and D are computed by the queries Q_C and Q_D, then **and**(C,D) can be obtained by the query Q_C INTERSECT Q_D. Some additional tentative translations from CLASSIC descriptions to SQL queries are illustrated in Table 1, which presents a selected set of mappings by way of a recursive function, called *getSQL*.

4.3 Problems with the naive translation

Unfortunately, the translations in Table 1 suffer from several problems.

To begin with, as first noted in [15], the translations are in certain cases incorrect because a relation r_Table only lists those pairs of values for which the relationship holds explicitly. This means that all the individuals that are in the domain of r but are not related to any specific individual (i.e., have zero fillers for r) will not appear in the first column of the table. Therefore the translations of descriptions **all**(r,C) and **at-most**(n,r) are wrong: for example, every object having no r-fillers also satisfies **all**(r,C), but our outer loop fails to return these. Therefore, our proposed translations for **all** and **at-most** are in fact really for the descriptions **and**(**all**(r,C),**at-least**(1,r)) and **and**(**at-most**(n,r),**at-least**(1,r)).

One proper translation for **all**(r,C) would be something like

```
SELECT x.id FROM ANY-OBJECT x
WHERE x.id NOT IN SELECT y.id FROM r_Table y
                  WHERE y.val NOT IN getSQL(C)
```

for which we would need to generate a table ANY-OBJECT of all the objects—the union of all primitive concept relations. In addition to the obvious size problem, this runs into the difficulty that different relations for primitives may have different kinds of keys.

The second class of problems concerns the fact that the queries generated by the above process are usually extremely inefficient, compared to hand-coded translations. The following are some of the sources of inefficiency:

1. *Nested queries:* in general, RDBMS are designed to optimize joins in queries that are in "canonical form" (where several variables ranging over relations are introduced in a single FROM clause) rather than having nested queries. Although some research has been done on the subject of un-nesting such queries (e.g., [22]), most RDBMS will not recognize that the four-level nested query generated for **all**(r,**all**(s,C)) (two levels for r and s each), may be rewritten in the following way

   ```
   SELECT x.id FROM ANY-OBJECT x WHERE true
   MINUS
   SELECT y.id FROM s_Table z,r_Table y
   WHERE y.val=z.id AND z.val NOT IN getSQL(C)
   ```

   which has a single join and a difference.

2. *Redundant joins*: the use of binary relations in description logics results in an extreme form of "normalization," where a relation such as PERSON(id,age,gender,height) is decomposed into Person_Table, age_Table, gender_Table, etc., so that finding all 45-year old male persons requires three relations to be joined. Although these relations are in fact views derived from the same base relation, PERSON, today's query processors are not capable of using functional dependency information (age and gender are functionally dependent on id) to eliminate the unnecessary joins.

3. *Unrealized loop fusion:* most SQL query processors do not perform detailed code analysis to discover that the SQL queries for the descriptions **at-least**(2,children) and **at-most**(4, children) can be executed in one single scan over children_Table, with conjoined predicates in the HAVING clause.

4. *Absence of common subexpression analysis and caching*: especially when issuing queries for a whole host of concepts in a subsumption hierarchy, we will find the same sub-query being executed over and over again for nested expressions. If we are in a situation where space is plentiful but time is scarce, we might want to cache previous results and reuse them. Again, there has been research on techniques for locating and reusing parts of previous query answers [16], but this has not found its way into implemented systems.

Our approach to these efficiency problems will be to manipulate the source *descriptions* to new forms, from which we can emit directly more efficient SQL queries. The special syntax of descriptions makes it much easier for us to perform such manipulations in the original

Descr. D	*getSQL*(D)
and(C,D)	*getSQL*(C) INTERSECT *getSQL*(D)
at-least(n,r)	SELECT x.id FROM r_Table x WHERE true GROUP BY id HAVING (n ≤ COUNT(*))
at-most(n,r)	SELECT x.id FROM r_Table x WHERE true GROUP BY id HAVING (n ≥ COUNT(*))
all(r,C)	SELECT x.id FROM r_Table x WHERE x.id NOT IN SELECT y.id FROM r_Table y WHERE y.val NOT IN *getSQL*(C)
fills(r,i)	SELECT x.id FROM r_Table x WHERE x.val = i

Table 1: Naive recursive translation function *getSQL*

form, rather than trying to optimize a translated SQL expression.

5 Correct and efficient translation

As we remarked earlier, constructors such as **all** and **at-most** may require the creation of the domain ANY-OBJECT. However, if such terms are always conjoined (with **and**) to some concept D that already has a table, then we can avoid computing the table ANY-OBJECT, and instead use set-difference to remove all objects that do not satisfy the **all** restriction. Therefore, **and**(D,**all**(r,C)) is translated as

```
SELECT d.id FROM D_Table d WHERE true
MINUS
SELECT y.id FROM r_Table y
WHERE y.val NOT IN getSQL(C)
```

incidentally eliminating one level of nesting in query evaluation. Henceforth, we therefore restrict our translation to those "safe" descriptions that have at least one primitive or named concept as a conjunct.

The following are additional transformations we apply to descriptions in order to be able to generate more efficient SQL queries.

One obvious way to reduce the level of nesting in queries is to collapse nested **and** terms according to the rule

and(B,**and**(C,D)) ≡ **and**(B,C,D)

so that queries can be obtained by conjoining with AND the restrictions due to the second and later subterms (C and D in this case) to the variable ranging over the primitive concept (B in this case).

In order to combine the loops due to **at-least** and **at-most** bounds on the same role, we introduce a new description constructor, **bounds**, which satisfies the equivalence

bounds(n,m,r) ≡ **and**(**at-least**(n,r),**at-most**(m,r))

Similarly, in order to combine the intersections caused by multiple checks for role fillers, we extend the syntax of the **fills** constructor so that it can take a list of individuals as arguments:

fills(r,{i1,i2,...}) ≡
and(**fills**(r,i1),**fills**(r,i2),...))

As far as nested descriptions are concerned, we observe that they always occur inside **all**-restrictions, where they are negated in the WHERE-clause of the query. A description such as **all**(r,**all**(s,C)), can be written in logical notation as $(\forall y)r(x,y) \Rightarrow ((\forall z)s(y,z) \Rightarrow C(z))$, which is equivalent to $\neg[(\exists y)r(x,y) \wedge (\exists z)s(y,z) \wedge \neg C(z)]$. The outer $\neg$ is encoded by the SQL MINUS construction, and the inner expression is just a join on the roles. If C has further nested **all**-restrictions, they can be similarly eliminated. As a result, our queries will have at most one level of nesting. For this reason, we will use a different translation function, called *getNotSql*, to translate nested concepts into their complements.

In order to avoid redundant joins involving features (which are functionally dependent on the concept key), we will essentially "build in" the joins in the original tables for primitive concepts. Specifically, we will require every feature to have associated some primitive concept, which acts as its domain, e.g., **age** might be associated with PERSON, attendsSchool with STUDENT.

Features will then appear as attributes in the relational view corresponding to the domain primitive concept, rather than as separate binary relations. For example, the relation PERSON_Table will have attributes **id**, **age**, **gender**, etc. Note that there is still exactly one tuple for every individual person, because these attributes are functionally dependent on the key *id*. Also, we will pre-process descriptions so that role and feature restrictions are grouped separately.

Finally, a basis for dealing with common subexpressions is the observation that when using defined concepts, such as USA-LOCATION, the same concept identifier usually occurs in several other definitions; therefore it would be reasonable to compute and save the relations corresponding to named concepts, as they are encountered. Also, the concept hierarchy should be traversed in an order where a concept is seen before it is used as a restriction in some other concept.[4] If redundant common subexpressions are still an important problem, we can analyze the original set of descriptions in the KB

[4]Recursive definitions are not allowed in CLASSIC.

```
function getSQL(and(B,D1,D2,...))
  var q : QUERY

  q.SELECT := [v.id];
  q.FROM   := [B_Table v]
  for every Di do
    if Di is a named concept or role restriction
     then getTopSql(v, Di, q)
    if Di is a feature restriction
     then translateFeatureRestr(v, Di, q)
  return q
```

Figure 1: Revised function *getSQL*

to find common subterms—a task that is much easier than the equivalent process in SQL, and is in fact done by some DL processors currently.

5.1 Translating descriptions more carefully

Suppose C is a description with at least one conjunct, B, which already has a corresponding relational table (because it is primitive or has already been computed). The revised function *getSQL*, presented in Figure 1, computes an SQL query corresponding to C by creating a record structure of type QUERY, with fields representing fragments of an SQL query: the SELECT, FROM, WHERE clauses, as well as lists of queries to be INTERSECT-ed and MINUS-ed with it. The function takes as argument a conjunctive description **and**(B,D_1,D_2,...), and uses function *getTopSql* to add new fragments to the query record structure q. *getTopSql*, and its internal sub-procedure *getNotSql*, are in turn described in Table 2, as case statements on the structure of their arguments, indicating what additions are made to the respective FROM, WHERE, INTRSC and MINUS fields of q. (Note that the extra argument v for *getTopSql* ranges over the named concept for which we already have a table (i.e., B), while argument z for *getNotSql* helps in translating nested role restriction concepts inside **all** constraints. Also note that both functions take as input/output parameter the query Q that is to be modified; this helps "hide" the top query, while processing each of its roles' restrictions.)

The function, *translateFeatureRestr* (not detailed here), translates to SQL restrictions on features (such as **same-as**, **all**, **fills**) by taking advantage of the fact that features are functionally dependent on the key of the concept table in which they are stored, thus avoiding the need for nested queries for **all** restrictions, for example. Due to lack of space we cannot provide the full details of translating feature restrictions (including translating **same-as** constraints), and content ourselves with giving an example translation. Suppose that primitive class PERSON has table PERSON_Table(id,age,...) (indicating that age is a feature with domain PERSON), while its primitive subclass STUDENT with features attends, year, etc., has table STUDENT_Table(id,attends,year,...). Then the description

```
and(STUDENT,
      all(attends,IVY-LEAGUE),
      all(year, (one-of Junior,Senior)),
      fills(age,22))
```

corresponding to a query for 22-year old students attending an Ivy League school in their junior or senior year, will generate the SQL query

```
SELECT s.id
FROM PERSON_Table p, STUDENT_Table s
WHERE (s.attends IN IVY-LEAGUE_Table) AND
      (s.year IN (Junior, Senior)) AND
      (s.id = p.id) AND (p.age = 22)
```

Note the re-use of variable s for both attends and year restrictions, and the join needed to access age, which is stored in a different table.

6 Cross references to individuals

One of the advantages of a simple association between database relations and base predicates in a data model based on predicate calculus is that there is no difficulty in translating the data in the database to or from arguments of the predicates, since they are both values such as integers and strings.

In our case the relationship is once again more complex, because we deal with an object-oriented world, while the database is still record-oriented. Hence we encounter some well-known problems [21], where the same individual may be identified by different keys in different relations (e.g., social security or employee number), or the same data value may have different meanings in different columns of tables (e.g., 'M' may mean male in attribute gender, and Monday in attribute dayOfWeek).

We therefore require for every primitive concept a function *DB-to-KB-id* which takes an attribute value and generates from it an appropriate object identifier name in the CLASSIC world. Moreover, in order to translate descriptions to queries we also need a function *KB-to-DB-id*, for use with **fills** or **one-of** constructors. In our experience, both these functions can be arbitrarily complex, and need to be context-dependent.

Additional problems arise because CLASSIC normally makes the open world assumption, while databases are usually considered to have complete information. From a practical point of view, the greatest difficulty is caused by null values, but we defer this issue to a more complete paper.

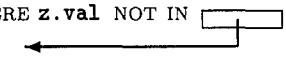
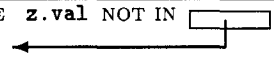
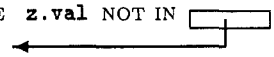
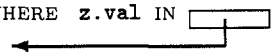
Descr. D	*getTopSql*(v,D,Q)	*getNotSql*(z,D,Q)
named concept C	FROM C_Table y WHERE v.id = y.id	WHERE z.val NOT IN C_Table
fills(r,i)	FROM r_Table y WHERE v.id = y.id	WHERE z.val NOT IN SELECT y.id FROM r y WHERE y.val=i
fills(r,{i_1,i_2,...})	INTRSC SELECT y.id FROM r_Table y WHERE y.val IN (i_1,i_2,...) GROUP BY id HAVING (COUNT(*) = n)	WHERE z.val NOT IN 
at-least(n,r)	INTRSC SELECT y.id FROM r_Table y GROUP-BY id HAVING (COUNT(*) $\geq$ n)	WHERE z.val NOT IN 
bounds(r,lo,hi)	INTRSC SELECT y.id FROM r_Table y GROUP-BY id HAVING (COUNT(*) $\geq$ lo) & (COUNT(*) $\leq$ hi)	WHERE z.val NOT IN 
at-most(n,r)	MINUS SELECT y.id FROM r_Table y GROUP-BY id HAVING (COUNT(*) < n)	WHERE z.val IN 
all(r,C)	MINUS Q′ where Q′= {SELECT z.id; FROM r_Table z;} and we have called *getNotSql*(z,C,Q′)	FROM r_Table y WHERE z.val= y.id call *getNotSql*(y,C,Q)
one-of(i_1,i_2,...)	—	WHERE z.id NOT IN (i_1,i_2,...)
range(n,m)	—	WHERE z.val< n OR z.val> m
test(f)	—	WHERE NOT f(z.val)

Table 2: Functions translating role restrictions to SQL

7 Strategies for connecting to the database

Our original data exploration context was such that users were looking at existing, static data, for which they were receiving monthly updates. In such an environment, it is reasonable to take a loose coupling approach where (1) the views for the various concepts in the conceptual hierarchy are generated in the relational database[5]; (2) the resulting data is transferred to the machine that holds the KB; (3) individuals are created and added to the KB through a "back door" that avoids all inferences; and (4) the KBMS is handed the results of some inferences performed "in bulk" through SQL queries, and a list of specific individuals and inferences that the description logic reasoner still must try.

The inferences performed by the description logic reasoner are taken care of as follows:

- *Concept classification*: the descriptions corresponding to the classes in the conceptual schema are processed in the description logic reasoner as usual, since this does not require access to the database. We therefore obtain a subsumption hierarchy for the concepts, which are guaranteed to be consistent.
- *Individual classification*: for every concept C in the concept subsumption hierarchy, we obtain a view definition C_Table with one tuple per instance for C. Since we desire only the *lowest* classes in which an individual is an instance, we use a query to exclude any tuples appearing in the tables corresponding to the immediate subclasses of C in the IS-A hierarchy.
- *Integrity checking*: for every primitive class C, we take the associated description D recording necessary conditions, and obtain a query expressing the concept **not**(D), which expresses the conditions of not being a C. When this query is run against the tuples in C_Table, any individuals returned fail to satisfy the necessary conditions, and are rejected by our system.
- *Propagation*: for a primitive class C, with necessary conditions described by D, the tuples in C_Table that are *not* returned by the SQL query corresponding to D correspond to individuals that are *not known to satisfy* the necessary conditions.[6] These individuals need to have propagation inferences performed, and this is done by "turning on" the CLASSIC system's inference machinery for them.
- *Rule firing*: Similarly, given a rule "If C then D,"

[5] As mentioned earlier, the concepts should be topologically sorted according to the "uses" hierarchy.

[6] Contrast this with the previous case, which retrieves individuals *known not to satisfy* the necessary conditions.

which in CLASSIC must be associated with some concept C', we obtain a query for the description `and(C,not(D))`, and the results of this query over `C'_Table` produces the individuals for which we must still apply the rule in the CLASSIC system, because the left hand side holds but the right hand side is not known to hold yet.

The result of this scheme is that the final state of the KB is identical to that which would have been achieved if all facts were added through the normal interface to CLASSIC, except that in the applications we have considered there are only a very few individuals, if any, for which propagation and rule-firing must still be done. The result is much more efficient database loading: we are currently able to load over twenty-thousand individuals in this way, with some individuals having eight pages of ASCII text retrieved from the database.

One problem we face when new facts are inferred is that they are not available in the original relational database. We believe it is neither possible nor desirable to translate the inferences back to the database (see [29] for relevant arguments). This means that the KBMS must be prepared to save the inferred facts, or additional objects created by its users, and connect them to the facts loaded from the database.

8 Summary

We have summarized the features of KBMS based on description logics, and have discussed a variety of situations where it is necessary to connect them to existing relational databases. Because of the object-oriented nature of DLs and the forward-chaining nature of the inferences performed by description logic reasoners, we have chosen a loose-coupling approach to loading database facts.

The novelty of our approach lies in the idea that individual objects and associated facts should be loaded from the database into the KB through a back door, where the usual inference machinery is turned off. Instead, many inferences are performed by asking the appropriate queries, leaving to the DBMS the task of optimizing their execution. This leaves the KBMS to deal with the comparatively fewer cases where complex reasoning is needed. Note that this is clearly different from standard approaches to the loose coupling of frame-based KBMS to DBMS (e.g., [1]);

We have also shown how the pre-processing of descriptions, with their special syntax that avoids variables, makes it possible to obtain forms that result in more efficient SQL queries being generated.

The ideas presented here are applicable to other DLs and query languages, not just CLASSIC and SQL. First, it is relatively easy to extend the *getSQL* mapping to other concept constructors. Second, Devanbu [15] has developed a general-purpose declarative notation for expressing succinctly mappings from DLs to other formalisms, and an interpreter that generates code from this notation. This can be used, among other things, to describe the translation of CLASSIC to SQL. The reader is referred to [15] for additional details.

We conclude by pointing out that the techniques developed here have been put to use in the context of a significant application, whose goal is to aid a data analyst in iteratively exploring large amounts of data [11]. This task of "data archaeology" is well-supported by the kind of object-centered, deductive view offered by CLASSIC, and the techniques described here are critical to letting the analyst view the data stored in large relational databases through the window of a description logic.

Acknowledgments

We gratefully acknowledge significant help received from Prem Devanbu. We also thank the key people who participated in implementation aspects of this research, most notably Charles Isbell and Boris Altman, as well as Lori Alperin Resnick and Tom Kirk. The comments of Dan Lieuwen sparked a major revision of material in Section 5, and for this we are most grateful. Alex Borgida is partially supported by NSF Grant IRI 91-19310.

References

[1] IntelliCorp, "Bridging the information gap," in *A Review of Products, Services, and Research*, AAAI-87, Seattle, 1987, pp. 70–71.

[2] Anwar, T. M., Beck, H., and Navathe, S., "Knowledge mining by imprecise querying: A classification-based approach," *Proc. 8th IEEE Data Engineering Conf.*, Tempe, AZ, February, 1992, pp. 622–630.

[3] Beck, H. W., Gala, S. K., and Navathe, S. B., "Classification as a query processing technique in the CANDIDE semantic data model," *Proc. 5th IEEE Data Engineering Conf.*, Los Angeles, February, 1989, pp. 572–581.

[4] Beck, H. W., Anwar, T. M., and Navathe, S. B., "Classification through conceptual clustering in database systems," *Proc. 1st Intl. Conf. on Information and Knowledge Management*, Baltimore, MD, November, 1992, pp. 465–472.

[5] Beck, H. W., Anwar, T. M., and Navathe, S. B., "A conceptual clustering algorithm for database schema design," to appear in *IEEE Trans. on Knowledge and Data Engineering.*

[6] Bergamaschi, S., Bonfatti, F., and Sartori, C., "Entity-Situation: A model for the knowledge representation module of a KBMS," *Proc. EDBT'88 – Advances in Database Technology*, 1988, pp. 578–582.

[7] Bergamaschi, S., and Sartori, C., "On taxonomic reasoning in conceptual design," *ACM Trans. on Database Systems* 13(3), September, 1992, pp. 385–422.

[8] Borgida, A., "A new look at the foundations and utility of Description Logics (or Terminological Logics are not just for the `Flightless Birds`)," Technical Report, Rutgers University, 1992.

[9] Borgida, A., Brachman, R. J., McGuinness, D. L., and Resnick, L. A., "CLASSIC: A structural data model for objects," *Proc. 1989 ACM SIGMOD Conf.*, Portland, OR, June, 1989, pp. 59–67.

[10] Borgida, A., and Brachman, R. J., "Intelligence in the interface," *Proc. 2nd Intl. Workshop on Intelligent and Cooperative Information Systems: Core Technology for Next Generation Information Systems*, Como, Italy, October, 1991, pp. 68–75.

[11] Brachman, R. J., Selfridge, P. G., Terveen, L. G., Altman, B., Borgida, A., Halper, F., Kirk, T., Lazar, A., McGuinness, D. L., and Resnick, L. A., "Knowledge representation support for data archaeology," *Proc. 1st Intl. Conf. on Information and Knowledge Management*, Baltimore, MD, November, 1992, pp. 457–464.

[12] Ceri, S., Gottlob, G., and Wiederhold, G., "Interfacing relational databases and Prolog efficiently," *Proc. 1st Intl. Conf. on Expert Database Systems*, Charleston, SC, April, 1986, pp. 207–223.

[13] Chen, Y-F., Nishimoto, M., and Ramamoorthy, C. V., "The C information abstraction system," *IEEE Trans. on Software Engineering*, March, 1990.

[14] Devanbu, P., Brachman, R. J., Selfridge, P. G., and Ballard, B. W., "LaSSIE: A knowledge-based software information system," *Comm. of the ACM, 34*(5), May, 1991, pp. 34–49.

[15] Devanbu, P., "Translating description logics to information server queries," Technical Report, AT&T Bell Laboratories, May, 1992.

[16] Finkelstein, S., "Common expression analysis in database applications," *Proc. 1982 ACM SIGMOD Conf.*, Orlando, 1982, pp. 235–245.

[17] Gosh, S., Lin, C. C., and Sellis, T., "Implementation of a Prolog-INGRES interface," *ACM SIGMOD Record*, Vol. 17, No. 2, June, 1988, pp. 77–88.

[18] Heinsohn, J., Kudenko, D., Nebel, B., and Profitlich, H.-J., "An empirical analysis of terminological representation systems," *Proc. AAAI-92*, San Jose, CA, July, 1992, pp. 767–773.

[19] Illarramendi, A., Blanco, J. M., and Goñi, A., "A uniform approach to design a federated system using BACK," *Proc. Terminological Logic Users Workshop*, KIT–Report 95, Technische Universität Berlin, Berlin, October, 1991, pp. 61–85.

[20] Jarke, M., Clifford, J., and Vassiliou, Y., "An optimizing Prolog front-end to a relational query system," *Proc. 1984 ACM SIGMOD Conf.*, Boston, May, 1984, pp. 296–306.

[21] Kent, W., "Limitations of record-based information models," *ACM Trans. on Database Systems* 4(4), March, 1976, pp. 9–36.

[22] Kim, W., "On optimizing an SQL-like nested query," *ACM Trans. on Database Systems* 7(3), Sept., 1982, pp. 443–469.

[23] Mark, W., Tyler, S., McGuire, J., and Schlossberg, J., "Commitment-based software development," *IEEE Trans. on Software Engineering*, Vol. 18, No. 10, October, 1992, pp. 870–885.

[24] Morris, K., Ullman, J. D., and Van Gelder, A., "Design overview of the Nail! system," *Proc. 3rd IEEE Symp. on Logic Programming*, 1986, pp. 554–568.

[25] Nebel, B., and Peltason, C., "Terminological reasoning and information management," in *Information Systems and Artificial Intelligence*. D. Karagianis, *ed.*. Springer-Verlag, 1991, pp. 181–212.

[26] Patel-Schneider, P. F., Brachman, R. J., and Levesque, H. J., "ARGON: Knowledge representation meets information retrieval," *Proc. 1st Conf. on Artificial Intelligence Applications*, Denver, December, 1984, pp. 280–286.

[27] Tou, F. N., Williams, M. D., Fikes, R., Henderson, A., and Malone, T., "RABBIT: An intelligent database assistant," *Proc. AAAI'82*, Pittsburgh, pp. 314–318.

[28] Vassiliou, Y., Clifford, J., and Jarke, M., "How does an expert system get its data?" *Proc. 9th VLDB Conf.*, Florence, 1983, pp. 70–72.

[29] Wiederhold, G., "Mediators in the architecture of future Information Systems," *IEEE Computer 21(3)*, March, 1992, pp. 38–50.