Interpreting a Reconstructed Relational Calculus: Extended Abstract

Aaron Watters
Computer and Information Sciences
New Jersey Institute of Technology
aaron@vienna.njit.edu

University Heights
Newark, NJ, 07102
(201)596-2666

Abstract

This paper describes a method for answering all relational calculus queries under the assumption that the domain of data values is sufficiently large. The method extends recent theoretical results that use extended relation representations to answer domain dependent queries, without the use of auxilliary variables or invented constants or an explicit enumeration of the active domain. The method is shown to be logically correct and to have polynomial data complexity. By identifying relational algebra operations with relational calculus queries this approach extends relational algebra to a full boolean algebra, where intersection, union, and difference are defined between any two relations, whether or not they are union compatible. An example illustrates that this approach can be useful in distributed query optimization.

1 Goals

This paper has two goals: the specific goal of this paper is to propose a method for directly calculating answers to *any* tuple relational calculus query regardless of whether the query is domain dependent; the broader goal of this paper is to advocate the interpretation of relations as *description filters.*

This research builds on the theoretical work of Hull and Su [7] and of Aylamazyan, Gilula, Stolboushkin, and Schwartz [2] which independently established (among several results)

Proposition 1 *If we assume that the domain of database constants is sufficiently large, and if we use an extended representation for relations then all relational calculus queries have domain independent answers.*

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0367...$1.50

The additional contribution of the present paper is to introduce a new extended relation representation that does not include invented constants or variables in answers or an explicitly enumerated active domain, as in both of the cited seminal papers, and to present an algorithm for directly interpreting relational calculus queries. The analysis of this algorithm establishes that the method is correct, and produces canonical results, and that the method possesses polynomial data complexity in the sense of Vardi [14]. The algorithm has been implemented for experimental purposes using Common Lisp.

2 Relational Calculus and Domains of Values

Relational theory traditionally offers two dual bases for query specification: query specification based on the relational algebra; and query specification based on a relational calculus. These dual approaches are not symmetrical, however, since a subset of well formed relational calculus queries are identified as "undesirable" - the domain dependent queries. Such queries are often considered unanswerable or meaningless.

A student of the history of mathematics might raise the following question: are the domain dependent queries unanswerable because they have no consistent answer - in the sense that $1 \div 0$ cannot represent a number without destroying the consistency of the number system - *or* are they unanswerable within the current space of relation representations but answerable in an extended space - in the sense that $\sqrt{-1}$ cannot be represented as a *real* number but may be consistently represented in the extended *complex* number system.

The answer to this question is that neither option is technically correct. A domain dependent query has no *unique* answer, since the answer will vary as the database domain varies. However, using a modified representation for relations and assuming the database domain is sufficiently large, domain dependent queries have answers that are independent of the domain. For

example the domain dependent query

$$\{x \mid x.\texttt{name} = \texttt{'nancy'} \wedge \forall y : y.\texttt{name} = \texttt{'nancy'}\}$$

has two possible answers, depending on the domain. If the domain contains only the constant **'nancy'** then the answer is **name->nancy**. However, if the domain is sufficiently large – in this case if it contains any other constant – the answer is always an empty relation (or **false** in the notation of this paper). For another example the domain dependent query

$$\{x \mid \neg x.\texttt{name} = \texttt{'nancy'}\}$$

may be answered in the present approach using an extended relation representation

```
true
except
name->"nancy"
```

with the interpretation that the relation accepts all descriptions aside from those with name **'nancy'**.

So, in a sense, the analogy with the real and complex number systems holds – just as the real number system can be expanded to allow representations for negative roots, so can the space of relational representations be expanded to include answers to all domain dependent queries, provided we assume the database domain is sufficiently large.

3 An Example Use

At this point a pragmatist might raise the question: does the extension of the relational model to include answers to domain dependent queries have any practical uses? The following example hopes to illustrate immediate value for these ideas in distributed query optimization, *even when the final query answers are standard relations.*

Consider a scenario where a salesperson maintains a local database which includes a list of current customers **Cust** and a list of target industries **Target** that the salesperson hopes to sell to. Suppose further that the salesperson subscribes to a remote information service which provides a large list of companies **Co**. From the local and remote database the salesman wants to derive a list of prospective customers Q – companies that are not current customers, but which are classified in one of the target industries. The schema for the databases is illustrated in Figure 1.

In tuple relational calculus the salesperson's query may be formulated as

$$Q = \{x \mid x \in \texttt{Co} \wedge \neg x \in \texttt{Cust} \wedge (\exists y : y \in \texttt{Target} \wedge y.\texttt{class} = x.\texttt{class})\}$$

A query optimizer might attempt to optimize this query using *local processing* [1], but there is no obvious local subquery to evaluate for this example within the traditional relational framework. However, using the extended relational methods described in this paper, a query optimizer could factor the query into a *domain dependent* local subquery

$$D = \{x \mid \neg x \in \texttt{Cust} \wedge (\exists y : y \in \texttt{Target} \wedge y.\texttt{class} = x.\texttt{class})$$

which can be evaluated using only local relations, producing an extended relation D. The relation D can then be transferred to the remote service where the final answer can be computed using

$$Q = \{x \mid x \in \texttt{Co} \wedge x \in D\}$$

and the result may then be transferred back to the salesperson's workstation.

Figure 2 illustrates the local evaluation of D using the author's lisp-based implementation of the tuple relational calculus.

4 Logical Framework

This paper uses a new (reconstructed) syntax and semantics for the tuple relational calculus inspired by the work of Imielinski and Lipski [8], with notation inspired by Gallier [6].

The language: The tuple relational calculus is constructed from a number of syntactic components: a set of attributes ATTS, with generic members $A_1, A_2, A_3, \ldots$; a domain of values VLS, with generic members $v_1, v_2, v_3, \ldots$; a set of relation constants RCNSTS, with generic members $r_1, r_2, r_3, \ldots$; and a set of tuple variables TVRS, with generic members $x_1, x_2, x_3, \ldots$. From these components the set of logical expressions EXPS is defined recursively by the BNF equation

$$\begin{aligned} E :== \quad & \text{true} \mid \text{false} \mid x_1.A_1 = v_1 \mid x_1.A_1 = x_2.A_2 \mid \\ & x_1 \in r_1 \mid \neg E \mid E \wedge E \mid \\ & E \vee E \mid \exists x_1 : E \mid (E) \end{aligned}$$

In addition to these basic constructs add universal quantification and implication which are defined by the following macro rewrites

$$\begin{aligned} E \rightarrow E' \quad &\equiv \quad (\neg E) \vee E' \\ \forall x_1 : E \quad &\equiv \quad \neg \exists x_1 : \neg E. \end{aligned}$$

Using the expression syntax, the syntax of a relational calculus query is given by

$$\{x_1 \mid E\}$$

where x_1 is the only *free* variable of the expression E – that is, x_1 is the only variable not previously bound by quantification in E.

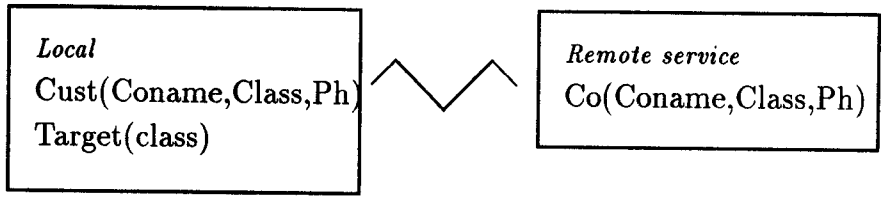

Figure 1: The local database and the remote information service.

```
> (set-rel cust
     coname->geq * class->retail * ph->1245 +
     coname->succ * class->lt-ind * ph->4854 +
     coname->cong * class->mech * ph->5752 +
     coname->prime * class->acctg * ph->8790 +
     coname->aleph * class->advt * ph->1334 +
     coname->perp * class->advt * ph->6443 +
     coname->zeta * class->taxi * ph->4321 +
     coname->floor * class->lt-ind * ph->4564 +
     coname->slash * class->law * ph->7897 +
     coname->tilde * class->retail * ph->1546 )
CUST
> (set-rel target
     class->retail +
     class->law +
     class->mktg)
TARGET
> (relcalc D x
     (* (- (in x cust))
        (ex y (in y target) (x_class=y_class))))
D
> (exrel-display D)
CLASS->RETAIL +
CLASS->LAW +
CLASS->MKTG
except
CLASS->RETAIL* PH->1546* CONAME->TILDE +
CLASS->RETAIL* PH->1245* CONAME->GEQ +
CLASS->LAW* PH->7897* CONAME->SLASH
```

The above interaction declares the values of the relations `Cust` *and* `Target` *using* `(setrel ...)`*, defines the relation* `D` *using* `(relcalc ...)` *with a domain dependent query equivalent to*

$$D = \{x \mid \neg x \in \texttt{Cust} \wedge (\exists y : y \in \texttt{Target} \wedge y.\texttt{class} = x.\texttt{class})\}$$

and displays the resulting value of D *using* `(exrel-display ...)`.

Figure 2: Deriving D using the lisp implementation.

Interpretation: Set theoretical meaning is assigned to relational calculus expressions and queries using a modified 'cylindric' model theoretic interpretation constructed using a number of sets:

- the set of tuples or "total descriptions" DSCS that map all attributes to values, which is defined as the function space

$$\text{DSCS} = (\text{ATTS} \longrightarrow \text{VLS}),$$

  where if f is a member of DSCS then the value assigned to an attribute by f is given by $f(A)$;

- The database instance space INSTS which includes all possible associations between relation names and subsets of total descriptions, which is defined as the function space

$$\text{INSTS} = (\text{RCNSTS} \longrightarrow \mathcal{P}(\text{DSCS}))$$

  ($\mathcal{P}(X)$ is the powerset of X - the set of all subsets of X); and

- the variable assignment space ASNS which includes all possible assignments of tuple variables to total descriptions, which is defined as the function space

$$\text{ASNS} = (\text{TVRS} \longrightarrow \text{DSCS}),$$

  where if s is a member of ASNS then the value assigned to a variable x on an attribute A under the assignment s is given by $s(x)(A)$;

A database $M \in$ INSTS is a function which assigns to every relation name $r \in$ RCNSTS a subset of total descriptions $M(r) \subseteq$ DSCS. Each database $M \in$ INSTS imposes an interpretation on every relational calculus expression $E \in$ EXPS, written $I_M(E) \subseteq$ ASNS which identifies the set of variable assignments which satisfies the expression E under the database instance M. This interpretation function is defined recursively as follows:

$$\begin{aligned} I_M(\text{true}) &= \text{ASNS} \\ I_M(\text{false}) &= \emptyset \\ I_M(x_1.A_1 = v_1) &= \{s \mid s \in \text{ASNS}, s(x_1)(A_1) = v_1\} \end{aligned}$$

$$
\begin{aligned}
I_M(x_1.A_1 = x_2.A_2) &= \{s \mid s \in \text{ASNS}, \\
&\qquad s(x_1)(A_1) = s(x_2)(A_2)\} \\
I_M(x_1 \in r_1) &= \{s \mid s \in \text{ASNS}, s(x_1) \in M(r_1)\} \\
I_M(E \wedge E') &= I_M(E) \cap I_M(E') \\
I_M(E \vee E') &= I_M(E) \cup I_M(E') \\
I_M(\neg E) &= \text{ASNS} - I_M(E)
\end{aligned}
$$

and $I_M(\exists x_1 : E)$ is defined to be the set of all $s \in \text{ASNS}$ where there is an $f \in \text{DSCS}$ such that if $s' \in \text{ASNS}$ is defined by

$$
s'(x) = \begin{cases} f & \text{if } x = x_1 \\ s(x) & \text{otherwise} \end{cases}
$$

then $s' \in I_M(E)$.

Furthermore the database M imposes on a query the following interpretation as a set of total descriptions:

$$
I_M(\{x \mid E\}) = \{s(x) \mid s \in I_M(E)\}
$$

i.e., the answer to a query $\{x \mid E\}$ consists of the set of total description substitutions for x which occur in a substitution satisfying E.

The interpretation function I_M gives rise to the notions of logical entailment and equivalence in the usual way.

Definition 1 *An expression E entails an expression E', written $E \Rightarrow E'$ if and only if for all $M \in$ INSTS $I_M(E) \subseteq I_M(E')$. The expressions are logically equivalent, written $E \Leftrightarrow E'$ if and only if $E \Rightarrow E'$ and $E' \Rightarrow E$.*

Furthermore an expression E is said to be *consistent* when $I_M(E)$ is non-empty for some M, otherwise E is said to be *inconsistent.*

5 The Boolean Space of Relations

The semantics described above is unusual in that the interpretation for expressions and queries are "bigger" than the standard interpretation. In particular if the attributes ATTS are $\{A, B, C\}$ and the values VLS are $\{1, 2, 3\}$ then the answer to the 'domain independent' query

$$
\{x \mid x.A = 1\}
$$

includes the total descriptions

```
A->1 * B->1 * C->1 +
A->1 * B->1 * C->2 +
A->1 * B->1 * C->3 +
A->1 * B->2 * C->1 +
A->1 * B->2 * C->2 + ...
```

(in this notation * represents logical *and* and + represents logical *or*) and this answer is clearly domain dependent, since it will grow larger if we insert the additional constant 4 into the domain. Nevertheless, we can represent this answer in a domain independent way as `A->1` with the understanding that the unlisted attributes "may take on any value in the domain." on the other attributes. Thus under the semantics described above the interpretation for most queries will be "domain dependent", but the answer can be represented using a notation that defines a domain independent "filter" which specifying which descriptions are included in the answer and which are excluded.

Previous examples introduced the filter representation for relations which may be defined by the nonterminal Γ in the following BNF description

$$
\begin{aligned}
\tau &::== A_1 = A_2 \mid A_1\texttt{->}v_1 \mid \tau\texttt{*}\tau \mid \texttt{true} \\
d &::== \tau \mid \tau\texttt{+}d \mid \texttt{false} \\
\Gamma &::== d \mid d \text{ except } \Gamma
\end{aligned}
$$

Here the nonterminal τ represents a slightly extended notion of a *tuple* and the nonterminal d represents the special case of the filter representation which is analogous to a "standard relation." Intuitively, a filter representation with no exceptions $\tau_1 + \tau_2 + \ldots + \tau_n$ denotes the set of all descriptions $f \in \text{DSCS}$ that match one of the τ_i's on all attribute values, but may take on any value for the other attributes; and a filter with exceptions d `except` Γ denotes all descriptions denoted by d but not denoted by Γ.

The precise logical interpretation for the filter notation is obtained by translation to tuple relational calculus expressions using the translation function $\text{EX}(v, \phi)$ as follows:

$$
\begin{aligned}
\text{EX}(x, A_1 = A_2) &= x.A_1 = x.A_2 \\
\text{EX}(x, A_1\texttt{->}v_1) &= x.A_1 = v_1 \\
\text{EX}(x, \texttt{true}) &= \text{true} \\
\text{EX}(x, \tau\texttt{*}\tau') &= \text{EX}(x, \tau) \wedge \text{EX}(x, \tau') \\
\text{EX}(x, \texttt{false}) &= \text{false} \\
\text{EX}(x, \tau\texttt{+}d) &= (\text{EX}(x, \tau)) \vee (\text{EX}(x, d)) \\
\text{EX}(x, d \texttt{ except } \Gamma) &= (\text{EX}(x, d)) \wedge \neg(\text{EX}(x, \Gamma))
\end{aligned}
$$

where in cases of ambiguity the later equation takes precedence. The relation denoted by a filter expression Γ is then defined as the answer to the query

$$
\{x \mid \text{EX}(x, \Gamma)\}
$$

under a given database M. For consistency *all* relations – answers as well as base relations – are assumed to correspond to filter expressions in this presentation.

This paradigm, where all relations "denote" sets containing elements of the same type (total descriptoins),

provides the fundamental intellectual basis for the interpretation of all relational calculus expressions. As an interesting side effect, this approach also places the interpretation for all relations within a single boolean algebra where the relational algebra operations of union, intersection, and difference may be extended to operate on pairs of relations that are not "union compatible."

For instance, suppose the relation r is represented by the filter expression

```
A->1 * B->2 +
A->3 * B->4 +
A->5 * B->3
```

and the relation s is represented by the filter expression

```
B->2 * C->6 +
B->4 * C->8 +
B->9 * C->7.
```

Then the union $r \cup s$ may be identified with the answer to the domain dependent query

$$\{x \mid x \in r \vee x \in s\}$$

with the filter representation

```
A->1* B->2 +
A->3* B->4 +
A->5* B->3 +
          B->9* C->7 +
          B->4* C->8 +
          B->2* C->6
```

Further more $s - r$ may be identified with the answer to the domain dependent query

$$\{x \mid x \in s \wedge \neg x \in r\}$$

with the filter representation

```
B->9* C->7 +
B->4* C->8 +
B->2* C->6
except
C->8* B->4* A->3 +
C->6* B->2* A->1
```

which cannot be represented as a standard relation. The case of intersection is left to the reader.[1]

At this point we state the following proposition which is a corollary of results to follow:

Proposition 2 *Under the assumption that the domain of values* VLS *is sufficiently large, the answer to any relational calculus query can be represented using a domain independent filter representation.*

[1] These representations were derived using the lisp implementation of the proposed method, with some white space added to increase readability.

This proposition uses the following modified definition for domain independence.

Definition 2 *A filter Γ is domain independent with respect to a query $\{x \mid E\}$ if it is constructed from domain values that appear in E or in the filter representations for the relations mentioned in E.*

A number of logical results follow which can be proven assuming a finite but "sufficiently large" domain. However, in order to simplify the presentation in this extended abstract *assume the domain* VLS *to be infinite* (which is certainly sufficiently large).

6 Representing Boolean Expressions

The ultimate goal of this paper is to define and analyze an interpreter for all relational calculus queries. The interpreter to be defined uses basic data structures and operations which embed the boolean part of the tuple relational calculus expressions (expressions formed from equalities combined using $\wedge$, $\vee$, and $\neg$) into a boolean algebra of data structures.

Pair representations for conjunctive expressions: First, consider the problem of constructing a canonical logical representation for expressions formed from simple conjunctions with the abstract syntax

$$E^c :== \text{true} \mid \mathbf{eq} \mid \mathbf{eq} \wedge E^c$$

where **eq** represents simple equality expressions of form $x_1.A_1 = v_1$ or $x_1.A_1 = x_2.A_2$. Any such consistent expression may be represented using the a pair $\rho = (e, \nu)$ where e is a set of equivalence classes of variable/attribute pairs and ν is a partial mapping of elements of e to values ($\nu \in (e \mapsto \text{VLS})$) - that is, for some $\epsilon \in e$ the value $\nu(e) \in \text{VLS}$ is defined, but for other $\epsilon \in e$ the value may not be defined. For example pair (e, ν) with equivalence relation

$$e = \left\{ \begin{array}{l} \{x_1.A_1, x_2.A_2, x_3.A_3\} \\ \{x_4.A_4\} \\ \{x_6.A_6, x_7.A_7, x_8.A_8, x_9.A_9\} \end{array} \right\}$$

and with partial mapping

$$\nu = \left\{ \begin{array}{l} \{x_1.A_1, x_2.A_2, x_3.A_3\} \mapsto v_1 \\ \{x_4.A_4\} \mapsto v_2 \end{array} \right\}$$

represents the expression

$$x_1.A_1 = v_1 \wedge x_2.A_2 = v_1 \wedge x_3.A_3 = v_1 \wedge$$
$$x_4.A_4 = v_2 \wedge$$
$$x_6.A_6 = x_9.A_9 \wedge x_7.A_7 = x_9.A_9 \wedge x_8.A_8 = x_9.A_9$$

and a number of other equivalent conjunctive expressions. Since elements of e are equivalence classes, all

pairs of elements of e must be disjoint. In addition we say a pair $\rho = (e, \nu)$ is *canonical* if for any $\epsilon \in e$ where ϵ contains only one element the value $\nu(\epsilon)$ is defined.[2]

The function $\mathrm{R}_c(E^c)$ defines a translation procedure which maps a *consistent* conjunctive expression E^c to its pair representations defined recursively as follows:

$$\begin{aligned}
\mathrm{R}_c(\text{true}) &= (\emptyset, \emptyset) \\
\mathrm{R}_c(x.A = v) &= (\{\{x.A\}\}, \{\{x.A\} \mapsto v\}) \\
\mathrm{R}_c(x.A = x.A) &= \mathrm{R}_c(\text{true}) \\
\mathrm{R}_c(x_1.A_1 = x_2.A_2) &= (\{\{x_1.A_1, x_2.A_2\}\}, \emptyset) \\
&\quad\ x_1.A_1, x_2.A_2 \text{ distinct} \\
\mathrm{R}_c(\alpha \wedge \beta) &= (e, \nu)
\end{aligned}$$

where if $\mathrm{R}_c(\alpha) = (e_1, \nu_1)$ and $\mathrm{R}_c(\beta) = (e_2, \nu_2)$ then e is the smallest set of equivalence classes and and ν the least defined partial function such that

- if $\epsilon_1 \in e_1$ then there is a $\epsilon \in e$ where $e_1 \subseteq e$ and if $\nu_1(\epsilon_1)$ is defined then $\nu_1(\epsilon_1) = \nu(\epsilon)$; and
- if $\epsilon_2 \in e_2$ then there is a $\epsilon \in e$ where $e_2 \subseteq e$ and if $\nu_2(\epsilon_2)$ is defined then $\nu_2(\epsilon_2) = \nu(\epsilon)$.

The conditions on the final equation cannot always be satisfied – sometimes no such pair (e, ν) exists – and in this case $\mathrm{R}_c(E^c)$ is undefined.

Define an inverse mapping $\mathrm{R}_c^-(\rho)$ which maps a pair (e, ν) to an appropriate logical interpretation to be the conjunction of equalities including for each $\epsilon \in e$ all equalities $p = p'$ whenever $p, p' \in \epsilon$ are distinct and whenever $\nu(\epsilon)$ is defined adding all equalities $\rho = \nu(\epsilon)$ for each $\rho \in \epsilon$.[3]

The $\mathrm{R}_c(E^c)$ provides a unique and correct representation for the logical meaning of a conjunctive expression E in the sense expressed by the following straightforward results.

Proposition 3 *For any conjunctive expression E^c the function value $\mathrm{R}_c(E^c)$ is defined if and only if E^c is consistent. For any consistent conjunctive expression E^c we have $E^c \Leftrightarrow \mathrm{R}_c^-(\mathrm{R}_c(E^c))$, and $\rho = \mathrm{R}_c(E^c)$ is the only canonical pair representation where $E^c \Leftrightarrow \mathrm{R}_c^-(\rho)$.*

We may also effectively characterize entailment among conjunctive expressions using pair representations as follows:

Definition 3 *For two pairs $\rho = (e, \nu)$ and $\rho' = (e', \nu')$ we write $\rho \Rightarrow \rho'$ if and only if we have for each $\epsilon' \in e'$ there is an $\epsilon \in e$ with $\epsilon' \subseteq \epsilon$ such that if $\nu'(\epsilon')$ is defined then $\nu'(\epsilon') = \nu(\epsilon)$.*

[2] A pair (e, ν) which is not canonical can be translated into an equivalent canonical pair by removing from e all singletons ϵ where $\nu(\epsilon)$ is not defined.

[3] Technically, fewer equalities for each ϵ would suffice, but the redundancy introduced here is required for results to follow.

This overloaded notation is justified by the following result.

Proposition 4 *For any pair of consistent conjunctive expressions E and E' we have $E_1^c \Rightarrow E_2^c$ if and only if $\mathrm{R}_c(E_1^c) \Rightarrow \mathrm{R}_c(E_2^c)$.*

For simplicity we may identify the conjunction between two pairs ρ and ρ' as $\rho \wedge \rho' = \mathrm{R}_c(\mathrm{R}_c^-(\rho) \wedge \mathrm{R}_c^-(\rho'))$ whenever the result is defined.[4] This identification has the expected property:

Proposition 5 *When $\rho \wedge \rho'$ is defined then $\mathrm{R}_c^-(\rho) \wedge \mathrm{R}_c^-(\rho')$ is consistent and $\mathrm{R}_c^-(\rho \wedge \rho') \Leftrightarrow \mathrm{R}_c^-(\rho) \wedge \mathrm{R}_c^-(\rho')$.*

Set representations for disjunctions of conjunctive expressions: Now consider the problem of finding a canonical logical representation for disjuncts of conjunctive expressions, with abstract syntax

$$E^d :== E^c \mid E^c \vee E^d.$$

It is tempting to represent such a disjunct $E_1^c \vee E_2^c \vee \ldots \vee E_n^c$ by the set of pairs $\{\mathrm{R}_c(E_1^c), \mathrm{R}_c(E_2^c), \ldots, \mathrm{R}_c(E_n^c)\}$, but such a representation would not be canonical since it maps some equivalent expressions to different representations. For instance the expressions

$$(x.A = 1) \vee (x.B = 2)$$

and

$$(x.A = 1) \vee (x.B = 2) \vee (x.A = 1 \wedge x.B = 2)$$

are equivalent, but would have different representations if we simply form the set of pair representations for conjunctive expressions. However, if we remove from the set of pairs those pairs which are not logically minimal (those which entail another pair) we obtain a canonical representation.

In particular define the function $\mathrm{R}_d(E)$ which maps a disjunction of conjunctive expressions $E^d = E_1^c \vee E_2^c \vee \ldots \vee E_n^c$ as follows

$$\mathrm{R}_d(E^d) = \textsc{wk}(\{\mathrm{R}_c(E_i^c) \mid \mathrm{R}_c(E_i^c) \text{ defined}, i \in \{1, \ldots, n\}\})$$

Here the WK() function returns only those pairs which do not entail another pair, using the entailment test described previously. To identify the valid range of disjunction representations we say that a set of pairs δ is a canonical disjunction set only if $\delta = \textsc{wk}(\delta)$ Note that an inconsistent disjunction of conjunctions is always represented by the empty set $\emptyset$, hence the function may be extended consistently by setting $\mathrm{R}_d(\texttt{false}) = \emptyset$. An 'inverse' for R_d may be chosen such that

$$\mathrm{R}_d^-(\delta) = \begin{cases} \text{false} & \text{if } \delta = \emptyset \\ \mathrm{R}_c^-(\rho_1) \vee \ldots \vee \mathrm{R}_c^-(\rho_n) & \text{if } \delta = \{\rho_1, \ldots, \rho_n\} \end{cases}$$

[4] The Lisp implementation defines the conjunction directly in terms of pairs, however.

The R_d function uniquely characterizes the meaning of a disjunction of conjunctions in the sense of the following result.

Proposition 6 *For any disjunction of conjunctions E^d we have $E^d \Leftrightarrow R_d^-(R_d(E^d))$, and $\delta = R_d(E^d)$ is the only canonical disjunction set such that $E^d \Leftrightarrow R_d(\delta)$.*

Furthermore transformations corresponding to disjunction and conjunction may be defined for disjunction sets as follows:

$$\begin{aligned}\delta \wedge \delta' &= \text{WK}\{\rho \wedge \rho' \mid \rho \in \delta,\ \rho' \in \delta',\ \rho \wedge \rho' \text{ is defined}\}\\ \delta \vee \delta' &= \text{WK}(\delta \cup \delta').\end{aligned}$$

Clearly, the conjunction and disjunction transformations always produce canonical disjunction sets. Furthermore the notation for these transformations are justified by the expected properties that

$$\begin{aligned}R_d^-(\delta \wedge \delta') &\Leftrightarrow R_d^-(\delta) \wedge R_d^-(\delta')\\ R_d^-(\delta \vee \delta') &\Leftrightarrow R_d^-(\delta) \vee R_d^-(\delta')\end{aligned}$$

List representations for boolean relational calculus expressions: Lists of disjunction sets of form $\Re = [\delta_1, \delta_2, \ldots, \delta_n]$ provide representations for expressions that include negations. A list of disjunctive sets corresponds to a boolean expression with the special syntax

$$E^* ::== E^d \mid E^d \wedge \neg(E^*)$$

which shall be called *exception expressions*. In order to reduce the number of parentheses required in exception expressions abbreviate $E^d \wedge \neg(E^*)$ by E^d **except** E^*. Intuitively an expression E^d **except** E^* denotes the set of variable assignments matching E^d with the any variable assignments matching E^* removed – for this reason E^d is called the condition of the expression and E^* is called the exceptions for the expression. An example of an exception expression is

$$\begin{aligned}&(x.\texttt{fname} = \texttt{john} \vee x.\texttt{fname} = \texttt{joe})\\ &\quad \textbf{except}\ (x.\texttt{fname} = \texttt{joe} \wedge x.\texttt{lname} = \texttt{smith})\end{aligned}$$

which represents variable assignments where the **fname** for x is **john** or **joe** unless the **fname** is **john** and the **lname** is **smith**.

A first attempt for representing exception expressions can be defined by

$$\begin{aligned}&R_*^1(E_1^d \textbf{ except } E_2^d \textbf{ except } \ldots \textbf{ except } E_n^d) =\\ &[R_d(E_1^d), R_d(E_2^d), \ldots, R_d(E_n^d)].\end{aligned}$$

Lists produced by this transformation of the form $\Re = [\delta_1, \delta_2, \ldots, \delta_n]$ will be referred to as exception lists. The null list [ ] is one of several possible representations for the expression false. It is convenient to introduce the alternative notation for exception lists

$$\delta_1 \textbf{ except } [\delta_2, \delta_3, \ldots, \delta_n] = [\delta_1, \delta_2, \ldots, \delta_n].$$

for some of the definitions and results to follow.

The natural inverse for the R_*^1 mapping is given by

$$R_*^-([\delta_1, \ldots, \delta_n]) = R_d^-(\delta_1) \textbf{ except } \ldots \textbf{ except } R_d^-(\delta_n)$$

which maps list representations to appropriate logical interpretations. To complete the definition set $R_*^-([\,]) = \text{false}$.

Unfortunately, the R_*^1 mapping is not canonical because it maps logically equivalent exception expressions, such as $x.A = 1$ **except** $x.A = 1$ and false, to different representations. The intuitive problem with $x.A = 1$ **except** $x.A = 1$ is that the condition $x.A = 1$ and the exception $x.A = 1$ "cancel each other." The CLEAN operation removes from exception lists such cancellations. To calculate $\text{CLEAN}([\delta_1, \delta_2, \ldots, \delta_n])$ first set

$$\begin{aligned}\delta_1' &= \text{WK}(\delta_1)\\ \delta_k' &= \delta_k \wedge \delta_{k-1}' \text{ for } k = 2 \ldots n\end{aligned}$$

then repeat the following sequence of assignments until δ_1' and δ_2' contain no common elements

$$\begin{aligned}e &:= \delta_1' \cap \delta_2';\\ \delta_1^* &:= \delta_1' - e;\\ \delta_2' &:= (\delta_2' \wedge \delta_1^*) \vee (\delta_1' \wedge \delta_2') \vee \delta_4;\\ \delta_1' &:= \delta_1^* \vee \delta_3';\\ \delta_k' &:= (\delta_k' \wedge \delta_1^*) \vee \delta_{k+2} \text{ for } k = 3 \ldots n.\end{aligned}$$

This definition uses the convention that $\delta_j = \emptyset$ if $j > n$. It is not difficulty to establish that this loop will terminate after at most $n - 1$ iterations. Finally, if $\delta_1' = \emptyset$ define $\text{CLEAN}([\delta_1, \delta_2, \ldots, \delta_n]) = [\,]$. Otherwise define

$$\text{CLEAN}([\delta_1, \delta_2, \ldots, \delta_n]) = \delta_1' \textbf{ except } \text{CLEAN}([\delta_2', \ldots, \delta_n'])$$

The CLEAN operation preserves logical meaning, *i.e.*

$$R_*^-(\Re) \Leftrightarrow R_*^-(\text{CLEAN}(\Re)).$$

Using CLEAN define the canonical representation $R_*(E^*)$ for an exception expression E^* by

$$R_*(E^*) = \text{CLEAN}(R_*^1(E^*)).$$

To identify the valid range for R_* we call any exception list $\Re$ satisfying $\Re = \text{CLEAN}(\Re)$ a canonical exception list. The R_* mapping provides a canonical representations for exception expressions in the following sense.

Proposition 7 *For any exception expression E^* we have $E^* \Leftrightarrow R_*^-(R_*(E^*))$ and $\Re = R_*(E^*)$ is the only canonical exception list such that $E^* \Leftrightarrow R_*^-(E^*)$.*

7 Compiling Boolean Expressions

Representing *all* boolean expressions with exception lists requires transformations that perform the boolean operations $\wedge$, $\vee$, and $\neg$ within the space of exception lists.

A canonical negation operation for exception lists may be obtained easily by simply "reversing the polarity" of the exceptions, setting

$$\neg\Re = \textsc{Clean}(\mathrm{R}_d(\text{true}) \ \textbf{except} \ \Re).$$

It is relatively straightforward to define conjunction and disjunction recursively in terms of each other – but such definitions tend to lead to exponential algorithms. The following (peculiar) definition provides a disjunction operation that is clearly polynomial. Define

$$[\delta_1, \delta_2, \ldots, \delta_n] \vee [\delta'_1, \delta'_2, \ldots, \delta'_m] = \textsc{Clean}([\delta''_1, \delta''_2, \ldots, \delta''_{n+m}])$$

where each

$$\delta''_k = \delta_k \vee \delta'_k \vee \bigvee\{\delta_i \wedge \delta'_j \mid i + j = k;\ i, j \text{ not both odd}\}.$$

This definition adopts the convention that $\delta_k = \emptyset$ whenever $k > n$ and $\delta'_k = \emptyset$ whenever $k > m$. Furthermore the $\bigvee\{x_1, x_2, \ldots, x_l\}$ operation represents the $x_1 \vee x_2 \vee \ldots \vee x_l$ computation.

Using the disjunction operation, a canonical conjunction operation for exception lists may be defined as

$$[\delta_1, \delta_2, \ldots, \delta_n] \wedge [\delta'_1, \delta'_2, \ldots, \delta'_m] =$$
$$\textsc{Clean}(\delta_1 \wedge \delta'_1 \ \textbf{except} \ ([\delta_2, \delta_3, \ldots, \delta_n] \vee [\delta'_2, \delta'_3, \ldots, \delta'_m]))$$

These operations have the appropriate logical properties, that is,

Proposition 8 *For any pair of canonical exception lists $\Re$ and $\Re'$*

$$\begin{aligned} \mathrm{R}^-_*(\Re \wedge \Re') &\Leftrightarrow \mathrm{R}^-_*(\Re) \wedge \mathrm{R}^-_*(\Re') \\ \mathrm{R}^-_*(\Re \vee \Re') &\Leftrightarrow \mathrm{R}^-_*(\Re) \vee \mathrm{R}^-_*(\Re') \\ \mathrm{R}^-_*(\neg\Re) &\Leftrightarrow \neg\mathrm{R}^-_*(\Re) \end{aligned}$$

Furthermore the computations $\Re \wedge \Re'$, $\Re \vee \Re'$, and $\neg\Re$ produce canonical exception lists.

Using these properties it is a simple matter to define an interpreter INTERP which translates any boolean expression into a canonical representation as exception lists. Such an interpreter may be defined recursively as follows.

$$\begin{aligned} \textsc{Interp}(\mathbf{eq}) &= \mathrm{R}_*(\mathbf{eq}) \text{ where } \mathbf{eq} \text{ is an equality.} \\ \textsc{Interp}(E_1 \wedge E_2) &= \textsc{Interp}(E_1) \wedge \textsc{Interp}(E_2) \\ \textsc{Interp}(E_1 \vee E_2) &= \textsc{Interp}(E_1) \vee \textsc{Interp}(E_2) \\ \textsc{Interp}(\neg E) &= \neg\textsc{Interp}(E). \end{aligned}$$

This interpreter may be extended to interpret 'inclusion terms' of form $x \in r$ using a function $\mathbf{f}$ that associates to each relation name r a filter representation $\mathbf{f}(r)$. In this case define

$$\textsc{Interp}(x \in r) = \textsc{Interp}(\textsc{Ex}(x, \mathbf{f}(r))).$$

Providing interpretations for *all* relational calculus expressions requires a transformation which translates existential assertions of form $\exists x : E$ into exception lists as well.

8 The Vanishing Quantifiers

It turns out that under the assumption of a sufficiently large (or infinite) domain quantified variables 'disappear' from canonical expressions. For example, the following expressions are equivalent

$$(\exists x : x.A = 1 \wedge y.B = 2 \wedge y.C = x.C) \Leftrightarrow (y.B = 2).$$

Intuitively speaking, the existential conditions vanish because there are always enough extra constants available in the domain to construct a substitution for x that satisfies all the equalities mentioning x whenever all other equalities are satisfied. Similarly, the universal quantifier also vanishes, but in the opposite direction. For example we have

$$(\forall x : x.A = 1 \wedge y.B = 2 \wedge y.C = x.C) \Leftrightarrow \text{ false}.$$

Intuitively, any universally quantified conjunction that includes any condition of the quantified variable is equivalent to false because a substitution can always be constructed for the variable that violates one of the conditions. More complex expressions require greater subtlety. For example,

$$\begin{aligned} &\exists x : (x.A = 1 \ \textbf{except} \ (x.A = 1 \wedge y.B = 2)) \\ &\quad \Leftrightarrow (\text{true} \ \textbf{except} \ y.B = 2). \end{aligned}$$

Intuitively a portion of the exception is preserved because the left-hand exception places no further restrictions on x.

More generally we may define transformation $\exists x : \alpha$ over the data structures of the previous sections as follows.

Definition 4 (Quantifier transformation) *For any pair $\rho = (e, \nu)$ compute the pair $\rho' = \exists x : \rho$ as follows: set $E^c_1 = \mathrm{R}^-_c(\rho)$; let E^c_2 be the result of removing from the conjunction E^c_1 all equalities involving the variable x; finally,[5] set $\rho' = \mathrm{R}_c(E^c_2)$. To extend the operation over disjunction sets define*

$$\exists x : \{\rho_1, \rho_1, \ldots, \rho_1\} = \textsc{wk}(\{\exists x : \rho_1, \exists x : \rho_2, \ldots \exists x : \rho_n\}$$

[5] The correctness of this method depends upon the way in which we defined R^-_c – alternative (correct) definitions for R^-_c require this definition to be restated.

To extend the existential transformation over exception lists define $\exists x : \Re$ *as follows:*

$$\begin{aligned} \exists x : [\,] &= [\,] \\ \exists x : [\delta] &= [\exists x : \delta] \end{aligned}$$

and for the inductive case suppose

$$\delta_1 \textbf{ except } \delta_2 \textbf{ except } \Re' = \textsc{Clean}(\Re)$$

and set

$$\exists x : \Re = \textsc{Clean}(\delta_1' \textbf{ except } \delta_2' \textbf{ except } \exists x : \Re')$$

where

$$\begin{aligned} \delta_1' &= \exists x : \delta_1 \\ \delta_2' &= \{\exists x : \rho_2 \mid \rho_2 \in \delta_2 \textit{ and for some } \rho_1 \in \delta_1 \\ &\qquad \textit{we have } \rho_2 = \rho_1 \wedge \exists x : \rho_2\} \end{aligned}$$

This function has the expected logical property:

Proposition 9 *If* $\Re$ *is an exception list then* $\mathrm{R}_*^-(\exists x : \Re) \Leftrightarrow \exists x : \mathrm{R}_*^-(\Re)$. *Furthermore* $\exists x : \Re$ *produces a canonical exception list.*

Thus the boolean interpreter defined above may be extended to interpret quantification as follows

$$\textsc{Interp}(\exists x : E) = \exists x : \textsc{Interp}(E)$$

where universal quantification is interpreted in terms of existential quantification in the usual manner.

9 Answering All Relational Calculus Queries

The extended interpretation function INTERP provides a canonical interpretation $\textsc{Interp}(E)$ for any relational calculus expression E, satisfying the correctness property:

Proposition 10 *For any expression* E, *supposing* $r_1, r_2, \ldots, r_n$ *are the relation names mentioned in* E, *we have*

$$\begin{aligned} &E \wedge \\ &(\forall x : (x \in r_1) \leftrightarrow \textsc{Ex}(x, \mathbf{f}(r_1))) \wedge \\ &(\forall x : (x \in r_2) \leftrightarrow \textsc{Ex}(x, \mathbf{f}(r_2))) \wedge \\ &\qquad \vdots \\ &(\forall x : (x \in r_n) \leftrightarrow \textsc{Ex}(x, \mathbf{f}(r_n))) \\ &\qquad\qquad \Leftrightarrow \mathrm{R}_*^-(\textsc{Interp}(E)) \wedge \\ &\qquad\qquad\quad (\forall x : (x \in r_1) \leftrightarrow \textsc{Ex}(x, \mathbf{f}(r_1))) \wedge \\ &\qquad\qquad\quad (\forall x : (x \in r_2) \leftrightarrow \textsc{Ex}(x, \mathbf{f}(r_2))) \wedge \\ &\qquad\qquad\qquad\quad \vdots \\ &\qquad\qquad\quad (\forall x : (x \in r_n) \leftrightarrow \textsc{Ex}(x, \mathbf{f}(r_n))) \end{aligned}$$

That is, the INTERP *function preserves the logical meaning of expressions when conjoined with the logical interpretation for the input relations.*

Here $A \leftrightarrow B$ stands for $(A \rightarrow B) \wedge (B \rightarrow A)$ as usual.

Furthermore the INTERP function has *polynomial data complexity* in the sense of Vardi [14].

Proposition 11 *For any fixed expression* E *the computation* $\textsc{Interp}(E)$ *completes in time polynomial in the size of the filter representations* $\mathbf{f}$ *for the input relations.*

This result follows essentially by inspecting each of the logical transformations for exception lists and concluding that they are each polynomial in the size of their inputs.

A filter representation for the answer to a relational calculus query $\{x \mid E\}$ may be calculated by translating $\mathrm{R}_*^-(\textsc{Interp}(E))$ into a filter representation that omits the variable x in a straightforward manner, by 'inverting' the EX transformation.

10 Additional Related Work

The notion of domain dependence was first introduced and studied by Fagin [5], and Nicolas and Delombe [11]. Theoretical studies by DiPaola [4] and Vardi [13] establish that the general question of whether a query is domain dependent cannot be decided. In view of the undecidability result it is important to note that the present proposed interpreter *does not decide domain dependence* – in particular, although queries that produce "non-standard" results are certainly domain dependent, queries which produce "standard" results may or may not be domain dependent.

The traditional approach for handling the problem of domain dependence is to identify a recognizable subset of domain independent queries – called "safe" queries – and reject all other queries. There are a number of proposed definitions for "safe" queries, a number of which are compared by Kifer [9]. The most recent of these is given by Van Gelder and Topor [12] together with analysis of other proposals and related issues. An alternative to the "safe query" methods is to redefine the query language to prevent domain dependent queries, such as the Pique proposal [10] which exploits a universal relation assumption to scope all variables.

The analysis of the application of powerdomain algebras to database problems, as set out by Buneman, Jung, and Ohori [3] motivated the algebraic methods described here.

11 Research Directions

There are two directions in which this research should be extended – towards generalizations of the method,

and towards applications of the method.

Some possible generalizations: The language of this paper omitted certain standard features of relational languages, such as arithmetic comparisons and calculations. Although such extensions complicate the procedure they do not present any insurmountable difficulties. More interesting extensions that should be investigated including the questions of supporting complex objects (or non-first normal form relations) and partial information representations in this framework.

Some possible applications: Existing optimizations can be characterized as semantically invariant rewrites within the relational calculus and this study may lead to the inspiration of new optimizations as well – optimizations that take advantage of the greater generality of the present paradigm. Beyond this, the "filter" paradigm may be directly useful in applications that are currently difficult to formulate within the standard relational model – just as numerical applications are sometimes difficult to formulate using real numbers, but are more easily addressed in the extended space of complex numbers.

Acknowledgements: Peter Buneman, Susan Davidson, Richard Hull, and Tomasc Imielinski provided many helpful discussions which aided in the development of these ideas. Special thanks to Dina Q. Goldin, of Brown University, who translated [2] into English, to Micheal Kifer and Richard Hull who diseminated her work.

References

[1] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 13(3):359–384, 1990.

[2] A. Ajiamazyan, Gilula M., A. Stolboushkin, and G. Schwartz. Reduction of the relational model with infinite domains to the case of finite domains. *Proc. of USSR Acad. of Science (Doklady)*, 286(2):308–311, 1986.

[3] P. Buneman, A. Jung, and A. Ohori. Using powerdomains to generalize relational databases. *Theoretical Computer Science*, 91:23–55, 1991.

[4] R. DiPaola. The recursive unsolvability of the decision problem of the class of definite formulas. *Journal of the ACM*, 16(2):324–324, 1969.

[5] R. Fagin. Horn clause queries and data base independence. *ACM Journal*, 29(4):952–985, 1982.

[6] J. Gallier. *Logic for Computer Science.* Harper and Row, 1986.

[7] R. Hull and Jianwen Su. Domain independence and the relational calculus. *Acta Informatica.* to appear.

[8] T. Imielinski and W. Lipski. The relational model of data and cylindrical algebra. *Journal of Computer and Systems Sciences*, pages 80–102, February 1984.

[9] M. Kifer. On safety, domain independence, and capturability of database queries. In *Knowledge and Database Conference Proceedings*, pages 405–415, 1988.

[10] D. Maier, D. Rozenshtein, S. Salveter, J. Stein, and D. S. Warren. Pique: A relational query language without relations. *Information Systems*, 12(3):317–335, 1991.

[11] J. Nicolas and R. Delombe. On the stability of relational queries. Technical report, ONERA-CERT, 1982.

[12] A. Van Gelder and R. Topor. Safety and translation of relational calculus queries. *ACM Transactions on Database Systems*, 16(2), 1991.

[13] M. Vardi. The decision problem for database dependencies. *Information Processing Letters*, 12(5):251–254, 1981.

[14] M. Vardi. The complexity of relational query languages. In *Proceedings of the 4th ACM Symposium on the Theory of Computing*, pages 137–146, 1982.