University of Umeå

Department of Computing Science
SE-901 87 Umeå, Sweden

UMINF-03.08
ISSN-0348-0542
June 2003

Data Integration under the Schema Tuple Query Assumption

by

Michael J. Minock
mjm@cs.umu.se

ABSTRACT

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. This report introduces a set of assumptions which enable users to obtain intensional descriptions of the certain, uncertain and missing answers to their queries given the available data sources. The general assumption is that query and source descriptions are written as tuple relational queries which return only whole schema tuples as answers. More specifically, queries and source descriptions must be within an identified sub-class of these ‘schema tuple queries’ which is closed over syntactic query difference. Because this identified query class is decidable for satisfiability, query containment and equivalence are also decidable. Sidestepping the schema tuple query assumption, the identified query class is more expressive than conjunctive queries with negated subgoals. The ability to directly express members of the query class in standard SQL makes this work immediately applicable in a wide variety of contexts.

1 Introduction

The need to answer queries over integrated databases has been a long explored topic. Normally the architecture governing such an effort is similar to that of figure 1.

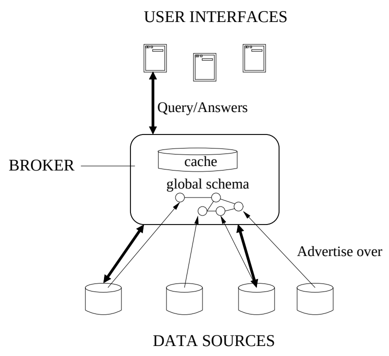

Figure 1: A typical data integration architecture

Here a *global schema* models the domain (e.g. movies, medical records, experimental findings, etc.) and autonomous *data sources* dynamically advertise their contents to a *broker* (or mediator) as views over the *global schema* (or mediated schema). A user may then query the broker using the relations of the global schema and receive answers over a large number of separate data sources. This eliminates the burden of manually visiting each data source, launching queries over local schemas and combining answers into a globally coherent answer. Furthermore, a cache within the broker limits needless network transmission of commonly accessed tuples.

We assume that data sources advertise their contents as views over the global schema (known as *local-as-view*). This is in contrast to the less extensible method of defining global schema relations as views over source database relations (known as *global-as-view*). Under local-as-view, data sources may dynamically join the data integration system without necessitating a reorganization of the global schema. See [28][17] for further discussion.

The global schemas we shall consider are relational and are constrained under sets of functional dependencies. We also assume that the data sources provide correct and *locally complete* views. Locally complete sources contain all of the information described by their views. Thus one may invoke the closed world assumption[26] over the space defined by a source's view on the global schema. Finally we assume that the local sources provide globally standard attribute value encodings through their views.

1.1 Answering user queries

Given that source descriptions are views over the global schema, the user's query may be answered by rewriting it as a query using just the 'views' defining the sources. This relates to the general problem of answering queries using views. In the context of data integration systems, the following steps are normally carried out[17]:

1.) Obtain a set of *maximally contained rewritings* of the user's query over the 'views' defining the sources.

2.) Execute query rewritings to obtain a maximal set of answers from the sources.

Informally step 1 means that we rewrite the user's query in terms of the source descriptions such that the resulting query is logically contained within the user's query. Furthermore such a contained rewriting must be maximal. That is there is no more general rewriting, with respect to a given query language, that is also logically contained by the user's query. When collectively the data sources cover the user's query, this reduces to finding an equivalent rewriting of the query using the data source descriptions. Normally, however, there will be gaps in coverage and we must obtain a set of maximally contained rewritings.

Step 2 considers how we actually obtain answers from the sources given the maximally contained rewritings. Certainly we may obtain tuples by simply executing the rewritings over the sources. However the notion of *certain* answers, introduced in [1], singles out a class of tuples that are of particular interest. Informally a tuple is a certain answer if it is an answer in all possible database instances (under the global schema) consistent with the contents of the given sources. See [1][17] for the formal definition of certain answers. As we shall see, the approach within this report eases the task of identifying certain answers considerably.

1.2 Broadening the System Response

Answers are generally presumed to be tuples. However because of gaps in coverage, users might be misled by a purely extensional response. Users might require an accompanying *intensional* response that puts the extensional answer within proper context[1]. Consider an example dialog over the data integration system pictured in figure 2.

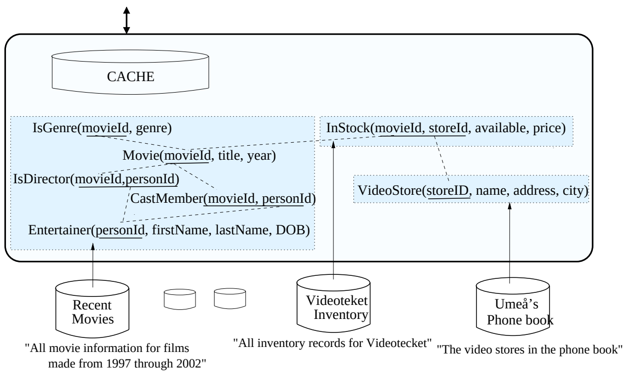
CACHE
IsGenre(movieId, genre)
Movie(movieId, title, year)
IsDirector(movieId,personId)
CastMember(movieId, personId)
Entertainer(personId, firstName, lastName, DOB)
InStock(movieId, storeId, available, price)
VideoStore(storeID, name, address, city)
Recent Movies
Videoteket Inventory
Umeå's Phone book
"All movie information for films made from 1997 through 2002"
"All inventory records for Videotecket"
"The video stores in the phone book"

Figure 2: An integrated movie system

[1]The terms extensional and intensional denote two basic ways of describing a set of objects. The extensional description of a set is simply an enumeration of the items within the set. Thus the set A may be extensionally described as $\{$'Alien', 'ET', ...$\}$. An intensional description of a set describes the set conceptually, by summarizing the properties that members have. Thus an intensional description of set A may be "Science fiction films of the 1980's".

User: give the films of the 90's that are available at video stores in Umeå.

Normally a system would be expected to present the certain set of tuples **A**. However, a broader response is more appropriate.

System: There are no sources that provide information on movies made before 1997. **A** is the set of movies made from 1997 through 1999 which are in the inventory of 'Videoteket'. **B** is the set of movies made from 1997 through 1999 that are not at 'Videotecket', but $\underline{\text{may}}$ be available at video stores in Umeå that are not providing inventory information.

In this case **A** is the *certain* answer set and **B** is termed the *uncertain* answer set. The certain answer set, in correspondence with [1], is the set of tuples that are answers to the query no matter what additional data sources are added to the system. The uncertain answer set is the set of tuples that could be rendered in or out of the answer set if more information is made available. Both **A** and **B** are extensional, consisting of sets of movie tuples. The intensional description, "no sources that provide information on movies made before 1997," is the *missing* answers description. This summarizes the set of movies for which no information whatsoever is provided. The intensional description, "the set of movies made from 1997 through 1999 that are not at 'Videotecket', but $\underline{\text{may}}$ be available at video stores in Umeå that are not providing inventory information," is the intensional description of the *uncertain answers*. Finally the description "the set of movies made from 1997 through 1999 which are in the inventory of 'Videoteket'," provides the *certain answer* set description.

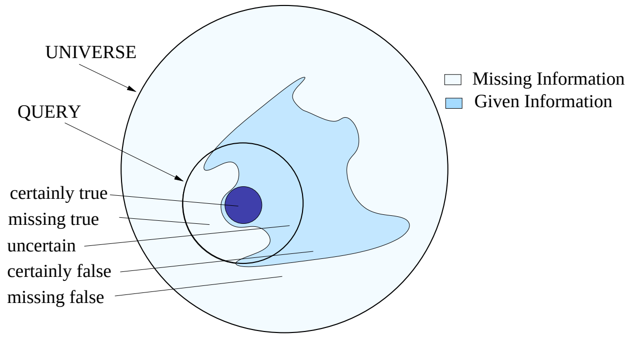

Figure 3: A query partitioning the universe of tuples

Figure 3 shows how the universe of tuples within a data integration environment is partitioned by a query. The five disjoint sets are the 'certainly true', 'uncertain', 'missing true', 'certainly false' and the 'missing false' tuples. We shall focus on generating correct and complete descriptions of the first three of these sets and we shall refer to these three sets by the shorter names of the 'certain', 'uncertain' and 'missing' answers.

Thus, in summary, we propose a third step to the process of answering queries over data integration systems.

3.) Present an intensional descriptions of the certain, uncertain and missing answers to a user's query with respect to the available data sources.

1.3 Schema Tuple Queries

To cleanly represent the partitioning of the universe of tuples of figure 3, we restrict the language in which user queries and data source descriptions may be expressed. The type of queries (and views) we consider are the *schema tuple queries*[2]. Such queries are tuple relational queries[10] 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 $\{(m.MovieId, m.Title, m.Year)|\, Movie(m) \wedge m.year = 1999\}$.

In this report schema tuple queries are further restricted by limiting their specification to the languages: $\mathcal{L}$, $\mathcal{L}_{pos}$, $\mathcal{Q}$ and $\mathcal{Q}_{pos}$. 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 $\mathcal{L}$. The languages $\mathcal{L}_{pos}$ and $\mathcal{Q}_{pos}$ are corresponding languages which disallow negation of existential quantifier sequences.

The restriction to return only whole tuples from relations of the schema, not arbitrary combinations of projected attributes, is in contrast to the flexibility of normal tuple relational queries and relational algebra. However schema tuple queries specified in $\mathcal{L}$ and $\mathcal{Q}$ may have their set differences calculated syntactically. This enables one to perform set theoretic operations over query answer sets without having to materialize tuples. In addition the complexity of generating understandable intensional descriptions of schema tuple queries is much less formidable than with classical tuple relational calculus and relational algebra. Finally the limited flexibility of schema tuple queries may often be overcome by considering highly decomposed schemas.

Given that we may invoke the schema tuple query assumption, the main practical result of this report is to show that if the data sources are described using the language $\mathcal{Q}_{pos}$ and the user's query is defined using the language $\mathcal{L}$, then one may: 1.) retrieve certain answers over the integrated system; 2.) generate intensional descriptions (within the language $\mathcal{Q}$) of the certain, the uncertain and the missing answers to the user's query over the available data sources.

1.4 Organization of this Report

Section 2 formally defines the language $\mathcal{L}$ and its closure over disjunction, $\mathcal{Q}$. Both $\mathcal{L}$ and $\mathcal{Q}$ are proven decidable for satisfiability; that is one may determine if there exists a database state for which the query will return a tuple. Section 3 shows that queries built using $\mathcal{Q}$ are closed over syntactic query difference and intersection. Based on the decidability of $\mathcal{Q}$, we may thus decide query containment and equivalence over $\mathcal{Q}$. Section 4 gives an algorithm for how to use $\mathcal{Q}$ within a data integration system to generate descriptions of the certain, uncertain and missing answers of the type in section 1.2. Section 5 relates this work to prior work and gives future directions. Section 6 gives conclusions.

[2]The term ‘schema tuple query’ is being newly introduced in this report. To the author's knowledge the notion is being newly introduced as well.

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 $\tau_1 = \langle \text{IsDirector} : [\text{‘367434’},\text{‘43252’}]\rangle$ and $\tau_2 = \langle \text{CastMember: } [\text{‘367434’},\text{‘43252’}]\rangle$ then $t_1 \neq t_2$. The positional access operator is extended to the schema tuples to mirror the standard tuple relational calculus. Thus $\tau_1[2] = \text{‘43252’}$. Furthermore we shall assume that tuple components may be accessed through attribute names (e.g. $\tau_1.personId$). 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.

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 the 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”

$$\begin{aligned}
&\{m_1|\ell_1\} = \{m_1|Movie(m_1)\wedge\\
&\quad(\exists y_1)(\exists y_2)(\\
&\qquad IsDirector(y_1) \wedge Entertainer(y_2)\wedge\\
&\qquad y_2.lastName = \text{‘Lucas’}\wedge\\
&\qquad m_1.movieId = y_1.movieId\wedge\\
&\qquad y_1.personId = y_2.personId)\}
\end{aligned}$$

- A query with negation: “The films made in the year 2000 that are not in stock in a video store in the city Umeå”

$$\begin{aligned}
&\{m_2|\ell_2\} = \{m_2|Movie(m_2)\wedge\\
&\quad\neg(\exists y_3)(\exists y_4)(\\
&\qquad VideoStore(y_3) \wedge InStock(y_4)\wedge\\
&\qquad y_3.city = \text{‘Umeå’}\wedge\\
&\qquad m_2.movieId = y_4.movieId\wedge\\
&\qquad y_4.storeId = y_3.storeId)\wedge\\
&\quad m_2.year = 2000\}
\end{aligned}$$

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 not in stock in <u>some</u> video store in Umeå,” is not expressible using $\mathcal{L}$.

We shall also identify the sub-language $\mathcal{L}_{pos} \subset \mathcal{L}$ where no quantifier sequences are negated (s_i is positive for i, $1 \leq i \leq k$). When $\ell \in \mathcal{L}_{pos}$ is expressed using a single quantifier sequence ($k = 1$), we say the query is in *outward* form. When the number of variables is minimized, but k is maximized, we say that the formula is in *inward* form.

We now arrive at the main result of this section.

Theorem 1 *($\mathcal{L}$ is decidable for satisfiability)*
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, $\{\tau/x\}\ell$.

Proof:
Because the test will be for satisfiability, the free variable of ℓ is considered existentially quantified: $\ell' = \exists x \ell$. The formula ℓ' may then be converted into an equivalent closed formula written in domain calculus: Range conditions are expanded into n-ary relations with introduced variables for each attribute of the corresponding relation. Tuple variable quantifications are replaced with corresponding sets of domain variable quantifications. Set conditions are re-written as disjunctions of simple conditions, and then all simple conditions are rewritten in their domain calculus equivalent form. All join conditions are also written in domain calculus using built in predicates such as $>(X,Y)$ or $=(X,Y)$. This transformation builds a logically equivalent expression to ℓ' within domain calculus.

The domain calculus expression then may then be converted into CNF in the standard way. All variables are existentially quantified in the input expression, but those that have a negation proceeding them, are converted to universal quantifiers in the standard way: $\neg(\exists y_1)...(\exists y_m)(\psi)$ is equivalent to $(\forall y_1)...(\forall y_m)\neg(\psi)$. Thus all quantifier prefixes over variables are of $\exists^*\forall^*$ form. Thus the Skolemization process will only introduce Skolem constants for variables.

The final function-free CNF expression may then be fed to a resolution theorem prover. Because the CNF has a finite Herbrand Universe, resolution will saturate after a finite number a steps. The CNF and hence ℓ' is unsatisfiable if and only if the empty clause is generated. $\Box$

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.

We also identify $\mathcal{Q}_{pos} \subset \mathcal{Q}$ where every formula in the disjunction is in $\mathcal{L}_{pos}$.

Theorem 2 *($\mathcal{Q}$ is decidable for satisfiability)*
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, $\{\tau/x\}q$.

Proof:
A satisfiability test may be conducted for each disjunct of $q \in \mathcal{Q}$. Based on theorem 1 each such test is decidable. Since the number of disjuncts is finite, this decision process will terminate with a correct answer. $\Box$

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

We now cover several important properties that hold for queries built over the languages $\mathcal{L}$ 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 $\mathcal{Q}$)*
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 $\{x_1|l_1\} - \{x_2|l_2\} = \{x_1|q\}$

Proof:
Assuming that ℓ_1 and ℓ_2 do not share any variable names in common,

$\{x_1|\ell_1\} - \{x_2|\ell_2\} = \{x_1|\ell_1 \wedge \{x_1/x_2\}\neg\ell_2\} =$
$\{x_1|\ell_1 \wedge \{x_1/x_2\}\neg(P(x_2) \wedge (\bigwedge_{i=1}^{k} \Phi_i) =$
$\{x_1|(\ell_1 \wedge \neg P(x_1))\vee$
$\quad(\ell_1 \wedge \neg\Phi_1)$
$\quad\vee\ldots.\vee$
$\quad(\ell_1 \wedge \neg\Phi_k)\} =$
$\{x_1|\ell'_0 \vee \ldots \vee \ell'_k\} =$
$\{x_1|q\}$ where $q = \ell'_0 \vee \ldots \vee \ell'_{k_2}$.$\square$

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

$\{m_1|\ell_1\} - \{m_2|\ell_2\} = \{m_1|\ell_1 \wedge \{m_1/m_2\}\neg\ell_2\} =$
$\{m_1|$
$\quad(Movie(m_1)\wedge$
$\quad\quad(\exists y_1)(\exists y_2)($
$\quad\quad\quad IsDirector(y_1) \wedge Entertainer(y_2)\wedge$
$\quad\quad\quad y_2.lastName = \text{‘Lucas’}\wedge$
$\quad\quad\quad m_1.movieId = y_1.movieId\wedge$
$\quad\quad\quad y_1.personId = y_2.personId)\wedge$
$\quad\quad(\exists y_3)(\exists y_4)$
$\quad\quad\quad(VideoStore(y_3) \wedge InStock(y_4)\wedge$
$\quad\quad\quad y_3.city = \text{‘Umeå’}\wedge$
$\quad\quad\quad m_1.movieId = y_4.movieId\wedge$
$\quad\quad\quad y_4.storeId = y_3.storeId))$
$\quad\vee$
$\quad(Movie(m_1)\wedge$
$\quad\quad(\exists y_1)(\exists y_2)($

$IsDirector(y_1) \wedge Entertainer(y_2) \wedge$
$y_2.lastName = \text{‘Lucas’} \wedge$
$m_1.movieId = y_1.movieId \wedge$
$y_1.personId = y_2.personId) \wedge$
$m_1.year \neq 2000)\}$

3.2 Closure of Intersection over $\mathcal{L}$

The following, although obvious, is of use later in this report.

Lemma 1 *(The closure of $\mathcal{L}$ over intersection)*
Let $\ell_1 \in \mathcal{L}$ and $\ell_2 \in \mathcal{L}$. Then there is an $\ell_3 \in \mathcal{L}$ with the property that for all database instances $\{x_1|\ell_1\} \cap \{x_2|\ell_2\} = \{x_1|\ell_3\}$

Proof:
By definition. □

3.3 Closure of Difference and Intersection over $\mathcal{Q}$

We now show that $\mathcal{Q}$ stays closed over syntactic difference and intersection. This means that we may describe the set difference (or intersection) of two queries built over $\mathcal{Q}$ with a third query built over $\mathcal{Q}$.

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

Proof:
By induction on the number of disjuncts in q_2.
Base case: Assume that q_1 consists of a finite number of disjuncts and q_2 consists of $n = 1$ disjunct. The result of set difference will simply be each disjunct of q_1 minus the single disjunct of q_2. Because $q_2 \in \mathcal{L}$, this may be carried out for each disjunct of q_1 as in theorem 3. The resulting formula, when all generated disjuncts are grouped together, is within $\mathcal{Q}$. Therefore the theorem holds in the base case.
Induction hypothesis: Assume the theorem holds when q_1 consists of a finite number of disjuncts and the number of disjuncts in q_2 is $n = k$.
Induction step: Consider q_2', written as $q_2 \vee \ell$ where $\ell \in \mathcal{L}$ and q_2 consists of k disjuncts. $\{x_1|q_1\} - \{x_2|q_2'\} = \{x_1|q_1\} - (\{x_2|q_2\} \cup \{x_2|\ell\}) = (\{x_1|q_1\} - \{x_2|q_2\}) - \{x_2|\ell\}$. From the induction hypothesis $(\{x_1|q_1\} - \{x_2|q_2\} = \{x_1|q_3\}$ where $q_3 \in \mathcal{Q}$. From the base case $\{x_1|q_3\} - \{x|\ell\} = \{x_1|q_3'\}$ where $q_3' \in \mathcal{Q}$. Therefore $\{x_1|q_1\} - \{x_2|q_2'\} = \{x_1|q_3'\}$ where $q_3' \in \mathcal{Q}$ for q_2' of $k+1$ disjuncts.

Thus the theorem has been proven by induction. □

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

Proof:
Since $q_1 \wedge q_2 \equiv q_1 \wedge \neg(q_1 \wedge \neg q_2)$, the truth of theorem 4 establishes the truth of this theorem. □

3.4 Expressing Integrity 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[2]. 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:
$(\forall x)(\forall y)(P(x) \wedge P(y) \wedge$
$\quad y.w_1 = x.w_1 \wedge ... \wedge y.w_m = x.w_m \Rightarrow$
$\quad\quad y.v_1 = x.v_1 \wedge ... \wedge y.v_n = x.v_n)$

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

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

Note also that other integrity constraints may be encoded as universally quantified formula. For example the constraint that all film years are greater than 1927 but less than or equal to 2002 could be encoded:

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

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 thus may be converted to CNF without Skolemization. When determining the satisfiability of $\ell \in \mathcal{L}$, the CNF representing Γ may simply be conjoined with the CNF representing ℓ before we start the resolution procedure.

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

Theorem 6 *(Decidability of $\subseteq$,= and disjointness over $\mathcal{Q}$)*
if $q_1 \in \mathcal{Q}$, $q_2 \in \mathcal{Q}$ and Γ expresses the integrity 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.*

Proof:
Predicate 1: $\{x_1|q_1\} \subseteq \{x_2|q_2\}$ under the constraints Γ iff $\{x_1|q_2\} - \{x_2|q_2\} = \emptyset$ under the constraints Γ. Based on the closure of $\mathcal{Q}$ over difference this reduces to a satisfiability test enriched with Γ over a query built over $\mathcal{Q}$. Because the CNF representing $\mathcal{Q}$ and Γ has a finite Herbrand universe, we may decide predicate 1. Predicate 2 follows directly from the ability to decide predicate 1. Predicate 3 is decided by deciding the satisfiability of $\{x_1|q_2\} \cap \{x_2|q_2\}$. Again this is decidable because the CNF representing $\mathcal{Q}$ and Γ has a finite Herbrand universe. □

4 Query Differencing in Data Integration

Let us return to the problem of answering queries over data integration systems. We shall assume that the user's query is $\{x|\ell\}$ where $\ell \in \mathcal{L}$ over the free variable x. Furthermore we shall assume that there exist n source databases where the i-th source database is represented by the 'view expression' $\{x_{s_i}|desc(s_i)\}$ where $desc(s_i) \in \mathcal{Q}$. This 'view expression' describes the *complete* information that the source s_i has over the global schema. Because we assume that the information in the source is complete, we may invoke the closed world assumption over the space of tuples satisfying $desc(s_i)$. In addition the broker holds a universally quantified formula Γ which states the functional dependency and other constraints of the domain. Finally we assume that the broker maintains a cache, which may be described based on the history of queries issued to sources.

In the example of figure 2 the three data sources have advertised the following 'views' over the global schema:

s_1: **Recent Movies**:

$$\begin{aligned}
&\{m|Movie(m)\wedge\\
&\quad m.year \le 2002 \wedge m.year \ge 1997\}\cup\\
&\{g|IsGenre(g) \wedge (\exists m')(Movie(m')\wedge\\
&\quad m'.year \le 2002 \wedge m'.year \ge 1997\wedge\\
&\quad g.movieId = m'.movieId)\}\cup\\
&\{d|IsDirector(d) \wedge (\exists m')(Movie(m')\wedge\\
&\quad m'.year \le 2002 \wedge m'.year \ge 1997\wedge\\
&\quad d.movieId = m'.movieId)\}\cup\\
&\{c|CastMember(c) \wedge (\exists m')(Movie(m')\wedge\\
&\quad m'.year \le 2002 \wedge m'.year \ge 1997\wedge\\
&\quad c.movieId = m'.movieId)\}\cup\\
&\{e|Entertainer(e) \wedge (\exists d')(\exists m')
\end{aligned}$$

$$
\begin{aligned}
&\quad (IsDirector(d') \wedge Movie(m') \wedge \\
&\quad m'.year \leq 2002 \wedge m'.year \geq 1997 \wedge \\
&\quad d'.movieId = m'.movieId \wedge \\
&\quad e.personId = d'.personId))\} \cup \\
&\{e | Entertainer(e) \wedge (\exists c')(\exists m') \\
&\quad (CastMember(c') \wedge Movie(m') \wedge \\
&\quad m'.year \leq 2002 \wedge m'.year \geq 1997 \wedge \\
&\quad c'.movieId = c'.movieId \wedge \\
&\quad e.personId = c'.personId))\}
\end{aligned}
$$

s_2: **Videoteket Inventory**:

$$
\begin{aligned}
&\{s | InStock(s) \wedge (\exists v')(VideoStore(v') \wedge \\
&\quad s.StoreId = v'.StoreId \wedge \\
&\quad v'.Name =' Videotecket' \wedge v'.city = \text{‘Umeå’}\}
\end{aligned}
$$

s_3: **Umeå’s Phone book**: The video store listings

$$\{v | VideoStore(v) \wedge v.city = \text{‘Umeå’}\}$$

In the ideal case a data source may answer any query about a tuple in its possession. Thus, for example, s_1 would be able to answer queries such as, “give the cast members of horror films that are in the inventory of Videoteket.” Note that a data source such as s_1 would not normally be able to answer such a question. Although it possesses all the correct answers, it is unable to determine which cast members played in horror films which are in the inventory of some video store in northern Sweden. Though the idealized case is rather unrealistic, it provides a good starting point for discussion.

The pseudo code below gives the certain answers to the user’s query under the ideal assumption. The method $ask(source_i, query)$ returns the answers to the *query* posed over the $source_i$ of n sources. Because data sources are assumed to be able to evaluate any condition over a tuple in their possession, such answers will be certain. The operation $A \leftarrow B$ inserts the tuples of set B into the set A. All uses of $-$ and $\cap$ are syntactic operations over query expressions.

```
$Process(query)$
begin
  $q = query;$
  $ANSWERS = \emptyset;$
  for $i = 1$ to $n$ do
      $ANSWERS \leftarrow ask(s_i, q \wedge desc(s_i));$
      $q = q \wedge \neg(q \wedge desc(s_i));$
  od
  $certain = query \wedge \neg q;$
  $missing = q;$
end
```

ANSWERS thus contains the tuples that satisfy the query over the space described by *certain*. Missing tuples are described with the expression *missing*.

Now we shall describe querying over three successively more complex scenarios where data sources do not have perfect knowledge about the tuples they possess. First we consider querying sources described with formula in $\mathcal{L}$. Second we consider the case of querying collections of $\mathcal{L}$ sources. Unfortunately the results break down for data sources using the full $\mathcal{L}$ language and we must limit the source descriptions to $\mathcal{L}_{pos}$. Finally we shall consider the full case of querying collections of Q_{pos} sources.

4.1 Querying $\mathcal{L}$ Data Sources

We now consider how to obtain answers to $\{x|\ell\}$ over a data source s described by $\{x_s|desc(s)\}$ where $desc(s) \in \mathcal{L}$. Just as in the idealized case, the query is intersected with the data source description and the resulting query is sent to the data source. However, unlike the ideal case, a 'real' data source is not always able to evaluate the conditions of this intersected query. For example assume we issue the following query over the following source:

ℓ_1: **Give year 2000 horror movies**:

$$\begin{aligned}&\{x|Movie(x) \land x.year = 2000\land\\&\quad(\exists y_1)(Genre(y_1) \land y_1.type = \text{‘horror’}\land\\&\qquad y_1.movieID = x.movieId)\}\end{aligned}$$

s_1': **Just movie information from** s_1:

$$\begin{aligned}&\{m|Movie(m)\land\\&\quad m.year \geq 1997 \land m.year \leq 2002\}\end{aligned}$$

Even though the source contains all of the tuples satisfying the query, it is not able to distinguish which of them are horror films. Thus the best the source may do is to drop the conditions it may not evaluate and return uncertain answers. The process is termed *query truncation* and it is carried out by simply *dropping* all non-free variables of the query formula ℓ from the formula $\ell \land desc(s)$. Dropping a variable simply means that the variable and any corresponding range conditions, simple conditions, set conditions and join conditions within the query are removed. The result of this process is signified $\{x|drop_\ell(\ell \land desc(s))\}$. For the example above the query evaluated over s_1' is:

$$\begin{aligned}&\{m|drop_{\ell_1}(\{m/x\}\ell_1 \land desc(s_1'))\} =\\&\quad\{m|Movie(m) \land m.year = 2000\}.\end{aligned}$$

All of the conditions in $\{x_s|drop_\ell(\{x_s/x\}\ell \land desc(s)\}$ may be evaluated by the source s. Moreover, if $drop_\ell(\{x_s/x\}\ell \land desc(s)) \equiv \{x_s/x\}\ell \land desc(s)$ then we know that truncation did not generalize the query sent to the source and thus the answers over the source will be certain with respect to the original query. Alternatively if $\{x_s/x\}drop(\ell) \land desc(s) \not\equiv \{x_s/x\}\ell \land desc(s)$ then the query was generalized and thus the answers to the query over the source will be uncertain. In either case $\{x_s|drop_\ell(\{x_s/x\}\ell \land desc(s))\}$ is posed over the source to obtain answers.

As a more complex example consider the following query posed over the following source:

ℓ_2: **Horror or action films not available in Umeå**:

$$\{x|Movie(x)\wedge$$
$$(\exists g)(Genre(g)\wedge g.genre\in\{\text{‘horror’},\text{‘action’}\}\wedge$$
$$g.movieId = m.movieId)\wedge$$
$$\neg(\exists s)(\exists v)(InStock(s)\wedge VideoStore(v)\wedge$$
$$s.movieId = m.movieId\wedge$$
$$s.storeId = v.storeId\wedge v.city = \text{‘Umeå’})\}$$

s_4: **Horror films not shown anywhere**

$$\{m|desc(s_4)\}\equiv\{m|Movie(m)\wedge$$
$$(\exists g)(Genre(g)\wedge g.genre = \text{‘horror’}\wedge$$
$$g.movieId = m.movieId)\wedge$$
$$\neg(\exists s)(InStock(s)\wedge s.movieId = m.movieId)\}$$

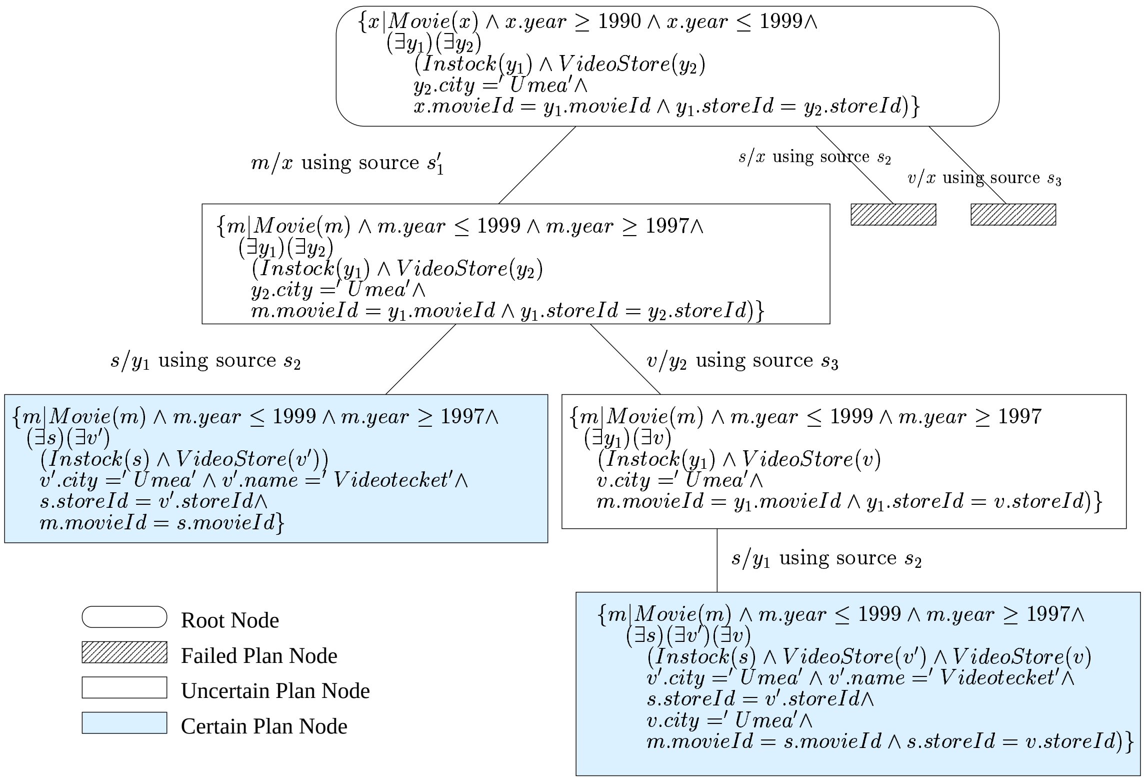
$\{x|Movie(x)\wedge x.year\geq 1990\wedge x.year\leq 1999\wedge (\exists y_1)(\exists y_2)(Instock(y_1)\wedge VideoStore(y_2)\ y_2.city='Umea'\wedge x.movieId = y_1.movieId\wedge y_1.storeId = y_2.storeId)\}$
m/x using source s'_1
s/x using source s_2
v/x using source s_3
$\{m|Movie(m)\wedge m.year\leq 1999\wedge m.year\geq 1997\wedge (\exists y_1)(\exists y_2)(Instock(y_1)\wedge VideoStore(y_2)\ y_2.city='Umea'\wedge m.movieId = y_1.movieId\wedge y_1.storeId = y_2.storeId)\}$
s/y_1 using source s_2
v/y_2 using source s_3
$\{m|Movie(m)\wedge m.year\leq 1999\wedge m.year\geq 1997\wedge (\exists s)(\exists v')(Instock(s)\wedge VideoStore(v'))\ v'.city='Umea'\wedge v'.name='Videotecket'\wedge s.storeId = v'.storeId\wedge m.movieId = s.movieId\}$
$\{m|Movie(m)\wedge m.year\leq 1999\wedge m.year\geq 1997\ (\exists y_1)(\exists v)(Instock(y_1)\wedge VideoStore(v)\ v.city='Umea'\wedge m.movieId = y_1.movieId\wedge y_1.storeId = v.storeId)\}$
s/y_1 using source s_2
$\{m|Movie(m)\wedge m.year\leq 1999\wedge m.year\geq 1997\wedge (\exists s)(\exists v')(\exists v)(Instock(s)\wedge VideoStore(v')\wedge VideoStore(v)\ v'.city='Umea'\wedge v'.name='Videotecket'\wedge s.storeId = v'.storeId\wedge v.city='Umea'\wedge m.movieId = s.movieId\wedge s.storeId = v.storeId)\}$
Root Node
Failed Plan Node
Uncertain Plan Node
Certain Plan Node

Figure 4: The search tree for the example of section 1

In this case one may verify that certain answers to the query are returned.

4.2 Querying Collections of $\mathcal{L}_{pos}$ Sources

Now we shall consider posing the query over a collection of n data sources, where the i-th source is described with $\{x_i | desc(s_i)\}$ where $desc(s_i) \in \mathcal{L}_{pos}$. As opposed to the simple case above, this time the n sources may be combined in an effort to provide certain answers[3]. Each combination is considered a *plan*, and a plan is a formula within $\mathcal{L}$.

Our strategy shall be to search the space of potential plans looking for the set of the most general plans that return certain answers to the query based on the process defined in section 4.1. Although computationally intensive, the manner in which this is achieved is straightforward. A search tree is built where each node contains a plan. The original query formula ℓ is the 'plan' within the root of the search tree. Nodes are expanded, by binding a source description formula to an unbound variable within the node's plan. This process results in a finite tree of nodes, each a different plan to answer the query. This tree is traversed and the set of certain (and uncertain) answer generating plans are obtained.

These certain (and uncertain) answer generating plans are executed to retrieve needed tuples from the sources. Using the retrieved tuples and tuples already within the cache, certain and uncertain answer sets are identified. These answer sets, along with the certain and uncertain answer generating plans, provide the basis for a mixed extensional/intensional response of the type described in section 1.

We shall now describe the critical parts of this process in more detail. Figure 4 shall be referred to during this discussion. Figure 4 shows the search tree built for the query of section 1 over the sources s'_1, s_2 and s_3. s'_1 is defined in section 4.1.

4.2.1 The root node and its initial expansion

The root note contains the user query ℓ as its plan with all of the variables within ℓ marked *unbound*[4]. The root node is expanded to n depth level one nodes, each containing the plan $\{x_s/x\}\ell \wedge desc(s_i)$. By lemma 1, these plans are in $\mathcal{L}$. In figure 4 we can see that only the plan over s'_1 is satisfiable.

4.2.2 Expansion of non-root nodes

Now we show how search tree nodes of depth level one and beyond are expanded. A single unbound variable y within the plan of the node is matched with a single source s_i. This match is the basis of the expansion of the node. Assume that $desc(s_i$ is written in outward form and ℓ' is the plan within the node to be expanded. The new node will have the plan $\ell'' \in \mathcal{L}$ where ℓ'' is ℓ' with the variable y replaced by the free variable of $desc(s_i)$, all of the existential variables of $desc(s_i)$ placed within the quantifier sequence under where y had appeared, and the single conjunct of $desc(s_i)$ conjoined with the conjunct over which y had ranged.

[3]Note that n may indeed be 1 and the case will be different than that in section 4.1. Specifically section 4.1 did not propose a strategy for how a source answers self joining queries.

[4]Figure 4 uses the variable names x and y_i to denote unbounded variables.

Since all combinations are considered, the branching factor of the tree is on the order of $O(nm)$ where n is the number of sources and m is the number of variables in ℓ. Figure 4 shows only the matches that make type sense.

4.2.3 Termination tests

There are three conditions that signal a search tree node does not need to be expanded:

1.) The node's plan is unsatisfiable.

2.) The node's plan, with unbound variables dropped, could be used to give certain answers to the query according to the procedure of section 4.1.

3.) There are no remaining unbound variables within the node's plan.

If a node meets the first condition, it is marked *failed*. If a node meets the second condition, the unbound variables are dropped from its plan and it is marked *certain*. The third condition guarantees termination by bounding the size of the search tree to $O((nm)^m)$.

4.2.4 Obtaining certain, uncertain and missing answer descriptions

At the end of this process the tree is traversed to obtain descriptions of the certain, uncertain and missing answers. The certain answers are the disjunction of the plans obtained from the nodes that are marked certain. Thus $certain \in \mathcal{Q}$. The uncertain answers are the disjunction of the non-failed plans at level one minus the description of the certain answers. Thus $uncertain \in \mathcal{Q}$. Finally the missing answers is the original query minus both $certain$ and $uncertain$. Thus $missing \in \mathcal{Q}$. From figure 4 one may verify that this results in the queries described in the example dialog of section 1.

4.2.5 Issuing queries to the sources

Once the certain and uncertain answer set descriptions are obtained, the records of source/variable bindings are used to materialize the needed tuples from the sources. This is achieved by building a query for each variable/source binding where the variable is free and the conditions are the simple and type conditions over the variable within the plan. Such queries are prepared for dispatch to their corresponding sources.

In figure 4: $\{m|Movie(m) \wedge m.year \leq 1999 \wedge m.year \geq 1999\}$ would be prepared for s'_1, $\{s|InStock(s)\}$ would be prepared for s_2 and $\{v|VideoStore(v)\}$ would be prepared for s_3. The reason that these queries are prepared, but not immediately sent, is because their answers, in part or whole, may already reside within the cache.

4.2.6 Using and maintaining the cache

A cache over all the relations of the global schema is maintained within the broker. The description of what is within the cache is $cache \in \mathcal{Q}_{pos}$. Before the prepared queries of section 4.2.5 are issued to sources, the information within *cache* is syntactically subtracted from them. The remaining portions of the prepared queries are then sent to the sources and, upon return of answers, the cache description is augmented to reflect the additional tuples.

Once all of the source/variable bindings have been used to materialize relevant tuples into the cache, the certain answer set description is applied to obtain the extension of the certain answers.

The uncertain answer set description has all of its non free variables dropped and is likewise applied to the cache. This materializes all of the uncertain and certain tuples. From this the certain answers are set subtracted to yield an approximation of the uncertain answer extension. Note that this is the only place in the whole system where a classical set difference operation is applied over an actual extension. To obtain the exact extension of uncertain answers, the certain answers from the complement of the original query must be solved for and in turn subtracted from the approximation of the uncertain answer extension.

4.3 Querying Collections of Q_{pos} Sources

At a logical level, extending the system to the full example where we have a set of sources each described by an expression in Q_{pos}, does not significantly alter the strategy of section 4.2. The Q_{pos} expressions are merely pooled into a very large Q_{pos} expression that fits the above case by treating each portion of a source as an 'independent source'.

For performance however, one would like to insure that joins that could be performed at the sources are performed there and that only the relevant portions of the resulting answers are sent to the broker. In addition we might wish to enable a principled way to include semi-join optimizations into the framework. No doubt a more sophisticated cache use and maintenance strategy will also be required. These considerations, however, are beyond the scope of this current work and shall be the subject of future work.

5 Discussion

5.1 Related Work

This work extends [23] which assumed a Universal Relation[15] 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.

We have identified a family of specification language for schema tuple queries. Since there is a direct translation from schema tuple queries to the domain relational calculus, this work may directly access the vast body of work on decidability classes for first order formulas[13]. The language $\mathcal{Q}$ essentially falls within the Schönfinkel-Bernays class[4]. This is is the class of function free formula that contain quantifier sequences of only the $\exists^*\forall^*$ form. The extension of $\mathcal{Q}$ toward more expressive schema tuple query languages, will no doubt borrow heavily from [13] and its sister collection [21] which chronicles interesting undecidable classes of first order formula.

As expected, the general problem of equivalence between relational algebra[10] expressions was shown to be undecidable[3]. Other work[8] 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[7]. Roughly speaking, $\mathcal{L}$ is contained within the class of conjunctive queries closed over complementation and $\mathcal{L}_{pos}$ is contained within the class of conjunctive queries closed over composition. Clearly neither $\mathcal{L}$ nor $\mathcal{L}_{pos}$ are *relationally complete*.

Recently most work around the question of query containment has adopted Datalog notion to represent queries. 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*[25] 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, otherwise the decision problem is *NP*-complete. This approach may be enriched to decide containment between conjunctive queries with inequalities[18]($CQ^{\neq}$). Containment between conjunctive queries with negation of extensional predicates within their bodies($CQ^{\neg}$) may also be decided[20]. The complexity of deciding containment over $CQ^{\neq}$ and $CQ^{\neg}$ is Π_P^2. Containment between Datalog programs (Support recursion, but not negation) is undecidable[27]. Containment of a Datalog program within a conjunctive query is doubly exponential[9], 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 our knowledge no prior work has invoked the ‘schema tuple query’ assumption. Certainly one could imagine a regime in which the schema tuple assumption would be enforced over non-recursive Datalog. In such a case $\mathcal{Q}$ 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 made no effort to gauge the complexity of query containment over $\mathcal{Q}$, 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 anticipate that the approach will scale in real world environments.

It should be noted that the description logics[5][12] 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. 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[12] which have sound and complete subsumtion procedures. A serious limitation of description logics however, is their restriction to one and two place predicates. This has limited their impact in database environments. The recent introduction of *DLR* based description logics which are based on a unary concept and *n*-ary relationships[6] may hold some promise. Still, as an example, it is difficult to envision current description logics representing concepts such as “the movies whose director plays an acting role.”

Most work that considers the task of obtaining maximally contained rewritings assumes conjunctive queries for both the user query as well as the source descriptions. The problem of finding maximally contained rewritings under such assumptions is NP-complete[11]. However since it is often the case that the the number of conditions within a query is bounded, rewriting algorithms are still able to scale up to large numbers of sources. The bucket algorithm[19], the inverse-rules approach[14] and recently the mini-con algorithm[24] are all efforts toward scalable rewriting algorithms for conjunctive queries over conjunctive views. The work here considers user queries including general negation over sources described with conjunctive views.

The problem of identifying certain answers over data integration systems has some rather negative complexity results[1]. Unfortunately the complexity measures are in data complexity (e.g. number of provided tuples), rather than query complexity. The only class that gives polynomial complexity is that of conjunctive views under open world assumption of the database. Even inequalities in the conjunctive queries pushes the system into intractability.

The status of the schema tuple query assumption with regard to these issues is still not settled. It does stand to reason, however, that complexity may be in terms of query size rather than data size. We have shown a method within section 4 to obtain a description of the certain answers to a query over a data integration environment. This yields a method by which to retrieve the full set of certain answers when the sources operate under a closed world assumption.

5.2 Future Directions

5.2.1 More expressive queries and schemas

Our contained rewriting algorithm of section 4.2 is unable to handle sources described using the full language $\mathcal{L}$ because substitution of source descriptions using negation results in plans outside of $\mathcal{L}$. We seek decidable schema tuple query languages that satisfy the same closure properties as $\mathcal{Q}$ which stay closed over such rewritings.

Another issue we shall consider, is extending the types of schemas that we may cover. The inclusion of union types seems relatively straight forward. For example, if we extend the example of figure 2 to include private video collections, we could introduce the union type of *HasVideo* of which the relations *InStock* and *InPersonalCollection* would be members. Such a union type would enable queries to be posed over the more general category using common attributes. Tuples, however, would still be members of one, and only one, member predicate. Formally queries using union types may always be rewritten in a form within $\mathcal{Q}$ that only references base level, non union type, schema relations.

Modeling 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 built, we conjecture that modeling true IS-A hierarchies is unnecessary for most domains. Finally it will be interesting to see how cardinality and foreign key constraints could be integrated into this work.

5.2.2 Annotating source advertisements

We intend to enable sources to claim open-world in addition to closed world coverage over their ‘views’ of the global schema. This would extend the breadth of application of our approach and would enrich intensional responses. Specifically responses would need to distinguish between the *certain and complete* and the *certain and partial* answers to the user’s query. Certainly the extension to open world data sources will have a significant complicating impact on the process of identifying certain answers.

We also acknowledge that the assumption that data sources provide only correct information is too optimistic. We shall explore how to discover and summarize differences of opinion between the data sources relative to a specific user query.

5.2.3 Simplification and natural language generation

Intensional descriptions within $\mathcal{Q}$ must be simplified before they are presented. Such simplification may be envisioned as an informed search process where utility is inversely correlated with expression complexity and operators split or fuse the disjuncts of the expression.

Assuming that the complexity measure is simply the number of disjuncts within an expression, the fact that there are a finite number of constants within the input expression, means that there are a finite number of reasonable splitting and fusion operations. Given these considerations, it seems likely that modern search techniques may be brought to bear on the problem of minimizing the number of disjunction within an expression in $\mathcal{Q}$. Given this, we shall focus on generating descriptions of each disjunct in the simplified query. Thus we focus on the problem of generating descriptions of expressions in $\mathcal{L}$.

An initial approach to generating natural language description from expressions within $\mathcal{L}$ has been proposed in [22]. The approach uses a phrasal lexicon and exploits the decidability of containment over $\mathcal{L}$.

6 Conclusions

This report has introduced the sub-classes of*schema tuple queries* specified in the languages $\mathcal{L}$, $\mathcal{L}_{pos}$, $\mathcal{Q}$ and $\mathcal{Q}_{pos}$. 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 $\mathcal{L}$. The languages $\mathcal{L}_{pos}$ and $\mathcal{Q}_{pos}$ are corresponding languages which disallow negation of existential quantifier sequences.

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. Thus the schema tuple query assumption may be appropriate for many, if not most, real world schemas.

Sidestepping the schema tuple query restriction, the queries specified using $\mathcal{L}$ are more expressive than conjunctive queries with negated subgoals, but are less expressive than non-recursive Datalog with negated subgoals. However, the query classes here are more naturally expressed in tuple relational calculus than in Datalog. Resting the form of these query classes within tuple calculus also enables us to directly access a large body of classical work on decidable classes of logical formulas.

There is a very direct correspondence between queries specified in $\mathcal{L}$ (and $\mathcal{Q}$) 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. Queries specified $\mathcal{Q}$ correspond to the `UNION` of such SQL expressions.

The main practical result of this report is to show that if data sources are described using the language $\mathcal{Q}_{pos}$ and the user's query is specified using the language $\mathcal{L}$, then one may: 1.) retrieve *certain answers* over the integrated system; 2.) generate intensional descriptions (within the language $\mathcal{Q}$) of the certain, the uncertain and the missing answers to the user's query over the available data sources.

It seems reasonable to suppose that other problems will have interesting simplifications when restricted to the schema tuple queries specified using $\mathcal{L}$ and $\mathcal{Q}$. Of particular interest are problems in updates over views, semantic query optimization and cooperative query answering[16].

7 Acknowledgments

I thank Steve Hegner for his valuable comments on an earlier draft of this report. I thank also the Department of Computing Science, at the University of Umeå for providing a wonderful environment in which to conduct this work.

8 Bibliography

References

[1] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *Sym. on Principles of Database Systems*, pages 254–263, 1998.

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

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

[11] J. Van den Busshe. Two remarks on the complexity of answering queries using views. In *Information Processing Letters*, 2000.

[12] 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.

[13] B. Dreben and W.D. Goldfarb. *The decision problem. Solvable classes of quantificational formulas*. Addison Wesley, 1979.

[14] O. Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, 2000.

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

[16] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *Intelligent Information Systems*, 1(2):127–157, 1992.

[17] A. Halevy. Answering queries using views: A survey. *VLDB Journal 10(4): 270-294*, 2001.

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

[19] A. Levy, A. Rajaraman, and J. Ordille. Query-answering algorithms for information agents. In *Proc. of the Thirteenth National Conference on Artificial Intelligence*, pages 40–47, 1996.

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

[21] H. Lewis. *Unsolvable Classes of Quantificational Formulas*. Addison Wesley, 1979.

[22] 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.

[23] 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.

[24] R. Pottinger and A. Halevy. A scalable algorithm for answering queries using views. In *The VLDB Journal*, pages 484–495, 2000.

[25] 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.

[26] R. Reiter. *Logic and Databases*, chapter On Closed World Databases. Plenum Press, 1978.

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

[28] J. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.