Knowledge Representation using Schema Tuple Queries

Michael J. Minock
Department of Computing Science, Umeå, Sweden
Email: mjm@cs.umu.se

Abstract

This paper introduces schema tuple queries and argues for their suitability in representing knowledge over standard relational databases. Schema tuple queries are queries that return only whole tuples of schema relations. In particular a subclass of the schema tuple queries is identified that is decidable for satisfiability and is closed over syntactic query difference. These properties enable the determination of query containment, equivalence and disjointness. Given this, the identified query class possesses many of the desirable properties of description logics. Additionally such schema tuple queries may be directly translated to SQL and applied over standard *n*-ary database relations.

1 Introduction

The relational model has maintained an enduring impact without rival. At a surface level the reason for this might seem to be the widely adhered to SQL language and the large number of commercial systems implemented around that standard. At a deeper level, however, the success of the relational model stems from its firm rooting within first order logic. And, of note, this includes being able to represent full *n*-ary relations such as the relation `AttendsFilm` in the following schema[1]:

```
Friend(name, age, gender)

Theater(name, address)

Movie(movieID, title, year)

AttendsFilm(friend, movieID, theater).

IsDirector(movieID, entertainerID)

IsCastMemeber(movieID, entertainerID)

Entertainer(entertainerID, name, dateOfBirth)
```

Administrators of relational databases commonly define such tables, often using many more attributes than three. We thus take as an initial requirement that *our knowledge representation language must be able to intensionally describe any finite extension over a set of value-oriented n-ary relations*. Such intensional expressions are called *queries* in this work. Of course in addition to denoting answer sets, 'queries' may be put to use in view definition, constraint specification, meta-data annotation, rule definition, etc.

[1] Primary keys are underlined, while foreign key attributes are *italicized* here.

The queries that we shall consider are the *schema tuple queries*[2]. Such queries are tuple relational queries[12] that return only whole tuples from relations of the schema, not arbitrary combinations of projected attributes. Thus, to obtain movies of the year 1999, we would write $\{m|Movie(m) \wedge m.year = 1999\}$ rather than $\{(m.movieID, m.title, m.year)|\ Movie(m) \wedge m.year = 1999\}$. The restriction to return only whole tuples from relations of the schema is in contrast to the flexibility of normal tuple relational queries and relational algebra. There are three principle reasons why the schema tuple restriction is made:

The first reason is that set difference is fully defined over schema tuple query answer sets. Since the atoms of sets are whole tuples, we can speak of taking the 'typed' difference between two tuple sets. For example we may speak of taking the difference between the set of all movies ($s_1 = \{m|Movie(m)\}$), and the movies made in the 1990's ($s_2 = \{m|Movie(m) \wedge m.year \geq 1990 \wedge m.year < 2000\}$). In fact we can derive such query differences syntactically without having to materialize answer sets ($s_1 - s_2 = \{m|Movie(m) \wedge m.year < 1990\} \cup \{m|Movie(m) \wedge m.year \geq 2000\}$). We may also speak of taking differences between tuple sets of heterogeneous types. For example the set of all movies (s_1) minus the set of all entertainers ($s_3 = \{e|Entertainer(m)\}$), is simply the set of all movies ($s_1 - s_3 = s_1$). If, however, we relax the schema tuple query restriction and permit projections and thus non-typed relations, then we face the prospect of taking differences between non-union compatible, non-typed relations. What meaning could $\{(m.movieID, m.title, m.year)|Movie(m)\} - \{(m.year)|Movie(m) \wedge m.year \geq 1990 \wedge m.year < 2000\}$ have? These relations are not union-compatible and thus set difference between them is not well defined.

The second reason for the schema tuple restriction centers around the perspicuity of intensional descriptions. Because tables correspond roughly to the nouns and verbs of a domain, descriptions should center around collections of such objects. Unrestricted projection would probably so complicate query expressions that the task of generating crisp, understandable natural language descriptions would be doomed.

The third, and perhaps the most compelling reason for invoking the schema tuple query restriction is the existence of a particular subclass of schema tuples queries that is decidable for satisfiability and is closed over complementation. Though the specification language of this class falls under the well known Schönfinkel-Bernays class[3] of function free formula with equality that contain quantifier sequences of only the $\exists^*\forall^*$ form, the specification language here is closed over complementation. The general Schönfinkle-Bernays class, in contrast, does not remain decidable when closed over complementation ($\neg\exists^*\forall^*\phi \equiv \forall^*\exists^*\neg\phi$). As shall be shown, the schema tuple query restriction is critical to maintaining the decidability of our specification language over complementation.

In this paper, we identify two classes of schema tuple queries which use the specification languages $\mathcal{L}$ and Q respectively. Formulas in the language $\mathcal{L}$ are signed quantifier sequences $\exists^*$ over conjunctions of predicates over a single free tuple variable[3]. Formulas in the language Q are finite disjunctions of formulas within $\mathcal{L}$. Of note the language Q is decidable for satisfiability and the class of queries specified using Q is closed under syntactic query difference. These properties enable the determination of query containment (subsumption), equivalence and disjointness for queries specified in Q.

[2]The term 'schema tuple query' is being newly introduced here and is based on the restriction that such queries return only *schema tuples*.

[3]$\mathcal{L}$ does **not** denote the set of all logical formulas in this work.

1.1 Organization of this Paper

Section 2 formally defines the language $\mathcal{L}$ and its closure over disjunction, $\mathcal{Q}$. Both $\mathcal{L}$ and $\mathcal{Q}$ are proven[4] decidable for satisfiability; that is one may determine if there exists a database state for which a query specified in $\mathcal{L}$ or $\mathcal{Q}$ would return a tuple. Section 3 shows that queries built using $\mathcal{Q}$ are closed over syntactic query difference and complementation. Based on the decidability of $\mathcal{Q}$, we may thus decide query containment and equivalence over $\mathcal{Q}$. Section 4 relates this work to prior work and gives future directions. Section 5 gives conclusions.

2 The Languages $\mathcal{L}$ and $\mathcal{Q}$

2.1 Foundations

We assume the existence of two disjoint, countable sets: $\mathcal{U}$, the *universal domain* of *atomic values*, and $\mathcal{P}$, *predicate names*. Let U be a distinct symbol representing the type of $\mathcal{U}$. A *relation schema* R is an n-tuple $[U,...,U]$ where $n \geq 1$ is called the *arity* of R. A *database schema* D is a sequence $\langle P_1 : R_1, ..., P_m : R_m \rangle$, where $m \geq 1$, P_i's are distinct predicate names and R_i's are relation schemas. A *relation instance* r of R with arity n is a finite subset of $\mathcal{U}^n$. A *database instance* d of D is a sequence $\langle P_1 : r_1, ..., P_m : r_m \rangle$, where r_i is an instance of R_i for $i \in [1..m]$.

Definition 1 *(Schema tuples)*
A *schema tuple* τ of the database instance d is the pair $\langle P_i : \mu \rangle$, where $1 \leq i \leq m$ and $\mu \in r_i$.

We say that the *type* of τ is P_i and we say the components of τ are the components of μ. The schema tuple τ_1 is equal to the schema tuple τ_2 if and only if τ_1 and τ_2 match on type and they match on <u>all</u> components. Thus if $\tau_1 = \langle$IsDirector : ['0133093','43252']$\rangle$ and $\tau_2 = \langle$ISCastMember: ['0133093','43252']$\rangle$ then $t_1 \neq t_2$. The positional access operator is extended to the schema tuples to mirror the standard tuple relational calculus. Thus $\tau_1[2] =$ '43252'. Furthermore we shall assume that tuple components may be accessed through attribute names (e.g. $\tau_1.entertainerID$). Finally we shall assume that $\mathcal{U}$ is totally ordered so that arithmetic comparison operators ($=,>,<,\geq,\leq$ and $\neq$) are well defined.

We now recursively define the set of *tuple relational formulas*. *Atomic formulas* provide the base for the inductive definition. The atomic formula $P(z)$, where P is a predicate name and z is a tuple variable, means that the tuple referred to by z is a schema tuple of type P. We term such formulas *range conditions*. $X\theta Y$ is an atomic formula where X and Y are either constants or component references (of the form $z.a$) and θ is one of the arithmetic comparison operators. We term such formulas to be *simple conditions* if either X or Y are constants and to be *join conditions* if both X and Y are component references. Lastly we include atomic formula $X\varepsilon C$ where X is a component reference, C is a set of constants and ε is a set membership operator ($\in$ and $\notin$).We term such formulas *set conditions*. Finally if F_1 and F_2 are tuple relational formula, where F_1 has some free variable z, then $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$, $(\exists z)F_1$ and $(\forall z)F_1$ are also tuple relational formulas.

We now define the notion of a schema tuple query.

[4]All of the proofs of the theorems in this paper appear in [17].

Definition 2 *(Schema tuple queries)*
A *schema tuple query* is an expression of the form $\{x|\varphi\}$, where φ is a tuple relational formula over the single free tuple variable x.

When we write the expression $\{x|\varphi\}$ we normally assume that the expression φ is over the free variable x. For the schema tuple τ, $\tau \in \{x|\varphi\}$ iff $\{\tau/x\}(\varphi)$ where $\{t/x\}\varphi$ means to substitute the term t in place of x in φ. Thus a *schema tuple query* is simply the intensional description of a schema tuple set. The actual tuples within this set are the extensional answers to the query.

We shall now turn our attention to several interesting sub-languages of the tuple relational formula that may be used to specify safe schema tuple queries.

2.2 $\mathcal{L}$ and its Decidability

A *basic query component* built over the free tuple variable x is a formula of the form $(\exists y_1)(\exists y_2)..(\exists y_n)\Psi$, where Ψ is a conjunction of range conditions, simple conditions, set conditions and join conditions using exactly the tuple variables $x,y_1,..,y_n$. A *signed basic query component* is either a basic query component or the formula $\neg\Theta$ where Θ is a basic query component.

Definition 3 *(The language $\mathcal{L}$)*
The language $\mathcal{L}$ consists of all formulas of the form:

$$P(x) \wedge (\bigwedge_{i=1}^{k} \Phi_i)$$

where $P(x)$ restricts the free tuple variable x to range over P and Φ_i is a signed basic query component over x.

The following two queries are built using $\mathcal{L}$:

- An SPJ query: "The movies directed by Lucas"

$$\begin{aligned}
&\{m_1|\ell_1\} = \{m_1|Movie(m_1)\wedge\\
&\quad(\exists y_1)(\exists y_2)(\\
&\qquad IsDirector(y_1) \wedge Entertainer(y_2)\wedge\\
&\qquad y_2.lastName = \text{`Lucas'}\wedge\\
&\qquad m_1.movieID = y_1.movieID\wedge\\
&\qquad y_1.entertainerID = y_2.entertainerID)\}
\end{aligned}$$

- A query with negation: "The films made in the year 2000 that have not been seen by any of my male friends"

$$\begin{aligned}
&\{m_2|\ell_2\} = \{m_2|Movie(m_2)\wedge\\
&\quad\neg(\exists y_3)(\exists y_4)(\\
&\qquad Friend(y_3) \wedge AttendsFilm(y_4)\wedge\\
&\qquad y_3.gender = \text{`male'}\wedge\\
&\qquad m_2.movieID = y_4.movieID\wedge\\
&\qquad y_4.friend = y_3.name)\wedge\\
&\quad m_2.year = 2000\}
\end{aligned}$$

Note that $\mathcal{L}$ does not allow for any mixed quantifier sequences within basic query components. Thus a query such as "The movies that are shown at all theaters" is not expressible using $\mathcal{L}$. Cardinality constraints may be expressed in $\mathcal{L}$. Thus, for example, we could specify a query for the movies that have more than 2 directors.

We now arrive at the main result of this section.

Theorem 1 *($\mathcal{L}$ is decidable for satisfiability) [17]*
For all $\ell \in \mathcal{L}$ over the free variable x, we may determine if there exists a database instance d where for some schema tuple τ, $\{\tau/x\}\ell$.

2.3 The Language $\mathcal{Q}$

We define the language $\mathcal{Q}$ as the set of formula which are disjunctions of formula within $\mathcal{L}$. Not surprisingly $\mathcal{Q}$ is decidable for satisfiability as well.

Definition 4 *(The language $\mathcal{Q}$)*
$q \in \mathcal{Q}$ if q is written as a finite expression $\ell_1 \vee ... \vee \ell_k$ where each $\ell \in \mathcal{L}$ and ℓ is over the free tuple variable x.

Theorem 2 *($\mathcal{Q}$ is decidable for satisfiability)[17]*
For all $q \in \mathcal{Q}$ over the free variable x, we may determine if there exists a database instance d where for some schema tuple τ, $\{\tau/x\}q$.

3 Reasoning over $\mathcal{L}$ and $\mathcal{Q}$

We now cover several important properties that hold for queries built over $\mathcal{L}$ and $\mathcal{Q}$.

3.1 Syntactic Query Difference over $\mathcal{L}$

This theorem states that we may describe the set difference of two queries built over $\mathcal{L}$ with a query built over $\mathcal{Q}$.

Theorem 3 *(Syntactic query difference over $\mathcal{L}$ is in $\mathcal{Q}$)[17]*
Let $l_1 \in \mathcal{L}$ and $l_2 \in \mathcal{L}$. Then there is a $q \in \mathcal{Q}$ with the property that for all database instances $\{x_1|l_1\} - \{x_2|l_2\} = \{x_1|q\}$

The following illustrates syntactic query difference between the two example queries of section 2.2.

$\{m_1|\ell_1\} - \{m_2|\ell_2\} = \{m_1|\ell_1 \wedge \{m_1/m_2\}\neg\ell_2\} =$
$\{m_1|$
$\quad (Movie(m_1) \wedge$
$\quad\quad (\exists y_1)(\exists y_2)($
$\quad\quad\quad IsDirector(y_1) \wedge Entertainer(y_2) \wedge$
$\quad\quad\quad y_2.lastName =$ 'Lucas' $\wedge$
$\quad\quad\quad m_1.movieID = y_1.movieID \wedge$
$\quad\quad\quad y_1.entertainerID = y_2.entertainerID) \wedge$

$(\exists y_3)(\exists y_4)($
$Friend(y_3) \wedge AttendsFilm(y_4) \wedge$
$y_3.gender =$ 'male' $\wedge$
$m_2.movieID = y_4.movieID \wedge$
$y_4.friend = y_3.name))$
$\vee$
$(Movie(m_1) \wedge$
$(\exists y_1)(\exists y_2)($
$IsDirector(y_1) \wedge Entertainer(y_2) \wedge$
$y_2.lastName =$ 'Lucas' $\wedge$
$m_1.movieID = y_1.movieID \wedge$
$y_1.entertainerID = y_2.entertainerID) \wedge$
$m_1.year \neq 2000)\}$

3.2 Closure of Difference and Intersection over Q

We now show that Q stays closed over syntactic difference and intersection. This means that we may describe the set difference (or intersection) of two queries built over Q with a third query built over Q. Note that because the entire universe of tuples may be described with a query specified in Q, syntactic query difference provides us with a way to take query complements.

Theorem 4 *(Syntactic query difference is closed over Q) [17]*
Let $q_1 \in Q$ and $q_2 \in Q$. Then there is a $q_3 \in Q$ such that for all database instances $\{x_1|q_1\} - \{x_2|q_2\} = \{x_1|q_3\}$

Theorem 5 *(Syntactic query intersection is closed over Q)[17]*
Let $q_1 \in Q$ and $q_2 \in Q$. Then there is a $q_3 \in Q$ such that for all database instances $\{x_1|q_1\} \cap \{x_2|q_2\} = \{x_1|q_3\}$

3.3 Expressing Constraints

Since functional dependencies constrain the set of legal database states, an expression that necessarily violates a functional dependency must be ruled to be unsatisfiable. We shall represent functional dependencies as universally quantified formulas[1]. In turn these formulas shall be included in the resolution process that decides satisfiability of queries.

Definition 5 *(Functional dependencies as formulas)*
The functional dependency $W \rightarrow V$ over the relation R where W is a set of m attributes and V is a set of n attributes is expressed as the universally quantified formula:
$(\forall x)(\forall y)(P(x) \wedge P(y) \wedge$
$y.w_1 = x.w_1 \wedge \ldots \wedge y.w_m = x.w_m \Rightarrow$
$y.v_1 = x.v_1 \wedge \ldots \wedge y.v_n = x.v_n)$

As an example, the functional dependency *movieId* $\rightarrow$ *title year* on the relation *Movie* is represented by:

$$(\forall m_1)(\forall m_2)(Movie(m_1) \wedge Movie(m_2) \wedge$$
$$m_1.movieId = m_2.movieId \Rightarrow$$
$$m_1.title = m_2.title \wedge m_1.year = m_2.year).$$

Note also that domain rules may be encoded as universally quantified formula and included in Γ. For example the constraint that all film years are greater than 1927 but less than or equal to 2002 could be encoded:

$$(\forall m_1)(Movie(m_1) \Rightarrow$$
$$Movie(m_1.year \geq 1927 \wedge m_1.year \leq 2002)$$

Finally existence constraints may be specified and added to Γ. For example a simple formula within $\mathcal{L}$ may express the fact that *some* movies were made in the 1930's with the exact title, identifier and year information remaining unknown.

We shall use the formula Γ to denote all of the integrity constraints of the domain. Γ is simply a conjunction of universally quantified formulas and formulas of the form $\exists^*\forall^*$. Thus Γ may be Skolemized without inducing an infinite Herbrand universe. When determining the satisfiability of $\ell \in \mathcal{L}$, the CNF representing Γ must simply be conjoined with the CNF representing ℓ before we start the resolution procedure.

3.4 Containment and Equivalence over $\mathcal{Q}$

Theorem 6 *(Decidability of $\subseteq$,= and disjointness over $\mathcal{Q}$) [17]*
if $q_1 \in \mathcal{Q}$, $q_2 \in \mathcal{Q}$ and Γ expresses the constraints over the domain, then there exists a sound and complete inference mechanisms to decide if the three predicates:
1.) $\{x_1|q_1\} \subseteq \{x_2|q_2\}$
2.) $\{x_1|q_1\} = \{x_2|q_2\}$
3.) $\{x_1|q_1\} \cap \{x_2|q_2\} = \emptyset$.
are necessarily true over the set of all database instances for which Γ holds.

3.5 Translation to standard SQL

There is a very direct correspondence between queries specified in $\mathcal{L}$ and standard SQL. Queries in $\mathcal{L}$ mirror the SQL of the form:

```
SELECT *
FROM R AS x
WHERE
 [NOT] EXISTS (sub-query) ... ;
```

where the `sub-query` is of the same form but may not use `NOT`. Naturally simple and join conditions may be added to such queries and there may be more than one sub-query. Clearly any general facility that translates $\mathcal{Q}$ expressions to SQL, will have to evaluate each $\mathcal{L}$ disjunction separately.

4 Related Work

This work extends [19] which assumed a *universal relation*[14] for the global schema and an extended form of relational algebra for the query language. The main improvement here is to lift these results to arbitrary relational schemas and to better specify the exact query form in standard logical notation.

An application of the work is to provide intensional descriptions of the certain, uncertain and incomplete portions of a user's query over a data integration system[17]. Typically data integration systems have significant gaps of coverage over the global (or mediated) schema they purport to cover. Given this reality, users are interested in knowing exactly which part of their query is supported by the available data sources. If data sources are described using the language Q_{pos}[5] and the user's query is specified using the language $\mathcal{L}$, then one may: 1.) retrieve *certain answers* over the data integration system; 2.) generate intensional descriptions (within the language Q) of the certain, the uncertain and the missing answers to the user's query over the available data sources. See [17] for an in depth description of this approach. Additionally an initial approach to generating the actual natural language description from expressions within $\mathcal{L}$ has been proposed[18]. The approach uses a phrasal lexicon and exploits the decidability of containment over $\mathcal{L}$.

4.1 Classical Logic

We have identified a family of specification languages for schema tuple queries. Since there is a direct translation from such queries to domain relational calculus, this work may directly access the vast body of work on decidability classes for first order formulas[4]. The language Q essentially falls within the Schönfinkel-Bernays class[3]. This is the class of function free formula with equality that contain quantifier sequences of only the $\exists^*\forall^*$ form. However, as mentioned in the introduction, the Schönfinkel-Bernays class does not remain decidable under complementation, whereas the specification language Q does.

4.2 Relational Query Equivalence and Containment

As expected, the general problem of equivalence between relational algebra[12] expressions was shown to be undecidable[2]. Other work[10] singled out conjunctive queries (the select-project-join queries of the relational algebra) as a special case where query equivalence is decidable. Subsequent work specified a *first order query hierarchy* over query languages[9]. Roughly speaking, $\mathcal{L}$ is contained within the class of conjunctive queries closed over complementation. Clearly $\mathcal{L}$ is not *relationally complete*.

Recently most work around the question of query containment has adopted Datalog notation. A *conjunctive query(CQ)* in Datalog is simply a query where each predicate in the body of a rule references an extensional database relation. To decide if $q_1 \subseteq q_2$ one first *freezes*[20] q_1 by replacing the variables of its body and head with constants. Then if q_2 includes the frozen head of q_1 in its answer set when applied over the *canonical* database consisting of just the frozen predicates of q_1, we may conclude that $q_1 \subseteq q_2$. When no predicate appears more than once in the body of the rule, a linear time algorithm exists to decide containment,

[5] Q_{pos} is a restriction of Q which disallows negation

otherwise the decision problem is NP-complete. This approach may be enriched to decide containment between conjunctive queries with inequalities[15]($CQ^{\neq}$). Containment between conjunctive queries with negation of extensional predicates within their bodies($CQ^{\neg}$) may also be decided[16]. The complexity of deciding containment over $CQ^{\neq}$ and $CQ^{\neg}$ is Π_P^2. Containment between Datalog programs (Supporting recursion, but not negation) is undecidable[21]. Containment of a Datalog program within a conjunctive query is doubly exponential[11], while the converse question is easier.

Though there has been a lot of effort to chart languages over which containment and equivalence may be decided, to the author's knowledge no prior work has invoked the 'schema tuple query' restriction. Certainly one could imagine a regime in which the schema tuple restriction would be enforced over non-recursive Datalog. In such a case Q would be contained in the class of non-recursive Datalog programs with negation in the rule bodies. The class of conjunctive queries with negation bears resemblance to the class $\mathcal{L}$, however $\mathcal{L}$ is not contained within $CQ^{\neg}$, because negation in $\mathcal{L}$ may span more than one predicate. For example a single query in $CQ^{\neg}$, could not express the second query of section 2.2.

While we have not yet made an effort to establish the complexity of query containment over Q, it is clearly NP-hard in terms of query lengths. In fact, based on the fact that general Schönfinkle-Bernays satisfiability is NEXP, it is likely that the general containment problem is quite complex. Still since the complexity is worst case and is in query length, we still anticipate that the approach will scale in real world applications.

4.3 Description Logics

The description logics[7][13] community has investigated 'query' containment under the name of *concept subsumption*. Description logics use unary and binary predicates which formalize traditional semantic network role/filler systems. As such they are interesting fragments of logic over predicates of at most two variables[6]. The ability of description logics to reason over incomplete information makes them suited to complex tasks in data management[5].

The notion of syntactic difference between the concepts $A(x)$ and $B(x)$ may be represented as $A(x) \sqcap \neg B(x)$. Thus the focus here is on the 'decidable' description logics which have the conjunction ($\sqcap$) and complement operator ($\neg$). These are the description logics of the type *ALC* and beyond[13] which have sound and complete subsumption procedures. A limitation of description logics however, is their restriction to one and two place predicates.

This may be illustrated by considering the following state of the relation `AttendsFilm`:

AttendsFilm(friend,	movieID,	theater).
Peter	0133093	SF
Peter	0234215	Royal
Anna	0234215	SF

If one were to represent this state using only binary relations, then one obvious candidate would be:

Visits(friend,	theater)
Peter	SF
Peter	Royal
Anna	SF

Sees(friend,	movieID)
Peter	0133093
Peter	0234215
Anna	0234215

Shows(friend,	movieID)
SF	0133093
SF	0234215
Royal	0234215

However if the state of AttendsFilm is represented this way, then the spurious fact that Peter has seen the movie with ID 0234215 at SF would be concluded.

One possible fix is to insert a special key attribute into the original relation and then build a unary relation over this key (AF_KEY(key)) and associate all values to values of this key in three separate relations. Here we see the altered original relation and one of the resulting binary relations.

```
AttendsFilm(key, person, movieID,  theater)      AF_Friend(key, friend)
            k1   Peter   0133093   SF                      k1   Peter
            k2   Peter   0234215   Royal                   k2   Peter
            k3   Anna    0234215   SF                      k3   Anna
```

and so forth with the binary relations AF_MovieId and AF_Theater.

This approach is problematic on several counts. First there has been 'surgery' over the original relation, which is not always an option – especially in data integration environments. Secondly, permitting the surgery were allowed, it would permit duplicate person-movieID-theater triples in the original relation. Finally such an approach would introduces a dummy variable 'key' of dubious conceptual status.

The recent introduction of *DLR* based description logics which are based on a unary concept and *n*-ary relationships[8] may hold some promise to represent arbitrary *n*-ary database relations. Still, as an example, it is difficult to envision how *DLR* would represent cyclic concepts such as "the movies whose director plays an acting role."

4.4 Future Directions

In addition to extending the expressivity of the class of schema tuple queries, we shall consider extending the expressivity of the schemas over which such queries apply. The inclusion of union types seems relatively straight forward. By 'union types' we mean a type that may have tuples from a heterogeneous set of base types where each member of a base type that satisfies the conditions of the union type are members of the union type. For example we might have $IsInvolved = IsCastMemeber \cup IsDirector$. The modeling of IS-A hierarchies through inclusion dependencies would violate the assumption that tuples are drawn from a single schema relation. Since union types may be handled, and since the decidability of containment enables subsumption hierarchies to be built, we conjecture that modeling true IS-A hierarchies using multiple relations is unnecessary for most domains. Further experiments will evaluate this claim.

The treatment of non-first normal form schemas shall also be considered. Collection or row valued attributes might pose a significant challenge, but one idea is to simply allow for schema attributes to be equal to relational variables to capture row types and to let schema attributes be equal to a set (or bag) of values to capture collection types. The effect that this would have has not been explored deeply, but such an extension is probably necessary if this work is to impact the XML/XQuery world.

Another topic of interest is to investigate the specification of rules, both categorical and default, through pairs of schema tuple queries. The rule $\alpha \rightarrow \beta$ expresses the constraint that $\alpha \wedge \neg\beta$ is unsatisfiable. It will be interesting to establish which forms the schema tuple queries α and β may take if the overall system is to remain decidable.

This work is being implemented in a system called STEP (Schema Tuple Expression Processor). STEP translates queries specified in $\mathcal{L}$ and $\mathcal{Q}$ into expression in domain calculus and then uses the theorem proving system SPASS (`http://spass.mpi-sb.mpg.de/`) to obtain satisfiability decisions. STEP has a broker subsystem that does query planning over data integration systems and STEP has a natural language generation subsystem that generates query description using a phrasal lexicon. STEP is able to uniformly perform complex subsumption tests in under one tenth of a second on a SPARC 10 workstation. STEP also routinely produces natural language descriptions of queries in under one second. Though initial results are very promising, there is still a fair amount of system development necessary before STEP can be applied robustly to real problems.

5 Conclusions

This paper has introduced the decidable sub-classes of *schema tuple queries* specified in the languages $\mathcal{L}$ and $\mathcal{Q}$. Formulas in the language $\mathcal{L}$ are signed quantifier sequences ($\exists^*$) over conjunctions of predicates over a single free tuple variable. Formulas in the language $\mathcal{Q}$ are finite disjunctions of formulas within $\mathcal{L}$. Query containment (subsumption), equivalence and disjointness may be decided for queries specified in $\mathcal{L}$ or $\mathcal{Q}$. Within the class of queries specified using $\mathcal{Q}$, one may express the difference between two queries as a third query within the class. This ability to calculate syntactic query difference enables the generation of intensional descriptions of query differences and intersections.

Though the approach is limited to only queries returning full schema tuples, such limitations may often be overcome by considering a virtual, highly decomposed version of the schema. Problems concerning aggregate operators appear to run deeper. Still, assuming that the global schema can be adequately decomposed and that support for aggregate operators can be isolated to 'last step' client processes, the schema tuple query assumption may be appropriate for many, if not most, real world schemas.

6 Bibliography

References

[1] S. Abiteboul, R. Viannu, and V. Hull. *Foundations of Database Systems 3rd edition*. Addison Wesley, 1995.

[2] A. Aho, Y. Sagiv., and J. Ullman. Equivalences of relational expressions. *SlAM Journal on Computing*, 8(2):218–246, 1979.

[3] P. Bernays and M. Schönfinkel. Zum entscheidungsproblem der mathematischen logik. *M.A.*, 99:342–372, 1928.

[4] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives of Mathematical Logic. Springer-Verlag, 1997.

[5] A. Borgida. Description logics in data management. *TKDE*, 7(5):671–682, 1995.

[6] A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence, 82:353–367*, 1996.

[7] R. Brachman and J. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[8] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. Description logic framework for information integration. In *Principles of Knowledge Representation and Reasoning*, pages 2–13, 1998.

[9] A. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer Systems and Sciences*, 25(1):99–128, 1982.

[10] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9 of the ACM Sym. on the Theory of Computing*, pages 77–90., 1977.

[11] S. Chaudhuri and M. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Sym. on Principles of Database Systems*, pages 55–66, 1992.

[12] E. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Database Systems*, pages 33–64. Prentice-Hall, 1972.

[13] F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. In G. Brewka, editor, *Studies in Logic, Language and Information*, pages 193–238, 1996.

[14] R. Fagin, A. Mendelzon, and J. Ullman. A simplified universal relation assumption and its properties. *ACM Transactions on Database Systems, 7*, 1982.

[15] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[16] A. Levy and Y. Sagiv. Queries independent of updates. In *Proc. of VLDB*, pages 171–181, 1993.

[17] M. Minock. Data integration under the schema tuple query assumption. Technical Report 03.08, The Univeristy of Umeå, Umeå, Sweden, June 2003.

[18] M. Minock. A phrasal generator for describing relational database queries. In *Proc. of the 9th EACL worshop on natural language generation*, Budapest, Hungary, April 2003.

[19] M. Minock, M. Rusinkiewicz, and B. Perry. The identification of missing information resource agents by using the query difference operator. In *COOPIS '99*. IEEE Computer Society Press, 1999.

[20] R. Ramakrishnan., Y. Sagiv, J. Ullman, and M. Vardi. Proof tree transformation theorems and their applications. In *Sym. on Principles of Database Systems*, pages 172–181, 1989.

[21] O. Shmueli. Decidability and expressiveness of logic queries. In *Sym. on Principles of Database Systems*, pages 237–249, 1987.