Managing Database Incompleteness with the Guarded Fragment

Michael J. Minock
Department of Computing Science
The University of Umeå, Sweden 90187
Phone: +46 90 786 6398 FAX: +46 90 786 6126
Email: *mjm@cs.umu.se*

Abstract

Traditional databases do not explicitly represent the portions over their schemas for which they are sound and complete. This paper proposes a method by which very fine grained meta-data descriptions may be attached to relational data sources to describe, among other things, the portions of various tables for which the database has complete data. Because of the formal properties of the meta-data description language, itself expressible within the guarded fragment of first-order logic, intensional descriptions of the coverage offered to a specific query may be generated. The techniques discussed in this paper are general and may apply to many scenarios where fine-grained meta-data descriptions are associated with tuple sets.

1 Introduction

Often one is prepared to make very specific declarations of soundness and completeness in a database. For example I am willing to say that my database has correct records for all of the commercial airports in Sweden. I may continue and declare that my database has all the schedule information for all of the flights to the Swedish airport with code 'UME' on a specific date. In any case these declarations of local soundness and completeness are fine-grained views[1] and are asserted as a dynamic part of populating or modifying the database.

The problem is that these declarations should be merged and managed so that when the database is queried, a coherent description of the database's completeness (or incompleteness) may be provided with respect to the specific query. Thus when the user asks for all the airport tuples in Sweden, Norway or Denmark they should be informed that their answer is complete for Sweden and not so for the remaining countries.

[1]Hereafter let us assume that such views are sound. The notion of complete in this case corresponds to the closed world assumption being applicable over the view.

More abstractly, consider that we have a set of n views: $\mathtt{v}_1, ..., \mathtt{v}_n$ that describe the portion of the schema for which the database has sound and complete coverage. The problem is, given a query $\mathtt{q}$, to calculate the portion of $\mathtt{q}$ that may be answered with *certainty*[2] using the views. The answer to this problem is a disjoint partition of the original query $\mathtt{q}$ into two queries $\mathtt{q}_{\mathtt{certain}}$ and $\mathtt{q}_{\mathtt{uncertain}}$ where $\mathtt{q}_{\mathtt{certain}}$ is the portion of the query that may be answered with certainty and $\mathtt{q}_{\mathtt{uncertain}}$ is the remainder that may not be answered with certainty over the views.

In relation to the general problem of answering queries using views(see [14]), the problem here is to obtain a maximally contained rewriting of a query using a set of closed-world views. Of note the language in which views (or queries) are expressed is well-beyond [6] the class of conjunctive queries [7]. The key to obtaining a solution to this problem, is the limitation of projection to whole schema tuples [16] and the restriction of tuple calculus formulas to a decidable class that may be translated to the guarded fragment of first-order logic [2].

The plan of this paper is as follows. Section 2 covers basic definitions and introduces the notion of *schema tuple queries* which give us a convenient way to describe sets of tuples within database tables. Section 3 presents the guarded schema tuple query language $\mathcal{L}_{GT}$ and its extension to full multi-typed disjunction, $\mathcal{Q}_{GT}$. Additionally section 3 shows that one can calculate syntactic difference and intersection as well as decide query containment for queries expressed within $\mathcal{Q}_{GT}$. Section 4 discusses some of the formal limitations of $\mathcal{L}_{GT}$ (and $\mathcal{Q}_{GT}$) and some weak methods that might remedy them. Section 5 presents a solution to the problem of deriving maximally contained rewritings where the query and view descriptions are expressed within the language $\mathcal{L}_{GT}$ and the certain answer description is expressed within the language $\mathcal{Q}_{GT}$. Section 6 discusses this work in relation to related work. Section 7 concludes this paper and outlines some of the prospects for future work in this area. The proofs are given in the appendix.

2 Basic Notions

Let us assume the existence of three disjoint sets: $\mathcal{U}$, the *universal domain* of *atomic values*, $\mathcal{P}$, *predicate names* and $\mathcal{A}$, *attribute names*. Without loss of generality, let us assume that $\mathcal{U}$ is a countably infinite, totally ordered set. Let U be a distinct symbol representing the type of $\mathcal{U}$. A *relation schema* R is a sequence $\langle A_1 : U, ..., A_n : U\rangle$ where $n \geq 1$ is called the *arity* of R and all $A_i \in \mathcal{A}$ are distinct attribute names. 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 a relation instance of R_i for $i \in [1..m]$. Hereafter we assume the database schema is fixed and thus drop reference to D.

[2]The notion of certain answers here is that of certain answers in [1] under closed-world views. Specifically an answer τ to query $\mathtt{q}$ is certain, if it is an answer to $\mathtt{q}$ over all databases where $\mathtt{v}_1, ..., \mathtt{v}_n$ are sound and complete views. See the formal definition of certain answers in section 5.

2.1 Tuple Relational Calculus

The tuple relational calculus is a ‘syntactic sugar’ over first order logic that lets us range variables over tuples of a relation, rather than over constants within a fixed argument position of a relation. Tuple calculus is widely used in database research and was introduced by Codd [9].

Let Z be the set of all tuple variables and for $z \in Z$ let $z.a$ be the *component reference* to attribute a of the relation over which z ranges. We say that such component references are *meaningful* if a is indeed an attribute of the relation over which z ranges. Let θ denote one of the operators ($=,>,<,\geq,\leq$ or $\neq$), let ε denote a set membership operator ($\in$ or $\notin$).

Let us inductively define the *tuple relational formulas*. The *atomic tuple formulas* are:

1. **Range Conditions:** $P(z)$, where P is a predicate name and $z \in Z$.
2. **Simple Conditions:** $X\theta Y$, where X is a component reference and $Y \in \mathcal{U}$.
3. **Join Conditions:** $X\theta Y$, where both X and Y are component references.
4. **Set Conditions:** $X\varepsilon Y$, where X is a component reference and Y is a set of constants drawn from $\mathcal{U}$. (e.g. $x.A \in \{y_1,...,y_n\}$)

All atomic formulas are tuple relational formulas and 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. The set $Free(\alpha) \subseteq Z$ denotes the free tuple variables of the formula α. We say a condition is *meaningful* if all of its component references are meaningful. We say that a formula is *safe* if all of its tuple variables range over a fixed number of relations.

2.2 Integrity Constraints

As is common, primary keys are expressed through functional dependencies and functional dependencies are expressed as universally quantified tuple formulas. Specifically the functional dependency $W \rightarrow A$ over the relation P where W is a set of m attributes and A is a single attribute of P is expressed as the universally quantified formula: $(\forall x)(\forall y)(P(x) \wedge P(y) \wedge y.w_1 = x.w_1 \wedge ... \wedge y.w_m = x.w_m \Rightarrow y.a = x.a)$. We shall use the symbol F to denote the set of all of the functional dependencies that hold over the database.

Foreign key constraints are simply a special case of inclusion dependencies. An inclusion dependency from the arbitrary sequence of attributes $a_1,...,a_m$ in relation P to the arbitrary sequence of attributes $b_1,...,b_m$ in P' is expressed as the formula: $(\forall x)(P(x) \Rightarrow (\exists y)(P'(y) \wedge x.a_1 = y.b_1 \wedge ... \wedge x.a_m = y.b_m))$. We shall use the symbol I to denote all of the inclusion dependencies that hold over the database.

Example 1 *(A relational database schema):*

City(name, country, latitude, longitude)
Airport(code, country, latitude, longitude)
Flight(fnum, carrier, $\text{from}_{Airport}$, $\text{to}_{Airport}$)
Schedule(flight_{Flight}, date, depTime, arrTime)

where the primary keys are underlined and the foreign keys are subscripted by the relation to which they refer.

2.3 Schema Tuple Queries

So much for relational orthodoxy, we now introduce the notion of typed schema tuples and schema tuple queries.

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

Example 2 *(Two example schema tuples)*
$\tau_1 =$ `<Airport: 'ULM','Germany',47,13>`
$\tau_2 =$ `<City: 'ULM','Germany',47,13>`

We say that $type(\tau) = P_i$ 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 $type(\tau_1) = type(\tau_2)$ and they match on all components. Thus $\tau_1 \neq \tau_2$ even though they have equal corresponding components.

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

When we write the expression $\{x|\alpha\}$ we normally assume that the expression α is over the free tuple variable x. For the schema tuple τ, $\tau \in \{x|\alpha\}$ if $\{\tau/x\}(\alpha)$ where $\{t/x\}\alpha$ means to substitute the term t in place of x in α. 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 schema tuple query.

3 Guarded Schema Tuple Query Languages

We shall now turn our attention to an interesting sub-language of the language of tuple relational formulas.

3.1 $\mathcal{L}_{GT}$ and its decidability

We shall inductively define the set of *guarded components*. Note that in this definition, we assume that the guarded component ψ over the variable y may be written as $s \cdot (\exists y)(P(y) \wedge \phi)$ where s is the sign (empty or $\neg$), $P(y)$ is the guard and ϕ is the *body* of the component.

Definition 3 *(Guarded components)*

1. The formula $(\exists y)(P(y) \wedge \phi)$ is a guarded component where ϕ is a conjunction of meaningful set and simple conditions limited to the operators $=$ and $\neq$ over y.

2. If ψ and ψ' are guarded components[3], then $(\exists y)(P(y) \wedge \phi \wedge s' \cdot (\exists y')(P'(y') \wedge y.a = y'.b \wedge \phi'))$ is a guarded component[4] where $y.a = y'.b$ is a meaningful equality join between the attribute a of P and the attribute b of P'.

3. If ψ and φ are guarded components, then $\psi \wedge \varphi$ and $\neg\psi$ are guarded components.

Note that the guarded component ψ may be written as $(\exists y)(P(y) \wedge \phi_0 \wedge \phi_1 \wedge \ldots \wedge \phi_n)$, where ϕ_i is the i-th conjunct of ϕ. We say that the guarded component ϕ is *normal* if it is written so that all the simple and set conditions over y are grouped in ϕ_0 and each ϕ_i for $0 < i \leq n$ involves exactly 1 normal child-component.

Definition 4 *(The language $\mathcal{L}_{GT}$)*
The language $\mathcal{L}_{GT}$ consists of all formulas of the form,

$$P(x) \wedge (\textstyle\bigwedge_{i=0}^{n} \phi_i)$$

obtained from a normal guarded component over the variable x.

Thus the language $\mathcal{L}_{GT}$ is obtained by simply removing the existential quantifier at the head of a normal guarded component. Note that queries built using $\mathcal{L}_{GT}$ are *non-cyclic*. A query is said to be *self-joining* if the queries descendent components have a variable that ranges over the same relation that the free variable x ranges over. Finally we speak of the *width* of a query as the depth of the deepest descendent of the head component of the query. Of the following queries, only q_3 is self joining and no query has a width greater than 3.

Example 3 *(Queries and views built using $\mathcal{L}_{GT}$)*

[3]Written $s \cdot (\exists y)(P(y) \wedge \phi)$ and $s' \cdot (\exists y')(P'(y') \wedge \phi')$ respectively.

[4]ψ' is said to be a *child-component* of ψ in this case. Likewise, ψ is said to be the *parent* of ψ'. This naturally leads to the corresponding notions *descendent* (*ancestor*) components of a given component.

$\mathbf{v_1}$:“The airports in Sweden”
$\{x|Airport(x) \wedge x.country = \text{“Sweden”}\}$

$\mathbf{q_1}$:“The airports in Scandinavia”
$\{x|Airport(x) \wedge x.country \in \{\text{“Sweden”}, \text{“Norway”}, \text{“Denmark”}\}\}$

$\mathbf{v_2}$:“Flights within Sweden”
$\{x|Flight(x) \wedge$
$\quad (\exists y_1)(Airport(y_1) \wedge y_1.code = x.to \wedge y_1.country = \text{“Sweden”}) \wedge$
$\quad (\exists y_2)(Airport(y_2) \wedge y_2.code = x.from \wedge y_2.country = \text{“Sweden”})\}$

$\mathbf{q_2}$:“Flights to Umeå, not from Arlanda”
$\{x|Flight(x) \wedge$
$\quad (\exists y_1)(Airport(y_1) \wedge$
$\qquad y_1.code = x.to \wedge y_1.code = \text{“UME”} \wedge y_1.country = \text{“Sweden”}) \wedge$
$\quad \neg(\exists y_2)(Airport(y_2) \wedge$
$\qquad y_2.code = x.from \wedge y_2.code = \text{“ARN”} \wedge y_2.country = \text{“Sweden”})\}$

$\mathbf{v_3}$:“Schedules for domestic flights to Umeå in first 2 days of December, 2003”
$\{x|Schedule(x) \wedge$
$\quad x.date \in \{\text{“20031201”}, \text{“20031202”}\}$
$\quad (\exists y_1)(Flight(y_1) \wedge x.flight = y_1.fnum \wedge$
$\qquad (\exists y_2)(Airport(y_2) \wedge y_1.from = y_2.code \wedge y_2.country = \text{“Sweden”}) \wedge$
$\qquad (\exists y_3)(Airport(y_3) \wedge y_1.to = y_3.code \wedge y_3.country = \text{“Sweden”} \wedge$
$\qquad\quad y_3.code = \text{“UME”}))\}$

$\mathbf{q_3}$:“Swedish airports that have flights to every other airport in Sweden”
$\{x|Airport(x) \wedge$
$\quad x.country = \text{“Sweden”} \wedge$
$\quad \neg(\exists y_1)(Airport(y_1) \wedge y_1.country = \text{“Sweden”} \wedge$
$\qquad \neg(\exists y_2)(Flight(y_2) \wedge y_2.from = x.code \wedge y_2.to = y_1.code))\}$

Note that in q_3 we assume that all airports have a flight to themselves.

Theorem 1 *($\mathcal{L}_{GT}$ is decidable for satisfiability)*
For all $\ell \in \mathcal{L}_{GT}$ over the free variable x, we may decide if there exists a database instance d, satisfying the inclusion dependencies I, for which $\{\tau/x\}\ell$ where τ is a schema tuple in d.

3.2 The Language $\mathcal{Q}_{GT}$

We define the language $\mathcal{Q}_{GT}$ as the closure of $\mathcal{L}_{GT}$ over disjunction where the free variable within each disjunct may range over a different relation.

Definition 5 *(The language $\mathcal{Q}_{GT}$)*
$q \in \mathcal{Q}_{GT}$ if q is written as a finite expression $\ell_1 \vee ... \vee \ell_k$ where each $\ell \in \mathcal{L}_{GT}$ and ℓ is over the free tuple variable x.

Theorem 2 *($\mathcal{Q}_{GT}$ is decidable for satisfiability)*
For all $q \in \mathcal{Q}_{GT}$ over the free variable x, we may determine if there exists a database instance d, satisfying the inclusion dependencies I, for which $\{\tau/x\}q$ where τ is a schema tuple in d.

3.3 Syntactic intersection and difference

We may describe the set difference (or intersection) of two queries built over $\mathcal{Q}_{GT}$ with a third query built over $\mathcal{Q}_{GT}$.

Theorem 3 *(Syntactic intersection of $\mathcal{Q}_{GT}$ queries)*
Let $q_1 \in \mathcal{Q}_{GT}$ and $q_2 \in \mathcal{Q}_{GT}$. Then there is a $q_3 \in \mathcal{Q}_{GT}$ such that for all database instances $\{x|q_1\} \cap \{x|q_2\} = \{x|q_3\}$.

Theorem 4 *(Syntactic difference of $\mathcal{Q}_{GT}$ queries)*
Let $q_1 \in \mathcal{Q}_{GT}$ and $q_2 \in \mathcal{Q}_{GT}$. Then there is a $q_3 \in \mathcal{Q}_{GT}$ such that for all database instances $\{x|q_1\} - \{x|q_2\} = \{x|q_3\}$.

3.4 Query containment

Theorem 5 *($\subseteq$,= and disjointness over $\mathcal{Q}_{GT}$)*
If $q_1 \in \mathcal{Q}_{GT}$, $q_2 \in \mathcal{Q}_{GT}$ and I are the set of inclusion dependencies, then one may decide if:

1. $\{x|q_1\} \subseteq \{x|q_2\}$
2. $\{x|q_1\} = \{x|q_2\}$
3. $\{x|q_1\} \cap \{x|q_2\} = \emptyset$.

are necessarily true over the set of all database instances for which I holds.

4 Weak Properties

There are two issues that remain formally unresolved in this work: *enforcing functional dependencies* and *simplifying queries*. In both cases we comment on the formal difficulties as well as on weaker methods to practically address these issues.

4.1 Enforcing Functional Dependencies

Unfortunately there are limitations when considering decidability of $\mathcal{L}_{GT}$ under functional dependencies. The guarded fragment is not decidable in the company of assertion statements [12]. Thus the following query is deemed to be satisfiable:

$$
\begin{aligned}
&\{x|Flight(x)\wedge\\
&\quad(\exists y_1)(Airport(y_1)\wedge x.from = y_1.code \wedge y_1.country = \text{“Sweden”})\\
&\quad(\exists y_2)(Airport(y_2)\wedge x.from = y_2.code \wedge y_2.country = \text{“Norway”})\}
\end{aligned}
$$

The approach here is to rewrite expressions in $\mathcal{L}_{GT}$ into enhanced expressions that capture at least some of the relevant functional dependencies. Thus for the functional dependency $W \to A$ over the relation P, a simple case of this is achieved by rewriting any component $(\exists y)(P(y) \wedge y.a = c)$ where $c \in \mathcal{U}$, to $(\exists y)(P(y) \wedge y.a = c \wedge \neg(\exists y')(P(y') \wedge y.w_1 = y'.w_1 \wedge \ldots \wedge y.w_m = y'.w_m \wedge y'.a \neq c))$. Clearly this does not cause any problems because the language $\mathcal{L}_{GT}$ stays closed under such a transformation.

Thus based on the functional dependency `code` $\to$ `country` holding over `Airport` we transform the query from above to:

$$
\begin{aligned}
&\{x|Flight(x)\wedge\\
&\quad(\exists y_1)(Airport(y_1)\wedge x.from = y_1.code \wedge y_1.country = \text{“Sweden”}\\
&\qquad\neg(\exists y_3)(Airport(y_3)\wedge y_1.code = y_3.code \wedge y_3.country \neq \text{“Sweden”}))\\
&\quad(\exists y_2)(Airport(y_2)\wedge x.from = y_2.code \wedge y_2.country = \text{“Norway”}\\
&\qquad\neg(\exists y_4)(Airport(y_4)\wedge y_2.code = y_4.code \wedge y_4.country \neq \text{“Norway”}))\}
\end{aligned}
$$

In this case the functional dependency is reflected and the query is correctly deemed to be unsatisfiable. This technique, by itself, still lacks strength. Consider the example query:

$$
\begin{aligned}
&\{x|Schedule(x)\wedge\\
&\quad(\exists y_1)(Flight(y_1)\wedge x.flight = y_1.fnum\wedge\\
&\qquad(\exists y_2)(Airport(y_2)\wedge y_1.from = y_2.code \wedge y_2.country = \text{“Sweden”}))\\
&\quad(\exists y_3)(Flight(y_3)\wedge x.flight = y_3.fnum\wedge\\
&\qquad(\exists y_4)(Airport(y_4)\wedge y_3.from = y_4.code \wedge y_4.country = \text{“Norway”}))
\end{aligned}
$$

In this query the above technique is insufficient because the functional dependencies over `Flight` never get a chance to impact the decision. What we need is an extended technique to push constants throughout the query based on functional dependencies. To get a clearer view of this process, consider that the child relationship between guarded components defines a path through the schema that uses either foreign key to primary key (*foreign key reference*) or primary key to foreign key (*primary key referenced by*) steps. Let us assume the existence of a function that maps a given primary key constant k to a corresponding foreign key constant within the same table. Such a function is defined for every pair of primary/foreign key attributes of a table. We denote this function on the relation R between the primary key K and the foreign key attribute A as $FK^R_{K,A}(\mathcal{U}^n) \rightarrow \mathcal{U}$ where n is the number of attributes in K.

If we Skolemize the translation of our query in FOL, then (at least) all of the domain variables corresponding to the free variable x will have Skolem constants[5]. We may then use these FK functions to push constants through foreign key references. Interestingly this technique works no matter what the sign of the quantifier and stands as an orthogonal technique to standard Skolemization. The process is guaranteed to generate a finite Herbrand universe because the queries are non-cyclic and thus the application of the FK function will be finite. If this technique is used along with the rewriting technique above then the problem query above is deemed unsatisfiable.

One can imagine an even more extended version of this approach where each descendent component of a query is individually tested as a separate satisfiability problem, complete with rewriting and FK based constant propagation. Though of practical importance, it is unlikely that such an approach will ever fully resolve the formal difficulties of enforcing functional dependencies over $\mathcal{L}_{GT}$. Further evidence for this limitation comes from the undecidability of inferring additional inclusion or functional dependencies from a given set of functional and inclusion dependencies [17][8]. Recent progress [15] has been made to tease out those cases where the interaction is limited, and it remains a goal to incorporate such results into the approach described here.

4.2 Simplification

Given an expression $q \in \mathcal{Q}_{GT}$ we say that q' is a *equivalent rewriting* of q if there is a finite sequence of semantically equivalent formulas $q_1, ...q_m$ such that q_1 is q and q_m is q' and each q_{i+1} is obtained from q_i by applying a *rewrite rule*. The rewrite rules that we use include the routine rewrite rules [19] and several special cases designed around the syntactic form of $\mathcal{Q}_{GT}$. Of interest we also include rewrite rules that test if two distinct existential variables can be merged or if one may drop variables, components or conditions wholesale. In such cases, an explicit equivalence test is issued to insure that a semantically equivalent query has been written.

In summary, we have a set of sound, but not necessarily complete set of rewrite rules for queries in $\mathcal{Q}_{GT}$. We employ these rewrites to search for *shorter* (in terms of the number of conditions) equivalent rewritings of a query. In practice we are able to simplify a wide class of queries within reasonable time.

[5]Note that the unique names assumption does not hold over such constants.

5 Reporting Completeness

We now shall solve a general case of the problem posed in the introduction. Namely for the query $\mathtt{q}$ and the n views $\mathtt{v_1}, ..., \mathtt{v_n}$ expressing sound and complete coverage, we seek to partition the query into the two disjoint pieces $\mathtt{q_{certain}}$ and $\mathtt{q_{uncertain}}$ where the portion $\mathtt{q_{certain}}$ may be evaluated over the database to retrieve all available certain answers to the query. Let us define certain answers in this context.

Definition 6 *(Certain answers)*
Given a set of sound and complete views $\mathtt{v_1}, ..., \mathtt{v_n}$, each written $\{x_i|\nu_i\}$, a tuple τ is a *certain answer* to the query $\{x|\alpha\}$, if and only if $\tau \in \{x|\alpha\}$ for all database states d consistent with the extensions of $\mathtt{v_1}, ..., \mathtt{v_n}$.

Naturally for the problem here we assume that $\alpha \in \mathcal{L}_{GT}$ and $\nu_i \in \mathcal{L}_{GT}$. The problem is thus to show a procedure to obtain a schema tuple query $\{x|\sigma\}$ that obtains only the certain answers to $\{x|\alpha\}$ given the sound and complete views.

5.1 Obtaining certain query descriptions from a single view

We observe that it is not necessarily the case that *any* tuple from a view whose description semantically intersects with the query description, will actually be a certain answer to the query. Formally, for the i-th view, if $\tau \in \{x_i|\{x_i/x\}\ell \wedge \nu_i\}$ then it is not necessarily true that τ is a certain answer to $\{x|\ell\}$.

For intuition consider that a schema tuple view written in $\mathcal{L}_{GT}$, on its own, in not able to evaluate conditions on other relations[6]. So, when considering a query posed over a single view, we just as well might *truncate* the query so that it is *definable* over the view. Formally query truncation is carried out by simply *dropping* all non-free variables of the query formula ℓ. Dropping a variable means that the variable and any corresponding components using the variable are removed. The result of this process is written $\{x|drop(\ell)\}$ and it always results in a more general query.

Theorem 6 *(Certainty test over single views)*
τ is a certain answer to the not self-joining query $\{x|\ell\}$ posed over the single view $\{x'|\nu\}$ if and only if for all database instances d, $\tau \in \{x'|\{x'/x\}drop(\ell) \wedge \nu\}$ and $\{x'|\{x'/x\}drop(\ell) \wedge \nu\} = \{x'|\{x'/x\}\ell \wedge \nu\}$.

5.2 Obtaining certain query descriptions over a set of views

In the actual algorithm we search out the most general view combinations that return certain answers to the query. We call each view combination a *plan*. Specifically a plan is a rewrite of ℓ where variables are singly matched with views from the given set of n views. The manner in which this is achieved is straightforward. A

[6]Self-joining queries are an exception to this statement.

search tree is built where each node contains a plan. The original query formula ℓ is the plan within the root node of a search tree. Nodes are expanded, by replacing an unmatched variable from the original ℓ with a view description from the set of n views. Formally, assume that the component $(\exists y)(P(y) \wedge \phi_0 \wedge \phi_1 \wedge, ..., \wedge \phi_n)$ occurs within the plan ℓ' and ν_i is $P(x_i) \wedge \phi'$. The match between y and $\mathtt{v_i}$ is reflected in the rewritten component $(\exists x_i)(P(x_i) \wedge \{x_i/y\}\phi_0 \wedge \{x_i/y\}\phi_1 \wedge, ..., \wedge \{x_i/y\}\phi_n \wedge \phi')$. We say that $\ell'' \in \mathcal{L}_{GT}$ is obtained from ℓ' where such a component has been rewritten. At the level of the search tree, we say the node with the plan ℓ'' is obtained from the node with the plan ℓ' by matching the variable y within ℓ' to the view $\mathtt{v_i}$.

Variable matching occurs top-down in the plan moving from parent to child components in ℓ. The termination test for node expansion is if the node meets one of the following three conditions:

1. The node's plan is unsatisfiable.
2. The node's plan, with unmatched variables dropped, can be used to give certain answers to the query.
3. There are no remaining unmatched variables within the node's plan.

This process results in a finite tree of $O((nm)^m)$ nodes where n is the number of views and m is the number of tuple variables in ℓ. This tree is traversed and the set of certain answer generating plans are obtained. These plans are pooled into an expression in Q_{GT} that intensionally describes the certain answers. This expression may be evaluated over the database to generate the actual certain answers, or it may be simplified and presented as a summary description[7].

Theorem 7 *(Certain answers over a set of views)*
Given a set of sound and complete views $\mathtt{v_1}, ..., \mathtt{v_n}$, each written as $\{x_i|\nu_i\}$ where $\nu_i \in \mathcal{L}_{GT}$ and the non self-joining query $\{x|\ell\}$, the certain answers to $\{x|\ell\}$ may be written as the expression $\{x|\sigma\}$ where $\sigma \in Q_{GT}$.

5.3 Examples of certainty reporting

Here we see the ideas of reporting completeness in action. These examples draw upon expressions from example 3 as well as the following additional inputs:

v$_4$:“Schedule information for all domesic Swedish flights for
November 30 and December 1, 2003”
$\{x|Schedule(x) \wedge$
$x.date \in \{$“20031130”, “20031201”$\} \wedge$

[7]It is possible that such descriptions include impossible states that violate the functional dependencies of the database. Thankfully, by definition, such descriptions are never answer yielding. Moreover if the techniques of section 4.1 are used, many empty answer reports will not occur in practice.

$(\exists y_1)(Flight(y_1) \wedge x.flight = y_1.fnum \wedge$
$\quad(\exists y_2)(Airport(y_2) \wedge y_1.from = y_2.code \wedge y_2.country = \text{“Sweden”}) \wedge$
$\quad(\exists y_3)(Airport(y_3) \wedge y_1.to = y_3.code \wedge y_3.country = \text{“Sweden”})\}$

$\mathbf{q_4}$:“Schedule information for flights arriving in Umeå or Luleå, Sweden”
$\{x|Schedule(x) \wedge$
$\quad(\exists y_1)(Flight(y_1) \wedge x.flight = y_1.fnum \wedge$
$\quad\quad(\exists y_2)(Airport(y_2) \wedge y_1.to = y_2.code \wedge y_2.code \in \{\text{“UME”}, \text{“LUL”}\} \wedge$
$\quad\quad\quad y_2.country = \text{“Sweden”}))\}$

Example 4 *(Example outputs)*
Assuming that the set of views are v_1,v_2,v_3 and v_4 from above, query $q_{1_{certain}}$ is exactly the form of v_1, for query q_2*:*

$\mathbf{q_{2_{certain}}}$:
$\{x|Flight(x) \wedge$
$\quad(\exists y_1)(Airport(y_1) \wedge$
$\quad\quad \wedge y_1.code = x.to \wedge y_1.code = \text{“UME”} \wedge y_1.country = \text{“Sweden”}) \wedge$
$\quad\neg(\exists y_2)(Airport(y_2) \wedge$
$\quad\quad y_2.code = x.from \wedge y_2.code = \text{“ARN”} \wedge y_2.country = \text{“Sweden”})$
$\quad(\exists \mathbf{y_3})(\mathbf{Airport}(\mathbf{y_3}) \wedge \mathbf{y_3.code} = \mathbf{x.from} \wedge \mathbf{y_3.country} = \textbf{“Sweden”})\}$

q_3 may be answered completely and for query q_4*:*

$\mathbf{q_{4_{certain}}}$:
$\{x|Schedule(x) \wedge$
$\quad\mathbf{x.date} \in \{\textbf{“20031130”}, \textbf{“20031201”}, \textbf{“20031202”}\} \wedge$
$\quad(\exists y_1)(Flight(y_1) \wedge x.flight = y_1.fnum \wedge$
$\quad\quad(\exists y_2)(Airport(y_2) \wedge y_1.to = y_2.code \wedge y_2.code \in \{\text{“UME”}, \text{“LUL”}\} \wedge$
$\quad\quad\quad y_2.country = \text{“Sweden”}))$
$\quad\vee$
$\quad Schedule(x) \wedge$
$\quad\mathbf{x.date} \in \{\textbf{“20031201”}, \textbf{“20031202”}\} \wedge$
$\quad(\exists y_1)(Flight(y_1) \wedge x.flight = y_1.fnum \wedge$
$\quad\quad(\exists y_2)(Airport(y_2) \wedge y_1.to = y_2.code \wedge \mathbf{y_2.code} = \textbf{“UME”} \wedge$
$\quad\quad\quad y_2.country = \text{“Sweden”}))\}$

Material in bold states what additional conditions are necessary for certainty given that the views v_1, v_2, v_3 and v_4 are sound and complete.

5.4 Simplification of a set of views

As a final issue, we consider that one would like to compile view descriptions off-line into more compact semantically equivalent forms to speed up on-line processing. This is simply the minimization process of 4.2 with the added constraint that any rewriting of the set of views $v_1, ..., v_n$ to $v'_1, ..., v'_m$ must be semantically equivalent and *must able to give certain answers to each input view* $v_1, ..., v_n$.

6 Related Work

The problem addressed in this work relates to the two problems of deriving maximally contained rewritings of a query over a set of views [14] and the problem of obtaining certain answers [1][13] over closed-world views. In fact, we combine these problems into the single problem of deriving a certain rewriting of a query, given a set of closed world views.

The derivation of the certain answer description, is greatly simplified by the schema tuple query assumption. To the author's knowledge no prior work has invoked the 'schema tuple query' restriction explicitly. Certainly one could imagine a regime in which the schema tuple restriction would be enforced over non-recursive Datalog. In such a case Q_{GT} would be correspond to the class of non-recursive Datalog programs with negation in their rule bodies.

Though it may seem to be an odd restriction, there are two 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[8]. Since the atoms of sets are whole tuples, we can speak of taking the 'typed' difference between two tuple sets. In fact we can derive such query differences syntactically without having to materialize answer sets (see theorems 3 and 4). We may also speak of taking differences between tuple sets of heterogeneous types. 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. 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 limitations of the schema tuple restriction may usually be overcome by considering a virtual, highly decomposed version of the schema. Alternatively one may enable projection and simple aggregation as an activity on the periphery of a core schema tuple expression processor. Finally it should be noted that there is a very direct correspondence between queries specified in $\mathcal{L}_{GT}$ and standard SQL. Queries in $\mathcal{L}_{GT}$ mirror the SQL of the form:

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

[8]Decidable description logics [10] of the type *ALC* and beyond also share this property ($A(x)$ - $B(x)$ may be represented as $A(x) \sqcap \neg B(x)$). Description logics typically 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 [4]. For an in depth discussion of this limitation with respect to databases see [16].

where the `sub-query` is of the same form. Naturally simple and join conditions may be added to such queries and there may be more than one sub-query.

The language $\mathcal{L}_{GT}$ is related to the language described in [16], when the author was unaware of the guarded fragment [2] of first-order logic. This prior language, now named $\mathcal{L}_{SBT}$ relied on the decidability of the $\exists^*\forall^*$ fragment [3] . Interestingly this fragment is not normally decidable under complementation ($\neg\exists^*\forall^*\rho \equiv \forall^*\exists^*\neg\rho$), yet $\mathcal{L}_{SBT}$ is. The reason that $\mathcal{L}_{SBT}$ stays closed under complementation is intimately connected with the aforementioned schema tuple query assumption. In any case it will be interesting to compare the practical and theoretical trade-offs between $\mathcal{L}_{SBT}$ and $\mathcal{L}_{GT}$.

It should finally be mentioned that the practical problem addressed in this work is very similar in spirit to the problem confronted by the PANORAMA system [18]. PANORAMA allows users to attach *properties* to views and then decides which other views inherit such properties. PANORAMA's view definition language is strictly conjunctive [7] and the set of properties include *soundness*, *completeness* and *emptiness*, but it is an open, extensible set. PANORAMA associates a meta-schema with the actual schema where the meta-schema encodes the view definitions with associated properties. Queries are processed over both the base relations as well as the meta relations. The normal extensional answer is augmented with a set of view/property pairs that cover the extensional answers. The technique is efficient, yet incomplete. That is not all relevant covering views will be reported in the meta-answer. Motro terms his approach *algebraic* and contrasts it with the logical based work in *semantic query optimization* [5].

7 Conclusions and future work

The main result of this paper is to show that if sound and complete materialized views are described using the language $\mathcal{L}_{GT}$ and the user's query is specified using the language $\mathcal{L}_{GT}$ and is not self-joining, then one may generate intensional descriptions (within the language $\mathcal{Q}_{GT}$) of the certain answers of the query over the views. Moreover one may directly apply the resulting query over the materialized views to obtain such certain answers.

The languages $\mathcal{L}_{GT}$ and $\mathcal{Q}_{GT}$ may be translated into the highly expressive guarded fragment of first order logic [2][12]. We envision the application of these results in a number of areas including, but not limited to, data warehousing, semantic query optimization and cooperative query answering.

8 Acknowledgments

I thank Steve Hegner for his comments on an earlier draft of this paper. I also thank the anonymous reviewers that pointed out several flaws in my initial definition of $\mathcal{L}_{\mathcal{GT}}$.

9 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] H. Andreka, J. van Benthem, and I. Nemeti. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27:217–274, 1998.

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

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

[5] U. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.

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

[7] 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.

[8] A. Chandra and M. Vardi. The implication problem for functional dependencies and inclusion dependencies in undecidable. *SIAM Journal Comput.*, 14(3):671–677, 1985.

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

[10] 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.

[11] H. Ganzinger and H. de Nivelle. A superposition decision procedure for the guarded fragment with equality. In *LICS*, pages 295–304, 1999.

[12] E. Grädel. On the restraining power of guards. *Symbolic Logic*, 64:1719–1742, 1999.

[13] G. Grahne and A. Mendelzon. Tableau techniques for querying information sources through global schemas. In *proc. of ICDT*, pages 332–347, 1999.

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

[15] M. Levene and G. Loizou. How to prevent interaction of functional and inclusion dependencies. *Information Processing Letters*, 71:115–125, 1999.

[16] M. Minock. Knowledge representation using schema tuple queries. In *KRDB '03*. IEEE Computer Society Press, 2003.

[17] J.C. Mitchell. The implication problem for functional and inclusion dependencies. *Informatica and Control*, 56:154–173, 1983.

[18] A. Motro. Panorama: A database system that annotates its answers to queries with their properties. *Intell. Inf. Syst.*, 7(1):51–73, 1996.

[19] J. Ullman. *Principles of Database and Knowledge-Base Bystems*. Computer Science Press, Rockville, Maryland, 1989.

10 Appendix

Proof of Theorem 1:

Let L be a first order language. The *atomic formulas* (or atoms) of L are of the usual forms: $v_1 = v_2$ for domain variables v_1, v_2 and $P(v_1, ..., v_n)$ for the n-ary relation symbol $P \in \mathcal{P}$.

Definition 7 *(Guarded formula [2])*
The *guarded formulas* of L are defined inductively as follows.

1. An atomic formula is a guarded formula
2. If φ, ψ are guarded formulas, then $\varphi \wedge \psi$ and $\neg\varphi$ are guarded formulas.
3. Let $\bar{v}$ be a finite, non-empty sequence of domain variables, ψ a guarded formula and G an atom such that $Free(\psi) \subseteq Free(G)$. Then $(\exists\bar{v})(G \wedge \psi)$ is a guarded formula where G is called the *guard* of the quantifier.

Sentences within the guarded fragment with equality are decidable for satisfiability, have the finite model property and have the tree model property [2][12]. If we can translate any expression $\ell \in \mathcal{L}_{GT}$ as well as well as the inclusions dependencies I into the guarded fragment, then theorem 1 will be proven.

Translation of $\ell \in \mathcal{L}_{GT}$ to the guarded fragment

Because we test for satisfiability, the free variable of ℓ is considered existentially quantified: $\ell' = (\exists x)(\ell)$. This re-frames the problem to that of showing that guarded components of the tuple calculus, defined inductively in definition 3 may always be translated from the ‘syntactic sugar’ of the tuple calculus into the guarded fragment of first order logic.

In what follows, the notation $\overline{y}$ corresponds to expansion of tuple variable y to a set of domain variables. These variables range over the first order predicate $\overline{P}$ which corresponds to P in the tuple calculus. When necessary, we write the individual variables in the first order case. That is we write, $(\exists y_1),...,(\exists y_n)(\overline{P}(y_1,...,y_n) \wedge ...)$. Constants are handled through introducing special unary ‘constant relations’ for each constant appearing in a query. Additionally we enforce mutual exclusion between such constant relations, for example $(\forall v)(Norway(v) \Rightarrow \neg Sweden(v))$

It suffices to show that each resulting guarded component in the inductive definition presented in definition 3 may be translated into a first order formula of the guarded fragment.

1. The tuple formula $(\exists y)(P(y) \wedge \phi)$ is translated to the formula $(\exists \overline{y})(\overline{P}(\overline{y}) \wedge \overline{\phi})$ where $\overline{\phi}$ corresponds to the first order expansion of all the conditions of the tuple formula ϕ. In performing this translation, set operators ($\in$ and $\notin$) are rewritten using $=$ and $\neq$ and all simple conditions are expressed through ranging domain variables over introduced constant relations. The resulting formula rests within the guarded fragment.

2. $(\exists y)(P(y) \wedge \phi \wedge s' \cdot (\exists y')(P'(y') \wedge y.a = y'.b \wedge \phi'))$ is a guarded component where $y.a = y'.b$ is a meaningful equality join between the i-th attribute of P and the j-th attribute of P'. The translation of this component into the guarded fragment is: $(\exists y_1),...,(\exists y_i),...,(\exists y_n)$ $(\overline{P}(\overline{y}) \wedge \overline{\phi} \wedge s' \cdot (\exists y'_1),...,(\exists y'_{j-1})$ $(\exists y'_{j+1}),...,(\exists y'_{n'})$ $(\overline{P'}(y'_1,...,y'_{j-1},y_i,y'_{j+1},...,y'_{n'}) \wedge \overline{\phi'}))$

3. Conjunction and negation of guarded components are merely translated down to the first order fragment, where they are permitted.

Translation of inclusion dependencies to the guarded fragment

Each inclusion dependency in I, is written in tuple calculus as $(\forall x)(P(x) \Rightarrow (\exists y)(P'(y) \wedge x.a_1 = y.b_1 \wedge ... \wedge x.a_m = y.b_m))$. Such expression may be translated to the guarded fragment as $\neg(\exists \overline{x})(\overline{P}(\overline{x}) \wedge \neg(\exists \overline{y'})(\overline{P'}(\overline{y'} \cup \overline{x'})$ where $\overline{y'}$ are the variables in $\overline{y}$ which are not joined on and $\overline{x'}$ are the variables in $\overline{x}$ which are joined on where it is assumed that such variables will be properly ordered in $\overline{P'}(\overline{y'} \cup \overline{x'})$. This resulting expression is within the first order guarded fragment.

The translation of the guarded component plus the translation of the inclusion dependencies are conjoined together and the resulting formula is decided for satisfiability [11]. □

Proof of Theorem 2:
Proof:

A satisfiability test may be conducted for each disjunct of $q \in Q_{GT}$. Since the number of disjuncts is finite, based on theorem 1 this process will terminate with a correct answer. $\Box$

Proof of Theorem 3:
A simple consequence of associativity and distributivity of $\wedge$ over $\vee$. $\Box$

Proof of Theorem 4:
A simple consequence of DeMorgan's law, associativity and distributivity of $\wedge$ over $\vee$. $\Box$

Proof of Theorem 5:
1 is true iff $\{x|q_1\} - \{x|q_2\} = \emptyset$ under the constraints I. This may be decided based on theorems 2 and 4. The decidability of 2 follows directly from the ability to decide predicate 1. The decidability of 3 follows directly from the theorems 2 and 3. $\Box$

Proof of Theorem 6:
If τ is a certain answer to $\{x|\ell\}$ then $\tau \in \{x|\ell\}$ for all database states d consistent with the extension $\{x'|\nu\}$. Clearly, therefore, $\tau \in \{x'|\nu\}$. More specifically $\tau \in \{x'|\{x'/x\}\ell \wedge \nu\}$. Now, since any possible state of affairs might exist for relations other than the single relation which ν ranges over, there are always states that could rule a tuple in or out based on whether ℓ's arbitrary sub-components are satisfied. Thus for an answer to be certain, that is unable to be ruled in or out based on states outside of ν, the sub-components of ℓ must not play a role. Thus $\tau \in \{x'|\{x'/x\}drop(\ell) \wedge \nu\}$ Since it is the case that $\tau \in \{x'|\{x'/x\}\ell \wedge \nu\}$, and dropping components may only generalize a query, it must be that $\{x'|\{x'/x\}drop(\ell) \wedge \nu\} = \{x'|\{x'/x\}\ell \wedge \nu\}$.

If $\{x'|\{x'/x\}drop(\ell) \wedge \nu\} = \{x'|\{x'/x\}\ell \wedge \nu\}$ for all database states and if $\tau \in \{x'|\ \{x'/x\}drop(\ell) \wedge \nu\}$ then $\tau \in \{x'|\{x'/x\}\ell \wedge \nu\}$. Therefore $\tau \in \{x|\ell\}$ and $\tau \in \{x'|\nu\}$ since, by definition all database states include $\{x'|\nu\}$. Therefore τ is a certain answer. $\Box$

Proof of Theorem 7:
Naturally we take σ to be the expression derived using the algorithm in section 5.2.

If $\tau \in \{x|\sigma\}$ then τ is a certain answer to $\{x|\ell\}$ given the sound and complete views $\mathtt{v}_1, ..., \mathtt{v}_\mathtt{n}$. This statement is true, because σ is written only using views from $\mathtt{v}_1, ..., \mathtt{v}_\mathtt{n}$ and simple and set conditions that may be evaluated over such views. Since the views are sound and complete, there is no special database instance d that would rule τ out of the answer set. Thus τ is certain.

If τ is a certain answer to $\{x|\ell\}$ given the sound and complete views $\mathtt{v}_1, ..., \mathtt{v}_\mathtt{n}$, then $\tau \in \{x|\sigma\}$. It is harder to establish this statement. Assume that τ is a certain answer to $\{x|\ell\}$ given the sound and complete views $\mathtt{v}_1, ..., \mathtt{v}_\mathtt{n}$. Now suppose that $\tau \notin \{x|\sigma\}$. This means that $\tau \in \{x|\alpha\}$, where α is satisfiable and $\alpha \wedge \sigma$ is not satisfiable. It is clear that α must be written only using views from $\mathtt{v}_1, \ldots, \mathtt{v}_n$ using simple and set conditions that may be evaluated over such views – otherwise some possible state outside of $\mathtt{v}_1, \ldots, \mathtt{v}_n$ could rule τ outside of $\{x|\ell\}$ and thus violate the assumption that τ is certain. Now the crux of the issue is whether α may be written in a more expressive language that would rule τ as a certain answer of $\{x|\ell\}$. But, since $\ell \in \mathcal{L}_{GT}$, any such enhancement of expressivity could not be expressed in ℓ. Given this, to make an impact on what gets ruled in or out of the answer set of ℓ, α must not be in a language that is more expressive than $\mathcal{L}_{GT}$. We may thus assume that $\alpha \in \mathcal{L}_{GT}$. Now given that ℓ employs only so many constants in its components and that a certain answer generating view, as a consequent of theorem 6, must use a tighter predication for its existential variables than ℓ, there are only a finite number of possible re-writings of the *n* views into an expression of some width of *k* or less. In fact the algorithm of section 5.2 calculates just this expression for the *k* set to the width of the widest disjunct of σ. The only possibility left is that α is a rewrite greater than width *k* that expresses a case of certainty that is not covered by σ. But since all additional predications cause further specialization, this is not possible and thus $\{x|\alpha\} \subseteq \{x|\sigma\}$. This is a contradiction of the assumptions above and thus $\tau \in \{x|\sigma\}$. $\square$