GEORGIA INSTITUTE OF TECHNOLOGY   OFFICE OF CONTRACT ADMINISTRATION

PROJECT ADMINISTRATION DATA SHEET

☒ ORIGINAL □ REVISION NO. ______

Project No. G-36-602 GTRI/~~GIT~~ DATE 2/16/83

Project Director: Marc H. Graham School/~~Lab~~ ICS

Sponsor: National Science Foundation
Washington, DC 20550

Type Agreement: Grant No. IST-8217441

Award Period: From 6/1/83 To 11/30/85 (Performance) 2/28/86 (Reports)

Sponsor Amount: Total Estimated: $100,993 Funded: $ 100,993

Cost Sharing Amount: $ 5,315 Cost Sharing No: G-36-338

Title: "Canonical Queries as a Query Answering Device (Information Science)"

ADMINISTRATIVE DATA OCA Contact John W. Burdette x4820

1) Sponsor Technical Contact:

Michael J. McGill
Information Science & Technology
Room 236
National Science Foundation
Phone: (202) 357-9554

2) Sponsor Admin/Contractual Matters:

Idele Kruithoff
National Science Foundation
Washington, DC 20550
Phone: (202) 357-9653

Defense Priority Rating: NA Military Security Classification: NA
(or) Company/Industrial Proprietary: NA

RESTRICTIONS

See Attached NSF Supplemental Information Sheet for Additional Requirements.

Travel: Foreign travel must have prior approval – Contact OCA in each case. Domestic travel requires sponsor approval where total will exceed greater of $500 or 125% of approved proposal budget category.

Equipment: Title vests with GIT

COMMENTS:

*Includes a 6 month unfunded flexibility period.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
FEB 1983
RECEIVED
Research Reports
Office

COPIES TO:

Research Administrative Network
Research Property Management
Accounting

~~Research Security Services~~
Reports Coordinator (OCA)
GTRI
Library

Research Communications (2)
Project File
Other PD
Other

GEORGIA INSTITUTE OF TECHNOLOGY **OFFICE OF CONTRACT [illegible]**

SPONSORED PROJECT TERMINATION/CLOSEOUT SHE[illegible]

[illegible]

Date 10-[illegible]

Project No. G-36-602 School [illegible]

Includes Subproject No.(s) N/A

Project Director(s) Marc H. Graham [illegible]

Sponsor National Science Foundation

Title "Canonical Queries as a Query Answering Device (Information Science)"

Effective Completion Date: 11-30-85 (Performance [illegible])

Grant/Contract Closeout Actions Remaining:

- □ None
- □ Final Invoice or Final Fiscal Report
- □ Closing Documents
- ☒ Final Report of Inventions Patent questionnaire sent to P.I.
- □ Govt. Property Inventory & Related Certificate
- □ Classified Material Certificate
- □ Other ______

Continues Project No. ______ Continued by Project No. ______

COPIES TO:

Project Director
Research Administrative Network
Research Property Management
Accounting
Procurement/GTRI Supply Services
Research Security Services
[redacted]
Legal Services

Library
GTRC
Research Communications (2)
Project File
Other ______

G-36-602

Annual Report 1984
NSF-IST-82-17441
Canonical Queries
Marc H. Graham, PI

Georgia Tech ICS graduate students Ke Wang, Steve Ornburn, Gita Rangarajan, and Amy Lapwing were supported under this grant for different periods.

The paper "The Power of Canonical Queries" was written with Alberto Mendelzon. The paper "On the Complexity and Axiomatizability of Consistent Database States," with Moshe Vardi appeared in Principles of Database Systems (PODS) March 84. These papers deal with the expressive power of Canonical Queries.

The Power of Canonical Queries

by

Marc H. Graham*†

and

Alberto O. Mendelzon**††

September 1983

*School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

**CSRG
University of Toronto
Toronto, Ontario
Canada

†Research supported in part by the National Science Foundation grant IST 8217441.

††Research supported in part by the Natural Science and Engineering Research Council of Canada.

This document was prepared with a Xerox Star 8010 system.

1. Introduction

The approach to formal data semantics which has come to be called "weak instance theory" began with the work of Honyeman [H] as a means of integrating the relations of a multirelation database for the purpose of checking constraints. It was soon recognized, by Sagiv [S] and also by Yannakakis [Y], that the theory leads naturally to an extremely powerful and concise query language, called window functions by some authors [MRW] [MUV] and called canonical queries here. This paper analyzes the expressive power of these queries. We show that canonical queries are expressible in first order logic if we allow infinitely many axioms. These axioms are shown to be full multirelational implicational dependencies as defined by Fagin [F]. The bulk of the paper is concerned with characterizing the case when canonical queries may be finitely axiomatized in first order logic. This occurs precisely when the query is expressible in the relational algebra.

Maier, Ullman and Vardi [MUV] have considered these questions as well. The present work was done independently and differs from their work in the following ways.

Proposition 1 establishes the existence of representative instances even when the chase is not guaranteed to terminate. We consider the result of a non-terminating chase to be undefined. In the same spirit, compactness is not used in the proof of Lemma 1. Proposition $\overset{6}{7}$. although a simple observation, does not appear in [MUV]. ?

A new definition of boundedness appears in Theorems 1 and 2. In these theorems we take up a problem ignored in [MUV]: the behaviour of canonical queries on inconsistent states. We do this for a technical reason: a relational algebra expression is defined for every state, canonical queries only for consistent ones. Theorems 1 and 2 present two different attacks on this problem. In Theorem 1, we remove all equality generating dependencies in the manner suggested by Beeri and Vardi [BV]. All states are then consistent and the results of canonical queries on the subset of originally consistent states are preserved. In Theorem 2, we consider queries which distinguish consistent and inconsistent states.

The proof of the "hard part" of these results (the implication $5 \Rightarrow 4$ in Theorem 1) is made very easy by the presence of Lemma ~~2~~3. This lemma establishes a well known piece of folklore as fact.

2. Definition and Notation

2.1 Basic Definitions

We begin with a finite set of attributes which we denote U and call the universe. Following standard notation in the area, upper case letters near the front of the alphabet: A, B, A_i, A_j, ... indicate single attributes. Those toward the end of the alphabet: X, Y, ... represent sets of attributes. The set $\{A\}$ is often denoted A and $X \cup Y$ is written XY. To each $A \in U$ is associated an infinite domain $dom(A)$. distinct A_i, A_j in U, $dom(A_i)$, $dom(A_j)$ are either identical or disjoint.

Let $R \subseteq U$. R is then a relation scheme. A tuple for R is a function assigning to each $A \in R$ a value in $dom(A)$. The term row will be freely used for tuple in certain contexts. If $X \subseteq R$, $t[X]$ denotes the restriction of the function t to the attributes in X.

A relation instance I for R is a set of tuples for R. The size of an instance I, denoted $|I|$, is the number of tuples it contains. We often restrict our attention to finite relations. If $X \subseteq R$, the projection onto X of an instance I of R, denoted

$\Pi_X(I) = \{t[X] \mid t \in I\}$. If $\mathbf{R}$ is a collection of relation schemes of U, then $\mathbf{R}$ is a <u>database schema for</u> U. We do not in general insist that $\mathbf{R}$ cover U, *i.e.*, that $\cup\mathbf{R} = U$.

A <u>state</u> ρ of a schema $\mathbf{R}$ is an assignment to each $R \in \mathbf{R}$ of an instance for R. The size of a state is the sum of the sizes of its instances:

$$|\rho| = \sum_{R \in \mathbf{R}} |\rho(R)|$$

We define an inclusion relation among states in a natural way. If ρ, σ are states of $\mathbf{R}$, $\rho \subseteq \sigma$ if $\rho(R) \subseteq \sigma(R)$ for every $R \in \mathbf{R}$. If I is an instance and $\mathbf{R}$ a schema for U, we define $\Pi_{\mathbf{R}}(I)$ to be the state ρ of $\mathbf{R}$ given by $\rho(R) = \Pi_R(I)$ for each $R \in \mathbf{R}$.

A <u>tableau</u> for a universe U is an instance for U over an extended set of domains. For each $A \in U$, we form the tableau domain *tdom(A)* from *dom(A)* by adding infinitely many variables. Elements of $\cup_{A \in U}$ *dom(A)* are called constants. If *dom(A$_i$)* = *dom(A$_j$)*, then *tdom(A$_i$)* = *tdom(A$_j$)*, else the sets are disjoint. From now on the term instance will denote a tableau without variables. A <u>tagged</u> <u>tableau</u> for U is a tableau over the set of attributes $U \cup \{Tag\}$ where *Tag* is an attribute assumed not to be in U. Further, *tdom(Tag)* is disjoint from any attribute domain in U. The values in *tdom(Tag)* will always be given a special interpretation, namely as relation schemes.

Tableaux provide a uniform notation for the expression of data dependencies, conjunctive queries and database states.

A <u>(unirelational)</u> <u>data</u> <u>dependency</u> for a universe U is a pair $d = \langle T, x \rangle$ where T is a tableau over U and x is one of the following if x is an equality assertion of the form $a=b$, then d is an equality generating dependency or egd. (It is conventional to assume the symbols a, b appear in the tableau T.)

if x is a tableau then d is a tuple generating dependency, or tgd. If every symbol appearing in the tableau x also appears in the tableau T, then d is a total or full dependency. In this case x may be assumed to contain (or abusively, to be) a single tuple. Otherwise, if x contains symbols not in T, then d is partial or embedded.

This definition of dependency is based on the work of Beeri and Vardi [BV]. The parallel to the implicational dependencies of Fagin [F] is immediate. The quantifier "unirelational" in the above indicates that these dependencies can be written in a first order language with a single predicate letter of arity the cardinality of U. The class of multi-relational dependencies can be captured through the use of tagged tableaux, as follows.

A multi-relational data dependency for a database schema **R** over a universe U, $d = <T,x>$ is a data dependency for U in which T, and x if d is a tgd, are tagged tableaux. Two extra conditions are imposed: (i) the tags of T and x are relation schemes of **R**: $\Pi_{Tag}(T \cup x) \subseteq \mathbf{R}$; (ii) tuples may agree only as they are allowed to by their tags: $t[A]=u[B]$ implies $A \in t[Tag]$ and $B \in u[Tag]$. (Of course, $t[A]=u[B]$ is possible only if $tdom(A)=tdom(B)$.) Finally, a multi-relational tgd $d=<T,S>$, is considered full if for each $s \in S$ and each $A \in s[Tag]$, $s[A]$ appears in T.

Let ρ be a state of a schema **R** over universe U. The tableau of ρ T_ρ, is a tagged tableau over U defined as follows (this definition gives T_ρ only up to isomorphic renaming of variables): For each $R \in \mathbf{R}$, each $t \in \rho(R)$, a row v of T_ρ has $v[R]=t$. For each $A \in U\text{-}R$, $v[A]$ is a variable appearing nowhere else in T_ρ. Finally, $v[Tag]=R$ and no other rows appear in T_ρ.

A <u>query</u> on a schema $\mathbf{R}$ with target list X, is a function from states of $\mathbf{R}$ to instances of X. A <u>conjunctive query</u> q on a schema $\mathbf{R}$ with target list X is a full multi-relational tgd on schema $\mathbf{R} \cup \{X\}$. If $q = <T,x>$, then T is a tagged tableau on $\mathbf{R}$, x is a single tuple and $x[Tag]=X$. To define the function described by a conjunctive query, we introduce the idea of a homomorphism.

Define $Sym(T) = \cup_{A \in U}(\Pi_A(T))$ where T is a tableau on universe U. A <u>homomorphism</u> on T is any function with domain $Sym(T)$. If η is a homomorphism on T, then we allow η to also represent its extension [illegible] and tableaux. That is, $\eta(t) = \eta \circ t$ (the composition of η and t), [illegible] $t \in T\}$. A homomorphism *preserves* a set of symbols C if it is the identity on C. A constant preserving homomorphism preserves the set of constants (recall this is the set $\cup_{A \in U} dom(A)$). A <u>tag preserving</u> homomorphism is a homomorphism extended by the identity on $tdom(Tag)$.

The relation between tableaux which is central to this paper is that of <u>homomorphic embeddability.</u> If T, S are tableaux on a universe U, then T is homomorphically embeddable into S if there exists a homomorphism η on T such that $\eta(T) \subseteq S$. T and S are homomorphically equivalent if each may be embedded into the other. For certain applications, we may require the homomorphism to preserve some set of symbols. If either of T or S or both are tagged and η is non-tag preserving we may write $\eta(T) \subseteq S$ to mean $(\eta(\Pi_U(T)) \subseteq (\Pi_U(S))$. In some circumstances the set of all homomorphisms embedding T into S is of interest, as in the following definition.

A tableau T_0 satisfies a dependency $d = <T,x>$ if for every homomorphism η embedding T into T_0

if x is the equality assertion $a=b$, then $\eta(a)=\eta(b)$;

if x is the tableau S, then η can be extended to a homomorphism μ on $Sym(S \cup T)$ (i.e. μ restricted to $Sym(T)$ is η) with $\mu(S) \subseteq T_0$.

(For multirelational dependencies we may consider only tag preserving homomorphisms.) A tableau satisfies a set of dependencies D if it satisfies each dependency in D.

We can now describe conjunctive queries as functions. Let $q = <T,\{t\}>$ be a conjunctive query for a schema **R** and ρ a state of **R**. The relation $q(\rho) = \{\eta(t) \mid \eta$ *a tag preserving homomorphism embedding* T *into* $T_\rho\}$. It is customary to further restrict the homomorphisms to be constant preserving. When that is done, the class of conjunctive queries includes all queries expressible by relational algebra expressions using a restricted form of selection, projection and product [CM]. Union may be modeled by considering finite sets of conjunctive queries [SY].

The chase [ABU], [MMS] is a fundamental process in the study of databases. It is a means of transforming, if possible, an arbitrary tableau into one which satisfies a given set of dependencies. Let $d = <T,x>$ be a dependency and η a homomorphism on T. The pair $\iota = <d,\eta>$ is called a transformation. If S is a tableau and η embeds T into S, then ι is said to be enabled. The application of an enabled transformation ι to a tableau S, denoted $\iota(S)$ is a tableau whose definition depends on the nature of the dependency d.

If d is an egd, so that x is $a=b$, then one of the symbols $\eta(a)$, $\eta(b)$ replaces the other everywhere it appears in S. It is customary to give a disambiguating rule for the choice of the replacement. When $\eta(a)$, $\eta(b)$ are distinct constants, ι is a contradiction and it is usual to assign $\iota(S)=\varnothing$.

If d is a tgd, so that $d=<T,V>$, then η is extended to a homomorphism μ on $T \cup V$ and $\iota(S)=S \cup \mu(V)$. The extension of η to μ is restricted so that μ is one-to-one on $Sym(V)-Sym(T)$ and for each $y \in Sym(V)-Sym(T)$, $\mu(y) \notin Sym(S)$, that is, $\mu(y)$ is a new variable.

It is customary to denote $chase_D(T)$ as the limit of the process of applying transformations whose dependencies are chosen from the set D, starting with the tableau T. If D contains only full dependencies and a disambiguating rule is given for the application of egds (see above), $chase_D(T)$ is unique and effectively computable. Otherwise, it is at best defined only upto isomorphism and whenever D contains partial dependencies, this limit is not clearly defined.

2.2 Consistency, weak instances, canonical queries

Let ρ be a state of a schema **R** over a universe U. Let D be a set of dependencies over U. Following Honeyman [H], see also [GMV], we define a weak instance for ρ with respect to D as an instance I of U such that $\rho \subseteq \Pi_{\mathbf{R}}(I)$ and I satisfies D. We denote the set of all such finite weak instances as $weak(D,\rho)$ and we say ρ is consistent with D if $weak(D,\rho) \neq \varnothing$. We denote the set of all states of **R** consistent with D as $CONS(\mathbf{R},D)$.

Let $Sym(\rho)=\cup_{R \in \mathbf{R}}(\cup_{A \in R}\Pi_A(\rho(R)))$ be the set of all symbols appearing in the state ρ. A representative instance for ρ with respect to a set of dependencies D is a uppercase

italics

possibly infinite weak instance for ρ such that every element of weak(D,ρ) is the image, under some Sym(ρ) preserving homomorphism, of the representative instance. We can show that every consistent state has a representative instance.

Proposition 1. If $\rho \in CONS(\mathbf{R},D)$, then ρ has a representative instance. ~~Further,~~ all ~~representative instances for~~ ρ are equivalent ~~via~~ ~~*Sym(ρ)*~~ preserving ~~ho~~momorphisms.

Proof. ~~Let I, J be elements of weak(D,ρ)~~. We take the direct product of the elements of *weak*(*i*,ρ). This is an instance over the universe *U* for which the attribute domains, to be denoted *xdom(A)*, are sequences of countable length. It is convenient and customary to consider these sequences as functions on the set of natural numbers, N. So for the instance we have for each *A*,

uppercase

as

xdom(A)$=\{f | f: \mathbf{N} \rightarrow dom(A)\}$. However, we may identify in *xdom(A)*, that function *f* such that $f(i) = a$ for each $i \in \mathbf{N}$ with the element $a \in dom(A)$. This allows us to consider *xdom(A)* as an extension of *dom(A)*.

Let I be this direct product. Its definition requires that we number the elements of *weak*(D,ρ). Having done so, we have by definition

$$\mathbf{I} = \{ <f_1, \ldots f_m> \mid <f_1(i), \ldots f_m(i)> \in I_i \in weak(D,\rho) \text{ for every } i \in \mathbf{N} \}$$

It is well known that dependencies are preserved under direct products, that is, I satisfies *D*.

Further, for $I_i \in weak(D,\rho)$ each, the natural map

$$\eta_i : < f_1, f_2 \ldots f_m > \vdash <f_1(i), \ldots f_m(i)>$$

replace $\eta_i(f) = f(i)$

italics

uppercase

homomorphically embeds I onto I_i. η_i is *Sym(ρ)* preserving since $f=a$ iff $f(i)=a$ for every i: therefore $\eta_i(f)=a$. It remains to show $\rho \subseteq \pi_R(\mathbf{I})$.

italics

Uppercase

for any $R \in \mathbf{R}$, let $u \in \rho(R)$. Each $I_i \in weak(D,\rho)$ contains a tuple with $u_i[R] = u$. Therefore I contains a tuple u such that $u[R] = u$. ⊣ remove

This proposition is stronger than the results of Honeyman [H], Sagiv [S], Mendelzon [Me] and Maier, Ullman and Vardi [MUV] in that it does not depend on the chase. As noted by those authors, $chase_D(T_\rho)$, if it exists, is ~~a representative~~ homomorphically embeddable into every element of weak(D, ρ) . ~~instance for ρ~~.

Let $X \subseteq U$. The canonical query on X with respect to schema $\mathbf{R}$ and set of dependencies D, denoted $?X[\mathbf{R},D]$ or just $?X$ when $\mathbf{R}$ and D are known from context, is a function from $CONS(\mathbf{R},D)$ to instances of X defined by

$$?X[\mathbf{R},D](\rho) = \bigcap_{I \in weak(D,\rho)} (\Pi_X(I))$$

The definition does not provide an effective computation of $?X[\mathbf{R},D]$ and indeed such a computation may not exist. However, the representative instance can be used to compute $?X[\mathbf{R},D]$ when it can itself be effectively found. Define C- projection Π^C as projection with respect to elements of C only:

$$\Pi^C_X(T) = \{t \mid t \in \Pi_X(T) \text{ and } t[A] \in C \text{ for each } A \in X\}$$

Proposition 2. :: $CONS(\mathbf{R},D)$ and I is a representative instance for ρ, then

$$?X(\rho) = \Pi_X^{Sym(\rho)}(\mathbf{I}) \quad ? \quad \in$$

Proof. We have $x \in ?\overset{\text{upper case}}{X}'(\rho)$ iff for each $I \in weak(D,\rho)$, ~~there~~ exists $t \in I$ with $t[X] = x$ iff there exists $t \in$ I with $\eta(t) = t$ where is a copy $Sym(\rho)$ preserving ~~homorphisim~~ remove homorphisim and $\eta(I) = I$. It is easy to see that for $A \in \overset{\text{upper case}}{X}$ $x[A] \in Sym(\rho)$. This comes from the fact that for each $I \in weak(D,\rho)$ there is a $J \in weak(D,\rho)$ with $Sym(I) \cap Sym(J) = \overset{-Sym(\rho)}{\emptyset}$. (indeed J, can be formed by isomorphically renaming each element of $Sym(I) - Sym(\rho)$ by an element not in $Sym(I)$.) therefore $x \in ?X(\rho)$ iff $x \in \Pi^{Sym(\rho)}(\mathbf{I})$. ⊣ Uppercase

Proposition 2 replaces an intersection of infinitely many projections with a single projection of an infinite relation. This brings us no closer to an effective computation. We now show, as stated earlier, that no such effective computation exists.

Proposition 3.

1) There exist X, **R**, D such that $?X/\mathbf{R},D/$ is not effectively computable.

2) There exists no uniform, effective procedure for determining if $?X/\mathbf{R},D/$ is computable for arbitrary X, **R**, D.

Proof. (1) The <u>completeness problem</u> is determining for all triples $\{<\rho,\mathbf{R},D>\}$, whether $?R(\rho)=\rho(R)$ for every $R \in \mathbf{R}$. The completeness problem is shown to be undecidable in [GMV] where (2) the set $\{<\mathbf{R}.D>$ / *completeness of states of* **R** *with respect to D is decidable*$\}$ is shown to be not recursive. ⊣

upper case Tρ

In contrast to proposition 3, in the case that $chase_D(\tau\rho)$ is effectively computable, so is $?X(\rho)$.

Proposition 4. If $chase_D(\tau\rho)$ exists, then

$$?X(\rho) = \pi^{O\,[illegible]}(chase_D(\tau\rho)) \quad .$$

Proof. In this case $chase_D(\tau\rho)$ is embeddable via $Sym(\rho)$ preserving homomorphisim <u>into</u> (not onto) every element of $weak(D.\rho)$ [GMV]. Therefore $\pi^{O\,[illegible]}(chase_D(\tau\rho)) \subseteq ?X(\rho)$. On the other hand, it is only a small abuse of notation to state $chase_D(\tau\rho) \in weak(D.\rho)$. So $\pi^{O\,[illegible]}(chase_D(\tau\rho)) \supseteq ?X(\rho)$. ⊣

A query E is said to be <u>monotonic</u> if $\rho \supseteq \sigma$ implies $E(\rho) \supseteq E(\sigma)$.

Proposition 5. Canonical queries are monotonic.

Proof. The inclusion $\rho \supseteq \sigma$ implies $weak(D.\rho) \subseteq weak(D.\sigma)$. The proposition follows. ⊣

We allow only finite states. Suppose however we were to allow states of arbitrary size. It would still be possible to define weak instances for these states and therefore cannónical queries as well. Proposition 5 remains true in this case without modification to its proof. Proposition 5 therefore establishes that canonical queries are monotonic everywhere, not merely over states of finite size. This is crucial to the development of section 4, below.

On the other hand, not all expressions of the relational algebra define monotonic queries. We will say that a query E on schema **R** is canonical if there is some set of dependencies D such that $E = ?X/\mathbf{R},D/$. We know then that not all queries expressed in relational algebra are canonical. T he reverse inclusion is also not true.

It is well known that no expression of the relational algebra is equivalent to the transitive closure of a binary relation. [AU], [Im], [ZI]. Let R be a binary relation symbol and let d be the dependency which expresses the transitivity of R:

$$\forall xyx(Rxy \wedge Ryz \Rightarrow Rxz)$$

Then $?R[\{R\}.\{d\}]$ is the transitive closure function, since, directly from the
remove
definition, $?R[\{R\}.\{d\}]I$ is the smallest relation containing I which is transitive. We state these facts as a proposition.

Proposition 6. The set of canonical queries is incomparable to the set of queries which may be expressed in the relational algebra. ⊣

3. The Logic of canonical queries.

In this section we present canonical queries in a logical framework. We do this to make more apparent the closeness of our approach to the approach of artificial

Lower case : e

intelligence which treats querying as logical inference. [GME] We also do it to prepare ourselves for the results of the next section.

Let U be a universe. It is necessary to fix an ordering on the elements of U. If $\mathbf{R}$ is a schema over U, the first order language (with equality) $L_{\mathbf{R}}$ has neither function nor constant symbols. T he predicates of $L_{\mathbf{R}}$ are the schemes of $\mathbf{R}$. Thus if $R \in \mathbf{R}$ is the set $\{A_{i_1}, \ldots A_{i_m}\}$, then $L_{\mathbf{R}}$ has an m-ary predicate symbol R. Let S be a relation scheme. For notational ease we will denote the language associated with a schema $\mathbf{R} \cup \{S\}$ as $L_{\mathbf{R},S}$ rather than $L_{\mathbf{R} \cup \{S\}}$. However, we always assume a <u>new</u> predicate symbol, that is, a symbol not in $L_{\mathbf{R}}$, appears for S in $L_{\mathbf{R},S}$, even when $S \in \mathbf{R}$.

Let $X \subseteq U$ and D be a set of unirelational dependencies on U. Consider the following set of sentences Σ in the language $L_{\mathbf{R},X,U}$

(containing instance)

(dependencies)

Projection axiom: $\forall c_1 \ldots c_n (U(c_1, \ldots, c_n) \Rightarrow X(c_{i_1} \ldots c_{i_m}))$

where $X = A_{i_1} \ldots A_{i_m}$

The finite models of Σ, denoted $struc(\Sigma)$, can be written as triples $<\rho, I, \xi>$ where $\rho \in CONS(\mathbf{R}, D)$, $I \in weak(D, \rho)$ and $\xi \supseteq [X/\mathbf{R}, D](\rho)$. Let C be formed by reducing $struc(\Sigma)$ to $L_{\mathbf{R},X}$; that is, $C = \{<\rho, \xi> \mid \rho \in CONS(\mathbf{R}, D) \text{ and } \xi \supseteq [X/\mathbf{R}, D](\rho)\}$. Let $D_{\mathbf{R},X}$ be the collection of consequences of Σ in the language $L_{\mathbf{R},X}$; that is, the elements of $D_{\mathbf{R},X}$ are sentences in $L_{\mathbf{R},X}$ which hold in every element of $struc(\Sigma)$. Clearly, the members of C satisfy the sentences of $D_{\mathbf{R},X}$; that is, $C \subseteq struc(D_{\mathbf{R},X})$. We now demonstrate the reverse inclusion.

Lemma 1. $C = struc(D_{\mathbf{R},X})$

Proof. We need only show $struc(D_{\mathbf{R},X}) \subseteq C$, by preceeding remarks. So let $<\rho,\xi> \notin C$. Let **dom** be the set of all values appearing in ρ and expand the language $L_{\mathbf{R},X,U}$ by adding each element of **dom** as a constant. In the expanded language, let

$$D_1 = \{R(a_1,\ldots,a_n) \mid \textit{for each } R \in \mathbf{R} \textit{ where } <a_1,\ldots,a_n> \in \rho(R)\}$$

$$D_2 = \{\neg X(a_1,\ldots,a_m) \mid <a_1,\ldots,a_m> \notin \xi \textit{ and } a_i \in \mathbf{dom}\}$$

$$D_3 = \{a \neq b \mid \textit{for each pair of distinct elements of } \mathbf{dom}\}.$$

Now $\Sigma' = \Sigma \cup D_1 \cup D_2 \cup D_3$ is an inconsistent set of sentences. Suppose otherwise. If M is a structure for Σ', then $M(U)$, the interpretation of U in M, is a weak instance for ρ with respect to D so $\rho \in CONS(\mathbf{R},D)$ [GMV]. I t must be therefore that $\xi \not\supseteq ?X(\rho)$, as $<\rho,\xi> \notin C$. So there is a tuple $x \in ?X(\rho) - \xi$. Now $x \in ?X(\rho)$ implies $x \in \Pi_X(M(U))$ so by the projection axiom, $x \in M(X)$. But $\neg X(x) \in \Sigma'$ so M is not a model of Σ'.

We note that $D_1 \cup D_2 \cup D_3$ is a finite set of sentences in the expanded language. Therefore, the conjunction of its elements, denoted d, is a quantifier free sentence of the language $L_{\mathbf{R},X}$ augmented with the set **dom** of constants. Furthermore, $<\rho,\xi>$ satisfies d. On the other hand, from the inconsistency of Σ' we may conclude, $\Sigma \vdash \neg d$. Noting that Σ is constant free, we may conclude $\Sigma \vdash (\forall \mathbf{x})(\neg d)$ where $\mathbf{x}$ is the vector of all elements of **dom** appearing in d, interpreted as variables. In short, $(\forall \mathbf{x})(\neg d)$ is an element of $D_{\mathbf{R},X}$, so $<\rho,\xi> \notin struc(D_{\mathbf{R},X})$. We may therefore conclude $struc(D_{\mathbf{R},X}) \subseteq C$. ⊣

Corollary 1. $D_{\mathbf{R},X}$ is equivalent to a set of total, multirelational equality and tuple generating dependencies.

Proof. Consider the formula $\neg d$ in the proof of the lemma. This may be rewritten as $(d_1 \rightarrow d_2 \vee d_3)$ where d_1 is ~~the~~ a conjunction of elements of D_1; d_2, d_3 ~~the~~ are disjunctions of the elements of D_2, D_3 respectively, these latter appearing in in positive (*i.e.*, unnegated) form. By a result due to McKinsey [McK], extended by Graham and Vardi [GV], since Σ contains only dependencies, for some atomic formula e of $D_2 \cup D_3$, we must have $\Sigma \vdash (\forall x)(d_1 \rightarrow e)$. ⊣

In light of this corollary, we will write $D_{R,X}$ as $E_R \cup T_{R,X}$ where E_R is the set of egd's and $T_{R,X}$ is the set of tgd's mentioned above. It is known [GV] that the set $struc(E_R)$ of finite models of E_R is exactly $CONS(R,D)$. It is natural to consider the set $struc(T_{R,X})$; that is, it is natural to consider canonical queries on states not required to be consistent. We can do this by removing all egd's from D and replacing them with "nearly equivalent" tgd's as follows.

Let $d = <T.a = b>$ be an egd. Let $A_1, \ldots, A_n$ be the attributes of u such that $\{a,b\} \subseteq dom(A_i)$. ~~For i=0 to n,~~ For each such i let w_{i_a}, w_{i_b} be a pair of tuples on the universe U, satisfying $w_{i_a}[A_i] = a$, $w_{i_b}[A_i] = b$, $w_{i_a}[B] = w_{i_b}[B]$ for all $B \neq A_i$, and $Sym(\{w_{i_a}, w_{i_b}\}) \cap Sym(T) = \{a,b\}$. The tgd translation of d is the set

$$\bigcup_{i=1} \{ \quad w_{i_a}\}, w_{i_b}>, <T \cup \{w_{i_b}\}, w_{i_a}> \}$$

The egd free version of a set of dependencies D, denoted D^{ef} is formed by replacing each egd in D with its tgd translation. Let E_R^{ef} and $T_{R,X}^{ef}$ be the set of egd's and tgd's respectively which make up $D_{R,X}^{ef}$ as in corollary 1.

Lemma 2

1) $E_R^{ef} = \emptyset$

2) $T_{R,X}^{ef} = T_{R,X}$

Proof. 1) Immediate. 2) [BV???] ⊣

Combining this result with lemma 1 we have,

Corollary 2. 1) $struc(T_{R,X}) = \{<\rho,\xi> / \xi \supseteq ?X[R,D^{ef}](\rho)\}$

2) If $\rho \in CONS(R,D)$, then $?X[R,D](\rho) = ?X[R,D^{ef}](\rho)$ ⊣

This result states the canonical queries defined with respect to D^{ef} are identical to the queries defined with respect to D when the former are restricted to $CONS(R,D)$. It is useful to state the following result, whose proof is immediate from corollary 2.

Corollary 3. $?X[R,D](\rho) = \cap\{\xi / <\rho,\xi> \in struc(T_{R,X})\}$ ⊣

These lemmas and their cor~~a~~ollaries can be viewed in the following way. They state that calculation of a canonical query is in ~~the~~ essence the derivation of a tuple generating dependency. The elements of $T_{R',x}$ are multirelation dependencies. They can be seen as uni ~~non~~relational dependencies by the simple expedient of ignoring the tag attribute in their tableaux. This transforms an element of $T_{R',x}$ into an embedded dependency in the language u which is easily seen to be logical consequence of D. Recalling a result of Beeri and Vardi's[BV?], we have that for each tuple $X \in ?X[R,D](\rho)$, there exists a chase sequence of finite length (possibly 0) which adds a row u to T_ρ with $u[X]$

This is replaced

replace this

uppercase

uppercase

The reader may wonder whether the set of finite structures $\{<\rho,\xi> / \xi = ?X[R,D](\rho)\}$ is first order axiomatizable. We have shown $T_{R,X}$ to be a first order axiomatization of structures containing "all the truth." Is it possible to axiomatize those structures containing "only the truth?" Interestingly, this question can be answered either way, depending on how it is phrased.

We have restricted ourselves to the consideration of finite structures only as models. Suppose that f is any function from database states to instances of the scheme X and consider the pair $<\rho, f(\rho)>$. As both ρ and $f(\rho)$ are finite, this pair may be described by a single sentence of the form "if the state is exactly ρ, then the instance of X is exactly $f(\rho)$." The set $\{<\rho, f(\rho)>\}$ is exactly the set of finite models of the (infinite) set of sentences so constructed. This procedure is hardly effective nor very informative. Furthermore, these sentences are not dependencies. The fact that $T_{\mathbf{R},X}$ contains only tuple generating dependencies of a particular form is vital to the results of the next section.

If we consider the collection of all models of a given set of first order sentences, we discover that it is impossible in general to axiomatize the exact answers to canonical queries. The transitive closure of a binary relation may serve us again as a counterexample. Let R, R^+ be binary relation symbols. Consider both as giving two different edge relations on the same set of nodes. For each k, it is possible to write "for no pair for which an arc appears in R^+, is there a path of length k between them in R." Each finite subset of the set of all such sentences is consistent with an axiomatization of the transitive closure of nonempty relations, should such an axiomatization exist. But by the principle of compactness, which applies here as the full collection of models is considered, no such axiomatization can exist. Every arc in R^+ must correspond to a path in [illegible] of some finite length.

This discussion justifies a belief that the result of corollary 3 is as close as one can get to canonical queries with first order sentences.

4. Algebraic canonical queries.

In this section we consider those queries which are both canonical and expressible in the relational algebra. (Queries expressible in the relational algebra will hereinafter be called algebraic.) We will rely on the well known equivalence of the relational algebra and relational calculus. We restrict the class of expressions we will allow in two ways. We consider these restrictions to be matters of convenience.

First, we do not allow constants. Dependencies are written in a constant-free language, as in the prior section. Allowing constants in our expression language merely confuses matters. Secondly, we do not allow equality. This is in conformity with the work of Chandra and Merlin [CM]. We adopt this restriction in this section (we abandon it in the next) as we are considering here the canonical queries defined by an egd-free set of dependencies. Every state is consistent with such a set of dependencies and thus each canonical query is defined on every state. This simplifies our discussion. When D is egd-free, the set of sentences Σ of section 3 is written in a language without equality. Thus our prohibition of equality is similar to our prohibition of constants.

Formulae of the relational calculus are customarily interpreted only in finite states. As they are also formulae of first order logic it will be convenient to interpret them over states of arbitrary size.

Lemma 3. Suppose E is ~~a N~~ an monotonic q· ·ry everywhere expressible in a relational calculus without equality. Th·· E is expressible as a union of conjunctive queries. ⊣

insert space Italics

Proof Suppose E may be expressed as $\psi(x_1,\ldots x_k)$ using 'domain' calculus notation. We show is (equivalent to) a positive, existential formula; i.e., it is constructed from atomic~~s~~ formulae using $\exists, \wedge, \vee$ as the only connectives. But these formulae are exactly unions of conjuntive queries.

We show first the existential part. Suppose ρ is a state and $\rho \models \psi(a_1,\ldots a_k)$; that is, the tuple $(a_1,\ldots a_k) \in E(\rho)$. Suppose we have a state σ related to ρ in the following way: for each scheme R, $\rho(R)$ is the intersection of $\sigma(R)$ with the appropriate cross product of a fixed set of ~~constraints~~ constants (fixed in the sense that the same set of constants is used to form each relation of ρ .) In this case, ρ is called a submodel of σ, σ an extension of ρ. As $\sigma \supseteq \rho$, we have remove $(a_1,\ldots,a_k) \in E(\sigma)$. Generalizing this arguement,. we see that the sentence ~~in~~ a language expanded with constants, remove $\psi(a_1,\ldots,a_k)$ is preserved under extensions; that is, if true in ~~any~~ a state it is true in all extensions of that state. Therefore ψ is existential by the dual of theorem 3.2.2 of uppercase Chang & Keisler[ck].

To show the positive part, we will show that everywhere monotonic queries are preserved under homomorphisim; that is, if $(a_1,\ldots,a_k) \in E(\rho)$ and ρ is homomorphically embeddable in a state σ via a homomorphisim h, then $<h(a_1),\ldots,h(a_k)> \in E(\sigma)$. But then the formula ψ expressing E is positive by theorem 3.2.4 of the above cited text.

We define a strong homomorphisim as one which preseves negative as well as positive atomic formulae; that is, if $<b\ldots b_n> \notin \rho(R)$ then $<h(b_1),\ldots,h(b_k)> \notin \sigma(R)$, where h takes ρ to σ. We leave to the reader the task of showing that any existential sentece is ~~preveived~~ preserved under strong homomorphisims. [cf,Enderton p91 f].

Now suppose h is a homomorphism of ρ into σ and $(a_1,\ldots,a_k) \in E(\rho)$. We must show $<h(a_1),\ldots,h(a_k)> \in E(\sigma)$. If h is strong we are done. For any $<b_1\ldots b_m> \notin \rho(R)$ with $<h(b_1)\ldots h(b_k)> \notin \sigma(R)$ add $<b_1\ldots b_m>$ to $\rho(R)$. Let ρ' be the result of all such additions. So $\rho' \supseteq \rho$ and therefore $(a_1,\ldots,a_k) \in E(\rho')$ by monotonicity. Now h is a strong homorphism, from ρ' to σ so $<h(a_1),\ldots,h(a_k)> \in E(\sigma)$since the sentence $\psi(a_1,\ldots,a_k)$ has been shown to be existential. (Recall ψ is the formula expressing E.) ⊣

In lemma 3 we depend on the assumption that the query is everywhere monotonic, not just monotonic on finite states. Of the theorems used in the proof, that concerning preservation under extensions has been shown to be false in the case that only finite states are considered. [Gurevich]. The status of the lemma itself is this case is unknown [op.cit].

We now produce now ~~dijunction~~ a characterization of those canonical queries which are algebriac.

It is obvious that the appearance of a tuple in the result of a canonical query depends upon the existence of certain tuples in the database state. We may wish to know how many such tuples must appear in the state to support a tuple in the query. If $t \in ?X(\rho)$, is it possible to bound the size of a substate $\sigma \subseteq \rho$ with $t \in ?X(\sigma)$ which bound is independent of the size of ρ? Note that in the case of transitive closure, it is not possible to do this. This motivates the following definition.

<u>Definition</u> A schema R is *X-bounded* with respect to a set of dependencies D (for some X a set of attributes), if there exists an integer k such that for every state ρ, $t \in ?X(\rho)$ implies there exists a substate $\sigma \subseteq \rho$ wi $|\sigma| \leq k$ and $t \in ?X(\sigma)$.

Maier, Ullman and Vardi [MUV] proposed notion of boundedness which we will show equivalent to ours. Their idea is based on the computation of canonical queries via the chase.

<u>Definition</u> A schema R is *X-chase-bounded* with respect to a set of dependencies D (for some X a set of attributes), if there exists an integer k such that for every state ρ, $t \in ?X(\rho)$ implies there exists a sequence of transformations on the dependencies in D which introduces a row w into T_ρ with $w[X] = t$, which sequence is of length not greater than k.

We should point out that qualification that D be finite is crucial to the meaningfulness of this definition. For if D is replaced with its semantic closure (the set of all dependencies it implies), then every schema is X-chase-bounded with $k=1$.

There is yet a third, equivalent notion of boundedness. Consider the set $T_{\mathbf{R},X}$ of the prior section. We will say that $T_{\mathbf{R},X}$ is finitely covered when it is equivalent to some finite subset of itself. By 'equivalence' here, we mean finite equivalence. $T_{\mathbf{R},X}$ is finitely equivalent to some set Σ if the equality $struc(T_{\mathbf{R},X})=struc(\Sigma)$ holds.

Theorem 1. Let $\mathbf{R}$ be a schema and D a finite set of unirelational dependencies on a universe U. The following are equivalent.

1) $\mathbf{R}$ is X-bounded with respect to D.

2) $\mathbf{R}$ is X-chase-bounded with respect to D.

3) $T_{\mathbf{R},X}$ is finitely covered.

4) $?X[\mathbf{R},D]$ is equivalent to a finite union of conjunctive queries.

5) $?X[\mathbf{R},D]$ is equivalent to an expression of t relational algebra.

Proof. $1 \Rightarrow 2$ Let k_0 be the integer required by e definition of X-bounded.

There are, up to isomorphism, finitely many states of size k_0. Each has only finitely many "X-consequences;" that is, $?X(\rho)$ is always finite. For each row of $?X(\rho)$ for each ρ there is a sequence of some finite length which introduces this consequence into T_ρ. The length, k_1, of the longest sequence among these proofs is the bound required for $\mathbf{R}$ to be X-chase-bounded.

$2 \Rightarrow 1$ Immediate.

1⇒3 As before, let k_0 be the bound required by the definition of X-bounded. We claim that $T_{R,X}$ need contain no dependency $d=<T,x>$ with $|T|>k_0$. But this is immediate.

3⇒4 Each of the dependencies in $T_{R,X}$ is identical in format to a conjunctive query on **R** with target list X. Set E to be the union of the (finitely many) conjunctive queries in $T_{R,X}$. We claim $E(\rho)=?X(\rho)$ for every state ρ.

By construction we have $<\rho,E(\rho)>\in struc(T_{R,X})$, that is $E(\rho)\supseteq ?X(\rho)$. For the reverse inclusion, we can show that for every $\xi\supseteq ?X(\rho)$, $\xi\supseteq E(\rho)$. That is, $E(\rho)=?X(\rho)$, by corollary 3.

So let $\upsilon\in E(\rho)$. By definition of E, there is an element $<T,x>\in T_{R,X}$ and some homomorphism η with $\eta(T)\subseteq T_\rho$ (η tag preserving) and $\eta(x)=\upsilon$. But $<T,x>$ is an element of $T_{R,X}$ so any ξ with $<\rho,\xi>\in struc(T_{R,X})$ must satisfy $<T,x>$; that is, $\eta(x)\in\xi$, that is $\upsilon\in\xi$.

4⇒5 Immediate.

5⇒4 From lemma 3 and proposition 5.

4⇒1 The bound is the number of conjuncts in the largest clause of the expression. ⊣

5. Boundedness with respect to consistency

In the preceding section we were concerned with the finiteness of the set $T_{R,X}$ of tgd's in the language $L_{R,X}$ implied by the dependencies, containing instance, and projection axioms defining the canonical queries. A similar question can be asked about the set E_R of equality generating dependencies so implied.

Fact: [GV] The set $E_{\mathbf{R}}$ is finite iff there is an integer k such that any inconsistent state of **R** has an inconsistent substate of size not exceeding k. ⊣

Thus we say **R** is bounded with respect to consistency if $E_{\mathbf{R}}$ is finite.

Despite the similarity of this fact to the equivalencies in theorem 1, we now show by example that boundedness with respect to consistency and algebraicness are mutually independent.

If D contains no egd's, then $E_{\mathbf{R}}$ is empty. So in particular, the transitive closure example (see section 4) is bounded with respect to consistency but not algebraic.

Let F be a set of functional dependencies over some universe U and let $SAT(U;F)$ be the set of all instances of U which satisfy F. Let $\mathbf{A}$ be the set $\{\Pi_V(I)|I \in SAT(U,F)\}$ for some $V \subseteq U$. As pointed out by Ginsberg and Zaddian [GZ], $\mathbf{A}$ need not be $SAT(V,G)$ for any set of functional dependencies G. Hull has recently shown that in that case $E_{\{V\}}$ is not finite [H]. But notice that $\mathbf{A} = CONS(\{V\},F)$ and that for any $X \subseteq U$, $?X[\{V\}].F$ is either identically empty (if $X \not\subseteq V$) or is equivalent to the appropriate projection. So $E_{\{V\},X}$ is certainly finite.

We will say a schema **R** is algebraic if for every X, $?X/\mathbf{R},D/$ is algebraic. If D contains only typed equality generating dependencies, algebraicness is implied by boundedness with respect to consistency.

Proposition 6. Suppose D is a set of typed egd's and **R** is bounded with respect to consistency with D. Then **R** is algebraic.

Proof. Suppose not. From the hypotheses and prior results, we know

1) there is an integer k_0 such that an inconsistent state of **R** contains an inconsistent substate of size not exceeding k_0;

2) For some $X \subseteq U$ and every integer k_1, there exists a state ρ with at least k_1 tuples and a tuple $x \in ?X(\rho)$ and $x \notin ?X(\sigma)$ for any proper substate σ of ρ.

Letting k be the integer of point 1 above, construct a consistent state as described in point 2 of size at least $k/X/$. Let this state be σ. Recalling that D contains only egd's, we note that the row of $chase_D(T)$ with x-value$x$ (x is the X-value given in point 2 above) must correspond to a tuple $v \in \sigma(R)$ for some R and $X \not\subseteq R$.

Let ν be a 1-1 mapping of $Sym(\sigma)$ which is the identity on symbols of the tuple v and takes all other symbols of σ to symbols not in σ. Let $\rho = \sigma \cup \nu(\sigma)$. To see that ρ is not consistent with D, let u be any tuple of σ such that $u/A/=x/A/$ for some $A \in X-R$. The egd $<T\rho, u[A]=\nu(u[A])>$ is a consequence of D but $u[A] \neq \nu(u[A])$ by construction. Consêquently there must be a substate $\rho_A \subseteq \rho$ with k or fewer tuples such that D implies $<T\rho_A, u[A]=\nu(u[A])>$. This substate must contain some rows of $\nu(\sigma)$ (although not necessarily u or $\nu(u)$).

Let η be the mapping on $T\rho_A$ defined by: $\eta(t)=t$ if t . $\eta(t)=v$ if $t \in \nu(\sigma)$. Now η is a homomorphism embedding $T_{\rho A}$ into T_σ since for every $v_0 \in \nu(\sigma)$, $v_1 \in \sigma$, and every attribute B, $v_0[B]=v_1[B]$ only if $v_0[B]=v[B]=v_1[B]$. So η is homomorphism enabling in T_σ a transformation on dependency $< T_{\rho A}, u/A/=\nu(u/A/)>$. Application of this transformation to T_σ will set $t/A/=x$. But $/\eta(T_{\rho A})/<k$. Repeat this arguement for each $A \in X-R$. This will uncover a substate $\sigma' \subseteq \sigma$ with $/\sigma'/<n/X/$ and $x?(\sigma')$. this contradicts our choice of σ.⊣

We now take up the task of tightening the results of Theorem 1. We wish to characterize algebraic canonical queries defined with respect to a set of dependencies which include egd's. Equivalently, we wish to consider queries defined exactly on the set *CONS*(**R**,*D*). We face an immediate syntactic difficulty: an expression of the relational algebra is necessarily defined on all states of **R**, without regard to their inclusion in *CONS*(**R**,*D*). Thus we must expand the domain of $?X[\mathbf{R},D]$ if we wish to find any algebra expression to which it is equivalent. A method of doing this is given by Corollaries 2 and 3 of section 3: replacing D with D^{ef}. This method is exploited in Theorem 1. We seek in this section an expansion which distinguishes consistent from inconsistent states more,precisely. Many such expansions are possible. We adopt the following.

For a set of attributes X of cardinality n, we define the X-product of a state ρ as

$$\times_{A\in X}(dom(A))\cap Sym(\rho)^n$$

That is, an X-product is the set of all combinations of symbols in ρ which respect the domain definitions. We define $?X[\mathbf{R},D](\rho)$ to be the X-product of ρ when $\rho\notin CONS(\mathbf{R},D)$. This definition reflects the standard logical notion that everything is a consequence of an inconsistent set of sentences. It also preserves the monotonicity of canonical queries, as any superset of an inconsistent state is inconsistent.

The expanded function will not always distinguish consistent from inconsistent states. Consider a four attribute universe with two schemes: $\{AB,CD\}$ and the functional dependency $A\rightarrow B$. If the domains of these attributes are pairwise disjoint (the "typed" case), then $?C$ is identically the C-product in every state. Similarly, $?CD$ is the CD-product in some consistent states. We can describe sets of attributes for which this behaviour is impossible.

Let $d = <T, a=b>$ be an egd. The repeating symbols of d are those elements of $Sym(T)$ with more than one appearance in d (a and b are presumably repeating symbols.) The agree set of d is the set of attributes labelling the columns of T in which the repeating symbols occur. (See Ginsburg and Hull ?? [GH].) If X contains the agree set of some egd in or implied by a set of dependencies D, then $?X[\mathbf{R}, D](\rho)$ satisfies d exactly when $\rho \in CONS(\mathbf{R}, D)$. [Not quite: we need 2 symbols of dom(A) in ρ.] We will exploit this fact in Theorem 2. We must first expand the class of relational algebra expressions we allow.

As we have allowed egd's in D, we must allow equality in our expressions. We define a conjuctive query with inequalities to be a conjunctive query plus a set of pairs of symbols called inequality assertions (and written $a \neq b$). So if q is a conjunctive query with inequality

$$q = < <T, x>, S>$$

and ρ is a state, $q(\rho) = \{\nu(x) | \nu(T) \subseteq T\rho$, ν a homomorphism and $\nu(a) \neq \nu(b)$ for each $a \neq b$ in $S\}$. (The expansion of conjunctive queries to include inequalities was first made by Klug [K].)

We recall that $D_{\mathbf{R},X}$ is the set of all multirelational egd's and full tgd's which are consequences of the set Σ defined in section 3. Again, $D_{\mathbf{R},X}$ is said to be finite if it is finitely equivalent to a finite subset of itself.

Theorem 2. Let X contain the agree set of some egd implied by a set of dependencies D. [Do I need this?] For any schema $\mathbf{R}$, the following are equivalent:

1) $\mathbf{R}$ is bounded with respect to consistency and X-bounded.

2) $D_{\mathbf{R},X}$ is finite.

3) $?X[\mathbf{R}, D]$ is equivalent to a union of conjunctive queries with inequalities.

4) $?X[\mathbf{R}, D]$ is equivalent to an expression of the relational algebra.

Proof. The equivalence 1⇔2 follows from Theorem 1 and the fact mentioned earlier. We show 2⇒3 by construction. (A proof of 2⇒4 exists which omits this step. We find this procedure more informative.)

Construct a conjunctive query for each element of $T_{\mathbf{R},X}$ as before. Let E_1 be the union of these queries For each element of $E_{\mathbf{R}}$, proceed as follows:

Let $<T.a=b>$ be an element of $E_{\mathbf{R}}$. Let $W=\{w_R \mid R \in \mathbf{R}, w_R[Tag]=R\}$ be a collection of tagged rows sharing no symbols with each other or with $Sym(T)$. Let $v_1,\dots,v_p$ be rows with tag X which rows result from permuting the symbols in $\cup_{R\in\mathbf{R}}(\cup_{A\in X}\{w_R[A] \mid A\in R\})$ in all ways consistent with the domain definitions. Construct the set of conjunctive queries with inequalities

$$\{<<T\cup W.v_i>.\{a\neq b\}> \mid 1\leq i\leq p\}.$$

Let E_2 be the union of all these queries.

We claim the union of $E_1 \cup E_2$ calculates $?X[\mathbf{R}, D]$. The proof is as before with the observation that if any element of $E_{\mathbf{R}}$ is violated by a state, the set of queries so constructed will force the result to be the appropriate X-product.

The equivalence 3⇔4 is as before. Note that a conjunctive query with inequality is monotonic, so Lemma 2, suitably modified, holds for the larger class of expressions considered here.

We complete the chain by demonstrating 3⇒2. If $<<T.x>.S>$ is an element of the union given by (3), construct the sentence

$$\forall y(T' \rightarrow x' \vee s')$$

where $T' = \wedge\{R(w[R]) \mid w[Tag] = R, w \in T\}$

$x' = X(x)$

$s' = \vee\{a = b \mid a \neq b \in S\}$

and y is the vector of all variables appearing in this sentence. We claim this sentence is implied by Σ (by (3)) and apply the result of McKinsey referenced earlier to reduce the resulting finite set to a subset of $D_{\mathbf{R},X}$, as before. We then claim this set to be finitely equivalent to $D_{\mathbf{R},X}$. **[Does this really work? I think so but I'm passing on. The next paragraph can also be used to prove this (or 4⇒2)]** ⊣

The weakening assumption in this theorem is a result of the particular expansion of canonical queries which we've adopted. Suppose we were to choose an expansion which distinguish consistent and inconsistent states via some first-order property. In other words, suppose there exists, with respect to this putative expansion, a sentence ψ on a single predicate (of arity the cardinality of X) such that ψ is true at $?X[\mathbf{R}, D](\rho)$ exactly when ρ is consistent. If $?X$ is algebraic, the first order formula ϕ which expresses $?X$ can be composed with ψ to produce a sentence of $L_{\mathbf{R}}$ true of a state ρ exactly when ρ is consistent. (This composition is the syntactic exercise of replacing the atomic formulae of ψ with the formula ϕ, due care being taken to rename variables as appropriate.) But in that case, $E_{\mathbf{R}}$ is finite, by the results of [GV].

6. Discussion and Conclusions

We have considered the question: When is a canonical query algebraic, i.e., equivalent to an expression of the relational algebra? It is natural to ask the converse question. When is an expression of the relational algebra equivalent to some canonical query? The answer is the same. Such an expression must be monotonic and therefore equivalent to a union of conjunctive queries. Each such query is essentially a multirelational tgd, which may be considered a unirelational tgd simply by ignoring the tags. Thus each monotonic expression E gives rise to a set of tgd's D such that $E = ?X[\mathrm{R}, D]$ (X the "target scheme" of E). Every monotonic expression is canonical for some set of dependencies.

In the above discussion, we chose D after having seen the expression E. The reader may object to this procedure, considering the dependencies to come "first" and the queries only "later". But is this order correct? The purpose of canonical queries, window functions [MRW] [MUV] and universal relation interfaces [KU] is to make some set of queries very easy to formulate. Which set of queries should this be? We believe the database administrator, in cooperation with the end users, knows very well which queries are important. The dependencies and perhaps even the schema may be derived from the queries, rather than conversely. It is usual to declare the dependencies to be derived from "nature", that is, from knowledge of the application. We do not dispute this. We have shown that they describe an inference engine for the calculation of certain pre-selected queries.

RJ 4207 (46285) 2/23/84
Computer Science

ON THE COMPLEXITY AND AXIOMATIZABILITY OF CONSISTENT DATABASE STATES[+]
(Extended Abstract)

Marc H. Graham

Georgia Institute of Technology
Atlanta, Georgia 30332

Moshe Y. Vardi*

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: A database is consistent with respect to a set D of dependencies if it has a weak instance. A weak instance is a universal relation that satisfies D, and whose projections on the relation schemes are supersets of the relations in the database. In this paper we investigate the complexity of testing consistency and the logics that can axiomatize consistency, relative to a fixed set D of dependencies. If D is allowed to include embedded dependencies, then consistency can be non-recursive. If D consists only of total dependencies, then consistency can be tested in polynomial time. The degree of the polynomial can, however, be arbitrarily high. Consistency can be axiomatized but not finitely axiomatized by equality generating dependencies. If embedded dependencies are allowed then consistency cannot be finitely axiomatized by any effective logic. If, on the other hand, only total dependencies are allowed then consistency can be finitely axiomatized by fixpoint logic.

[+] To be presented in the 3rd ACM Symp. of Principles of Database Systems, Waterloo, April 1984.

* This research reported here was done while this author was at Stanford University.

ON THE COMPLEXITY AND AXIOMATIZABILITY OF CONSISTENT DATABASE STATES[†]

Extended Abstract

Marc H. Graham
Georgia Institute of Technology

Moshe Y. Vardi[‡]
IBM Research Laboratory, San Jose

Abstract

A database is consistent with respect to a set Σ of dependencies if it has a weak instance. A weak instance is a universal relation that satisfies Σ, and whose projections on the relation schemes are supersets of the relations in the database. In this paper we investigate the complexity of testing consistency and the logics that can axiomatize consistency, relative to a fixed set Σ of dependencies. If Σ is allowed to include embedded dependencies, then consistency can be non-recursive. If Σ consists only of total dependencies, then consistency can be tested in polynomial time. The degree of the polynomial can, however, be arbitrarily high. Consistency can be axiomatized but not finitely axiomatized by equality generating dependencies. If embedded dependencies are allowed then consistency cannot be finitely axiomatized by any effective logic. If, on the other hand, only total dependencies are allowed then consistency can be finitely axiomatized by fixpoint logic.

1. Introduction

Soon after the introduction of the relational model [C1], the important role of *semantic specification* was realized [C2,AN]. The purpose of semantic specification is to define which databases are semantically meaningful, called *consistent* in database terminology. The languages used for semantic specification are logical languages. Thus, the database is consistent if and only if it satisfies certain sentences in the language. An example of such a language is the language of *functional dependencies* [C2].

Traditionally, the logic used for semantic specification languages was *first-order logic*. The reason for that is probably the fact that this is the logic that most researchers and practitioners were familiar with. Recently, however, researchers in the area of semantic specification realized that there does not seem to be a straightforward way of specifying semantics of databases with *incomplete information* by means of first-order logic [Ho].

The situation is as follows. In principle, there is a conceptual database with complete information, called *weak instance* in database terminology, that completely describes reality. The semantics of this idealized database is given in first-order logic. In practice, however, we very often do not have all the information needed to describe reality. That is, the actual database does not contain enough information to uniquely determine the conceptual database. How we do know whether our partial description of reality is semantically meaningful? The intuitive answer is that it is semantically meaningful if it can be completed to a full description of reality. This is the justification for the definition in [Ho] that an actual database, which may have incomplete information, is consistent if it can be completed to a consistent database with complete information.

[†] To be presented in the 3rd ACM Symp. on Principles of Database Systems, Waterloo, April 1984

[‡] The research reported here was done while this author was at Stanford University and supported by a Weizmann Post-Doctoral Fellowship and AFOSR grant 80-0212.

While this definition was readily adopted by researchers and triggered numerous investigations of its implications (e.g., [GMV, Sa, MUV]), its logical aspects were not yet investigated.

A logic consists of three essential components: a *language*, a class of *structures* and a *satisfaction relationship* between structures and sentences in the language. The notion of structure in database theory is well understood: databases are essentially *finite* relational structures. What we are interested here is in the language and satisfaction relationship components. Specifically, we try to answer the two following questions:

(1) What is the *complexity* of testing consistency?

(2) What is the language required to *axiomatize* consistency?

More formally, we are given a set Σ of first-order sentences that the conceptual database (with complete information) is supposed to satisfied. Let $CONS(\Sigma)$ be the class of actual database (with incomplete information) that can be completed to satisfy Σ. We try to find out what is the complexity of recognizing databases in $CONS(\Sigma)$ and whether we can axiomatize it, that is, construct a set (preferably finite) Σ' of sentences in some language such that $CONS(\Sigma)$ is exactly the class of actual databases that satisfy Σ'. We are interested here in the case where the conceptual database is required to satisfy first-order sentences of a special form, the so called *data dependencies* [BV1, Fa2]. This class of sentences is considered to be appropriate to semantic specification of databases with complete information.

Our first finding is that there exists a set Σ of eid's such that $CONS(\Sigma)$ is not recursive! We are hence forced to restrict ourselves to the subclass of *total* (or *full*) dependencies [BV1, Fa2]. In this case we show that $CONS(\Sigma)$ is in PTIME. The degree of the polynomial can, however, be arbitrarily high!

With this in mind we turn to the issue of axiomatizability. By using classic model-theoretic techniques, we show that consistency is axiomatizable by first-order logic and even by dependencies, but is not *finitely* axiomatizable by first-order logic. The fact that consistency can be tested in polynomial time, and the strong connection between polynomial time computation and *fixpoint logic* shown in [Im, Var2], suggest that fixpoint logic might be the right logie to axiomatize consistency. Indeed, the deepest result in the paper is that consistency is *finitely* axiomatizable by fixpoint logic.

We discuss some "philosophical" aspects of our work in the concluding part of the paper.

2. Basic Definitions

2.1. Tuples, Relations, and Databases

Attributes are symbols taken from a given finite set U called the *universe*. We use the letters $A, B, C, \cdots$ to denote attributes and $X, Y, \cdots$ to denote sets of attributes. Sets of attributes are also called *relation schemes* for reasons to become clear shortly. As a convention, we do not distinguish between the attribute A and the set $\{A\}$, and we denote the union of X and Y by XY.

With each attribute A is associated an infinite set called its *domain*, denoted $DOM(A)$. The domain of a set X of attributes is $DOM(X) = \bigcup_{A \in X} DOM(A)$. An *X-value* is a mapping $w: X \rightarrow DOM(X)$, such that $w(A) \in DOM(A)$ for all $A \in X$. A *tuple* is an X-value for some X. A *relation* on a relation scheme X is a finite set of X-values. We use $a, b, c, \cdots$ to denote elements of the domains, $s, t, \cdots$ to denote tuples, and $I, J, \cdots$ to denote relations.

A *database scheme* is a sequence $\mathbf{R} = (R_1, \ldots, R_k)$ of relation schemes such that $U = \bigcup_{i=1}^{k} R_i$. We will occasionally consider U as a database scheme, meaning (U). A sequence $\mathbf{I} = (I_1, \ldots, I_k)$ of relations on $R_1, \ldots, R_k$, correspondingly, is called a *database* on $\mathbf{R}$. Let $\mathbf{I} = (I_1, \ldots, I_k)$ and $\mathbf{J} = (J_1, \ldots, J_k)$ be databases on $\mathbf{R}$. We say that $\mathbf{I}$ is contained in $\mathbf{J}$, denoted $\mathbf{I} \subseteq \mathbf{J}$, if $I_m \subseteq J_m$

for $m = 1, \ldots, k$.

For an X-value w and a set $Y \subseteq X$ we denote the restriction of w to Y by $w[Y]$. We do not distinguish between $w[A]$, which is an A-value, and $w(A)$, which is an element of $DOM(A)$. Let I be a relation on X. Then its *projection* on Y, denoted $I[Y]$, is a relation on Y, $I[Y] = \{w[Y] : w \in I\}$. Let **R** be a database scheme. We associate with **R** a projection map $\pi_{\mathbf{R}}$, defined as follows. Let I be a relation on U. Then $\pi_{\mathbf{R}}(I)$ is the sequence $(I[R_1], \ldots, I[R_k])$, which is a database on **R**. The set of all attribute values in a relation I is $VAL(I) = \bigcup_{A \in X} I[A]$, and the set of values in a database **I** is $VAL(\mathbf{I}) = \bigcup_{j=1}^{k} VAL(I_j)$. The database **I** is *nonempty* if $VAL(\mathbf{I}) \neq \emptyset$.

2.2. Dependencies

A *valuation* is a partial mapping $\alpha : DOM(U) \rightarrow DOM(U)$ such that for all $A \in U$ and $a \in DOM(A)$ we have $\alpha(a) \in DOM(A)$. We say that α is a valuation on a tuple w (resp., relation I, database **I**) if it is defined on $VAL(w)$ (resp., $VAL(I)$, $VAL(\mathbf{I})$). Let α be a valuation on a tuple w, then $\alpha(w)$ is the tuple $\alpha \circ w$ (i.e., α composed with w). Valuations are defined on relations and databases in the natural way, i.e., they are defined on relations tuple-wise, and they are defined on databases relation-wise.

For any given application only a subclass of all possible databases is of interest. This subclass is defined by semantic constraints that are to be satisfied by the databases of interest. A family of constraints that was extensively studied in the literature is the family of *dependencies*.

A *tuple generating dependency* (abbr. tgd) says that if some tuples, satisfying certain equalities exist in the database, then some other tuples (possibly with some unknown values), must also exist in the database. Formally, a tgd on a database scheme **R** is a pair <**I**,**J**> of nonempty databases on **R**. It is satisfied by a database **K** on **R** if for every valuation α on **I**, such that $\alpha(\mathbf{I}) \subseteq \mathbf{K}$, there exist a valuation β on **I** and **J** that agrees with α on $VAL(\mathbf{I})$ such that $\beta(\mathbf{J}) \subseteq \mathbf{K}$. If $VAL(\mathbf{J}) \subseteq VAL(\mathbf{I})$ then <**I**,**J**> is a *total* tgd (abbr. ttgd).

An *equality generating dependency* (abbr. egd) says that if some tuples, satisfying certain equalities exist in the database, then some values in these tuples must be equal. Formally, an egd on a database scheme **R** is a pair <**I**,$a_1 = a_2$> where **I** is a database and $\{a_1, a_2\} \subseteq VAL(\mathbf{I})$. It is satisfied by a database **K** on **R** if for every valuation α on **I** such that $\alpha(\mathbf{I}) \subseteq \mathbf{K}$ we have $\alpha(a_1) = \alpha(a_2)$. A *functional dependency* (abbr. fd) is a statement $X \rightarrow Y$. It is satisfied by a relation I on U if for every two tuples u and v in I, if $u[X] = v[X]$ then $u[Y] = v[Y]$. It is equivalent to an egd on U.

We will use the term *dependencies* or *embedded dependencies* to refer to the class tgd's and egd's, and we will use the term *total* to refer to the class of ttgd's and egd's. We note that dependencies are equivalent to first-order sentences of a special syntax [Fa2].

2.3. Satisfaction and Consistency

If we are given a database scheme **R** and a set Σ of dependencies on **R**, then it is quite obvious how to define the class of semantically meaningful databases on **R**. It is just the collection

$$SAT(\mathbf{R}, \Sigma) = \{\mathbf{I} : \mathbf{I} \text{ is a database on } \mathbf{R} \text{ that satisfies } \Sigma\}.$$

However, a basic idea in database theory is that of universal relation interface [MUV]. According to this approach, conceptually the database is a single relation on U, and consequently the semantic specification has to be given as a set of dependencies on U. In practice, however, information is often given to us not as tuples on U but in smaller units, tuples on subsets of U, and some information may even be missing. The database scheme $\mathbf{R} = \{R_1, \ldots, R_k\}$ describes the actual database, and its relations reflects parts of the bigger conceptual database.

Such a database on R is semantically meaningful if indeed it reflects a meaningful conceptual relation on U.

This lead Honeyman [Ho] to the following definition[1]. Let Σ be a set of dependencies on U, and let $\mathbf{I}=(I_1,\ldots,I_k)$ be a database on a database scheme $\mathbf{R}=(R_1,\ldots,R_k)$. We say that $\mathbf{I}$ is *consistent* with respect to Σ, if there exists a relation I on U, such that $I \in SAT(U,\Sigma)$ and $\mathbf{I} \subseteq \pi_{\mathbf{R}}(I)$. I is called a *weak instance* for $\mathbf{I}$. Note that $\mathbf{I}$ does not reflect exactly the breakdown of the information in I to smaller units of information, but rather it reflects a subset of that information, since I_j can be a proper subset of $I[R_j]$. We denote the set of databases on R that are consistent with respect to Σ by $CONS(\mathbf{R},\Sigma)$.

We now define a condition on database schemes that will play an important role when it comes to axiomatizability of consistency. A set Σ of dependencies over U is said to be *m-bounded* with respect to a database scheme R, for some natural number m, if for every database $\mathbf{I}$ on R, we have that $\mathbf{I}$ is in $CONS(\mathbf{R},\Sigma)$ if and only if for all $\mathbf{J} \subseteq \mathbf{I}$ with $|VAL(\mathbf{J})| \leq m$, we have that $\mathbf{J}$ is in $CONS(\mathbf{R},\Sigma)$. We say that Σ is *bounded* with respect to R if it is m-bounded with respect to R for some m.

3. Complexity

Several researchers investigated the complexity of testing satisfaction and consistency [BV2,GMV,MSY,Y]. What they tried to do is to find the complexity of the set $\{\langle U,I,\Sigma\rangle : I \in SAT(U,\Sigma)\}$ and the set $\{\langle \mathbf{R},\mathbf{I},\Sigma\rangle : \mathbf{I} \in CONS(\mathbf{R},\Sigma)\}$. In this context several lower bounds were shown. We find these lower bounds somewhat misleading. In a specific application the database administrator has a specific universe U_0, a specific database scheme $\mathbf{R}_0$, and a specific set Σ_0 of dependencies that describe the semantics of the application. Thus, he has no interest in the complexity of the above mentioned sets, but rather he is interested in the complexity of the sets $SAT(U_0,\Sigma_0)$ and the set $CONS(\mathbf{R}_0,\Sigma_0)$. Thus, what seems to be of interest in general is the complexity of the sets $SAT(U,\Sigma)$ and $CONS(\mathbf{R},\Sigma)$ for fixed U, R, and Σ. In the terms of [Var2] we are interested here the the *data complexity* rather then the *expression complexity* or the *combined complexity*.

Let us consider first satisfaction.

Lemma 1. [Cha] Let R be a database scheme and let Σ be a finite set of dependencies on U. Then $SAT(\mathbf{R},\Sigma)$ is in LOGSPACE. ■

Unlike satisfaction, the complexity of consistency depends on the kind of dependencies we have in Σ.

Theorem 1.

(1) Let R be a database scheme, and let Σ be a finite set of embedded dependencies on U. Then $CONS(\mathbf{R},\Sigma)$ is recursively enumerable.

(2) There exist a universe U and a finite set Σ of embedded dependencies on U such that $CONS(U,\Sigma)$ is not recursive.

(3) The set of pairs $(\mathbf{R},\Sigma)$, where Σ is a finite set of embedded dependencies on U and $CONS(\mathbf{R},\Sigma)$ is recursive, is not recursive.

Idea of Proof.

(1) Given a database on R, we just have to enumerate all relations on U and check whether any of them is a weak instance for the database.

(2) First, by reduction from the word problem for finite semigroups [Gu], we construct a universe U and a finite set Σ of dependencies on U such that the set $\{\sigma : \sigma \text{ is an egd and } \Sigma \text{ logically implies } \sigma\}$ is not recursive. Then, we show that this set is Turing-reducible to $CONS(U,\Sigma)$. The reduction involves exponentially (in the length of the given egd) many tests for consistency.

(3) The claim follows from a general characterization of undecidable properties of sets of dependencies in [Var1]. ■

[1] We use the generalization in [GMV] of the original ideas in [Ho].

Theorem 1 strengthens the results in [GMV] that the set

$\{\langle \mathrm{R}, \mathrm{I}, \Sigma \rangle :$ Σ is a set of embedded dependencies and

$\mathrm{I} \in CONS(\mathrm{R}, \Sigma)\}$.

is not recursive. Both results indicate very strongly that the weak instance approach is not practical when embedded dependencies are necessary to specify the semantics of the application. When all dependencies in Σ are total, the situation is radically different.

Theorem 2.

(1) Let R be a database scheme, and let Σ be a finite set of total dependencies on U. Then $CONS(\mathrm{R}, \Sigma)$ is in PTIME.

(2) There is a universe U and a finite set Σ of total dependencies on U such that $CONS(U, \Sigma)$ is logspace complete in PTIME.

(3) For every natural number k, there exist a universe U_k and finite set Σ_k of total dependencies on U_k, such that $CONS(U_k, \Sigma_k)$ can not be accepted in DTIME(n^k).

Idea of Proof.

(1) In [GMV,Ho] there is an algorithm to test for consistency. Given a database, the algorithm tries to construct a weak instance. It either succeeds, demonstrating consistency, or it fails, proving that there does not exists a weak instance. The complexity of the algorithm is $O(n^l)$, where n is the size of the database and l is the size of R and Σ.

(2) Hardness for PTIME is proven by reduction from the path system problem of [JL].

(3) By a generic reduction from deterministic polynomial time Turing machines. ■

Theorem 2 strengthens the result in [GMV] that the set

$\{\langle \mathrm{R}, \mathrm{I}, \Sigma \rangle :$ Σ is a set of total dependencies and

$\mathrm{I} \in CONS(\mathrm{R}, \Sigma)\}$.

is logspace complete in EXPTIME. It shows that testing consistency of I with respect to Σ is polynomial in the size of I and exponential in the size of Σ.

It is interesting to note in connection with Theorem 2, that if Σ consists of fd's, then $CONS(\mathrm{R}, \Sigma)$ can be accepted in time $O(n \log n)$ and linear space, by computing the closure of some congruence relation as in [DST].

Let us now consider bounded sets of dependencies. Intuitively, it seems that it should be easier to test consistency with respect to bounded sets than for general ones.

Theorem 3. Let R be a database scheme, and let Σ be a set of dependencies on U, such that Σ is bounded with respect to R. Then $CONS(\mathrm{R}, \Sigma)$ is in LOGSPACE.

Idea of Proof. Assume that Σ is m-bounded with respect to R. To check that $\mathrm{I} \in CONS(\mathrm{R}, \Sigma)$ it suffices to check that $\mathrm{J} \in CONS(\mathrm{R}, \Sigma)$ for all $\mathrm{J} \subseteq \mathrm{I}$ such that $|VAL(\mathrm{J})| \leq m$. It is easy to verify that checking each J requires space logarithmic in the size of I. ■

4. Axiomatizability

A subject of great interest in mathematical logic is that of *axiomatizability*. Given a class Ω of structures, the logician tries to axiomatize it by defining a logic Λ, which consists of a language L and a satisfaction relationship between structures and sentences in L. Ω is *axiomatizable* by Λ if there exists a set Σ of sentences of Λ, such that a structure M is in Ω if and only if M satisfies all sentences in Σ. If Σ is finite, then Ω is *finitely axiomatizable* by Λ. This notion of axiomatizability enables us to classify the expressive power of logics according to the classes of structures that they can axiomatize or finitely axiomatize.

We first try to axiomatize consistency by first-order logic. We have to bear in mind, however, that every class of databases is axiomatizable by first-order logic. This follows from the fact that every database can be described, up to isomorphism, by a single first-order sentence. The axioms for the class are the negations of the descriptions of all databases not in the class. In fact, one can show that every class of databases is even axiomatizable in a proper subset of first-order logic. This subset, which we call *universal-existential* logic, is the set of all first-order sentences whose prefix consists of a string of universal quantifiers followed by a string of existential quantifiers. Thus, axiomatizability results for first-order logic are not interesting, unless they talk about finite axiomatizability or about a proper subset of universal-existential logic.

The proof of next theorem uses *disjunctive equality-generating dependencies*. A disjunctive equality-generating dependency (abbr. degd) on a database scheme R is a pair $<I,\delta>$, where I is a finite database and δ is a sequence of equalities $a_1 = b_1, \ldots, a_k = b_k$ with $\{a_1, \ldots, b_k\} \subseteq VAL(I)$. It is satisfied by a database K on R if for every valuation α on I such that $\alpha(I) \subseteq K$ we have that either $\alpha(a_1) = \alpha(b_1)$, or ... or $\alpha(a_k) = \alpha(b_k)$. Observe that an egd is a degd where the sequence of equalities is of unit length.

Theorem 4. Let R be a database scheme, and let Σ be a set of dependencies on U. Then $CONS(R,\Sigma)$ is axiomatizable by egd's.

Idea of Proof. The proof goes in three steps. First, using the method of *diagrams* [CK] we show that $CONS(R,\Sigma)$ is axiomatizable by degd's. That is, there exists a set Σ' of degd's on R such that $CONS(R,\Sigma) = SAT(R,\Sigma')$. Now, using the fact that Σ is a set of dependencies, which are *Horn* sentences, we show that $CONS(R,\Sigma)$ is closed under *direct products*. Finally, using the last fact, we prove by McKinsey's technique [McKi] that we can assume without loss of generality that all the degd's in Σ' are actually egd's. ■

The above result is interesting theoretically, but does not really have practical significance because the set of egd's promised by the theorem can be non-recursive! What we would like to have is finite axiomatizability by first-order logic, because then we would be able to apply Lemma 1, and get logarithmic space complexity. Now, Theorem 3 gives us a case where consistency can be tested in logarithmic space, namely, when the given set of dependencies is bounded with respect to the database scheme. Can it be that Theorem 3 is just a corollary of Lemma 1? The answer is positive.

Theorem 5. Let R be a database scheme, and let Σ be a set of dependencies on U. Then $CONS(R,\Sigma)$ is finitely axiomatizable by egd's if and only if Σ is bounded with respect to R.

Idea of Proof. If $CONS(R,\Sigma)$ is finitely axiomatizable by egd's, then $CONS(R,\Sigma) = SAT(R,\Sigma')$ for some finite set Σ' of egd's. Let $m = \max\{k : <I,a_1 = a_2> \in \Sigma' \text{ and } |VAL(I)| = k\}$. Then Σ is m-bounded with respect to R. Conversely, if Σ is m-bounded with respect to R, then $CONS(R,\Sigma)$ is axiomatizable by egd's $<I,a_1 = a_2>$ with $|VAL(I)| = m$. ■

Theorem 5 leaves open the possibility that consistency is finitely axiomatizable by first-order logic though not by egd's. However, since first-order satisfaction can be tested in logarithmic space, finite axiomatizability of consistency by first-order logic will entail, by Theorem 2, that PTIME = LOGSPACE! This suggests the following result.

Theorem 6. There is a universe U, a finite set Σ of total dependencies on U, and a database scheme R, such that $CONS(R,\Sigma)$ is not finitely axiomatizable by first-order logic.

Idea of Proof. Let $U = \{A,B,C\}$, $R = \{AB,AC\}$, and $\Sigma = \{A \rightarrow C, B \rightarrow C\}$. We now show by an *ultraproduct* argument[2] [CK] that $CONS(R,\Sigma)$ is not finitely axiomatizable by first-order logic. ■

In view of the last two theorems, we would like to be able to tell, given a database scheme R and a set of dependencies Σ, whether Σ is bounded with respect to R. Unfortunately, there is no effective test for boundedness.

Theorem 7. The following set of pairs (U,Σ), where Σ is a finite set of dependencies on U and Σ is bounded with

[2] Thus we have to go to infinite structures in order to prove a claim about finite structures.

respect to U, is not recursive.

Idea of Proof. The claim follows from a general characterization of undecidable properties of sets of dependencies in [Var1]. ■

We do not know whether boundedness is decidable when we restrict ourselves to total dependencies. We believe that if we restrict ourselves to functional dependencies, then it is decidable.

Since we can not finitely axiomatize consistency by first-order logic, we try to do it by higher-order logics. Studying the definition of consistency we observe that essentially it consists of existentially quantifying over weak instances, which are relations over a possibly extended domain. The logic of such definition is called in mathematical logic *many-sorted projective logic* [Fe]. It is a very powerful logic, whose satisfaction relationship is not necessarily recursive (by Theorem 1; see also [Ha]). One can try to bound the size of the extended domain in order to make the satisfaction relationship recursive [MZ], but Theorem 1 implies that when the given dependencies are embedded this can not be done.

Let us now consider the case that the given dependencies are total. As we shall see in this consistency can be finitely axiomatized by the *fixpoint logic* of [AU,CH].

Let P be a new n-ary relation name, and let $L(\mathrm{R},P)$ be the language obtained by adding P to $L(\mathrm{R})$. The fixpoint sentences of $L(\mathrm{R})$ are of the form $LFP(\varphi)$, where φ is a first-order formula of $L(\mathrm{R},P)$ with free variables $x_1,\ldots,x_n$, where P occurs positively. Let M be a structure of $L(\mathrm{R})$ with domain D. Let Q be the minimal n-ary relation on the domain of M, such that the sentences $\forall x_1 \cdots x_n(P(x_1,\ldots,x_n)\equiv\varphi)$ is satisfied in the structure (M,Q) of the language $L(\mathrm{R},P)$. The relation Q is the *least fixpoint* of φ in the structure M. We now define the satisfaction relationship: M satisfies $LFP(\varphi)$ if $Q=D^n$. The following facts hold for fixpoint logic.

(1) Any class of databases that is finitely axiomatizable in fixpoint logic is in PTIME [CH].

(2) There is a class of databases that is finitely axiomatizable in fixpoint logic and is logspace complete in PTIME [Var2].

(3) Let Ω be a class of databases that include a linear order relation, such that Ω is in PTIME. Then Ω is finitely axiomatizable by fixpoint logic [Im,Var2]. (The linear order seems to be essential in order to simulate Turing machines.)

There are two reasons to suspect that consistency with respect to total dependencies can be finitely axiomatized by fixpoint logic. The first reason is, in view of the aforementioned facts, that consistency with respect to total dependencies can be tested in polynomial time. The second reason is that from the algorithm for testing consistency of [GMV,Ho] it follows that consistency with respect to total dependencies can be axiomatized by fixpoint logic over extended domains. Both observation show that with some "extra" tool, either a linear order or an extended domain, we can finitely axiomatized consistency by fixpoint logic. The question is whether we can do it without the "extra" tool. The answer is positive.

Theorem 8. Let R be a database scheme, and let Σ be a finite set of total dependencies on U, then $CONS(\mathrm{R},\Sigma)$ is finitely axiomatizable by fixpoint logic.

Idea of Proof. It turns out that the extended domain is not essential. The information conveyed by the new elements can be captured by relations over the old elements. These relations can be defined by fixpoint logic. The construction, however, is very involved. The length of the fixpoint sentence needed to axiomatize $CONS(\mathrm{R},\Sigma)$ is exponential in the length of Σ! ■

5. Philosophical Remarks

Another use of logical languages in relational database management system is as query languages. The result of applying a formula of the language to a database is the set of all tuples that satisfy the formula. An example of such a language is the relational calculus [C3]. The logic used for query languages was also traditionally first-order logic. However, in the last few years, it was realized that first-order logic does not have a sufficient expressive power as a query language. This was realized first by Aho and Ullman [AU], who observed that transitive closure is not first-order definable (this fact was originally proven in [Fa1]). Following that observation, several works investigated higher-order logics for query languages, e.g., [CH, MZ, Var2].

One can also object to the exclusive use of first-order logic in database theory on an "ideological" basis. The reason for the prominence of first-order logic in mathematical logic is that first-order logic is mathematically tractable and has very rich proof and model theories, e.g., we have completeness and compactness theorems. However, mathematical logic usually deals with general structures, either finite or infinite. In database theory, one usually wishes to consider only finite structures. Under this restriction many of the nice properties of first-order logic evaporate. In particular, we do not have completeness and compactness. Thus, there is no a priori reason to prefer first-order logic to other logics, and one should base his preference on practical considerations, such as ease of use and computational complexity.

First-order logic has the advantage of almost being a "lingua franca". It is a logic with which many practitioners are familiar, unlike the more esoteric higher-order logics. On the other hand, if one takes polynomial time as a yardstick for computational tractability, then there is evidence that fixpoint logic is the natural logic for finite structures [Im, Var]. Our results strengthen this evidence by showing that fixpoint logic rather than first-order logic is the adequate logic to specify semantics of databases with incomplete information. We believe that fixpoint logic should be given far more attention than it has been given in the past.

References

[AN] ANSI/X3/SPARC Interim Report 75-02-08, FDT-SIGMOD 7(1975).

[AU] Aho, A.V., Ullman, J.D.: Universality of data retrieval languages. Proc. 6th ACM Symp. on Principles of Programming Languages, San Antonio, 1979, pp. 110-117.

[BV1] Beeri, C., Vardi, M.Y.: A proof procedure for data dependencies. Technical Report, Dept. of Computer Science, The Hebrew University of Jerusalem, 1980.

[BV2] Beeri, C., Vardi, M.Y.: On the complexity of testing implications of data dependencies. Technical Report, Dept. of Computer Science, The Hebrew University of Jerusalem, 1980.

[C1] Codd., E.F.: A relational model for large shared data banks, Comm. of ACM 13(1970), pp. 377-387.

[C2] Codd, E.F.: Further normalization of the database relational model. In *Data Base Systems* (R. Rustin, ed.), Prentice-Hall, 1972, pp. 33-64.

[C3] Codd, E.F.: Relational completeness of database sublanguage. In *Data Base Systems* (R. Rustin, ed.), Prentice Hall, 1972, pp. 65-98.

[CH] Chandra, A.K., Harel, D.: Structure and complexity of relational queries. J. of Computer and Systems Sciences 25(1982), pp. 99-128.

[Cha] Chandra, A.K.: Programming primitives for database languages. Proc. 8th ACM Symp. on Principles of Programming Languages, 1981, pp. 50-62.

[CK] Chang, C.C., Keisler, H.J.: Model Theory. North-Holland, 1977.

[DST] Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. of ACM 27(1980), pp. 758-771.

[Fa1] Fagin, R.: Monadic generalized spectra. Zeitschr. f. math. Logik und Grundlagen d. Math. 21(1975), pp. 89-96.

[Fa2] Fagin, R.: Horn clauses and database dependencies. J. of ACM 29(1982), pp. 252-285.

[Fe] Feferman, S.: Two notes on abstract model theory - Properties invariant on the range of definable relations between structures. Fundamenta Math. 82(1974), pp. 153-165.

[Gu] Gurevich, Y.: The word problem for certain classes of semigroups (in Russian). Algebra and Logic 5(1966), pp. 25-35.

[GMV] Graham, M.H., Mendelzon, A.O., Vardi, M.Y.: Notions of dependency satisfaction. Technical Report STAN-CS-83-979, Stanford University, August 1983.

[Ha] Hajek, P.: Some remarks on observational model-theoretic languages. Proc. 2nd Conf. on Set Theory and Hierarchy Theory, 1975. Lecture Note in Mathematics - Vol. 537, Springer-Verlag, 1976, pp. 335-345.

[Ho] Honeyman, P.: Testing satisfaction of functional dependencies. J. ACM 29(1982), pp. 668-677.

[Im] Immerman, N.: Relational queries computable in polynomial time. Proc. 14th ACM Symp. on Theory of Computing, San Francisco, 1982, pp. 147-152.

[JL] Jones, N.D., Laaser, W.T.: Complete problems in deterministic polynomial time. Theoretical Computer Science 3(1977), pp. 105-117.

[McKi] McKinsey, J.C.C.: The decision problem for some classes of sentences without quantifiers. J. of Symbolic Logic 8(1943), pp. 61-76.

[MSY] Maier, D., Sagiv, Y., Yannakakis, M.: On the complexity of testing implications of functional and join dependencies. J. of ACM 28(1981), pp. 680-695.

[MZ] Makowsky, J.A., Zvieli, A.: Definable queries. Unpublished manuscript, Technion-Israel Inst. of Technology, 1982.

[MUV] Maier, D., Ullman, J.D., Vardi, M.Y.:The revenge of the JD. Proc. 2nd ACM Symp. on Principles of Database Systems, Atlanta, 1983, pp. 279-287.

[Sa] Sagiv, Y.: Can we use the universal instance assumption without using nulls? ACM Symp. on Management of Data, 1981, pp. 108-120.

[Var1] Vardi, M.Y.: Global decision problem for relational databases. Proc. 22nd IEEE Symp. on Foundation of Computer Science, Nashville, October 1981, pp. 198-202.

[Var2] Vardi, M.Y.: The complexity of relational query languages. Proc. 14th ACM Symp. on Theory of Computing, San Francisco, 1982, pp. 137-146.

[Y] Yannakakis, M.: Algorithms for acyclic databases. Proc. Int'l Conf. on Very Large Data bases, Nice, 1981, pp. 82-94.

G-36-602

Annual Report 1985
NSF-IST-82-17441
Canonical Queries
Marc H. Graham, PI

Georgia Tech ICS graduate students Ke Wang, Steve Ornburn, Gita Rangarajan, and Amy Lapwing were supported under this grant for different periods.

The paper "On the Equivalence of and EGD with a set of FD's" was written with Ke Wang for journal submission. The paper "Constant Time Maintenance" was begun during the contract period. A preliminary version has been submitted to PODS. A complete version is still under construction.

On the Equivalence of FD and EGD

Marc H. Graham

School of Information and Computer Science
Georgia Institute of Technology, Atlanta, GA 30332

Ke Wang

School of Information and Computer Science
Georgia Institute of Technology, Atlanta, GA 30332

ABSTRACT

Functional dependency has bounded structures(exponential size) in the size of schema.Many well done results have been obtained. *equality generating dependency* (EGD) as extension of FD has arbitraryly large structure for given database schema. Isolating the proper subset of EGD each of which is equivalent to a set of FD will be useful to apply the well done results for FD to EGD. A necessary and sufficient condition for the equivalence of a EGD and a set of FD will be found, and a polynomial algorithm for testing such condition is given. Finally, the equivalence condition for simple EGD will be shown to be equivalent to β- acyclicity of the EGD.

September 19, 1985

On the Equivalence of FD and EGD

Marc H. Graham

School of Information and Computer Science

Georgia Institute of Technology, Atlanta, GA 30332

Ke Wang

School of Information and Computer Science

Georgia Institute of Technology, Atlanta, GA 30332

1. Notation and concept

Attributes are symbols taken from a given finite set U called universe. Sets of attributes are called relation scheme.

With each attribute A is associated a infinite set called its domain,denoted as dom(A). We assume that different attribute A has disjoint dom(A).An Y-value is a mapping $Y \rightarrow \bigcup_{A \in Y} DOM(A)$ such that $m(A) \in dom(A)$ for all $A \in Y$. A tuple is an Y-value for some Y.A relation on a relation scheme U is a finite set of U-value.

A ***equality generating dependency*** on a relation scheme R is a pair $\sigma = < T, a = b >$,where tableau T is a set of variable tuples (or simply tuple), and a and b are two variables from the tuples in T. The EGD is typed if no variable appears in more than one column of T and a and b come from the same column of T. Notice that FD is a EGD with T containing two tuples.A homomorphism h from tableau T_1 to tableau T_2 is a mapping from symbols in T_1 to symbols in T_2 such that if v is a symbol of A column in T_1,then h(v) is also a symbol of A column in T_2.

Chase of a set of dependencies D on tableau T is a repeating process of finding a homomorphism from tableau T' of some $d \in D$ to T, make the transformation specified by d on T(either add a tuple or equate two symbols) until no change can be made.For a set of dependencies D and a EGD $\sigma = < T, a = b >$,testing if $D \models \sigma$ can be done by chasing D on T and seeing if a and b are ever equated during the chase, $D \models \sigma$ iff a and b are equated (see [1]).

Let $\sigma = < T = \{ t_{1,} t_2 \ldots t_k \}, x_1 = x_2 >$ be typed EGD on universe U, where x_1 and x_2 are symbols of X column and X ∈ U. We define

$$a(t_i, t_j) = \bigcup_{1 \le i < j \le k} \{ A \mid A \in U \text{ and } t_i[A] = t_j[A] \},$$

we define $< S_1, S_2 >$ to be a partition of T such that $S_1 \subseteq T, S_2 \subseteq T, S_1 \bigcup S_2 = T$ and $\prod_X (S_1) \bigcap \prod_X (S_2) = \emptyset$

Let

$$FD_{< S_1, S_2 >}(\sigma) = \{ Y \rightarrow X \mid Y = \bigcup_{t_i \in S_1, t_j \in S_2} a(t_i, t_j) \},$$

define

$$FD(\sigma) = \{ FD_{< S_1, S_2 >}(\sigma) \mid all < S_1, S_2 > \text{for } \sigma \}.$$

Lemma 1: For any typed EGD σ on U,$\sigma \models FD(\sigma)$.

Proof: since $FD(\sigma) = \{ FD_{< S_1, S_2 >}(\sigma) \mid all < S_1, S_2 > \text{for } \sigma \}$,we only need to prove $\sigma \models FD_{< S_1, S_2 >}(\sigma)$ for all $< S_1, S_2 >$. by the construction of $FD_{< S_1, S_2 >}(\sigma)$,$FD_{< S_1, S_2 >}(\sigma)$ looks like that in figure 1. Obviously,there exists a homomorphism h such that

$$h(t_i) = s_1' \text{ for } all \ t_i \in S_{2,}$$
$$h(t_j) = s_2' \text{ for } all \ t_j \in S_{2,}$$

and

$$h(x_1) = y_{1,}$$
$$h(x_2) = y_{2,}$$

so $\sigma \models FD_{< S_1, S_2 >}(\sigma)$.⊣

We have proved that $FD(\sigma)$ is implied by σ.next lemma shows that $FD(\sigma)$ is the only FD implied by σ.

Lemma 2: For any typed EGD σ, if $\sigma \models W \rightarrow A$ then $FD(\sigma) \models W \rightarrow A$,where $W \subseteq U, A \in U$.

Proof:

Case 1,$A = X$.

tableau form for $W \rightarrow A$ is shown in figure 2. since $\sigma \models W \rightarrow A$,there exists a homomorphism h such that $h(T) = T'$ and $h(x_1) = y_1, h(x_2) = y_2$,that is ,$T$ can be partitioned into S_1 and S_2 such that

$$t_i \in S_1 \text{ if } h(t_i) = S_1',$$
$$t_j \in S_2 \text{ if } h(t_j) = S_2',$$

obviously,$\prod_X (S_1) \bigcap \prod_X (S_2) = \emptyset$,

where $W \supseteq \bigcup_{t_i \in S_1, t_j \in S_2} a(t_i, t_j)$,

but $\bigcup_{t_i \in S_1, t_j \in S_2} a(t_i, t_j) \rightarrow X \in FD(\sigma)$,

so $FD(\sigma) \models W \rightarrow X$.

Case 2, $A \neq X$,

Since h and σ is typed, there exists no homomorphism h such that

$$h(x_1) = y_1,$$
$$h(x_2) = y_2,$$

because x_1 and x_2 are value of X, y_1 and y_2 are values of A, so $\sigma \not\models W \rightarrow A$,contradiction.so A $\neq$ X can not be the case.$\dashv$

Corollary 3: for any typed EGD σ, if σ is equivalent to a set of FD F, then $F \models\dashv FD(\sigma)$.

Proof: by lemma 2, we have $FD(\sigma) \models F$. Since $F \models \sigma$, and by lemma 1, $\sigma \models FD(\sigma)$, so $F \models FD(\sigma)$,therefore, $F \models\dashv FD(\sigma)$.$\dashv$

2. Equivalence of a EGD and a set of FD

From corollary 3, we only need to consider the equivalence between σ and $FD(\sigma)$.

Lemma 4: For any typed EGD σ on U, $FD(\sigma) \models \sigma$ if there exists a chain $t_{i_1}, t_{i_2} \cdots t_{i_l}$, where t_{i_1} is any tuple in T containing symbol x_1, t_{i_l} is any tuple in T containing symbol x_2 ,$t_{i_k} \in T$, such that either $X \in a(t_{i_k}, t_{i_{k+1}})$ or $a(t_{i_k}, t_{i_{k+1}}) \supseteq W$ for some W, where $W \rightarrow X \in FD(\sigma)$, for $1 \leq k < l$.

Proof: let $W_{i_k} \rightarrow X \in FD(\sigma)$ correspond to pair t_{i_k} and $t_{i_{k+1}}$ such that $a(t_{i_k}, t_{i_{k+1}}) \supseteq W_{i_k}$(if $X \notin a(t_{isub}\, k, t_{i_{k+1}})$), we apply FD sequence $W_{i_1} \rightarrow X, W_{i_2} \rightarrow X \cdots W_{i_{l-1}} \rightarrow X$ to T, we always change the symbol of second tuple to that of first one. Finally, we will change x_2 to x_1,therefore $FD(\sigma) \models \sigma$.⊣

In the next of the paper,we will say that the chain $t_{i_1}, t_{i_2}, \cdots t_{i_l}$ having the property in lemma 4 is covered by $FD(\sigma)$. Now we give the main theorem for equivalence condition.

Theorem 5 : For any typed EGD $\sigma=<T,x_1=x_2>$ on U,σ is equivalent to a set of FD iff there exists a chain of T $t_{i_1}, t_{i_2} \cdots t_{i_l}$ covered by $FD(\sigma)$.

Proof: if

by lemma 4, $FD(\sigma) \models \sigma$, by lemma 1, $\sigma \models FD(\sigma)$, so $FD(\sigma) \models\!\dashv \sigma$.

only if

if there is no such chain covered by $FD(\sigma)$, then x_1 and x_2 can never be equated during the chase of T by $FD(\sigma)$, so $FD(\sigma) \not\models \sigma$, by corollary 3 , σ is not equated to set of FD.⊣

3. Algorithm for testing the equivalence.

The algorithm will test whether there exists a chain $t_{i_1}, t_{i_2} \cdots t_{i_l}$ in T covered by $FD(\sigma)$.Given a EGD σ, the algorithm constructs a agree set graph $G_\sigma=(V,E)$, where $V=\{t_1,t_2 \cdots t_k\}$,$E=\{e_{ij} \mid e_{ij}$ *is labled by* $a(t_i,t_j), i\neq j\}$. Let T_{x_1} be the set of all $t_i \in T$ such that $t_i[X]=x_1$, let T_{x_2} be the set of all $t_j \in T$ such that $t_j[X]=x_2$,the algorithm will test whether deleting attributes in $a(t_i,t_j)$ from G_σ make T_{x_1} and T_{x_2} disconnect. If so, (t_i,t_j) will be a candidate edge for the chain covered by $FD(\sigma)$, because $a(t_i,t_j) \supseteq FD_{<S_1,S_2>}(\sigma)$, where S_1 contains T_{x_1},and S_2 contains T_{x_2},and S_1 is disconnected from S_2 after deleting attributes in $a(t_i,t_j)$ form G_σ.Having found all the candidate edges, a path of candidate edges and edges containing X from $t_i \in T_{x_1}$ to $t_j \in T_{x_2}$ will be a chain covered by $FD(\sigma)$ (if it exists).

Algorithm : Test whether $FD(\sigma) \models\!\dashv \sigma$.

Input: A typed EGD $\sigma=<T=\{t_1,t_2,\ldots t_k\},x_1=x_2>$ on universe U.

Output: "yes" and a chain covered by $FD(\sigma)$ if $FD(\sigma) \models\!\dashv \sigma$; "no", otherwise.

Method:

1. Construct a undirected graph $G_\sigma=(V,E)$,where $V=\{t_1,t_2 \cdots t_k\}$,

$E=\{e_{ij}$ *labled with* $a(t_i,t_j) \mid i \neq j\}$.

2. for each $e_{ij} \in E$,do following : copy G_σ into G'_σ ,deleting attributes in $a(t_i,t_j)$ from G'_σ . for each $t_i \in T_{s_1}$, run DFS algorithm on the resulting graph to decide whether t_i is connected to some $t_j \in T_{s_2}$. If no such $t_j \in T_{s_1}$ exists, mark e_{ij} as candidate in G_σ.

3. Delete e_{ij} in G_σ if $a(t_i,t_j)$ is neither marked candidate, nor $X \in a(t_i,t_j)$, call the resulting graph G''_σ .Notice that when deleting e_{ij} from G_σ, the attributes in $a(t_i,t_j)$ is not deleted from other edge in G_σ.

4. For each $t_i \in T_{s_1}$, run DFS on G''_σ . If some t_i is connected to some $t_j \in T_{s_2}$ by path $t_{i_1},t_{i_2}\ldots t_{i_l}$, return "yes" and the path , stop. otherwise, return "no " and stop. $\dashv$

Theorem 6 : The algorithm is correct.

Proof: If $FD(\sigma) \mid\!\!=\!\!\mid \sigma$, from theorem 4, there exists a chain $t_{i_1},t_{i_2}\ldots t_{i_l}$covered by $FD(\sigma)$. That is , for each $(t_{i_k},t_{i_{k+1}})$, either $a(t_{i_k},t_{i_{k+1}}) \supseteq W$ for some $W \rightarrow X \in FD(\sigma)$, or $X \in a(t_{i_k},t_{i_{k+1}})$, where $t_{i_1} \in T_{s_1}$, $t_{i_l} \in T_{s_2}$, $W = FD_{<S_1,S_2>}(\sigma)$ for some partition $<S_1,S_2>$ and $T_{s_1} \subseteq S_1, T_{s_2} \subseteq S_2$. Obviously, (if $X \notin a(t_{i_k},t_{i_{k+1}})$) , deleting $a(t_{i_k},t_{i_{k+1}})$ will disconnect S_1 from S_2, so $e_{i_k i_{k+1}}$ will be marked as candidate, so $t_{i_1},t_{i_2}\ldots t_{i_l}$ is a path consisting of either candidate edges or edges containing X, so the algorithm will return "yes" and a chain that can be covered by $FD(\sigma)$.

If $FD(\sigma) \not\mid\!\!=\!\!\mid \sigma$, from theorem 5, there is no chain covered by $FD(\sigma)$.Suppose that the algorithm returns "yes", then the path returned by the algorithm will be a chain covered by $FD(\sigma)$, contradiction, so the algorithm must return "no". $\dashv$

In general, there are $2^{|T|}$ partitions $<S_1,S_2>$,so a obvious algorithm for finding $FD(\sigma)$ will be exponential time of $|T|$. If σ is not reduced in the sense that the number of tuples in σ is not minimal, the algorithm will be very time-consuming.

4. β-acyclicity and equivalence property for simple, reduced EGD

A EGD $\sigma = <T, x_1 = x_2>$ is simple if there is at most one kind of repeating symbol in each column of T. formally, that is

$$a(t_i,t_j) \bigcap a(t_k,t_l) \subseteq a(t_i,t_k) \text{ for } all \;\; t_i,t_j,t_k,t_l \in T$$

A EGD $\sigma=<T,x_1=x_2>$ is reduced if T contains minimal number of tuples, that is ,removing any tuple from T will result in an unequivalent EGD.

For reduced EGD σ , we can assume that G_σ consists of one connected component including both tuples containing x_1 and tuples containing x_2. Now we define hypergraph for reduced EGD σ. Hypergraph $H_\sigma=(N,E)$ for EGD $\sigma=(T,x_1=x_2)$ is defined as

$$E=\{e_i=\bigcup_{t_j\in T} a(t_i,t_j) \mid t_i\in T\}, N=all\ \ attributes\ \ in\ \ E,$$

that is, each hyperedge e_i in E is a repeating attributes (the attributes having repeating value in T) for tuple $t_i\in T$. Since σ is reduced , we assume that H_σ is one connected component.There is one to one correspondence between H_σ and G_σ: each node t_i and its outgoing edges in G_σ corresponds to hyperedge e_i, the union of outgoing labels for t_i is equal to e_i; each edge e_{ij} in G_σ corresponds to $e_i\bigcap e_j$. On the other hand, each node A in H_σ corresponds to attribute A in the labels of G_σ, and each edge e_i in H_σ corresponds to a node t_i and its outgoing edges. We adopt standard notations of subhypergraph, induced hypergraph, articulation set ,and acyclicity for hypergraph H_σ from [2].

Corresponding notations should be easily defined for G_σ;there whenever a attribute is removed from one edge, it is also removed from whole G_σ. In following theorem, the proof is based on G_σ, since it is easily related to covered chain in G_σ.

One important property of simple EGD is that each node A in H_σ (or label A in G_σ) corresponds to exactly one repeating value in A column of T. Therefore no distinction needs to be made between attribute and its repeating value in H_σ (or G_σ). From this we have another important property of simple EGD, edge transitivity, that is, in H_σ,

$$A\in e_i\bigcap e_j \text{ and } A\in e_j\bigcap e_k \ \ implies\ \ A\in e_i\bigcap e_k .$$

or in G_σ,

$$A\in e_{ij} \text{ and } A\in e_{jk} \ \ implies\ \ A\in e_{ik} .$$

We will see that this transitivity property play a crucial rule in the proof of following theorem.

Let (N,E) be a hypergraph, and let F be a subset of E, let u be the set of nodes that is the union of the members of F. We say that F is guarded if there is an edge f (called guard) in F such that for each edge e of the hypergraph that is not in F, we have $e \bigcap u \subseteq f$. We say that F is closed if for each edge e of the hypergraph there is an edge f in F such that $e \bigcap u \subseteq f$. It follows easily that every guarded set of edges is closed.

We will use α-acyclicity, β-acyclicity defined in [3]. Especially, we will use the definition that a reduced hypergraph is α-acyclicity iff every nontrivial, connected closed set of (full) edges has an articulation set (see [3]), and we make use of definition that a hypergraph is β-acyclic iff every subhypergraph of it is α-acyclic.

Lemma 7 : Suppose σ is typed, simple ,reduced EGD on U, and $FD(\sigma) \models \sigma$ and $t_1 t_2, \ldots t_l$ is a chain covered by $FD(\sigma)$, then

$$a(t_i, t_j) \subseteq a(M, M+1) \quad i \leq M < j,$$

for any $1 \leq i < j \leq l$.

Proof: Since $FD(\sigma) \models \sigma$,there is a shortest chain $t_1, t_2 \ldots t_l$ that is covered by $FD(\sigma)$ and since σ is reduced ,$t_1, t_2 \ldots t_l$ are the only tuples in σ. We first note that X can appear in G_σ at most once, otherwise,by simplicity of σ , we can get a shorter chain that is covered by $FD(\sigma)$ (dashed chain in figure 3).

For any $1 \leq M < |T|$,and for any $1 \leq i \leq M, M+1 \leq j \leq |T|$:

1. if $i = 1, j = l$,

   if $X \notin a(M, M+1)$,then $a(1,l) \subseteq a(M, M+1)$,otherwise,

   if $X \in a(M, M+1)$,then

   if $M = 1, M+1 = l$ then $a(1,l) \subseteq a(M, M+1)$,otherwise,

   if $1 < M$ *or* $M+1 < l$ then for all $1 \leq k < M$ *or* for $M+1 \leq k' < l$,from the case we have proved, we have

   either $a(1,l) \subseteq a(k, k+1)$ (since $X \notin A(k, k+1)$),

   or $a(1,l) \subseteq a(k', k'+1)$ (since $X \notin A(k', k'+1)$).

But σ is simple, by the edge transitivity, we have $a(1,l) \subseteq a(M, M+1)$.

2. if $1 < i$ or $j < l$,

if $X \notin a(M,M+1)$, suppose $a(i,j) \not\subseteq a(M,M+1)$,then $a(i,j)-a(M,M+1) \neq \emptyset$, so removing attributes in $a(M,M+1)$ will not disconnect t_i and t_j, but since $t_1,t_2 \ldots t_l$ is covered by $FD(\sigma)$, there must be some $k < i$ (or $k \geq j$) such that $a(k,k+1) \subseteq a(M,M+1)$. Since σ is simple, that is,

$$a(i_1,i_2) \bigcap a(i_3,i_4) \supseteq a(i_1,i_4)$$

so $a(k,k+1) = a(k,k+1) \bigcap a(M,M+1) \subseteq a(k,M+1)$, that is, $a(k,k+1) \subseteq a(k,M+1)$. Since $M \geq k+1$ (if $k < i$), we have a shorter chain $t_1,t_2 \ldots t_k,t_{M+1} \ldots t_j \ldots t_l$ that is covered by $FD(\sigma)$ (see figure 4), contradiction to that the original one is shortest, so $a(i,j) \subseteq a(M,M+1)$ for $i \leq M < j$.

if $X \in a(M,M+1)$,then

if $M=i, M+1=j$ then $a(i,j) \subseteq a(M,M+1)$,

if $i < M$ or $M+1 < j$, then from the case we have proved, for all $i \leq k < M$ or all $M+1 \leq k' < j$,

either $a(i,j) \subseteq a(k,k+1)$ (since $X \notin a(k,k+1)$),

or $a(i,j) \subseteq a(k',k'+1)$ (since $X \notin a(k',k'+1)$).

Since σ is simple ,by edge transitivity, we have $a(i,j) \subseteq a(M,M+1)$.⊣

Another way to state this theorem is that if $t_1,t_2 \ldots t_l$ is a chain covered by $FD(\sigma)$, deleting attributes in $a(i,i+1)$ $(1 \leq i < l)$,including those containing X, will disconnect T_{s_1} and T_{s_2}.

Lemma 8 : let C be a guarded set of edges of a hypergraph, an articulation set for C is an articulation set for the entire hypergraph.

Proof : see lemma 6.1 in [4].⊣

Lemma 9: Suppose EGD σ is β-acyclic and σ is simple, if $S_i \bigcap S_j$ is an articulation set of σ such that deleting attributes in $S_i \bigcap S_j$ produces connected components $C_1,C_2 \ldots C_k$, let $C_1',C_2' \ldots C_k'$ be $C_1,C_2 \ldots C_k$ with the deleted attributes added back, then the subhypergraph C_i' containing S_i and subhypergraph C_j' containing S_j both are guarded set of σ.

Proof: let node(H) be all the attributes in hypergraph H.Obviously,$\sigma = \bigcup_{1 \leq l \leq k} C_l'$. since C_i' and

C_j' are symmetric, we only prove for C_i' .

For any edge $S_r \in H_\sigma - C_i'$,let $S_r \in C_l'$, $l \neq i$, for any attributes $A \in S_r$:

case 1: if $A \in node\,(C_i')$, since node occuring in only one edge does not appear in the graph, so by simplicity of σ , A will connect C_i' and C_l' ,so $A \in S_i \bigcap S_j$,otherwise C_i and C_l will keep connected after deleting $S_i \bigcap S_j$, so $A \in S_i$.

case 2: if $A \notin node\,(C_i')$, then $A \notin S_r \bigcap node\,(C_i')$.

In all case,$S_r \bigcap node\,(C_i') \subseteq S_i$ ⊣

Lemma 10: Suppose σ is typed EGD and H_1 and H_2 are subhypergraph of H_σ. let $H_1 \Gamma H_2$ denote that H_2 is guarded set of H_1,then for subhypergraphs C_1,C_2 and C_3 of H_σ with $C_1 \supseteq C_2 \supseteq C_3$ (hyperedge containment), if $C_1 \Gamma C_2$ and $C_2 \Gamma C_3$, then $C_1 \Gamma C_3$.

Proof : Let $e_3 \in C_3$ be the guarded edge of C_3 with respect to C_2,let $e_2 \in C_2$ be the guard of C_2 with respect to C_1, for any $e_1 \in C_1$, we have

$$e_1 \bigcap node\,(C_3) \subseteq e_1 \bigcap node\,(C_2) \bigcap node\,(C_3) \subseteq e_2 \bigcap node\,(C_3) \subseteq e_3$$

so $e_1 \Gamma C_3$ ⊣

In the following, we will use G_σ instead of H_σ, and all the notations of hypergraph and lemmas about hypergraph are applicable to G_σ.

Theorem 11 : Suppose σ is reduced, simple EGD,then $FD\,(\sigma) \models \sigma$ iff σ is β-acyclic.

Proof :

only if

If $FD\,(\sigma) \models \sigma$, there exists a chain $t_1,t_2 \ldots t_l$ covered by $FD\,(\sigma)$. Since σ is reduced,$t_1,t_2 \ldots t_l$ are the only tuples in σ. Suppose there exists a β-cycle C in hypergraph of σ, C has at least one edge $(t_k\,,t_{k+1})$ on chain $t_1,t_2 \cdots t_l$ (see figure 5). From lemma 7, $a\,(i\,,j\,) \subseteq A\,(k\,,k+1)$, for any t_i and t_j that lay at opposite side of $(t_k\,,t_{k+1})$, all such pairs will be disconnected after removing attributes in $a\,(k\,,k+1)$,so C is not β-cycle, contradiction.So no β-cycle in hypergraph of σ, σ is β-acyclic.

if

Since σ is simple and reduced, the hypergraph of σ is exactly one component containing both t_{s_1} and t_{s_2}.

If $FD(\sigma) \models \sigma$, and since σ is simple and reduced, from the proof of lemma 7, σ has exactly one tuple t_{s_1} containing x_1 and one tuple t_{s_2} containing x_2. So if σ contains more than one t_{s_1} or more than one t_{s_2}, we can immediately conclude that $FD(\sigma) \not\models \sigma$, therefore, we only prove for the case that σ contains exactly one t_{s_1} and t_{s_2}. For any articulation set e_{ij}:

If removing e_{ij} does not disconnect t_{s_1} from t_{s_2},then work on the subhypergraph obtained by adding back the deleted attributes to the component containing t_{s_1} and t_{s_2}.

If removing e_{ij} disconnects t_{s_1} and t_{s_2} , work on the 2 subhypergraphs containing t_i and t_j ,respectively, mark e_{ij}

From lemma 9 and lemma 10, each proper subhypergraph we work on is guarded set of σ. Furthermore, since σ is β-acyclic and each guarded set is closed set ,and guarded set we are working on is connected, so it is also α-acyclic.From lemma 8, any its articulation set is also that of whole hypergraph of σ ,we search for articulation set in the subhypergraph, but treat this articulation set as that of σ, and classify it as above with respect to whole hypergraph. Therefore, by working on some subhypergraph,we should really means working on "most recent subhypergraph" contained in "global subhypergraph" suggested as above.

This is a recursive decomposition, each time ,we work on proper subhypergraph of provious one.(also a subhypergraph of σ) . Since σ has only finite number of edges, finally we will end up with the situation that the subhypergraph to be decomposited is single node for G_σ (or single hyperedge for H_σ) edge, in this case, our decomposition ends.

Since all the marked edges form connection between subhypergraphs in the decomposition hierarchy,all these edges are connected and since at any level of decomposition, t_{s_1} and t_{s_2} are in same or different "local subhypergraph",so t_{s_1} and t_{s_2} will be connected by these marked edges, then any path from t_{s_1} to t_{s_2} in these connected articulation sets is covered by $FD(\sigma)$. so $FD(\sigma) \models \sigma$.$\dashv$

Reference

[1] Jefferey D. Ullman, "principle of database systems"

[2] David Maier, "The theory of relational database"

[3] Ronald Fagin, "types of acyclicity for hypergraphs and relational databases schemes",Research Report, IBM Research Laboratory, San Jose, California 95193

[4] Catried Beeri, Ronald Fagin, David Maier and Mihalis Yannakakis, "On the desirability of acyclic database schemes" ,JACM,Vol.3, No. 3, July 1983, pp 479-513

$$\begin{array}{lll} & \overbrace{a_1\ a_2 \cdots a_k}^{\bigcup_{t_i \in S_1, t_j \in S_2} a(t_i, t_j)} \cdots b_1\ b_3 \cdots & \overset{x}{y_1} \\ s_1' & & \\ s_2' & a_1\ a_2 \cdots a_k \cdots b_2\ b_4 \cdots & y_2 \\ \hline & \qquad y_1 = y_2 & \end{array}$$

figure 1

$$T' = \left\{ \begin{array}{l} s_1' \\ s_2 \end{array} \right. \quad \begin{array}{ll} \overbrace{a_1\ a_2 \cdots a_\ell}^{W} \cdots & \overset{x}{y_1} \\ a_1\ a_2 \cdots a_\ell \cdots & y_2 \\ \hline \qquad y_1 = y_2 & \end{array}$$

figure 2

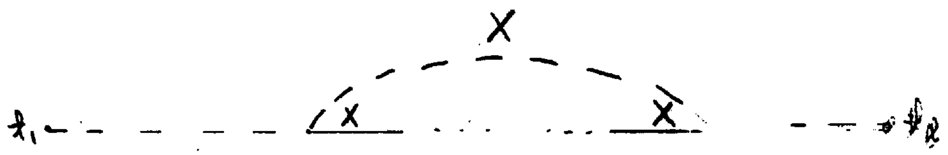


figure 3

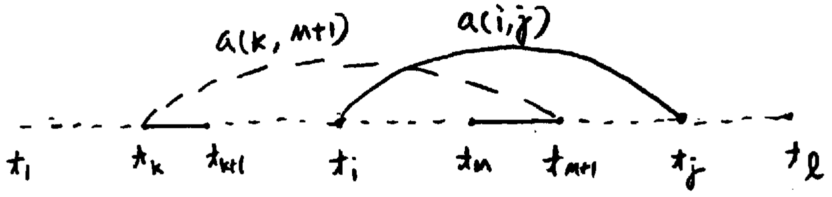


figure 4

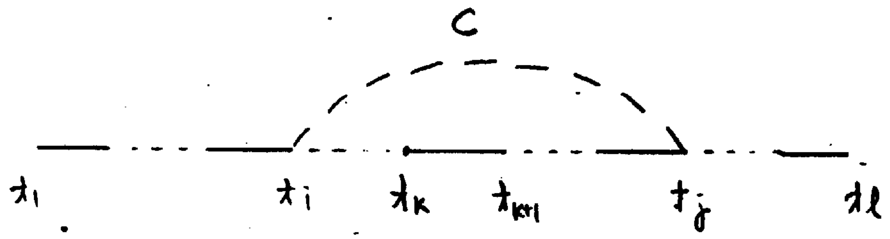


figure 5.

Constant Time Maintenance or The Triumph of the fd.

Marc H. Graham

Ke Wang

School of Information and Computer Science

Georgia Institute of Technology

Atlanta, GA 30332

1. Introduction.

Independence has been studied by many researchers in database theory. As mentioned by Beeri et. al. [BBG], independence meets the aesthetic principle of 'separation' or 'one thing in one place.' The research presented here constitutes an attack on the desirability of independence within the context of weak instance theory.

In all the contexts in which independence has been studied, it has the following essential description: Some set of 'local' properties is sufficient to guarantee some set of 'global.' properties. Within weak instance theory, it takes the following form: A database state within which each relation satisfies the dependencies local to it has a weak instance, i.e., is consistent. This problem does not arise in practice. In practice one does not encounter states about which only this local satisfaction property is known. One encounters instead the following problem: Given a state which is known to be consistent and a suggested modification to that state, should the modification be allowed; that is, will the modified state be consistent? We call this the maintenance problem.

As mentioned in [GY], very fast solutions to the maintenance problem are needed in practice. We assume that states are large and modifications are frequent. In such an environment, reading the entire state is not a viable solution to the maintenance problem. Thus 'very fast' must mean sublinear. One of our key results is that all sublinear solutions to the maintenance problem are constant time solutions. In other words, a very fast maintenance algorithm runs in time independent of the size of the database state. (This depends on our model of computation.)

Within weak instance theory, independence has been studied by Sagiv [S1] [S2], Graham and Yannakakis [GY] and Honeyman and Sciore [HS]. These authors restricted themselves to the case that the universal dependencies are a set of functional dependencies plus the join dependency for the scheme. In this case, independent schemes have a constant time maintenance algorithm.[1] It is not hard to see that there are schemes having a constant time maintenance algorithm which are not independent. The classic 'City, Address, Zip' problem is a case in point. The design algorithm of Biskup et. al. [BDB] produces the scheme $\{CAZ, ZC\}$ (recall the fd's are $CA \rightarrow Z$, $Z \rightarrow C$). This scheme is not independent, but it is constant time maintainable. (For example, to insert a tuple $\langle z, c \rangle$ into ZC, retrieve any tuple of the form $\langle c', a, z \rangle$ in CAZ and verify $c' = c$.)

We give below a characterization of constant time maintainable schemes *in the general case,* i.e., without restricting the set of universal dependencies. (We consider only typed dependencies here, primarily for notational convenience.) A key result is that the set of consistent states of such a scheme is axiomatized by a set of embedded functional dependencies only. (Were we to relax the restriction to typed dependencies, these would become binary egd's.) We then turn to the

[1] This does not hold in general. It does not hold if the local egd's are not binary.

case of fd's plus a single jd. We discuss a recognition algorithm and the maintenance algorithm.

Brosda and Vossen [BV] address concerns quite similar to ours. They use the modified foreign key constraint first introduced by Sagiv [S1]. They present a technique for verifying inserts in constant time. Their method of handling deletes, however, takes linear time in our model. For us, deletes are free and we do not take their constraint.

As shown by Sagiv [S3] and Atzeni and Chan [AC], schemes independent with respect to a set of fd's and a single jd are *algebraic*, that is, the canonical queries [GM] or projections of representative instances can be expressed in the relational algebra. Although we have yet to complete this phase of the research, it would seem that the same property holds for schemes constant time maintainable with respect to such a set of dependencies and indeed, by the same techniques. (This will not hold in the general case, by results of Sagiv.)

It seems then that the two concrete advantages of independent schemes are also advantages of constant time maintainable ones. Independent schemes are left with only the aesthetic advantage of separation. This is intuitively unsatisfying. Aesthetics should translate into practice.

(However, we must be honest. The recognition algorithm given below is exponential time. A lower bound is not known. Independence may reassert itself here, but at least one of the authors thinks it will not. Should the recognition problem prove intractable, that would be a concrete, but rather odd, justification of independence *viz a viz* constant time maintainability.)

2. Definitions and notation

In the interest of time and space, we will assume the reader to be familiar with the concepts of relational theory. In particular, we assume familiarity with the notions *tagged and untagged tableaux, functional and join dependencies, tuple and equality generating, single and multirelational dependencies, homomorphisms, satisfaction, closure of a set of attributes, embedded fd's, local satisfaction.* We denote by *SAT(D)* the set of all states, of the scheme in which the (possibly multirelational) dependencies of D are written, which satisfy D. We also assume the reader to be familiar with the basic concepts of weak instance theory. We denote by *CONS(*$\mathbf{R}$*,D)* the set of states of the scheme $\mathbf{R}$ which are consistent with (have a weak instance satisfying) D. Here D is *universal* for $\mathbf{R}$, that is, is a set of single relation dependencies over the underlying universe $\bigcup\mathbf{R}$.

Note that consistency is in its essence an existential, second order notion. We ask "Does there exist a weak instance for the state?" On the other hand, satisfaction is a first order notion. We ask "Is the state a model of D?" The two ideas are connected by the following result, due to Graham and Vardi:

Fact.[GV] For every $\mathbf{R},D$ there exists a set of equality generating dependencies Σ such that

$$CONS(\mathbf{R},D)=SAT(\Sigma)$$

Of course the dependencies in Σ are multirelational and their tableaux all use the scheme $\mathbf{R}$; whereas, the dependencies in D are all universal for $\mathbf{R}$. One should note that Σ is not necessarily recursive. By the same token, neither is $CONS(\mathbf{R},D)$ [GMV].

Now let T be a tableau or state.[2] For $\{x,y\}\subseteq T$ we define

[2] The terms "tableau" and "state" are synonymous in this paper.

$$v(x,y)=\{c \mid x[A]=c=y[A] \text{ for } \textit{some } A\}$$

$$v(x)=\bigcup_{\substack{y\in T\\ y\neq x}} v(x,y)$$

In short, $v(x)$ is the set of *symbols* or *values* of x which repeat. The function v can be extended further in a natural way so that we have

$$v(T)=\bigcup_{x\in T} v(x)$$

So, $v(T)$ is the set of repeating symbols of T. We also define a function a which returns attributes of values. For any set of values V, $a(V)=\{A \mid V\bigcap dom(A)\neq\emptyset\}$. We consider only disjoint attribute domains (the typed case) so a is well defined. The notation $a(x,y)$ abbreviates $a(v(x,y))$. Similarly for $a(x)$, $a(T)$. We often need to consider values, rather than attributes, as we will consider tableaux which are not *simple*, that is, in which more than one repeating symbol may appear in a given column. So v gives us more information than a does.

3. Extensions and extensibility.

Let $T=\{t_0, \ldots, t_m\}$ be a tableau. An *extension* of t_0 *in* T is a sequence $\langle u_0, \ldots, u_p\rangle$ where $u_0=t_0$ and $u_i\in T$. Duplicates are allowed in an extension.

We associate with an extension as defined above, a sequence of sets of attributes $\langle Y_0, \ldots, Y_p\rangle$ whose definition depends on a set of universal dependencies D. We set $Y_0=a(t_0)$ and for $0<i\leq p$

$$Y_i=(\bigcup\{a(u_j,u_i)\bigcap Y_j \mid j<i\})^+\bigcap R(u_i)$$

where we define

1) for any tuple *t*, *R(t)* is the relation or tag of *t*;

2) X^+ denotes the closure of X under the fd's implied by D.

The notion of an extension is a generalization of the notion of extension join defined by Honeyman [H].

Given an extension $E = \langle u_0, \ldots, u_p \rangle$, we define the set

$$v(E) = \bigcup_{i=0}^{p} \{u_i[A] \mid A \in Y_i\}$$

Fix, for the remainder of this paper, a scheme **R** and a set of unirelational dependencies D, which is universal for **R**. Let T be a tableau over **R** and let $t \in T$. We say t is ***extendible*** in T if there exists an extension of t in T, E such that $v(E) \supseteq v(T)$. The inclusion may be improper as $v(E)$ may contain non-repeating symbols of T. We say T is ***everywhere extendible*** if each $t \in T$ is extendible.

4. Model of Computation

We assume the database to be stored on an associative memory which responds to requests of the form

$$\langle R, \Psi \rangle$$

where R is a relation scheme (in **R**) and Ψ is a boolean combination of equality formula $A = a$ where $A \in R$ and $a \in dom(A)$. The memory responds by returning, if it exists, some tuple of R making Ψ true, where this is defined in the natural way. The request is said to ***succeed*** in this case. Otherwise it is said to ***fail***. We charge unit time for each request. We do this to discard from consideration the problems of data structuring. As we are interested primarily in lower bounds, these will carry over to the more realistic case.

5. The maintenance problem

Let Σ be the set of egd's such that $SAT(\Sigma)=CONS(\mathbf{R},D)$ (where $\mathbf{R}$, D were fixed earlier). Let $\langle T,x=y\rangle\in\Sigma$, $t\in T$. The maintenance problem $\langle t,\langle T,x=y\rangle,\mathbf{R},D\rangle$ is the decision problem which has as its set of instances

$$\{\langle u,\rho\rangle \mid u \ an \ R(t) \ tuple, \ \rho\in CONS(\mathbf{R},D)\}$$

A "yes" instance of $\langle t,\langle T,x=y\rangle,\mathbf{R},D\rangle$ is defined by: for every homomorphism $h:T\to\rho\bigcup\{u\}$, if $h(t)=u$ then $h(x)=h(y)$.

Suppose an algorithm $\mathbf{A}$ solves the $\langle t,\langle T,x=y\rangle,\mathbf{R},D\rangle$ problem. For any instance $\langle u,\rho\rangle$ of this problem define $\#\,\mathbf{A}(\langle u,\rho\rangle)$ to be the number of requests made by $\mathbf{A}$ on the instance $\langle u,\rho\rangle$. We say $\mathbf{A}$ solves $\langle t,\langle T,x=y\rangle,\mathbf{R},D\rangle$ in *constant time* if there exists an integer k such that

$$k\geq\#\,\mathbf{A}(\langle u,\rho\rangle) \text{ for } all \ \langle u,\rho\rangle$$

Theorem 1. There exists a constant time algorithm solving $\langle t,\langle T,x=y\rangle,\mathbf{R},D\rangle$ if and only if t is extendible in T.

For the proof of theorem 1 to work, it is necessary to make, without loss of generality, an assumption on the elements of Σ having the effect that $v(T)$ is not unnecessarily large. The assumption will also ensure that Σ is nonredundant.

The $\langle\langle T,x=y\rangle,\mathbf{R},D\rangle$ and $\langle\mathbf{R},D\rangle$ maintenance problems can be defined in the obvious way. We have as an immediate corollary of theorem 1:

Corollary. The $\langle\langle T,x=y\rangle,\mathbf{R},D\rangle$ maintenance problem can be solved in constant time if and only if T is everywhere extendible.

Less immediately, we also have

Theorem 2. The $\langle \mathbf{R}, D \rangle$ maintenance problem can be solved in constant time if and only if every element of Σ is everywhere extendible.

For theorem 2 we need to show that the condition of the theorem implies that Σ is finite. We show that the tableaux in Σ are simple.

The everywhere extendibility of a tableau T depends only on the set of embedded functional dependencies. We can show that, if T is everywhere extendible, the egd $\langle T, x = y \rangle \in \Sigma$ is implied by the embedded fd's. Therefore we have the following result, which gives us the subtitle of this paper.

Theorem 3. The $\langle \mathbf{R}, D \rangle$ problem has a constant time solution only if $CONS(\mathbf{R}, D) = CONS(\mathbf{R}, F)$ for a set of embedded, functional dependencies, F.

From inspection of the proof of theorem 1 we also have

Theorem 4. If the $\langle \mathbf{R}, D \rangle$ problem has no constant time solution, it has no sublinear solution.

6. The case of fd's and a single jd.

We now consider the case that D is of the form $F \bigcup *\mathbf{R}$ where F is a set of functional dependencies. We say that $F \bigcup *\mathbf{R}$ is *constant time maintainable*, or ctm, if the maintenance problem $\langle \mathbf{R}, F \bigcup *\mathbf{R} \rangle$ has a constant time solution.

Theorem 5. Let G be the set of functional dependencies implied by $F \bigcup *\mathbf{R}$. If $F \bigcup *\mathbf{R}$ is ctm, then $\mathbf{R}$ embeds a cover of G and $CONS(\mathbf{R}, F \bigcup *\mathbf{R}) = CONS(\mathbf{R}, G)$.

Note that theorem 2 of [GY] is a corollary of this result. We now give a brief discussion of an algorithm for determining if $F \bigcup *\mathbf{R}$ is ctm and another for solving the $\langle \mathbf{R}, F \bigcup *\mathbf{R} \rangle$ maintenance problem, if it is.

We borrow from the work of [GY]. The algorithm given there determines that $F \bigcup *\mathbf{R}$ is not independent by finding a non-trivial, multirelational egd in

Σ. We check this egd to see if its tableau is everywhere extendible. If it is, the algorithm must look for other violations of independence. We must be able to decide when the algorithm has seen enough.

Borrowing and modifying the notation of [GY], we construct a set of tableaux $\Upsilon=\{T_R^i(A) \mid R \in \mathbf{R}, A \in \bigcup \mathbf{R}, \textit{ i an integer}\}$ with the following property. For any state ρ of $\mathbf{R}$, for any $t \in \rho(R)$, if E is a non-trivial extension of t in ρ with $A \in a(E)$, there is an element $T_R^i(A) \in \Upsilon$ and a homomorphism $h: T_R^i(A) \rightarrow E$. Further, if we assume E is not unnecessarily long, then h is onto. (The tableau E is the set of tuples in the sequence E.) We can show $|\Upsilon|$ to be bounded by an exponential in $||F \bigcup *\mathbf{R}||$ as each $T_R^i(A)$ corresponds to a derivation of $R \rightarrow A$. (In fact, Υ need not be explicitly formed, so that the algorithm does not require exponential space. It remains to be seen whether it can be made to run in polynomial time.) Pairs of elements of Υ of the form $T_R^i(A)$, $T_R^j(A)$, $i \neq j$ form, in the manner of the proof of theorem 4 of [GY], the egd's we check for everywhere extendibility. (This is assuming $A \notin R$. If $A \in R$, each $T_R^i(A)$ is an egd.) We call this set of egd's Σ'. (Include the embedded fd's in Σ' ..) If the elements of Σ' are everywhere extendible, then we can show $SAT(\Sigma')=SAT(\Sigma)=CONS(\mathbf{R},F \bigcup *\mathbf{R})$.

First though, we show how to solve the $\langle \mathbf{R},F \bigcup *\mathbf{R} \rangle$ problem when $F \bigcup *\mathbf{R}$ is ctm. Given an instance $\langle u,\rho \rangle$ of $\langle \mathbf{R},F \bigcup *\mathbf{R} \rangle$, check that $\rho \bigcup \{u\}$ is locally satisfying and then extend u maximally in ρ. At this writing, this process requires having an embedded cover H of G where $H_i \equiv G^+ \mid R_i$, a cover which is hard to obtain. There is some hope that this problem can be made to disappear. This algorithm requires no more than $|H|$ requests and the algorithm accepts $\langle u,\rho \rangle$ if the extension reveals no contradiction. If that occurs, we can show $\rho \bigcup \{u\} \in SAT(\Sigma')$; that is, the extension of *no* tuple of ρ produces a contradiction in $\rho \bigcup \{u\}$.

To show $SAT(\Sigma')=CONS(\mathbf{R},F\bigcup\!*\mathbf{R})$, we follow the proof of theorem 5 of [GY]. We expand any element of $SAT(\Sigma')$ to a join consistent state. The key is to show that each step of the expansion preserves membership in $SAT(\Sigma')$. The join of the final state is a weak instance for the original state, as in [GY].

References

[AC] Paola Atzeni and Edward P. F. Chan, "Efficient Query Answering in the Representative Instance Approach," *PODS 85,* pp. 181-188

[BBG] Catriel Beeri, Philip A. Bernstein, Nathan Goodman, "A Sophisticate's Introduction to Database Normalization Theory," *VLDB 78* pp. 113-124

[BV] V. Brosda, G.Vossen, "Updating a Relational Database through a Universal Scheme Interface," *PODS 85,* pp. 66-75

[GM] Marc H. Graham, Alberto Mendelzon, "On the Power of Canonical Queries," unpublished

[GMV]Marc H. Graham, Alberto Mendelzon, Moshe Vardi, "Notions of Dependency Satisfaction," to appear, *JACM*

[GV] Marc H. Graham, Moshe Vardi, "On the Complexity and Axiomatizability of Consistent Database States," IBM RJ 4207, 1984

[GY] Marc H. Graham, Mihalis Yannakakis, "Independent Database Schemes," *JCSS, (28,1), 1984,* pp. 121-141

[H] Peter Honeyman, "Extension Joins," (publication data missing)

[HS] Peter Honeyman, Edward Sciore, "A New Characterization of Independence," (publication data missing)

[S1] Yehoshua Sagiv, "Can we use the Universal Instance Assumption without using Nulls?" *ACM-SIGMOD, 1981* pp. 108-120

[S2] Yehoshua Sagiv, "A Characterization of Globally Consistent Databases and their Correct Access Paths," *TODS (8,2) 1983,* pp. 266-286

[S3] Yehoshua Sagiv, "On Computing Restricted Projections of Representative Instances," *PODS 85* pp. 171-180

NATIONAL SCIENCE FOUNDATION
Washington, D.C. 20550

FINAL PROJECT REPORT
NSF FORM 98A

PLEASE READ INSTRUCTIONS ON REVERSE BEFORE COMPLETING

PART I—PROJECT IDENTIFICATION INFORMATION

1. Institution and Address Georgia Tech Research Corporation Georgia Institute of Technology Atlanta, GA 30332-0420	2. NSF Program IST	3. NSF Award Number 8217441
	4. Award Period From 6/1/83 To 11/30/85	5. Cumulative Award Amount $100,993

6. Project Title
"Canonical Queries As a Query Answering Device"

PART II—SUMMARY OF COMPLETED PROJECT *(FOR PUBLIC USE)*

The following reports and papers were prepared with partial support from this grant:

a. GIT-ICS-85/34, Graham, M.H., Griffeth, N.D. and Moss, E.; "Abstraction in Concurrency Control and Recovery Management" (November 1985);

b. GIT-ICS-84/22, Graham, M.H., Griffeth, N.D. and Smith-Thomas, B.; "Algorithms for Providing Reliability in Unreliable Database Systems" (October 1984);

c. GIT-ICS-84/20, Graham, M.H., Griffeth, N.D. and Moss, E.; "Recovery of Actions and Subactions in a Nested Transaction Systems" (October 1984);

d. GIT-ICS-84/12, Graham, M.H., Griffeth, N.D. and Moss, E.; "Committing and Aborting Subactions of Nested Transactions" (March 1984);

e. GIT-ICS-83/15, Graham, M.H., Griffeth, N.D. and Smith-Thomas, B.; "Reliable Scheduling of Database Transactions for Reliable Systems" (August 1983);

f. GIT-ICS-83/03, Graham, M.H., "Path Expressions in Databases" (February 1983);

g. GIT-ICS-82/04, Graham, M.H., "Functions in Database" (May 1982).

PART III—TECHNICAL INFORMATION *(FOR PROGRAM MANAGEMENT USES)*

1. ITEM *(Check appropriate blocks)*	NONE	ATTACHED	PREVIOUSLY FURNISHED	TO BE FURNISHED SEPARATELY TO PROGRAM	
				Check (✓)	Approx. Date
a. Abstracts of Theses					
b. Publication Citations					
c. Data on Scientific Collaborators					
d. Information on Inventions					
e. Technical Description of Project and Results					
f. Other *(specify)*					

2. Principal Investigator/Project Director Name *(Typed)* Marc H. Graham	3. Principal Investigator/Project Director Signature acting Dir. Unavailable for signature	4. Date 9/16/87

Form Approved OMB No. 3145-0058

ICS 85-37

an abbreviated version of this appeared in 1986 ACM-SIGMOD Intl Conf on Management of Data

Abstraction in Concurrency Control and Recovery Management

Marc H. Graham

Nancy D. Griffeth

School of Information and Computer Science

Georgia Institute of Technology

Atlanta GA 30332

J. Eliot B. Moss

Department of Computer and Information Science

University of Massachusetts

Amherst MA 01003

1. Introduction

The database literature contains many examples of actions on abstract data types which can be correctly implemented with nonserializable schedules of reads and writes. We mention one such example here.

Example 1. Consider transactions T_1 and T_2, each of which adds a new tuple to a relation in a relational database. Assume the tuples added have different keys. A tuple add is processed by first allocating and filling in a slot in the relation's tuple file, and then adding the key and slot number to a separate index. Assume that T_j's slot updating (S_j) and index insertion (I_j) steps can each be implemented by a single page read followed by a single page write (written RT_j, WT_j for the tuple file, and RI_j, WI_j for the index).

Here is an interleaved execution of T_1 and T_2:

$$RT_1 WT_1 RT_2 WT_2 RI_2 WI_2 RI_1 WI_1.$$

This is a serial execution of S_1 S_2 I_2 I_1. Now I_1 and I_2 clearly commute, since they are insertions of different keys to the index. Furthermore, I_1 cannot possibly conflict with S_2, since they deal with entirely different data structures. So the intermediate level sequence of steps is equivalent to the sequence S_1 I_1 S_2 I_2, which is a serial execution of T_1 T_2. We have demonstrated serializability of the original execution in layers, appealing to the meaning (semantics) of the intermediate level steps (S_j and I_j). But note that the sequence we gave may be a non-serializable execution of T_1 T_2 in terms of reads and writes, since the order of accesses to the tuple file and the index are opposite. If the same pages are used by both transactions, it will be a non-serializable execution. It is instructive also to observe that the sequence RT_1 RT_2 WT_1 WT_2 ... is not serializable even by layers. It does not correctly implement the intermediate operations S_1 and S_2.

A similar observation, which has received less attention, applies to recovery from action failure. The following example is an illustration of this interesting phenomenon.

Example 2. Consider T_1 and T_2 as defined above, but suppose that the index insertion steps I_1 and I_2 each require reading and possibly writing several pages (as they might, for example, in a B-tree). We now write $RI_j(p)$, $WI_j(p)$ for reading and writing index page p. Consider the following interleaved execution of T_1 and T_2:

$$RT_1WT_1RT_2WT_2RI_2(p)RI_2(q)\,WI_2(q)\,WI_2(r)\,WI_2(p)RI_1(p)\,WI_1(p)$$

The pair of index page writes $WI_2(q)\,WI_2(r)$ may be interpreted as a page split. This is serializable by layers, since at the level of the slot and index operations we are executing the sequence S_1 S_2 I_2 I_1, as in Example 1. But we encounter the following difficulty if we subsequently decide to abort T_2: The index insertion I_1 has seen and used page p, which was written by T_2 in its index insertion step. If we attempt to reproduce the page structure which preceded the page operations of T_2, we will lose the index insertion for T_1. Worse yet, if T_1 continues trying to operate on the index based on what it has seen of p, the structural integrity of the index could be violated. Thus it appears that we cannot reverse the page operations of T_2 without first aborting T_1. But there is still a way to reverse the index insertion of T_2, just by deleting the key inserted by T_2. Consider the following sequence:

$$S_1 \, S_2 \, I_2 \, I_1 \, D_2$$

The illustrated schedule is clearly correct, as long as the keys inserted by T_1 and T_2 are distinct, because we do not care whether the original page structure has been restored. We only need to restore the absence of the key in the index.

In this work, we present generalizations of serializability and atomicity which account for many such examples. The generalization arises from the observation that a transaction (or atomic action) is frequently a transformation on *abstract states* which is implemented by a sequence of actions on *concrete states*. The usual definition of serializability requires equality of concrete states. We call this *concrete serializability*, to distinguish it from equality of abstract states, which we call *abstract serializability*. Since many different concrete states in an implementation may represent the same abstract state, abstract serializability is a less restrictive correctness condition than concrete serializability. An immediate application of abstract serializability is to explain the correctness of apparently nonserializable schedules such as those described by Schwarz and Spector in [8] and by Weihl in [10]. If results returned by actions are considered part of the state, correctness conditions for read-only transactions, such as those described by Garcia-Molina in [2], can also be expressed.

The generalization of atomicity is analogous. The usual definition of an atomic action requires that it execute to completion or appear not to have happened at all. We introduce the idea of *abstract atomicity*, which is analogous to abstract serializability: A schedule of actions is abstractly atomic if it results in the same abstract state as some schedule in which only the non-aborted actions have run. *Concrete atomicity* corresponds to the more usual definition: the final state is the same as one that would have resulted from running only the concrete actions which were called by non-aborted abstract actions.

A widely accepted folk theorem states that it is necessary to use knowledge of the semantics of actions to achieve more concurrency than serialization allows. While we could address the semantics of specific atomic actions case by case, this is a tedious process. Instead, we describe a systematic method of using easily obtained knowledge about their semantics. A basic theorem of this paper, in a result related to the results of Beeri et. al. in [1], says that we can serialize at the individual levels of abstraction. Between levels, we need only to insure that the serialization order is preserved. Thus, in the above example, once the slot manipulation has been completed, locks on the page may be released. It is not necessary to wait until T_1 is complete. This has the effect of shortening transactions and thereby increasing concurrency and throughput. The analogous result holds for atomicity: we show that, for schedules which are serializable by layers, atomicity need only be enforced within each level of abstraction.

Another contribution is a much more realistic (but slightly more complicated) model than the usual straight-line model of transactions (as presented, for example, by Papadimitriou in [7]). The model presented here accounts for the flow of control in programs, such as "if-then-else" and "while" statements, without introducing nearly as much complexity as is present in [1]. The most interesting result involving the model is that, while it affects the classes of abstractly serializable and concretely serializable schedules in potentially profound ways, the class of CPSR schedules is essentially the same. This is because interchanges of non-conflicting actions preserves the flow of control within an action as well as the resulting state. It does not appear that any authors have previously addressed this issue.

The definitions of abstract and concrete serializability and atomicity do not suggest practical implementations. It is widely accepted, however, that the largest class of serializable schedules which is recognizable in any practical sense is the class of conflict-preserving serializable schedules. A similar situation may hold for atomicity. We define here a class of conflict-based atomic schedules which can be executed efficiently. This is the class of *restorable* schedules, in which no action is aborted before any action which depends on it. This class may be viewed as dual to the class of *recoverable* schedules defined by Hadzilacos in [4]: A schedule is recoverable if no action commits before any action which it depends on. In a restorable schedule, aborts can be efficiently implemented by executing state-based undo actions for each child action of an aborted action.

Finally, this work addresses a problem mentioned but not specifically addressed by Beeri et. al. in [1], which is the use of knowledge about abstract data types and state equivalence in serialization. The "fronts" of [1], which must be computed from an actual history of the system, can be determined in this context from information easily provided by a programmer: namely, from the call structure of the system and a "may conflict predicate" which describes which actions may conflict (i.e., not commute) with each other. The use of knowledge about abstractions and state equivalence permit description of legal interleavings in a simpler and more direct manner than in [1] or in Lynch's multi-level model in [6], where the set of legal interleavings must be given directly.

Similarly, the semantic information used for recovery can be provided easily by the programmer. The undos must themselves be actions (which will have to be coded if they are not "natural" actions for the abstraction). In each action, there must be a case statement which specifies the undo action for each set of states. For example, if the forward action is "Add key x to index I" then for the set of index states in which the index does not already contain x, the undo is "Delete key x from index I". For the set of index states in which the index already contains x, the undo action is the identity action.

2. The Model

We first describe the model for a single level of abstraction. The essential difference between this model and the straight-line model used by Papadimitriou in [7] is that the flow of control is reflected in the model. The essential difference between this model and those in [1] and [6] is that the construction of the set of legal interleavings is simple and visible in the model. Some notation will be needed to describe the levels of abstraction.

Notation: Let S_1 be an abstract state space and let S_0 be a concrete state space. Let A_1 be a set of abstract actions and A_0 be a set of concrete actions. Let $\rho:S_0 \rightarrow S_1$ be a partial function from concrete to abstract states. If $\rho(t) = s$ for concrete state t and abstract state s, then t ***represents*** s.

The intuition is that concrete states are used to represent abstract states and concrete actions are used to implement abstract actions. Not every concrete state represents a valid abstract state. Furthermore, the same abstract state may be represented by several different concrete states. However, we do expect that every abstract state is represented by some concrete state, that is, $\rho(S_0) = S_1$.

Actions map states to states according to a meaning function. The ***meaning function*** for a concrete [abstract] action is a function $m : A_0 \rightarrow 2^{S_0 \times S_0}$ [$m : A_1 \rightarrow 2^{S_1 \times S_1}$]. It is interpreted as follows: if $(s,t) \in m(a)$ for an action a then when executed on state s, the action a can terminate in state t. Actions are nondeterministic, that is, there may be more than one terminal state t for a given initial state s.

Abstract actions are implemented by programs over concrete actions. These programs generate sequences of concrete actions. For the sake of concreteness, we present one way of generating these sequences here. However, we do not assume that any particular method of generating the sequences is used. In proofs, we assume only that each program is associated with a set of sequences of concrete actions, which is the set of sequences the program would generate when running alone, and that new programs can be constructed from existing programs by concatenation. This operation amounts to running the first program to completion and then initiating the second program. The reader should note that when two programs run concurrently, one or both of them may generate a sequence of actions that would not be generated if they ran alone. Such sequences may be unacceptable.

A single concrete action is a program, and we will also regard any regular expression over actions as a program. We borrow notation from dynamic logic (see Harel, [5]) for a concise way to describe a program. If α and β are programs, new programs may be formed by concatenation ($\alpha ; \beta$ is a program); union ($\alpha \cup \beta$ is a program); or closure (α^* is a program).

The meanings of these constructs are defined recursively as follows:

The meaning of $\alpha ; \beta$ is to execute first α and then β:

$$m(\alpha ; \beta) = \{(s,t) | (\exists u)((s,u) \in m(\alpha) \text{ and } (u,t) \in m(\beta))\}.$$

Since concatenation of actions is clearly associative, we write $\alpha_1; \cdots ;\alpha_n$ for concatenation of n programs, ignoring the order of concatenation.

. The meaning of $\alpha \cup \beta$ is to execute either α or β:

$$m(\alpha \cup \beta) \;=\; m(\alpha) \cup m(\beta).$$

The meaning of α^* is to execute α zero or more times.

$$m(\alpha^*) \;=\; \{(s_0,s_n) | (\exists s_1,s_2,\cdots,s_{n-1})(\forall 1 \leq i \leq n)((s_{i-1},s_i) \in m(\alpha))\}.$$

Conditional execution of statements is modelled by actions which are identity on all states on which they are defined. These actions are called ***predicate actions*** and can be described by giving a predicate which is true for all states on which the predicate action is to be defined. For example the action $(x = 0)?$ is identity on all states in which the variable x is 0 and undefined elsewhere. The statement "if $x > 100$ then $x := x - 100$ else $x := 0$" is then modelled by $(p\ ;\ a) \cup (\ \bar{p}\ ;\ b)$, where p is the predicate action $(x > 100)?$, a is the action $x := x - 100$, and b is the action $x := 0$.

Notation: For any subset C of $S_0 \times S_0$ let

$$\rho(C) = \{(s,t)\ | (\exists (x,y) \in C)(\rho(x) = s \text{ and } \rho(y) = t)\}$$

We say that an abstract action is implemented by a program of concrete actions if ρ maps the meaning of the concrete program to the meaning of the abstract action. We will also require that if the program is initiated in a valid state then it must terminate in a valid state.

Definition: A concrete program α ***implements*** an abstract action a if and only if

(1) $m(a) = \rho(m(\alpha))$ and

(2) for every pair $(a,b) \in m(\alpha)$, if $\rho(a)$ is defined then $\rho(b)$ is also defined.

We now prove a technical lemma about implementations which will be useful in a subsequent section.

Lemma 1. Let a and b be abstract actions implemented by concrete programs α and β, respectively. Then $m(a;b) = \rho(m(\alpha;\beta))$.

Proof: First we show that $\rho(m(\alpha;\beta)) \subset m(a;b)$. Let $(s,t) \in \rho(m(\alpha;\beta))$. Then there are states c and d with $\rho(c)=s$ and $\rho(d)=t$ and $(c,d) \in m(\alpha;\beta)$. Thus there is a state b with $(c,b) \in m(\alpha)$ and $(b,d) \in m(\beta)$. Since α implements a and $\rho(c)$ is defined, $\rho(b)$ is also defined. Therefore, $(\rho(c),\rho(b)) \in \rho(m(\alpha)) = m(a)$ and $(\rho(b),\rho(d)) \in \rho(m(\beta)) = m(b)$. It follows from the definition of concatenation that $(s,t) = (\rho(c),\rho(d)) \in m(a;b)$.

Now we show that $m(a;b) \subset \rho(m(\alpha;\beta))$. Let $(s,t) \in m(a;b)$. There is a state $u \in S_1$ such that $(s,u) \in m(a)$ and $(u,t) \in m(b)$. Since $m(a) = \rho(m(\alpha))$ and $m(b) = \rho(m(\beta))$ there are states b, c, $d \in S_0$ such that $\rho(c) = s$, $\rho(d) = t$, and $\rho(b) = u$; $(c,b) \in m(\alpha)$; and $(b,d) \in m(\beta)$. Therefore $(c,d) \in m(\alpha;\beta)$ and $(s,t) = (\rho(c),\rho(d)) \in \rho(m(\alpha;\beta))$.

Corollary 1 to Lemma 1. Let a and b be abstract actions implemented by concrete programs α and β. Then the abstract action c having $m(c) = m(a;b)$ can be implemented by the concrete program $\gamma = \alpha;\beta$.

Proof: From Lemma 1, we have that $m(c) = m(a;b) = \rho(m(\alpha;\beta)) = \rho(m(\gamma))$. We need only show that if $(s,t) \in m(\gamma)$ and $\rho(s)$ is defined, then $\rho(t)$ is defined. But if $(s,t) \in m(\gamma)$ then $(s,t) \in m(\alpha;\beta)$ and therefore there is a $u \in S_0$ such that $(s,u) \in m(\alpha)$ and $(u,t) \in m(\beta)$. Assume that $\rho(s)$ is defined. Since α implements a, $\rho(u)$ is defined. Since β implements b, $\rho(t)$ is defined.

Corollary 2 to Lemma 1. Let $a_1, \cdots, a_n$ be abstract actions implemented by concrete actions $\alpha_1, \cdots, \alpha_n$. Then the abstract action c defined by $a_1; \cdots ;a_n$ can be implemented by the program $\alpha_1; \cdots ;\alpha_n$.

Proof: The proof is by a simple induction on the number of actions n.

Induction Base: If there is only one action a_1, the result is immmediate from the definitions.

Induction Hypothesis: For all sets of abstract actions of size less than or equal to $n-1$, the abstract action $a_1; \cdots ;a_{n-1}$ is implemented by the concrete program $\alpha_1; \cdots ;\alpha_{n-1}$.

Induction Step: By the induction hypothesis, $a_1; \cdots ;a_{n-1}$ is implemented by $\alpha_1; \cdots ;\alpha_{n-1}$. Using Corollary 1 to Lemma 1, we conclude that

$$a_1; \cdots ;a_{n-1};a_n = (a_1; \cdots ;a_{n-1});a_n$$

is implemented by

$$(\alpha_1; \cdots ;\alpha_{n-1});\alpha_n = \alpha_1; \cdots ;\alpha_{n-1};\alpha_n.$$

In keeping with the use of an initializing action in [7], we assume that the database has been initialized to concrete state I in the domain of ρ ($\rho(I)$ is the initial abstract state). It will often be useful to restrict the meaning function to those pairs whose initial state is I.

Notation: The restricted meaning function for program α is defined $m_I(\alpha)=\{(I,j) | (I,j) \epsilon m(\alpha)\}$. The restricted meaning function for abstract action a is defined $m_{\rho(I)}(a)=\{(\rho(I),\rho(j)) | (\rho(I),\rho(j)) \epsilon m(a)\}$.

If α implements a then $m_{\rho(I)}(a) = \rho(m_I(\alpha))$. Associated with each program is a set of possible computations of the program, one for each sequence of concrete actions which can be executed to completion.

Definition: A *computation* of an abstract action a having program α is a sequence $C = c_1; \cdots ;c_n$ of concrete actions in the regular set defined by the program, such that $m_I(C)$ is nonempty.

A computation of a set $a_1, \cdots ,a_n$ of concurrent abstract actions is an interleaving of the concrete actions in computations for $\alpha_1, \cdots ,\alpha_n$ which can be run to completion.

Definition: A *concurrent computation* of the set $a_1, \cdots ,a_n$ of abstract actions is an interleaving C of computations of the individual actions such that $m_I(C)$ is nonempty.

3. Serializable Computations

3.1. Serializability of Abstract Actions

The set of concurrent computations for a collection of actions will in general be hard to characterize. It may be even harder to characterize the ones which are correct. We discuss a relatively simple subset of these computations, those that behave, in some sense, like serial (non-interleaved) computations.

Definition: A *(complete) log* L is a set A_L of abstract actions, a sequence C_L of concrete actions, and a mapping $\lambda_L : C \rightarrow A$ such that $\lambda_L(c)$ is the abstract action $a \in A_L$ on whose behalf c is run. C_L is a prefix of some concurrent computation of A_L. A *partial log* L is a log in which C_L is a prefix of a concurrent computation of A_L.

Definitions are stated and results proved for complete logs unless otherwise stated. Usually, the extension to partial logs is trivial.

Notation: We will write $m(C_L)$ for $m(c_1; \cdots ;c_n)$ where $C_L = \{c_1, \cdots ,c_n\}$ and we assume that c_i precedes c_j for $i < j$.

We will write $c <_L d$ when c precedes d in the sequence C_L.

We consider serial computations to be correct.

Definition: Consider a log L containing abstract actions $A_L = \{a_1, \cdots ,a_n\}$ implemented by programs $\{\alpha_1, \cdots ,\alpha_n\}$. The log L is *serial* if C_L is a computation of the program $\alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)}$ for some permutation π of $\{1, \cdots ,n\}$.

We also consider a computation to be correct if it results in an abstract state that would result from some serial log. The following definition allows the use of knowledge about abstractions in determining the correctness of an interleaving. Depending on the abstraction, this can be a very different class of interleavings from those that would ordinarily be viewed as serializable.

Definition: A log L is *abstractly serializable* if and only if there is a permutation π of $\{1, \cdots ,n\}$ such that $\rho(m_I(C_L)) \subset m_{\rho(I)}(a_{\pi(1)}; \cdots ;a_{\pi(n)})$.

The next definition defines a class of serializable logs more closely related to the usual class of serializable schedules.

Definition: A log L is *concretely serializable* if and only if there is a permutation π of $\{1, \cdots ,n\}$ such that $m_I(C_L) \subset m_I(\alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)})$.

Definition: For both abstract and concrete serializability, the sequence $\pi(1), \cdots ,\pi(n)$ is called the *serialization order* of L.

A partial log L is serial (concretely serializable, abstractly serializable) if there is a complete serial (concretely serializable, abstractly serializable) log M such that C_L is a prefix of C_M.

Concrete serializability, which requires that concrete states be the same, is more restrictive than abstract serializability, which requires only that abstract states be the same.

Theorem 1: If the log L is concretely serializable then it is abstractly serializable.

Proof: Let $A_L = \{a_1, \cdots, a_n\}$ and let α_i implement a_i. Since L is concretely serializable, there is a permutation π of $\{1, \cdots, n\}$ such that

$$m_I(C_L) \subset m_I(\alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)}).$$

We define an abstract action $b = a_{\pi(1)}; \cdots ;a_{\pi(n)}$. By Corollary 2 to Lemma 1, b can be implemented by the concrete program $\beta = \alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)}$. In other words,

$$m(a_{\pi(1)}; \cdots ;a_{\pi(n)}) = m(b) = \rho(m(\beta)) = \rho(m(\alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)})).$$

It follows from this that

$$\begin{aligned}\rho(m_I(C_L)) &\subset \rho(m_I(\alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)})) \\ &= m_{\rho(I)}(a_{\pi(1)}; \cdots ;a_{\pi(n)})).\end{aligned}$$

This theorem can easily be extended to partial logs. For a partial log L which is concretely serializable, there is a concretely serializable complete log M such that C_L is a prefix of C_M. By the above theorem, M is also abstractly serializable; hence L is abstractly serializable.

Concrete serializability is not identical to SR as defined in [7] because of the nondeterminism and because it is necessary to check that the reordered collection of actions is a computation. If abstract actions are implemented only by straight-line programs, as in [7], then any serial schedule of the concrete actions in a concurrent computation is still a computation. But this is not the case in our model. Consider abstract actions A_1 and A_2, where $A_1 = ((x \leq 0)? ; (y := 1)) \cup ((x > 0)? ; (y := 2))$ and $A_2 = (x := 1)$. Suppose that in the initial state x is 0. The sequence

$$(x := 1) ; (x \leq 0)? ; (y := 1)$$

is not a computation, although any other interleaving of these concrete actions is. Thus we cannot interchange actions of a computation arbitrarily and expect the result to remain a computation. A subsequent lemma gives one mechanism by which we can verify that a transformation of a computation is still a computation.

It should be noted that this model reduces to the model in [7] if the concrete actions are deterministic reads and writes with the obvious meanings assigned to them and if all programs are constructed by concatenation only. It was shown in [7] for these concrete actions that concrete serializability is NP-complete. Without more information about the semantics of the actions, however, and about the abstraction function, we cannot say anything about the complexity class of either concrete or abstract serializability.

For this reason, neither abstract nor concrete serializability has significance as a definition of a class of schedules which we can recognize. However, abstract serializability is a valuable correctness condition for explaining the correctness of schedules such as the one in the opening example. In a subsequent section, we generalize this use of abstract serializability to explain the correctness of a large class of schedules, many of which are not concretely serializable. But first, we translate another standard serializability result to the new model of program execution.

Definition: Actions a and b *commute* if $m(a\ ;\ b) = m(b\ ;\ a)$. Otherwise, a and b *conflict.*

Definition: Let C and D be sequences of concrete actions. We say that $C \approx D$ if they are identical except for interchanging the order of two nonconflicting concrete actions, that is, actions c and d such that $m(c;d) = m(d;c)$. The transitive, reflexive closure of $\approx$ is denoted by $\approx*$.

The following lemma provides the basic mechanism for establishing that a permuted computation is still a computation. We use it to verify that a serial (non-interleaved) sequence of concrete actions could actually have been requested by the given atomic actions, that is, it is a semantically as well as syntactically valid sequence of actions.

Lemma 2: If L is a log and if $D \approx* C_L$ and D is constructed from C_L by interchanging nonconflicting operations c and d such that $\lambda(c) \neq \lambda(d)$, then there is a log M with

$A_M = A_L$, $C_M = D$ and $\lambda_M = \lambda_L$. Furthermore, $m(C_L) = m(C_M)$.

Proof: There are sequences of concrete actions γ and δ such that $C_L = \gamma;c;d;\delta$ and $D = \gamma;d;c;\delta$. Therefore

$$m(C_L) = \{(s,t) \mid (\exists u,v)((s,u) \in m(\gamma) \text{ and } (u,v) \in m(c;d) \text{ and } (v,t) \in m(\delta)\}.$$

Since $m(c;d) = m(d;c)$ and $\lambda_L(c) \neq \lambda_L(d)$, we have that

$$\begin{aligned} m(D) &= \{(s,t) \mid (\exists u,v)((s,u) \in m(\gamma) \text{ and } (u,v) \in m(d;c) \text{ and } (v,t) \in m(\delta)\} \\ &= m(C_L). \end{aligned}$$

Therefore D is a computation of A_L (or prefix of a computation of A_L) exactly when C_L is, and M is a log exactly when L is. Since we did not use the completeness of L, the results hold for either complete or partial logs.

Definition: Logs L and M are ***equivalent*** if $A_L = A_M$, $\lambda_L = \lambda_M$, and $C_L \approx^* C_M$. If L is equivalent to M for a serial log M, then L is ***conflict-preserving serializable***.

Theorem 2: If a log L is conflict-preserving serializable, then it is concretely serializable.

Proof: Let $A_L = \{a_1, \cdots, a_n\}$. If L is conflict-preserving serializable then there is a serial log M such that $A_M = A_L$, $C_M \approx^* C_L$, and $\lambda_M = \lambda_L$. By a simple induction using Lemma 2 to prove the induction step, $m_I(C_L) = m_I(C_M)$.

Suppose that a_i is implemented by α_i. By the definition of a serial log, there is a permutation π of $\{1, \cdots, n\}$ such that $C_M = \alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)}$. Hence for this permutation

$$m_I(C_L) = m_I(C_M) = m_I(\alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)}).$$

Therefore L is concretely serializable.

3.2. Layered Serializability

In this section, the definitions of serializability are extended to multiple levels of abstraction and the basic result on serializability is stated. We make two simplifying assumptions; how to weaken them will be discussed subsequently. The assumptions are:

(1) the levels of abstraction are totally ordered; and

(2) an action calls subactions belonging to the next lower level of abstraction only.

We assume a system with n levels of abstraction.

Notation: The concrete state at level i is S_{i-1}. The abstract state is S_i. The abstraction mapping at level i is $\rho_i : S_{i-1} \rightarrow S_i$. The set of concrete actions is C_i. The set of abstract actions is $A_i = \{a_{i,1}, \cdots a_{i,k_i}\}$. The number of abstract actions at level i is k_i. Concrete actions at level i are abstract actions at level $i-1$. Thus $C_i = A_{i-1}$.

Given a collection A_n of top-level actions, concurrent execution of the actions is described by a collection of logs.

Definition: A *complete system log* L is a set of complete logs $L_1, \cdots, L_n$ such that L_i is a complete log for level i and the concrete actions in the log L_i are the same as the abstract actions in the log L_{i-1}. A *partial system log* L is a set of partial logs $L_1, \cdots, L_n$ such that L_i is a partial log for level i and the concrete actions in the log L_i are a subset of the abstract actions in the log L_{i-1}. The *top-level log* for a system log L consists of the top-level abstract actions (A_n), the bottom-level concrete actions (C_1), and the mapping from concrete to abstract actions constructed by composing $\lambda_1, \cdots, \lambda_n$.

Definition: The system log L is *abstractly (concretely) serializable by layers* if each L_i is abstractly (concretely) serializable and there is a serialization order on A_{i-1} which is the same as the total order on C_i. We will denote this serialization order π_i.

The following theorem justifies the practice of "serializing by layers", that is, providing serialization for the individual levels of abstraction and forgetting subaction conflicts (e.g., releasing locks) as soon as the action at the next higher level is complete.

Theorem 3: If a system log L is abstractly serializable by layers then its top-level log is abstractly serializable.

Proof: Assume first that L is complete. Then by the definition of abstract serializability by layers, the following holds for each i:

$$\rho_i(m_I(C_{L_i})) \subset m_{\rho_i(I)}(a_{i,\pi_i(1)}; \cdots ; a_{i,\pi_i(k_i)})$$

where π_i gives the serialization order, and $A_{L_i} = C_{L_{i+1}} = a_{i,\pi_i(1)}; \cdots ; a_{i,\pi_i(k_i)}$. It follows by induction on the number of levels that

$$\rho_1 \circ \cdots \circ \rho_n(m_I(C_{L_n})) \subset m_{\rho_1 \circ \cdots \circ \rho_n(I)}(a_{n,\pi_n(1)}; \cdots ; a_{n,\pi_n(k_n)})$$

If L is partial, then we can extend the sequence of concrete actions to a computation having the above properties. Thus the result also holds for partial logs.

Corollary 1 to Theorem 3: If a system log **L** is concretely serializable by layers, then its top-level log is abstractly serializable.

Proof: By Theorem 1, the log is abstractly serializable by layers. It follows immediately from Theorem 3 that the log described is abstractly serializable.

Definition: If a system log is serializable by layers and if each log L_i is conflict-preserving serializable, then the set of logs is called ***conflict-preserving serializable by layers*** (LCPSR). Since all practical serialization methods recognize only subsets of the set of CPSR logs, the following two results are the interesting ones, from the practical point of view.

Corollary 2 to Theorem 3: If a system log **L** is conflict-preserving serializable by layers then its top-level log is abstractly serializable.

Proof: By Theorem 2, the system log is concretely serializable by layers. Hence it is abstractly serializable by layers and the result follows from Theorem 3.

Theorem 4: Membership in LCPSR can be tested in time $O(c+a^2)$ where c is the number of concrete actions in the system log and a is the number of abstract actions in the system log.

Proof: For each i, construct the conflict graph for level i as described in [9]. The nodes of this graph are the abstract actions in A_{L_i}. There is an edge from node a to node b if there are concrete actions $c,d \in C_{L_i}$ such that $\lambda(c) = a$, $\lambda(d) = b$, c and d conflict, and $c <_{L_i} d$. This graph can be constructed in time proportional to the number of actions in

C_{L_i}. If the graph is acyclic, then level i is CPSR. Acyclicity can be tested in time proportional to the square of the number of actions iin A_{L_i}.

It only remains to test whether there is a serialization order π_i on level i which is consistent with the order $<_{L_{i+1}}$. This can be tested at the time the edges are added to the graph for level i: if there is an edge from a to b then there is a serialization order consistent with $<_{L_{i+1}}$ if and only if $a <_{L_{i+1}} b$.

In practice, the only order that would be known for a system log would be the order on C_1. The order $<_{L_i}$ is any topological sort of the order given by the conflict graph for level $i-1$. Any topological sort is acceptable, because if there is no sequence of edges between a and b then there is no conflict between any children of abstract actions a and b in a computation of $\{a,b\}$, so that $\lambda_{L_i}^{-1}(a);\lambda_{L_i}^{-1}(b) \approx* \lambda_{L_i}^{-1}(b);\lambda_{L_i}^{-1}(a)$. Also, there can be no other conflicting actions between any children of a and b. Therefore, a and b can be viewed as having executed in either order.

3.3. Ordering the Layers

We are not usually given a linearly ordered collection of levels of abstraction in a system. Instead we may have *packages* of actions. We expect that there will be pairs of actions within a single package may conflict. Usually, actions in different packages will not conflict, but there are exceptions. Consider a relational database which may be accessed by two packages: one of the packages consists of relational operators, the other of matrix operators. We can imagine relations which are entirely numerical which may be accessed by both packages. Thus operations may conflict between packages.

We describe, intuitively, how to determine a linear collection of levels. We require that all actions in a single package are at the same level. Also, any two packages containing actions which may potentially conflict must be at the same level. Finally, two packages must be at the same level if they have members which recursively call each other.

To compute a linear order which satisfies these conditions, draw a directed graph representing the call structure of the system: if an action in package A calls an action in package B, then there is an edge from A to B. Add edges in both directions between packages containing potentially conflicting actions. Collapse all cycles in this graph to a single node (these cycles represent either conflict or mutual recursion or a combination), and label the new node by the set of packages on the cycle. The resulting acyclic graph defines a partial order on sets of packages. This partial order can be converted to a total order by picking an equivalence relation on the node labels which is a *congruence* with respect to the partial order: that is, if $P_1 < P_2$ in the partial order, then for every $Q_1 \equiv P_1$ and $Q_2 \equiv P_2$, $Q_1 < Q_2$.

Our second simplifying assumption was that an action only calls subactions which are at the next lower level of abstraction. But in practice, actions may call subactions at the same level or may skip several levels. In the former case, we may treat the calls to the same level as "invisible", and use only calls to the next lower level of abstraction in serializing. (In fact, this is the current practice: there are two levels, the top level and the read/write level. Only calls to reads and writes are noticed by the serialization mechanism.) In the latter case, we may insert subactions at each intervening level which do nothing but call the next lower level.

4. Recovery from Action Failure

One method of enforcing serializability is to abort actions which violate serializability constraints, and every practical serialization technique sometimes uses aborts for this purpose. Thus serialization contains the possibility of action failure and it is necessary to guarantee correct recovery from failure to guarantee serializability. The converse is not true, and so we initially consider failure atomicity without assuming serializability.

The rest of this paper discusses recovery from the failure of a single action by eliminating its partial effects. Two methods of eliminating partial effects are in common use. One is to roll the action back by *undoing* each change it has made. The other is to restore the system from a checkpoint taken prior to initialization of the action, *redoing* each subsequent concrete action other than those called by the aborted action. We develop the conditions which permit use of *redos* in section 4.1 and the conditions which permit use of *undos* in section 4.2. In both sections, we assume a single level of abstraction.

In section 4.3, the results are extended to a multi-level system and a result analogous to the result for layered serializability is stated. In a multi-level system, serializability is required to establish that the required sequence of concrete actions in a level of abstraction was implemented by the next lower level.

4.1. Aborting Actions

An abstract action is not inherently atomic, since it is implemented by a sequence of concrete actions. If it fails after execution of some of the concrete actions, then the effects of those actions which have been completed must be eliminated. The process of eliminating any partial effects of a failed abstract action will be referred to below as an abort of the action.

To abort an action correctly, it is necessary to change the current state to a state that could have occurred if the action had not executed at all. Let LOGS be the set of all logs. (Remember that a log L consists of a set A_L of abstract actions, a sequence C_L of concrete actions, and a mapping $\lambda_L : C \rightarrow A$.) We define an operator which chooses a concrete abort action when it is given a log and abstract action to be aborted:

$$ABORT : LOGS \times A \rightarrow (S_0 \rightarrow S_0).$$

The abort must restore some state which could have occurred in executing the abstract actions in $A_L - \{a\}$.

Definition: An action generated by the ABORT operator is called an *abort*. An action is said to be *aborted* if its last action is an abort.

A log which contains aborts should appear to be a log which contains all of the non-aborted actions and none of the aborted actions. We call such a log abstractly atomic.

Definition: A complete log L is *abstractly atomic* if there is a complete log M having the following properties:

(1) $A_M = A_L - \{a \mid a \text{ is aborted in } L\}$ and

(2) $\rho(m_I(C_L)) \subset \rho(m_I(C_M))$.

Note that we have not required that the logs be serializable. Any computation will do according to the above definition. Later, to achieve "layered atomicity", we will assume serializability.

Definition: A complete log L containing aborted actions is ***concretely atomic*** if it there is a complete log M having the following properties:

(1) $A_M \;= A_L - \{a \mid a \text{ is aborted in } L\}$;

(2) $m_I(C_L) \subset m_I(C_M)$.

We extend the definition of atomicity to partial logs in the obvious way.

Definition: A partial log L is abstractly (concretely) atomic if there is a complete abstractly (concretely) atomic log M such that $A_M = A_L$, C_L is a prefix of C_M, and λ_L is λ_M restricted to C_L.

It follows immediately from the definitions that concrete atomicity implies abstract atomicity.

One way to implement abstract atomicity is to restore state I and rerun the actions in A_M. The state I then serves as a checkpoint. However, an arbitrary choice of M in the above definition may require re-running the abstract actions, not just the concrete actions. In an on-line, high-volume transaction system, this is not a practical method. The programs for the abstract actions may not even be available after they terminate. In such a system, we want aborts to be simpler. For this reason we will require that the log M have a very simple relationship to the log L, in fact, that C_M is simply C_L minus the children of aborted actions. In this case, we can restore a final state for $m_I(C_L - \lambda_L^{-1}(a))$ to implement atomicity.

Notation: As long as it is clear what log is involved, we will write $ABORT(a)$ for $ABORT(L,a)$.

Definition: Let L be a log in which action a has not been aborted. $ABORT(a)$ is a ***simple abort*** of a for L if $m_I(C_L\,;ABORT(a)) \neq \phi$ and $m_I(C_L\,;ABORT(a)) \subset m_I(C_L - \lambda_L^{-1}(a))$.

Clearly, a simple abort of action a in log L exists if and only if $m_I(C_L - \lambda_L^{-1}(a))$ is a prefix of some computation of A_L. The following definitions lead to a characterization of logs and actions for which simple aborts exist.

Notation: Given a log L and action $c \epsilon C_L$, let $BEFORE(c)$ be the partial log having concrete actions $C_{BEFORE(c)} = \{b \mid b \epsilon C_L \text{ and } b <_L c\}$, abstract actions A_L, and mapping $\lambda_{BEFORE(c)}$ which is the restriction of λ_L to the set $C_{BEFORE(c)}$. Let $C_{AFTER(c)} = \{b \mid b \epsilon C_L \text{ and } c <_L b\}$. (Note that in general we cannot define a log $AFTER(c)$.)

The following definition says that an abstract action b depends on an abstract action a if it has a concrete subaction which follows and conflicts with a concrete subaction of a. If an action b depends on an action a, and if we restrict ourselves to simple aborts, then it may be necessary to abort b when a is aborted.

Definition: An action b *depends on* an action a in a log L if there is some $d \epsilon \lambda_L^{-1}(b)$ and some $c \epsilon \lambda_L^{-1}(a)$ such that d follows c in the order of C_L, a is not aborted in the log $BEFORE(d)$, and d and c conflict.

Definition: An action a of a log L is *removable* if no action depends on it. A log L is *restorable* if every aborted action is removable.

Restorability may be viewed as a dual condition to *recoverability*, which requires that no action be committed before any action which it depends on. Restorability says that no action is aborted before any action which depends on it.

Definition: Let C be a sequence of actions ordered by $<$ and let $F \subset C$. F ***is final in*** C if for every $f \epsilon F$ and $c \epsilon C - F$ either $c < f$ or f and c commute.

Note that the set $\lambda_L^{-1}(a)$ is final in C_L for any removable action a. It follows from this that it is the terminal subsequence of some sequence $D \approx* C_L$.

Lemma 3: If action a of log L is removable, then $C_L - \lambda_L^{-1}(a)$ is a prefix of a computation of A_L.

Proof: We will show by induction on the number of actions in any final set F of operations of C_L, that $C_L - F$ is a prefix of a computation. The lemma then follows from the fact that $\lambda_L^{-1}(a)$ is final in C_L for all removable actions a.

Induction Base (F contains only 1 action): Let $F = \{c\}$. Then $C_L = \gamma;c;\delta$ for some sequences γ and δ, such that for every $d \in \delta$, $m(c;d) = m(d;c)$. Hence $C_L \approx* \gamma;\delta;c$ and therefore $C_L - \{c\} = \gamma;\delta$ is a prefix of a computation.

Induction Hypothesis: For every final set F in C_L, if $|F| < n$, then $C_L - F$ is a prefix of a computation of A_L.

Induction Step: Suppose $|F| = n$. Let $F' = F - \{c\}$, where c is the first (or minimal) element of F with respect to $<_L$. Then F' is final in C_L and by the induction hypothesis, $C_L - F'$ is a prefix of a computation. Since c does not conflict with any later action in $C_L - F'$, we can use reasoning similar to the case $n = 1$ to show that $C_L - F' \approx* C_L - F;c$ and therefore $C_L - F$ is a prefix of a computation.

Since $C_L - \lambda_L^{-1}(a)$ is a prefix of a computation of $A_L - \{a\}$, we can restore checkpoint I and rerun all actions in $C_L - \lambda_L^{-1}(a)$ in the order given by $<_L$. In fact, the checkpoint can be taken at any point before the initialization of a. Let c be the first action of a. Let $d \in \{c\} \cup C_{BEFORE(c)}$. Then there is a state t such that $(I,t) \in m(C_{BEFORE(d)})$ and $m_t(C_{AFTER(d)} - \lambda_L^{-1}(a)) \neq \phi$. Any such state t can be used as a checkpoint state.

Lemma 3 can be applied inductively to show that if no dependencies were formed on abstract actions before they were aborted by a simple abort, then atomicity is guaranteed.

Theorem 5: If L is restorable and if every abort in L is simple, then L is atomic.

Proof: Let $\{a_1, \cdots, a_n\}$ be the set of aborted actions. Construct the log M such that $A_M = A_L - \{a_1, \cdots, a_n\}$, $C_M = C_L - \lambda_L^{-1}(\{a_1, \cdots, a_n\})$, and $\lambda_M = \lambda_L$ restricted to C_M. Since L is restorable, every aborted action in L is removable. Using Lemma 3 inductively, we see that $C_L - \lambda_L^{-1}(\{a_1, \cdots, a_n\})$ is a prefix of a computation of A_M. This verifies that M is a log.

Now we must verify that $m_I(C_L) = m_I(C_M)$. To do this, we observe that there exist $\gamma_1, \cdots, \gamma_{n+1}$ such that

$$C_L = \gamma_1;ABORT(a_1);\gamma_2;ABORT(a_2); \cdots ;\gamma_n;ABORT(a_n);\gamma_{n+1}.$$

The meaning of C_L is given by

$$m_I(C_L) = \{(I,t) \mid (\exists u)((I,u) \in m_I(\gamma_1;ABORT(a_1)) \text{ and } (u,t) \in m_I(\gamma_2;ABORT(a_2);\cdots;ABORT(a_n);\gamma_{n+1}))\}$$

But by the hypothesis of the theorem, every abort is simple, so that

$$m_I(\gamma_1;ABORT(a_1,L)) \subset m_I(\gamma_1 - \lambda_L^{-1}(a_1)).$$

and therefore

$$m_I(C_L) \subset m_I(C_L - \lambda_L^{-1}(a_1)).$$

Proceeding inductively, we see that $m_I(C_L) \subset m_I(C_L - \lambda_L^{-1}(\{a_1, \cdots, a_n\}) = m_I(C_M)$.

Theorem 5 suggests a general procedure for aborting actions. When an action a is to be aborted, abort the set of actions

$$D(a) = \{b \mid b \textit{ depends on } a\} \cup \{a\}.$$

The abort is done by restoring any concrete state which existed prior to the first concrete action in $\lambda_L^{-1}(D(a))$ and then re-running the actions in $C_L - \lambda_L^{-1}(D(a))$ from that point on.

4.2. Rolling Back Actions

A potentially much faster implementation than checkpoint/restore would simply roll back the concrete actions in the computation of an aborted action a. For this purpose, we define an UNDO operator on concrete actions which chooses an inverse concrete action to perform the roll back. The plan is to implement the ABORT operator on abstract actions as a sequence of UNDO actions, one for each concrete action called by the abstract action, applied in reverse order of execution of the concrete actions.

$$UNDO: C \times S_0 \rightarrow (S_0 \rightarrow S_0)$$

This *UNDO* operator chooses a state-dependent inverse action which will transform the current state to the state in which the forward action was initiated. Thus we must define the *UNDO* so that $m(c \,;\, UNDO(c,t) = \{(t,t)\}$. It follows from this definition that if c is the last concrete action in C_L and $(I,t) \in m(C_L - \{c\})$ then $m(C_L \,;\, UNDO(c,t)) = \{(I,t)\}$. Furthermore, if

$(I,t) \notin m(C_L - \{c\})$ then $m(C_L \; ; \; UNDO(c,t)) = \delta$. In other words, if the final action c was initiated in state t, then $UNDO(c,t)$ restores the state to t and to nothing else.

Actually, to undo an action c, it is not actually necessary that c be the last action of C_L, only that c is rot followed by any action which conflicts with $UNDO(c,t)$ for the state t in which c was initiated. This is stated in the following lemma.

Lemma 4: If the following conditions hold:

(1) $c \in C_L$;

(2) $(I,t) \in m(C_{BEFORE(c)})$;

(3) no action of $C_{AFTER(c)}$ conflicts with $UNDO(c,t)$; and

(4) $UNDO(c,t) \notin C_{AFTER(c)}$

then

$$m_I(C_L \; ; \; UNDO(c,t)) = \{(I,u) \mid (t,u) \in m(C_{AFTER(c)})\}.$$

Proof: By the definitions of $C_{BEFORE(c)}$ and $C_{AFTER(C)}$, $C_L = C_{BEFORE(c)};c;C_{AFTER(c)}$. By the hypothesis of the Lemma, for every $d \in C_{AFTER(c)}$,

$$m(d;UNDO(c,t)) = m(UNDO(c,t);d)$$

and

$$C_L;UNDO(c,t) \approx * \; C_{BEFORE(c)};c;UNDO(c,t);C_{AFTER(c)}.$$

It follows that

$$\begin{aligned} m(C_L;UNDO(c,t)) &= m(C_{BEFORE(c)};c;UNDO(c,t);C_{AFTER(c)}) \\ &= \{(s,w) \mid (\exists u,v)((s,u) \in m(C_{BEFORE(c)}) \\ &\quad \text{and } (u,v) \in m(c;UNDO(c,t)) \text{ and } (v,w) \in m(C_{AFTER(c)})\} \\ &= \{(s,w) \mid (s,t) \in m(C_{BEFORE(c)}) \text{ and } (t,w) \in m(C_{AFTER(c)})\}. \end{aligned}$$

Therefore, $m_I(C_L;UNDO(c,t)) = \{(I,u) \mid (t,u) \in m(C_{AFTER(c)})\}$.

The sequence of concrete actions called by an aborted abstract action a in a complete log L should be a prefix $c_1; \cdots ;c_k$ of a computation $c_1; \cdots ;c_n$ of a followed by $UNDO(c_k,t_k); \cdots ;UNDO(c_1,t_1)$. We extend the definition of concurrent computations to allow such sequences.

Definition: The concurrent computations of a set A of abstract actions include all interleavings C of sequences $c_1; \cdots ;c_k;UNDO(c_k,t_k); \cdots ;UNDO(c_1,t_1)$ such that

(1) $c_1; \cdots ;c_n$ is a computation of $a \in A$ for some $n>k$;

(2) $m_I(C) \neq \phi$;

(3) there is at most one $UNDO$ action in C for each $c \in C$;

(4) if there is an action $UNDO(c,t)$ for $c \in C$ then c precedes $UNDO(c,t)$ in C and $(I,t) \in m_I(C_{BEFORE(c)})$.

(5) each concrete action is called by exactly one abstract action.

Definition: If an action a has called an $UNDO$ then we say that a is *aborted* and is *rolling back*. If it has called an $UNDO$ for every forward action it called, then we say that a is *rolled back*.

The definition of a log is unchanged except for the expanded set of computations.

Definition: The *rollback of action a depends on* action b in a log L if there is a child c of a and a child d of b such that $c <_L d$; $UNDO(c,t) \notin C_{BEFORE(d)}$ and $UNDO(d,w) \notin C_{BEFORE(UNDO(c,t))}$; and d conflicts with $UNDO(c,t)$.

Definition: A log L is *revokable* if for each action $a \in A_L$, the rollback of a does not depend on any $b \in A_L$.

Theorem 6: If a complete log L is revokable then it is atomic.

Proof: We show that if L is revokable then $m_I(C_L) \subset m_I(C_M)$ for the log M with

$$A_M = A_L - \{a \mid a \text{ is rolled back in } L\} \text{and}$$
$$C_M = C_L - \{c \mid UNDO(c,t) \in C_L\} - \{UNDO(c,t) \mid t \in S_0\}.$$

Since for a complete log L, $A_M = A_L - \{a \mid a \text{ is aborted in } L\}$, it follows that L is atomic.

The proof is by induction on the number k of $UNDOs$ in C_L.

Induction Base ($k=1$): Let c be the action with $UNDO(c,t) \in C_L$ and let $\lambda_L(c)=a$. Because L is revokable, there is no action b such that the rollback of a depends on b. In other words, for every concrete action d in C_L, if $c <_L d <_L UNDO(c,t)$ then d commutes with $UNDO(c,t)$. This implies that

$$C_L \approx* C_{BEFORE(c)};c;UNDO(c,t);C_{AFTER(c)}$$

and therefore

$$\begin{aligned} m_I(C_L) &= m_I(C_{BEFORE(c)};c;UNDO(c,t);C_{AFTER(c)}) \\ &\subset m_I(C_{BEFORE(c)};C_{AFTER(c)}) \\ &= m_I(C_M). \end{aligned}$$

Induction Hypothesis: If there are fewer than k $UNDOs$ in C_L, then $m_I(C_L) \subset m_I(C_M)$ for some log M with

$$\begin{aligned} A_M &= A_L - \{a \mid a \textit{ is aborted in } L\}. \\ C_M &= C_L - \{c \mid UNDO(c,t) \in C_L\} - \{UNDO(c,t) \mid t \in S_0\}. \end{aligned}$$

Induction Step: Suppose there are k $UNDOs$ in C_L. Consider the first $UNDO$ in the order $<_L$. Suppose that it is $UNDO(c,t)$. Since it is the first, there is no $UNDO(d,w)$ such that $c <_L UNDO(d,w) <_L UNDO(c,t)$. Since L is revokable, $UNDO(c,t)$ commutes with every action d such that $c <_L d <_L UNDO(c,t)$. Therefore, using the same reasoning as for the induction base, and applying the induction hypothesis,

$$\begin{aligned} m_I(C_L) &\subset m_I(C_{BEFORE(c)};C_{AFTER(c)}) \\ &\subset m_I(C_M). \end{aligned}$$

If the log L is partial, we can extend L to a complete log by adding UNDOs for every incomplete action to the end of the log. The order of the UNDOs should be the reverse of the order of the forward actions. The new log is complete and revokable, therefore by Theorem 6 it is atomic.

Theorem 6 suggests the following algorithm for aborting actions. If the rollback of an action will not depend on any action in A_L, then executed a sequence of $UNDOs$ in reverse order of the forward actions. If the rollback will depend on some action, recursively abort the action on which the rollback will depend. Of course, the cascaded aborts can be avoided. To avoid them, it is necessary to block an abstract action if a rollback-dependency would develop.

4.3. Layered Atomicity

In this section, we describe the correct abortment of actions in a multi-level system. As in section 3.3, suppose that we have a system log **L** $= \{L_1, \cdots ,L_n\}$. To guarantee that the sequence of concrete actions at level i+1 is implemented by the abstract actions at level i, we must be able to say that there is an ordering on the non-aborted abstract actions in A_{L_i} which is the same as the ordering on these actions when they are viewed as concrete actions at level i+1. But this requires that each level be both serializable and atomic.

Definition: Let L be a complete log containing aborted actions. Let $A_L - \{a \mid a$ *is aborted in* $L\} = \{a_1, \cdots a_n\}$. L is ***abstractly serializable and atomic*** if there is a permutation π of $\{1, \cdots ,n\}$ such that

$$\rho(m_I(C_L)) \subset m_{\rho(I)}(a_{\pi(1)}; \cdots ;a_{\pi(n)}).$$

L is ***concretely serializable and atomic*** if there is a permutation π of $\{1, \cdots ,n\}$ such that

$$m_I(C_L) \subset m_I(\alpha_{\pi(1)}; \cdots ;\alpha_{\pi(n)}).$$

This is similar, in combining the aspects of computational atomicity with failure atomicity, to Weihl's definition of atomicity [10]. As usual, concrete serializability and atomicity implies abstract serializability and atomicity.

Definition: A system log **L** is ***abstractly serializable and atomic by layers*** if each log L_i is abstractly serializable and atomic; $C_{L_{i+1}} = A_{L_i} - \{a \mid a$ *is aborted in* $L_i\} = \{a_{i,1}; \cdots ;a_{i,k_i}\}$; and there is a serialization order π_i on level L_i such that $C_{L_{i+1}} = a_{i,\pi_i(1)}; \cdots ;a_{i,\pi_i(k_i)}$.

Theorem 7: If a system log **L** is abstractly serializable and atomic by layers then its top-level log is abstractly serializable and atomic.

Proof: The proof is by induction on the number n of levels.

Induction Base: If there is only one level, then the top-level log is identical to the log for that level and is therefore abstractly serializable and atomic by the definition of layered serializability and atomicity.

Induction Hypothesis: The top-level log is abstractly serializable and atomic if the system log is abstractly serializable and atomic by layers and there are fewer than n levels.

Induction Step: Suppose that the system log has n levels. By the definition of layered serializability and atomicity the level 1 log is abstractly serializable and atomic. Therefore there is a log M such that $A_M = A_{L_1} - \{a \mid a \textit{ is aborted in } L_1\}$ and

$$\rho_1(m_I(C_{L_1})) \subset \rho_1(m_I(C_M))$$
$$= m_{\rho_1(I)}(a_{1,\pi_1(1)}; \cdots ; a_{1,\pi_1(k_1)}).$$

By the definition of layered serializability and atomicity $C_{L_2} = a_{1,\pi_1(1)}; \cdots ; a_{1,\pi_1(k_1)}$. Therefore

$$m_{\rho_1(I)}(a_{1,\pi_1(1)}; \cdots ; a_{1,\pi_1(k_1)}) = m_{\rho_1(I)}(C_{L_2}).$$

Applying the induction hypothesis to the system log M consisting of the logs $L_2, \cdots, L_n$, the top level log for M is abstractly serializable and atomic, that is,

$$\rho_2 \circ \cdots \circ \rho_n(m_{\rho_1(I)}(C_{L_2})) \subset m_{\rho_1 \circ \rho_2 \circ \cdots \circ \rho_n(I)}(C_N)$$

for some log N with $A_N = A_{L_n} - \{a \mid a \textit{ is aborted in } L_n\}$. It follows that

$$\rho_2 \circ \cdots \circ \rho_n(m_{\rho_1(I)}(C_{L_1}) \subset m_{\rho_1 \circ \rho_2 \circ \cdots \circ \rho_n(I)}(C_N)$$

for this same log N.

Corollary 1 to Theorem 7: If each level of a system log L is serializable and restorable, then its top-level log is abstractly atomic.

Corollary 2 to Theorem 7: If each level of a system log L is serializable and revokable, then its top-level log is abstractly atomic.

5. Conclusions and Further Work

In summary, we have shown that, with respect to both serializability and failure atomicity, the correctness of atomic actions can be assured by guaranteeing their correctness at each level of abstraction. The result for serializability alone follows from the results presented by Beeri et. al. in [1]; but the relative simplicity of the proofs presented

3 Vassos Hadzilacos, *An Operational Model for Database System Reliability,* Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (March 1983), 244-257.

4 Theo Haerder and Andreas Reuter, *Principles of Transaction-Oriented Database Recovery,* ACM **Computing Surveys,** volume 15, number 4 (December 1983), 287-318.

5. David Harel, **First-Order Dynamic Logic,** Lecture Notes in Computer Science, ed. by G. Goos and J. Hartmanis (Springer-Verlag, New York, 1979).

6. Nancy A. Lynch, *Multilevel Atomicity -- A New Correctness Criterion for Database Concurrency Control,* ACM **Transactions on Database Systems,** volume 8, number 4 (December 1983), 484-502.

7. C. H. Papadimitriou, *Serializability of Concurrent Updates,* **Journal of the ACM,** volume 26, number 4 (October 1979), 631-653.

8. Peter M. Schwarz and Alfred Z. Spector, *Synchronizing Shared Abstract Types,* ACM **Transactions on Computer Systems,** volume 2, number 3 (August 1984), 223-250.

9. Jeffrey D. Ullman, **Principles of Database Systems,** (Computer Science Press, 1982).

10. William E. Weihl, *Specification and Implementation of Atomic Data Types,* PhD. Dissertation, MIT/LCS/TR-314, March, 1984.

Annual Report 1984
NSF-IST-82-17441
Canonical Queries
Marc H. Graham, PI

Georgia Tech ICS graduate students Ke Wang, Steve Ornburn, Gita Rangarajan, and Amy Lapwing were supported under this grant for different periods.

The paper "The Power of Canonical Queries" was written with Alberto Mendelzon. The paper "On the Complexity and Axiomatizability of Consistent Database States," with Moshe Vardi appeared in Principles of Database Systems (PODS) March 84. These papers deal with the expressive power of Canonical Queries.

The Power of Canonical Queries

by

Marc H. Graham*÷

and

Alberto O. Mendelzon**÷÷

September 1983

*School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

**CSRG
University of Toronto
Toronto, Ontario
Canada

÷Research supported in part by the National Science Foundation grant IST 8217441

÷÷Research supported in part by the Natural Science and Engineering Research Council of Canada.

This document was prepared with a Xerox Star 8010 system.

1. Introduction

The approach to formal data semantics which has come to be called "weak instance theory" began with the work of Honyeman [H] as a means of integrating the relations of a multirelation database for the purpose of checking constraints. It was soon recognized, by Sagiv [S] and also by Yannakakis [Y], that the theory leads naturally to an extremely powerful and concise query language, called window functions by some authors [MRW] [MUV] and called canonical queries here. This paper analyzes the expressive power of these queries. We show that canonical queries are expressible in first order logic if we allow infinitely many axioms. These axioms are shown to be full multirelational implicational dependencies as defined by Fagin [F]. The bulk of the paper is concerned with characterizing the case when canonical queries may be finitely axiomatized in first order logic. This occurs precisely when the query is expressible in the relational algebra.

Maier, Ullman and Vardi [MUV] have considered these questions as well. The present work was done independently and differs from their work in the following ways.

Proposition 1 establishes the existence of representative instances even when the chase is not guaranteed to terminate. We consider the result of a non-terminating chase to be undefined. In the same spirit, compactness is not used in the proof of Lemma 1. Proposition ~~3~~6. although a simple observation, does not appear in [MUV]. ?

A new definition of boundedness appears in Theorems 1 and 2. In these theorems we take up a problem ignored in [MUV]: the behaviour of canonical queries on inconsistent states. We do this for a technical reason: a relational algebra expression is defined for every state, canonical queries only for consistent ones. Theorems 1 and 2 present two different attacks on this problem. In Theorem 1, we remove all equality generating dependencies in the manner suggested by Beeri and Vardi [BV]. All states are then consistent and the results of canonical queries on the subset of originally consistent states are preserved. In Theorem 2, we consider queries which distinguish consistent and inconsistent states.

The proof of the "hard part" of these results (the implication 5⇒4 in Theorem 1) is made very easy by the presence of Lemma ~~2~~3. This lemma establishes a well known piece of folklore as fact.

2. Definition and Notation

2.1 Basic Definitions

We begin with a finite set of attributes which we denote U and call the universe. Following standard notation in the area, upper case letters near the front of the alphabet: A, B, A_i, A_j, ... indicate single attributes. Those toward the end of the alphabet: X, Y, ... represent sets of attributes. The set $\{A\}$ is often denoted A and $X \cup Y$ is written XY. To each $A \in U$ is associated an infinite domain $dom(A)$. distinct A_i, A_j in U, $dom(A_i)$, $dom(A_j)$ are either identical or disjoint.

Let $R \subseteq U$. R is then a relation scheme. A tuple for R is a function assigning to each $A \in R$ a value in $dom(A)$. The term row will be freely used for tuple in certain contexts. If $X \subseteq R$, $t[X]$ denotes the restriction of the function t to the attributes in X.

A relation instance I for R is a set of tuples for R. The size of an instance I, denoted $|I|$, is the number of tuples it contains. We often restrict our attention to finite relations. If $X \subseteq R$, the projection onto X of an instance I of R, denoted

$\Pi_X(I)=\{t[X] \mid t\in I\}$. If $\mathbf{R}$ is a collection of relation schemes of U, then $\mathbf{R}$ is a database schema for U. We do not in general insist that $\mathbf{R}$ cover U, *i.e.*, that $\cup\mathbf{R}=U$.

A state ρ of a schema $\mathbf{R}$ is an assignment to each $R\in\mathbf{R}$ of an instance for R. The size of a state is the sum of the sizes of its instances:

$$|\rho| = \sum_{R\in\mathbf{R}} |\rho(R)|$$

We define an inclusion relation among states in a natural way. If ρ, σ are states of $\mathbf{R}$, $\rho\subseteq\sigma$ if $\rho(R)\subseteq\sigma(R)$ for every $R\in\mathbf{R}$. If I is an instance and $\mathbf{R}$ a schema for U, we define $\Pi_{\mathbf{R}}(I)$ to be the state ρ of $\mathbf{R}$ given by $\rho(R)=\Pi_R(I)$ for each $R\in\mathbf{R}$.

A tableau for a universe U is an instance for U over an extended set of domains. For each $A\in U$, we form the tableau domain $tdom(A)$ from $dom(A)$ by adding infinitely many variables. Elements of $\cup_{A\in U} dom(A)$ are called constants. If $dom(A_i)=dom(A_j)$, then $tdom(A_i)=tdom(A_j)$, else the sets are disjoint. From now on the term instance will denote a tableau without variables. A tagged tableau for U is a tableau over the set of attributes $U\cup\{Tag\}$ where Tag is an attribute assumed not to be in U. Further, $tdom(Tag)$ is disjoint from any attribute domain in U. The values in $tdom(Tag)$ will always be given a special interpretation, namely as relation schemes.

Tableaux provide a uniform notation for the expression of data dependencies, conjunctive queries and database states.

A (unirelational) data dependency for a universe U is a pair $d=<T,x>$ where T is a tableau over U and x is one of the following if x is an equality assertion of the form $a = b$, then d is an equality generating dependency or egd. (It is conventional to assume the symbols a, b appear in the tableau T.)

if x is a tableau then d is a tuple generating dependency, or tgd. If every symbol appearing in the tableau x also appears in the tableau T, then d is a total or full dependency. In this case x may be assumed to contain (or abusively, to be) a single tuple. Otherwise, if x contains symbols not in T, then d is partial or embedded.

This definition of dependency is based on the work of Beeri and Vardi [BV]. The parallel to the implicational dependencies of Fagin [F] is immediate. The quantifier "unirelational" in the above indicates that these dependencies can be written in a first order language with a single predicate letter of arity the cardinality of U. The class of multi-relational dependencies can be captured through the use of tagged tableaux, as follows.

A multi-relational data dependency for a database schema **R** over a universe U, $d = <T, x>$ is a data dependency for U in which T, and x if d is a tgd, are tagged tableaux. Two extra conditions are imposed: (i) the tags of T and x are relation schemes of **R**: $\Pi_{Tag}(T \cup x) \subseteq \mathbf{R}$; (ii) tuples may agree only as they are allowed to by their tags: $t[A] = u[B]$ implies $A \in t[Tag]$ and $B \in u[Tag]$. (Of course, $t[A] = u[B]$ is possible only if $tdom(A) = tdom(B)$.) Finally, a multi-relational tgd $d = <T, S>$, is considered full if for each $s \in S$ and each $A \in s[Tag]$, $s[A]$ appears in T.

Let ρ be a state of a schema **R** over universe U. The tableau of ρ T_ρ, is a tagged tableau over U defined as follows (this definition gives T_ρ only up to isomorphic renaming of variables): For each $R \in \mathbf{R}$, each $t \in \rho(R)$, a row v of T_ρ has $v[R] = t$. For each $A \in U\text{-}R$, $v[A]$ is a variable appearing nowhere else in T_ρ. Finally, $v[Tag] = R$ and no other rows appear in T_ρ.

A <u>query</u> on a schema $\mathbf{R}$ with target list X, is a function from states of $\mathbf{R}$ to instances of X. A <u>conjunctive query</u> q on a schema $\mathbf{R}$ with target list X is a full multi-relational tgd on schema $\mathbf{R} \cup \{X\}$. If $q = <T,x>$, then T is a tagged tableau on $\mathbf{R}$, x is a single tuple and $x[Tag] = X$. To define the function described by a conjunctive query, we introduce the idea of a homomorphism.

Define $Sym(T) = \cup_{A \in U}(\Pi_A(T))$ where T is a tableau on universe U. A <u>homomorphism</u> on T is any function with domain $Sym(T)$. If η is a homomorphism on T, then we allow η to also represent its extension [illegible] and tableaux. That is, $\eta(t) = \eta \cdot t$ (the composition of η and t), [illegible] $t \in T\}$. A homomorphism *preserves* a set of symbols C if it is the identity on C. A constant preserving homomorphism preserves the set of constants (recall this is the set $\cup_{A \in U} dom(A)$). A <u>tag preserving</u> homomorphism is a homomorphism extended by the identity on $tdom(Tag)$.

The relation between tableaux which is central to this paper is that of <u>homomorphic embeddability.</u> If T, S are tableaux on a universe U, then T is homomorphically embeddable into S if there exists a homomorphism η on T such that $\eta(T) \subseteq S$. T and S are homomorphically equivalent if each may be embedded into the other. For certain applications, we may require the homomorphism to preserve some set of symbols. If either of T or S or both are tagged and η is non-tag preserving we may write $\eta(T) \subseteq S$ to mean $(\eta(\Pi_U(T)) \subseteq (\Pi_U(S))$. In some circumstances the set of all homomorphisms embedding T into S is of interest, as in the following definition.

A tableau T_0 satisfies a dependency $d=<T,x>$ if for every homomorphism η embedding T into T_0

if x is the equality assertion $a=b$, then $\eta(a)=\eta(b)$;

if x is the tableau S, then η can be extended to a homomorphism μ on $Sym(S \cup T)$ (i.e. μ restricted to $Sym(T)$ is η) with $\mu(S) \subseteq T_0$.

(For multirelational dependencies we may consider only tag preserving homomorphisms.) A tableau satisfies a set of dependencies D if it satisfies each dependency in D.

We can now describe conjunctive queries as functions. Let $q=<T,\{t\}>$ be a conjunctive query for a schema $\mathbf{R}$ and ρ a state of $\mathbf{R}$. The relation $q(\rho)=\{\eta(t) \mid \eta$ *a tag preserving homomorphism embedding* T *into* $T_\rho\}$. It is customary to further restrict the homomorphisms to be constant preserving. When that is done, the class of conjunctive queries includes all queries expressible by relational algebra expressions using a restricted form of selection, projection and product [CM]. Union may be modeled by considering finite sets of conjunctive queries [SY].

The chase [ABU], [MMS] is a fundamental process in the study of databases. It is a means of transforming, if possible, an arbitrary tableau into one which satisfies a given set of dependencies. Let $d=<T,x>$ be a dependency and η a homomorphism on T. The pair $\iota=<d,\eta>$ is called a transformation. If S is a tableau and η embeds T into S, then ι is said to be enabled. The application of an enabled transformation ι to a tableau S, denoted $\iota(S)$ is a tableau whose definition depends on the nature of the dependency d.

If d is an egd, so that x is $a=b$, then one of the symbols $\eta(a)$, $\eta(b)$ replaces the other <u>everywhere it appears in</u> S. It is customary to give a disambiguating rule for the choice of the replacement. When $\eta(a)$, $\eta(b)$ are distinct constants, ι is a contradiction and it is usual to assign $\iota(S)=\varnothing$.

If d is a tgd, so that $d=<T,V>$, then η is extended to a homomorphism μ on $T \cup V$ and $\iota(S)=S \cup \mu(V)$. The extension of η to μ is restricted so that μ is one-to-one on $Sym(V)-Sym(T)$ and for each $y \in Sym(V)-Sym(T)$, $\mu(y) \notin Sym(S)$, that is, $\mu(y)$ is a new variable.

It is customary to denote $chase_D(T)$ as the limit of the process of applying transformations whose dependencies are chosen from the set D, starting with the tableau T. If D contains only full dependencies <u>and</u> a disambiguating rule is given for the application of egds (see above), $chase_D(T)$ is unique and effectively computable. Otherwise, it is at best defined only upto isomorphism and whenever D contains partial dependencies, this limit is not clearly defined.

2.2 Consistency, weak instances, canonical queries

Let ρ be a state of a schema **R** over a universe U. Let D be a set of dependencies over U. Following Honeyman [H], see also [GMV], we define a <u>weak instance</u> for ρ with respect to D as an instance I of U such that $\rho \subseteq \Pi_{\mathbf{R}}(I)$ <u>and</u> I satisfies D. We denote the set of all such finite weak instances as $weak(D,\rho)$ and we say ρ is <u>consistent</u> with D if $weak(D,\rho) \neq \varnothing$. We denote the set of all states of **R** consistent with D as $CONS(\mathbf{R},D)$.

Let $Sym(\rho) = \cup_{R \in \mathbf{R}}(\cup_{A \in R}(\Pi_A(\rho(R))))$ be the set of all symbols appearing in the state ρ. A representative instance for ρ with respect to a set of dependencies D is a uppercase

italics

possibly infinite weak instance for ρ such that every element of weak(D,ρ) is the image, under some Sym(ρ) preserving homomorphism, of the representative instance. We can show that every consistent state has a representative instance.

Proposition 1. If $\rho \in CONS(\mathbf{R},D)$, then ρ has a representative instance. ~~Further, all representative instances for ρ are equivalent via $Sym(\rho)$ preserving homomorphisms.~~

Proof. ~~Let I, J be elements of $weak(D,\rho)$.~~ We take the direct product of the elements of *weak*(uppercase *i*,ρ). This is an instance over the universe U for which the attribute domains, to be denoted $xdom(A)$, are sequences of countable length. It is convenient and customary to consider these sequences as functions on the set of natural numbers, N. So for the instance we have for each A, $xdom(A)=\{f|f:\mathrm{N}\rightarrow dom(A)\}$. However, we may identify in $xdom(A)$, that function f such that $f(i)=a$ for each $i \in \mathrm{N}$ with the element $a \in dom(A)$. This allows us to consider $xdom(A)$ as an extension of $dom(A)$.

Let I be this direct product. Its definition requires that we number the elements of $weak(D,\rho)$. Having done so, we have by definition

$$\mathrm{I} = \{<f_1, \ldots f_m> \mid <f_1(i), \ldots f_m(i)> \in \mathrm{I}_i \in weak(D,\rho) \text{ for every } i \in \mathrm{N}\}$$

It is well known that dependencies are preserved under direct products, that is, I satisfies D.

Further, for $I_i \in weak(D,\rho)$ each, the natural map

$$\eta_i : <f_1, f_2 \ldots f_m> \vdash <f_1(i), \ldots f_m(i)>$$

replace $\eta_i(f) = f(i)$ italics

uppercase

homomorphically embeds I onto I_i. η_i is $Sym(\rho)$ preserving since $f=a$ iff $f(i)=a$ for every i; therefore $\eta_i(f)=a$. It remains to show $\rho \sqsubseteq \pi_R(\mathrm{I})$.

italics Uppercase

for any $R\in\mathbf{R}$, let $u\in\rho(R)$. Each $I_i\in weak(D,\rho)$ contains a tuple with $u_i[R]=u$. Therefore I contains a tuple u such that $u[R]=u$. ⊣ remove

This proposition is stronger than the results of Honeyman [H], Sagiv [S], Mendelzon [Me] and Maier, Ullman and Vardi [MUV] in that it does not depend on the chase. As noted by those authors, $chase_D(T_\rho)$, if it exists, is homomorphically ~~a representat~~ive embeddable into every element of weak(D, ρ) i~~nstance for ρ~~.

Let $X\subseteq U$. The canonical query on X with respect to schema $\mathbf{R}$ and set of dependencies D, denoted $?X[\mathbf{R},D]$ or just $?X$ when $\mathbf{R}$ and D are known from context, is a function from $CONS(\mathbf{R},D)$ to instances of X defined by

$$?X[\mathbf{R},D](\rho)=\bigcap_{I\in weak(D,\rho)}(\Pi_X(I))$$

The definition does not provide an effective computation of $?X[\mathbf{R},D]$ and indeed such a computation may not exist. However, the representative instance can be used to compute $?X[\mathbf{R},D]$ when it can itself be effectively found. Define C-projection Π^C as projection with respect to elements of C only:

$$\Pi_X^C(T)=\{t\mid t\in\Pi_X(T)\text{ and }t[A]\in C\text{ for each }A\in X\}$$

Proposition 2. If $\rho\in CONS(\mathbf{R},D)$ and I is a representative instance for ρ, then

$$?X(\rho)=\Pi_X^{Sym(\rho)}(\mathbf{I})$$

Proof. We have $x\in ?X(\rho)$ upper case iff for each $I\in weak(D,\rho)$, ~~there~~ exists $t\in I$ with $t[X]=x$ copy iff there exists $t\in$ I with $\eta(t)=t$ where is a $Sym(\rho)$ preserving homorphisim and $\eta(I)=I$. remove It is easy to see that for $A\in X$ $x[A]\in Sym(\rho)$. upper case This comes from the fact that for each $I\in weak(D,\rho)$ there is a $J\in weak(D,\rho)$ with $Sym(I)\cap Sym(J)=\emptyset$. $-Sym(\rho)$ (indeed J, can be formed by isomorphically renaming each element of $Sym(I)-Sym(\rho)$ by an element not in $Sym(I)$.) Therefore $x\in ?X(\rho)$ iff $x\in\Pi_X^{Sym(\rho)}(\mathbf{I})$. ⊣ Uppercase

Proposition 2 replaces an intersection of infinitely many projections with a single projection of an infinite relation. This brings us no closer to an effective computation. We now show, as stated earlier, that no such effective computation exists.

Proposition 3.

1) There exist X, **R**, D such that $?X/\mathbf{R},D/$ is not effectively computable.

2) There exists no uniform, effective procedure for determining if $?X/\mathbf{R},D/$ is computable for arbitrary X, **R**, D.

Proof. (1) The <u>completeness problem</u> is determining for all triples $\{<\rho,\mathbf{R},D>\}$, whether $?R(\rho)=\rho(R)$ for every $R\in\mathbf{R}$. The completeness problem is shown to be undecidable in [GMV] where (2) the set $\{<\mathbf{R},D>$ / *completeness of states of* **R** *with respect to D is decidable*$\}$ is shown to be not recursive. ⊣

upper case $T\rho$

In contrast to proposition 3, in the case that $chase_D(\tau\rho)$ ~~is~~ effectively computable, so is $?X(\rho)$.

Proposition 4. If $chase_D(T\rho)$ exists, then

$$?X(\rho) = \pi_X^{O\,[illegible]}(chase_D(\tau\rho)) \,.$$

Proof. In this case $chase_D(\tau\rho)$ is embeddable via $Sym(\rho)$ preserving homomorphisim <u>into</u> (not onto) every element of $weak(D,\rho)$ [GMV]. Therefore $\pi_X^{Q\,[illegible]}(chase_D(\tau\rho)) \subseteq ?X(\rho)$. On the other hand, it is only a small abuse of notation to state $chase_D(\tau\rho) \in weak(D,\rho)$. So $\pi_X^{P\,[illegible]}(chase_D(\tau\rho)) \supseteq ?X(\rho)$. ⊣

A query E is said to be <u>monotonic</u> if $\rho\supseteq\sigma$ implies $E(\rho)\supseteq E(\sigma)$.

Proposition 5. Canonical queries are monotonic.

Proof. The inclusion $\rho\supseteq\sigma$ implies $weak(D,\rho)\subseteq weak(D,\sigma)$. The proposition follows. ⊣

We allow only finite states. Suppose however we were to allow states of arbitrary size. It would still be possible to define weak instances for these states and therefore cannonical queries as well. Proposition 5 remains true in this case without modification to its proof. Proposition 5 therefore establishes that canonical queries are monotonic everywhere, not merely over states of finite size. This is crucial to the development of section 4, below.

On the other hand, not all expressions of the relational algebra define monotonic queries. We will say that a query E on schema **R** is canonical if there is some set of dependencies D such that $E = ?X/\mathbf{R},D/$. We know then that not all queries expressed in relational algebra are canonical. T he reverse inclusion is also not true.

It is well known that no expression of the relational algebra is equivalent to the transitive closure of a binary relation. [AU], [Im], [Zl]. Let R be a binary relation symbol and let d be the dependency which expresses the transitivity of R:

$$\forall xyx(Rxy \wedge Ryz \Rightarrow Rxz)$$

Then $?R\,[\{R\}.\{d\}]$ is the nsitive closure function, since, directly from the
remove
definition, $?R[\{R\}.\{d\}]I$ s the smallest relation containing I which is transitive. We state these facts as a pr position.

Proposition 6. The set of canonical queries is incomparable to the set of queries which may be expressed in the relational algebra. ⊣

3. The Logic of canonical queries.

In this section we present canonical queries in a logical framework. We do this to make more apparent the closeness of our approach to the approach of artificial

Lower case : e

intelligence which treats querying as logical inference. [GME] We also do it to prepare ourselves for the results of the next section.

Let U be a universe. It is necessary to fix an ordering on the elements of U. If $\mathbf{R}$ is a schema over U, the first order language (with equality) $L_{\mathbf{R}}$ has neither function nor constant symbols. T he predicates of $L_{\mathbf{R}}$ are the schemes of $\mathbf{R}$. Thus if $R \in \mathbf{R}$ is the set $\{A_{i_1}, \ldots A_{i_m}\}$, then $L_{\mathbf{R}}$ has an m-ary predicate symbol R. Let S be a relation scheme. For notational ease we will denote the language associated with a schema $\mathbf{R} \cup \{S\}$ as $L_{\mathbf{R},S}$ rather than $L_{\mathbf{R} \cup \{S\}}$. However, we always assume a <u>new</u> predicate symbol, that is, a symbol not in $L_{\mathbf{R}}$, appears for S in $L_{\mathbf{R},S}$, even when $S \in \mathbf{R}$.

Let $X \subseteq U$ and D be a set of unirelational dependencies on U. Consider the following set of sentences Σ in the language $L_{\mathbf{R},X,U}$

(containing instance)

(dependencies)

Projection axiom: $\forall c_1 \ldots c_n (U(c_1, \ldots, c_n) \Rightarrow X(c_{i_1}, \ldots c_{i_m}))$

where $X = A_{i_1} \ldots A_{i_m}$

The finite models of Σ, denoted $struc(\Sigma)$, can be written as triples $<\rho, I, \xi>$ where $\rho \in CONS(\mathbf{R}, D)$, $I \in weak(D, \rho)$ and $\xi \supseteq ?X/\mathbf{R},D/(\rho)$. Let C be formed by reducing $struc(\Sigma)$ to $L_{\mathbf{R},X}$; that is, $C = \{<\rho, \xi> \mid \rho \in CONS(\mathbf{R}, D) \text{ and } \xi \supseteq ?X/\mathbf{R},D/(\rho)\}$. Let $D_{\mathbf{R},X}$ be the collection of consequences of Σ in the language $L_{\mathbf{R},X}$; that is, the elements of $D_{\mathbf{R},X}$ are sentences in $L_{\mathbf{R},X}$ which hold in every element of $struc(\Sigma)$. Clearly, the members of C satisfy the sentences of $D_{\mathbf{R},X}$; that is, $C \subseteq struc(D_{\mathbf{R},X})$. We now demonstrate the reverse inclusion.

Lemma 1. $C = struc(D_{\mathbf{R},X})$

Proof. We need only show $struc(D_{\mathbf{R},X})\subseteq C$, by preceeding remarks. So let $<\rho,\xi>\notin C$. Let **dom** be the set of all values appearing in ρ and expand the language $L_{\mathbf{R},X,U}$ by adding each element of **dom** as a constant. In the expanded language, let

$$D_1=\{R(a_1,\ldots,a_n)\,/\,\textit{for each } R\in\mathbf{R} \textit{ where } <a_1,\ldots,a_n>\in\rho(R)\}$$

$$D_2=\{\neg X(a_1,\ldots,a_m)\,/\,<a_1,\ldots,a_m>\notin\xi \textit{ and } a_i\in\mathbf{dom}\}$$

$$D_3=\{a\neq b\,/\,\textit{for each pair of distinct elements of }\mathbf{dom}\}.$$

Now $\Sigma'=\Sigma\cup D_1\cup D_2\cup D_3$ is an inconsistent set of sentences. Suppose otherwise. If M is a structure for Σ', then $M(U)$, the interpretation of U in M, is a weak instance for ρ with respect to D so $\rho\in CONS(\mathbf{R},D)$ [GMV]. I t must be therefore that $\xi\supseteq ?X(\rho)$, as $<\rho,\xi>\notin C$. So there is a tuple $x\in ?X(\rho)-\xi$. Now $x\in ?X(\rho)$ implies $x\in\Pi_X(M(U))$ so by the projection axiom, $x\in M(X)$. But $\neg X(x)\in\Sigma'$ so M is not a model of Σ'.

We note that $D_1\cup D_2\cup D_3$ is a finite set of sentences in the expanded language. Therefore, the conjunction of its elements, denoted d, is a quantifier free sentence of the language $L_{\mathbf{R},X}$ augmented with the set **dom** of constants. Furthermore, $<\rho,\xi>$ satisfies d. On the other hand, from the inconsistency of Σ' we may conclude, $\Sigma\vdash\neg d$. Noting that Σ is constant free, we may conclude $\Sigma\vdash(\forall\mathbf{x})(\neg d)$ where $\mathbf{x}$ is the vector of all elements of **dom** appearing in d, interpreted as variables. In short, $(\forall\mathbf{x})(\neg d)$ is an element of $D_{\mathbf{R},X}$, so $<\rho,\xi>\notin struc(D_{\mathbf{R},X})$. We may therefore conclude $struc(D_{\mathbf{R},X})\subseteq C$. $\dashv$

Corollary 1. $D_{\mathbf{R},X}$ is equivalent to a set of total, multirelational equality and tuple generating dependencies.

Proof. Consider the formula $\neg d$ in the proof of the lemma. This may be rewritten as $(d_1 \rightarrow d_2 \vee d_3)$ where d_1 is ~~the~~ a conjunction of elements of D_1; d_2, d_3 ~~the~~ are disjunctions of the elements of D_2, D_3 respectively, these latter appearing in in positive (*i.e.*, unnegated) form. By a result due to McKinsey [McK], extended by Graham and Vardi [GV], since Σ contains only dependencies, for some atomic formula e of $D_2 \cup D_3$, we must have $\Sigma \vdash (\forall x)(d_1 \rightarrow e)$. $\dashv$

In light of this corollary, we will write $D_{R,X}$ as $E_R \cup T_{R,X}$ where E_R is the set of egd's and $T_{R,X}$ is the set of tgd's mentioned above. It is known [GV] that the set $struc(E_R)$ of finite models of E_R is exactly $CONS(R,D)$. It is natural to consider the set $struc(T_{R,X})$; that is, it is natural to consider canonical queries on states not required to be consistent. We can do this by removing all egd's from D and replacing them with "nearly equivalent" tgd's as follows.

Let $d = <T, a = b>$ be an egd. Let $A_1, \ldots, A_n$ be the attributes of u such that $\{a, b\} \subseteq dom(A_i)$. ~~For i=0 to n,~~ For each such i let w_{i_a}, w_{i_b} be a pair of tuples on the universe U, satisfying $w_{i_a}[A_i] = a$, $w_{i_b}[A_i] = b$, $w_{i_a}[B] = w_{i_b}[B]$ for all $B \neq A_i$, and $Sym(\{w_{i_a}, w_{i_b}\}) \cap Sym(T) = \{a, b\}$. The tgd translation of d is the set

$$\bigcup_{i=1}^{n} \{ <T \cup \{w_{i_a}\}, w_{i_b}>, <T \cup \{w_{i_b}\}, w_{i_a}> \}$$

The egd free version of a set of dependencies D, denoted D^{ef} is formed by replacing each egd in D with its tgd translation. Let $E_R{}^{ef}$ and $T_{R,X}{}^{ef}$ be the set of egd's and tgd's respectively which make up $D_{R,X}{}^{ef}$ as in corollary 1.

Lemma 2

1) $E_R{}^{ef} = \varnothing$

2) $T_{R,X}{}^{ef} = T_{R,X}$

Proof. 1) Immediate. 2) [BV???] $\dashv$

Combining this result with lemma 1 we have,

Corollary 2. 1) $struc(T_{\mathbf{R},X}) = \{<\rho,\xi> / \xi \supseteq ?X/\mathbf{R},D^{ef}/(\rho)\}$

2) If $\rho \in CONS(\mathbf{R},D)$, then $?X/\mathbf{R},D/(\rho) = ?X/\mathbf{R},D^{ef}/(\rho)$ ⊣

This result states the canonical queries defined with respect to D^{ef} are identical to the queries defined with respect to D when the former are restricted to $CONS(\mathbf{R},D)$. It is useful to state the following result, whose proof is immediate from corollary 2.

Corollary 3. $?X/\mathbf{R},D/(\rho) = \cap\{\xi / <\rho,\xi> \in struc(T_{\mathbf{R},X})\}$ ⊣

These lemmas and their corollaries can be viewed in the following way. They state that calculation of a canonical query is ~~the~~ in essence the derivation of a tuple generating dependency. This is replaced The elements of $T_{R',x}$ are multirelation dependencies. They can be seen as ~~non~~ uni relational dependencies by the simple expedient of ignoring the replace this tag attribute in their tableaux. This transforms an element of $T_{R',x}$ into an embedded dependency in the language uppercase $\mathcal{U}$ which is easily seen to be logical consequence of D. Recalling a resuit f Beeri and Vardi's[BV?], we have that for each tuple $X \in ?X/\mathbf{R},D/(\rho)$, there ex s chase sequence of finite length (possibly 0) which adds a row u to T_ρ with uppercase $u/\emptyset/$

The reader may wonder whethe :he set of finite structures $\{<\rho,\xi> / \xi = ?X/\mathbf{R},D/(\rho)\}$ is first order ıxiomatizable. We have shown $T_{\mathbf{R},X}$ to be a first order axiomatization of structures containing "all the truth." Is it possible to axiomatize those structures containing "only the truth?" Interestingly, this question can be answered either way, depending on how it is phrased.

We have restricted ourselves to the consideration of finite structures only as models. Suppose that f is any function from database states to instances of the scheme X and consider the pair $<\rho,f(\rho)>$. As both ρ and $f(\rho)$ are finite, this pair may be described by a single sentence of the form "if the state is exactly ρ, then the instance of X is exactly $f(\rho)$." The set $\{<\rho,f(\rho)>\}$ is exactly the set of finite models of the (infinite) set of sentences so constructed. This procedure is hardly effective nor very informative. Furthermore, these sentences are not dependencies. The fact that $T_{\mathbf{R},X}$ contains only tuple generating dependencies of a particular form is vital to the results of the next section.

If we consider the collection of all models of a given set of first order sentences, we discover that it is impossible in general to axiomatize the exact answers to canonical queries. The transitive closure of a binary relation may serve us again as a counterexample. Let R, R^+ be binary relation symbols. Consider both as giving two different edge relations on the same set of nodes. For each k, it is possible to write "for no pair for which an arc appears in R^+, is there a path of length k between them in R." Each finite subset of the set of all such sentences is consistent with an axiomatization of the transitive closure of nonempty relations, should such an axiomatization exist. But by the principle of compactness, which applies here as the full collection of models is considered, no such axiomatization can exist. Every arc in R^+ must correspond to a path in R of some finite length.

This discussion justifies a belief that the result of corollary 3 is as close as one can get to canonical queries with first order sentences.

4. Algebraic canonical queries.

In this section we consider those queries which are both canonical and expressible in the relational algebra. (Queries expressible in the relational algebra will hereinafter be called algebraic.) We will rely on the well known equivalence of the relational algebra and relational calculus. We restrict the class of expressions we will allow in two ways. We consider these restrictions to be matters of convenience.

First, we do not allow constants. Dependencies are written in a constant-free language, as in the prior section. Allowing constants in our expression language merely confuses matters. Secondly, we do not allow equality. This is in conformity with the work of Chandra and Merlin [CM]. We adopt this restriction in this section (we abandon it in the next) as we are considering here the canonical queries defined by an egd-free set of dependencies. Every state is consistent with such a set of dependencies and thus each canonical query is defined on every state. This simplifies our discussion. When D is egd-free, the set of sentences Σ of section 3 is written in a language without equality. Thus our prohibition of equality is similar to our prohibition of constants.

Formulae of the relational calculus are customarily interpreted only in finite states. As they are also formulae of first order logic it will be convenient to interpret them over states of arbitrary size.

Lemma 3. Suppose E is ~~a N~~ an monotonic query everywhere expressible in a relational calculus without equality. Then E is expressible as a union of conjunctive queries. ⊣

[illegible]space italics

Proof Suppose E may be expressed as $\psi(x_1, \ldots x_k)$ using 'domain' calculus notation. We show is (equivalent to) a positive, existential formula; i.e., it is constructed from atomic~~s~~ formulae using $\exists, \wedge, \vee$ as the only connectives. But these formulae are exactly unions of conjuntive queries.

We show first the existential part. Suppose ρ is a state and $\rho \models \psi(a_1, \ldots a_k)$: that is, the tuple $(a_1, \ldots, a_k) \in E(\rho)$. Suppose we have a state σ related to ρ in the following way: for each scheme R, $\rho(R)$ is the intersection of $\sigma(R)$ with the appropriate cross product of a fixed set of ~~constraints~~ constants (fixed in the sense that the same set of constants is used to form each relation of ρ .) In this case, ρ is called a submodel of σ, σ an extension of ρ. As $\sigma \supseteq \rho$, we have $(a_1, \ldots, a_k) \in E(\sigma)$. Generalizing this arguement, we see that the sentence~~s, in~~ remove a language expanded with constants~~,~~ remove $\psi(a_1, \ldots, a_k)$ is preserved under extensions; that is, if true in ~~any~~ a state it is true in all extensions of that state. Therefore ψ is existential by the dual of theorem 3.2.2 of uppercase Chang & Keisler[CK].

To show the positive part, we will show that everywhere monotonic queries are preserved under homomorphisim; that is, if $(a_1, \ldots, a_k) \in E(\rho)$ and ρ is homomorphically embeddable in a state σ via a homomorphisim $\underset{\sim}{h}$, then $<h(a_1), \ldots, h(a_k)> \in E(\cancel{\rho}\,\sigma)$. But then the formula ψ expressing E is positive by theorem 3.2.4 of the above cited text.

We define a strong homomorphisim as one which preseves negative as well as positive atomic formulae; that is, if $<b \ldots b_n> \notin \rho(R)$ then $<h(b_1), \ldots, h(b_k)> \notin \sigma(R)$, where h takes ρ to σ. We leave to the reader the tas~~t~~k of showing that any existential sentece is ~~preveived~~ preserved under strong homomorphisims. [cf,Enderton p91 f].

Now suppose h is a homomorphism of ρ into σ and $(a_1, \ldots, a_k) \in E(\rho)$. We must show $<h(a_1), \ldots, h(a_k)> \in E(\cancel{\rho}\,\sigma)$. If h is strong we are done. For any $<b_1 \ldots b_n> \notin \rho(R)$ with $<h(b_1) \ldots h(b_k)> \notin \sigma(R)$ add $<b_1 \ldots b_m>$ to $\rho(R)$. Let ρ' be the result of all such additions. So $\rho' \supseteq \rho$ and therefore $(a_1, \ldots, a_k) \in E(\rho')$ by monotonicity. Now h is a strong homorphism, from ρ' to σ so $<h(a_1), \ldots, h(a_k)> \in E(\sigma)$since the sentence $\psi(a_1, \ldots, a_k)$ has been shown to be existential. (Recall ψ is the formula expressing E.) $\dashv$

In lemma 3 we depend on the assumption that the query is everywhere monotonic, not just monotonic on finite states. Of the theorems used in the proof, that concerning preservation under extensions has been shown to be false in the case that only finite states are considered. [Gurevich]. The status of the lemma itself is this case is unknown [op.cit].

We now produce now ~~dijunction~~ a characterization of those canonical queries which are algebriac.

It is obvious that the appearance of a tuple in the result of a canonical query depends upon the existence of certain tuples in the database state. We may wish to know how many such tuples must appear in the state to support a tuple in the query. If $t \in ?X(\rho)$, is it possible to bound the size of a substate $\sigma \subseteq \rho$ with $t \in ?X(\sigma)$ which bound is independent of the size of ρ? Note that in the case of transitive closure, it is not possible to do this. This motivates the following definition.

<u>Definition</u> A schema R is *X-bounded* with respect to a set of dependencies D (for some X a set of attributes), if there exists an integer k such that for every state ρ, $t \in ?X(\rho)$ implies there exists a substate $\sigma \subseteq \rho$ wi $|\sigma| \leq k$ and $t \in ?X(\sigma)$.

Maier, Ullman and Vardi [MUV] proposed notion of boundedness which we will show equivalent to ours. Their idea is based on the computation of canonical queries via the chase.

<u>Definition</u> A schema R is *X-chase-bounded* with respect to a set of dependencies D (for some X a set of attributes), if there exists an integer k such that for every state ρ, $t \in ?X(\rho)$ implies there exists a sequence of transformations on the dependencies in D which introduces a row w into T_ρ with $w[X] = t$, which sequence is of length not greater than k.

We should point out that qualification that D be finite is crucial to the meaningfulness of this definition. For if D is replaced with its semantic closure (the set of all dependencies it implies), then every schema is X-chase-bounded with $k=1$.

There is yet a third, equivalent notion of boundedness. Consider the set $T_{\mathbf{R},X}$ of the prior section. We will say that $T_{\mathbf{R},X}$ is finitely covered when it is equivalent to some finite subset of itself. By 'equivalence' here, we mean finite equivalence. $T_{\mathbf{R},X}$ is finitely equivalent to some set Σ if the equality $struc(T_{\mathbf{R},X})=struc(\Sigma)$ holds.

Theorem 1. Let $\mathbf{R}$ be a schema and D a finite set of unirelational dependencies on a universe U. The following are equivalent.

1) $\mathbf{R}$ is X-bounded with respect to D.

2) $\mathbf{R}$ is X-chase-bounded with respect to D.

3) $T_{\mathbf{R},X}$ is finitely covered.

4) $?X[\mathbf{R},D]$ is equivalent to a finite union of conjunctive queries.

5) $?X[\mathbf{R},D]$ is equivalent to an expression of the relational algebra.

Proof. $1\Rightarrow2$ Let k_0 be the integer required by the definition of X-bounded.

There are, up to isomorphism, finitely many states of size k_0. Each has only finitely many "X-consequences;" that is, $?X(\rho)$ is always finite. For each row of $?X(\rho)$ for each ρ there is a sequence of some finite length which introduces this consequence into T_ρ. The length, k_1, of the longest sequence among these proofs is the bound required for $\mathbf{R}$ to be X-chase-bounded.

$2\Rightarrow1$ Immediate.

1$\Rightarrow$*3* As before, let k_0 be the bound required by the definition of X-bounded. We claim that $\mathrm{T}_{\mathbf{R},X}$ need contain no dependency $d=<T,x>$ with $|T|>k_0$. But this is immediate.

3$\Rightarrow$*4* Each of the dependencies in $\mathrm{T}_{\mathbf{R},X}$ is identical in format to a conjunctive query on $\mathbf{R}$ with target list X. Set E to be the union of the (finitely many) conjunctive queries in $\mathrm{T}_{\mathbf{R},X}$. We claim $E(\rho)=?X(\rho)$ for every state ρ.

By construction we have $<\rho,E(\rho)>\in struc(\mathrm{T}_{\mathbf{R},X})$, that is $E(\rho)\supseteq ?X(\rho)$. For the reverse inclusion, we can show that for every $\xi\supseteq ?X(\rho)$, $\supseteq E(\rho)$. That is, $E(\rho)=?X(\rho)$, by corollary 3.

So let $\upsilon\in E(\rho)$. By definition of E, there is an element $<T,x>\in\mathrm{T}_{\mathbf{R},X}$ and some homomorphism η with $\eta(T)\subseteq T_\rho$ (η tag preserving) and $\eta(x)=\upsilon$. But $<T,x>$ is an element of $\mathrm{T}_{\mathbf{R},X}$ so any ξ with $<\rho,\xi>\in struc(\mathrm{T}_{\mathbf{R},X})$ must satisfy $<T,x>$; that is, $\eta(x)\in\xi$, that is $\upsilon\in\xi$.

4$\Rightarrow$5 Immediate.

5$\Rightarrow$4 From lemma 3 and proposition 5.

4$\Rightarrow$1 The bound is the number of conjuncts in the largest clause of the expression. ⊣

5. Boundedness with respect to consistency

In the preceding section we were concerned with the finiteness of the set $T_{\mathbf{R},X}$ of tgd's in the language $L_{\mathbf{R},X}$ implied by the dependencies, containing instance, and projection axioms defining the canonical queries. A similar question can be asked about the set $E_{\mathbf{R}}$ of equality generating dependencies so implied.

Fact: [GV] The set $E_{\mathbf{R}}$ is finite iff there is an integer k such that any inconsistent state of $\mathbf{R}$ has an inconsistent substate of size not exceeding k. $\dashv$

Thus we say $\mathbf{R}$ is bounded with respect to consistency if $E_{\mathbf{R}}$ is finite.

Despite the similarity of this fact to the equivalencies in theorem 1, we now show by example that boundedness with respect to consistency and algebraicness are mutually independent.

If D contains no egd's, then $E_{\mathbf{R}}$ is empty. So in particular, the transitive closure example (see section 4) is bounded with respect to consistency but not algebraic.

Let F be a set of functional dependencies over some universe U and let $SAT(U,F)$ be the set of all instances of U which satisfy F. Let A be the set $\{\Pi_V(I) | I \in SAT(U,F)\}$ for some $V \subseteq U$. As pointed out by Ginsberg and Zaddian [GZ], A need not be $SAT(V,G)$ for any set of functional dependencies G. Hull has recently shown that in that case $E_{\{V\}}$ is not finite [H]. But notice that $A = CONS(\{V\},F)$ and that for any $X \subseteq U$, $?X[\{V\}].F$ is either identically empty (if $X \not\subseteq V$) or is equivalent to the appropriate projection. So $_{\{V\},X}$ is certainly finite.

We will say a schema $\mathbf{R}$ is algebraic if for every X, $X(\mathbf{R},D)$ is algebraic. If D contains only typed equality generating dependencies, algebraicness is implied by boundedness with respect to consistency.

Proposition 6. Suppose D is a set of typed egd's and $\mathbf{R}$ is bounded with respect to consistency with D. Then $\mathbf{R}$ is algebraic.

Proof. Suppose not. From the hypotheses and prior results, we know

1) there is an integer k_0 such that an inconsistent state of **R** contains an inconsistent substate of size not exceeding k_0;

2) For some $X \subseteq U$ and every integer k_1, there exists a state ρ with at least k_1 tuples and a tuple $x \in ?X(\rho)$ and $x \notin ?X(\sigma)$ for any proper substate σ of ρ.

Letting k be the integer of point 1 above, construct a consistent state as described in point 2 of size at least $k|X|$. Let this state be σ. Recalling that D contains only egd's, we note that the row of $chase_{D}(T)$ with x-value$x$ (x is the X-value given in point 2 above) must correspond to a tuple $\upsilon \in \sigma R)$ for some R and $X \not\subseteq R$.

Let ν be a 1-1 mapping of $Sym(\sigma)$ which is the identity on symbols of the tuple υ and takes all other symbols of σ to symbols not in σ. Let $\rho = \sigma \cup \nu(\sigma)$. To see that ρ is not consistent with D, let u be any tuple of σ such that $u[A]=x[A]$ for some $A \in X-R$. The egd $<T\rho, u[A]=\nu(u[A])>$ is a consequence of D but $u[A] \neq \nu(u[A])$ by construction. Consequently there must be a substate $\rho_A \subseteq \rho$ with k or fewer tuples such that D implies $<T\rho_A, u[A]=\nu(u[A])>$. This substate must contain some rows of $\nu(\sigma)$ (although not necessarily u or $\nu(u)$).

Let η be the mapping on $T\rho_A$ defined by: $\eta(t)=t$ if $t\cdot$. $\eta(t)=\upsilon$ if $t \in \nu(\sigma)$. Now η is a homomorphism embedding T_{ρ_A} into T_σ since for every $\upsilon_0 \in \nu(\sigma)$, $\upsilon_1 \in \sigma$, and every attribute B, $\upsilon_0[B]=\upsilon_1[B]$ only if $\upsilon_0[B]=\upsilon[B]=\upsilon_1[B]$. So η is homomorphism enabling in T_σ a transformation on dependency $<T_{\rho_A}, u[A]=\nu(u[A])>$. Application of this transformation to T_σ will set $t[A]=x$. But $|\eta(T_{\rho_A})|<k$. Repeat this arguement for each $A \in X-R$. This will uncover a substate $\sigma' \subseteq \sigma$ with $|\sigma'|<n|X|$ and $x?(\sigma')$. this contradicts our choice of σ.$\dashv$

We now take up the task of tightening the results of Theorem 1. We wish to characterize algebraic canonical queries defined with respect to a set of dependencies which include egd's. Equivalently, we wish to consider queries defined exactly on the set *CONS*(**R**,*D*). We face an immediate syntactic difficulty: an expression of the relational algebra is necessarily defined on all states of **R**, without regard to their inclusion in *CONS*(**R**,*D*). Thus we must expand the domain of $?X[\mathbf{R},D]$ if we wish to find any algebra expression to which it is equivalent. A method of doing this is given by Corollaries 2 and 3 of section 3: replacing D with D^{ef}. This method is exploited in Theorem 1. We seek in this section an expansion which distinguishes consistent from inconsistent states more precisely. Many such expansions are possible. We adopt the following.

For a set of attributes X of cardinality n, we define the X-product of a state ρ as

$$\times_{A\in X}(dom(A))\cap Sym(\rho)^{n}$$

That is, an X-product is the set of all combinations of symbols in ρ which respect the domain definitions. We define $?X[\mathbf{R},D](\rho)$ to be the X-product of ρ when $\rho\notin CONS(\mathbf{R},D)$. This definition reflects the standard logical notion that everything is a consequence of an inconsistent set of sentences. It also preserves the monotonicity of canonical queries, as any superset of an inconsistent state is inconsistent.

The expanded function will not always distinguish consistent from inconsistent states. Consider a four attribute universe with two schemes: $\{AB,CD\}$ and the functional dependency $A\rightarrow B$. If the domains of these attributes are pairwise disjoint (the "typed" case), then $?C$ is identically the C-product in every state. Similarly, $?CD$ is the CD-product in some consistent states. We can describe sets of attributes for which this behaviour is impossible.

Let $d = <T, a=b>$ be an egd. The repeating symbols of d are those elements of $Sym(T)$ with more than one appearance in d (a and b are presumably repeating symbols.) The agree set of d is the set of attributes labelling the columns of T in which the repeating symbols occur. (See Ginsburg and Hull ?? [GH].) If X contains the agree set of some egd in or implied by a set of dependencies D, then $?X[\mathbf{R}, D](\rho)$ satisfies d exactly when $\rho \in CONS(\mathbf{R}, D)$. **[Not quite: we need 2 symbols of dom(A) in ρ.]** We will exploit this fact in Theorem 2. We must first expand the class of relational algebra expressions we allow.

As we have allowed egd's in D, we must allow equality in our expressions. We define a conjuctive query with inequalities to be a conjunctive query plus a set of pairs of symbols called inequality assertions (and written $a \neq b$). So if q is a conjunctive query with inequality

$$q = << T, x >, S >$$

and ρ is a state, $q(\rho) = \{\nu(x) | \nu(T) \subseteq T\rho,\ \nu$ a homomorphism and $\nu(a) \neq \nu(b)$ for each $a \neq b$ in $S\}$. (The expansion of conjunctive queries to include inequalities was first made by Klug [K].)

We recall that $D_{\mathbf{R},X}$ is the set of all multirelational egd's and full tgd's which are consequences of the set Σ defined in section 3. Again, $D_{\mathbf{R},X}$ is said to be finite if it is finitely equivalent to a finite subset of itself.

Theorem 2. Let X contain the agree set of some egd implied by a set of dependencies D. **[Do I need this?]** For any schema $\mathbf{R}$, the following are equivalent:

1) $\mathbf{R}$ is bounded with respect to consistency and X-bounded.

2) $D_{\mathbf{R},X}$ is finite.

3) $?X[\mathbf{R}.D]$ is equivalent to a union of conjunctive queries with inequalities.

4) $?X[\mathbf{R}.D]$ is equivalent to an expression of the relational algebra.

Proof. The equivalence 1⇔2 follows from Theorem 1 and the fact mentioned earlier. We show 2⇒3 by construction. (A proof of 2⇒4 exists which omits this step. We find this procedure more informative.)

Construct a conjunctive query for each element of $T_{\mathbf{R},X}$ as before. Let E_1 be the union of these queries For each element of $E_{\mathbf{R}}$, proceed as follows:

Let $<T.a=b>$ be an element of $E_{\mathbf{R}}$. Let $W=\{w_R / R\in\mathbf{R}.\ w_R[Tag]=R\}$ be a collection of tagged rows sharing no symbols with each other or with $Sym(T)$. Let $v_1,\ldots,v_p$ be rows with tag X which rows result from permuting the symbols in $\cup_{R\in\mathbf{R}}(\cup_{A\in X}\{w_R[A]/A\in R\})$ in all ways consistent with the domain definitions. Construct the set of conjunctive queries with inequalities

$$\{<<T\cup W.v_i>.\{a\neq b\}>/1\leq i\leq p\}.$$

Let E_2 be the union of all these queries.

We claim the union of $E_1\cup E_2$ calculates $?X[\mathbf{R}.D]$. The proof is as before with the observation that if any element of $E_{\mathbf{R}}$ is violated by a state, the set of queries so constructed will force the result to be the appropriate X-product.

The equivalence 3⇔4 is as before. Note that a conjunctive query with inequality is monotonic, so Lemma 2, suitably modified, holds for the larger class of expressions considered here.

We complete the chain by demonstrating 3⇒2. If $<<T.x>.S>$ is an element of the union given by (3), construct the sentence

$$\forall y(T' \to x' \vee s')$$

where $T' = \wedge\{R(w[R]) \mid w[Tag] = R, w \in T\}$

$x' = X(x)$

$s' = \vee\{a = b \mid a \neq b \in S\}$

and y is the vector of all variables appearing in this sentence. We claim this sentence is implied by Σ (by (3)) and apply the result of McKinsey referenced earlier to reduce the resulting finite set to a subset of $D_{\mathrm{R},X}$, as before. We then claim this set to be finitely equivalent to $D_{\mathrm{R},X}$. **[Does this really work? I think so but I'm passing on. The next paragraph can also be used to prove this (or 4⇒2)]** ⊣

The weakening assumption in this theorem is a result of the particular expansion of canonical queries which we've adopted. Suppose we were to choose an expansion which distinguish consistent and inconsistent states via some first-order property. In other words, suppose there exists, with respect to this putative expansion, a sentence ψ on a single predicate (of arity the cardinality of X) such that ψ is true at $?X[\mathbf{R}, D](\rho)$ exactly when ρ is consistent. If $?X$ is algebraic, the first order formula ϕ which expresses $?X$ can be composed with ψ to produce a sentence of L_{R} true of a state ρ exactly when ρ is consistent. (This composition is the syntactic exercise of replacing the atomic formulae of ψ with the formula ϕ, due care being taken to rename variables as appropriate.) But in that case, E_{R} is finite, by the results of [GV].

6. Discussion and Conclusions

We have considered the question: When is a canonical query algebraic, i.e., equivalent to an expression of the relational algebra? It is natural to ask the converse question. When is an expression of the relational algebra equivalent to some canonical query? The answer is the same. Such an expression must be monotonic and therefore equivalent to a union of conjunctive queries. Each such query is essentially a multirelational tgd, which may be considered a unirelational tgd simply by ignoring the tags. Thus each monotonic expression E gives rise to a set of tgd's D such that $E \equiv ?X[\mathrm{R}, D]$ (X the "target scheme" of E). Every monotonic expression is canonical for some set of dependencies.

In the above discussion, we chose D after having seen the expression E. The reader may object to this procedure, considering the dependencies to come "first" and the queries only "later". But is this order correct? The purpose of canonical queries, window functions [MRW] [MUV] and universal relation interfaces [KU] is to make some set of queries very easy to formulate. Which set of queries should this be? We believe the database administrator, in cooperation with the end users, knows very well which queries are important. The dependencies and perhaps even the schema may be derived from the queries, rather than conversely. It is usual to declare the dependencies to be derived from "nature", that is, from knowledge of the application. We do not dispute this. We have shown that they describe an inference engine for the calculation of certain pre-selected queries.

RJ 4207 (46285) 2/23/84
Computer Science

ON THE COMPLEXITY AND AXIOMATIZABILITY OF CONSISTENT DATABASE STATES+ (Extended Abstract)

Marc H. Graham

Georgia Institute of Technology
Atlanta, Georgia 30332

Moshe Y. Vardi*

IBM Research Laboratory
San Jose, California 95193

ABSTRACT: A database is consistent with respect to a set D of dependencies if it has a weak instance. A weak instance is a universal relation that satisfies D, and whose projections on the relation schemes are supersets of the relations in the database. In this paper we investigate the complexity of testing consistency and the logics that can axiomatize consistency, relative to a fixed set D of dependencies. If D is allowed to include embedded dependencies, then consistency can be non-recursive. If D consists only of total dependencies, then consistency can be tested in polynomial time. The degree of the polynomial can, however, be arbitrarily high. Consistency can be axiomatized but not finitely axiomatized by equality generating dependencies. If embedded dependencies are allowed then consistency cannot be finitely axiomatized by any effective logic. If, on the other hand, only total dependencies are allowed then consistency can be finitely axiomatized by fixpoint logic.

+ To be presented in the 3rd ACM Symp. of Principles of Database Systems, Waterloo, April 1984.

* This research reported here was done while this author was at Stanford University.

ON THE COMPLEXITY AND AXIOMATIZABILITY OF CONSISTENT DATABASE STATES[†]

Extended Abstract

Marc H. Graham
Georgia Institute of Technology

Moshe Y. Vardi[‡]
IBM Research Laboratory, San Jose

Abstract

A database is consistent with respect to a set Σ of dependencies if it has a weak instance. A weak instance is a universal relation that satisfies Σ, and whose projections on the relation schemes are supersets of the relations in the database. In this paper we investigate the complexity of testing consistency and the logics that can axiomatize consistency, relative to a fixed set Σ of dependencies. If Σ is allowed to include embedded dependencies, then consistency can be non-recursive. If Σ consists only of total dependencies, then consistency can be tested in polynomial time. The degree of the polynomial can, however, be arbitrarily high. Consistency can be axiomatized but not finitely axiomatized by equality generating dependencies. If embedded dependencies are allowed then consistency cannot be finitely axiomatized by any effective logic. If, on the other hand, only total dependencies are allowed then consistency can be finitely axiomatized by fixpoint logic.

1. Introduction

Soon after the introduction of the relational model [C1], the important role of *semantic specification* was realized [C2,AN]. The purpose of semantic specification is to define which databases are semantically meaningful, called *consistent* in database terminology. The languages used for semantic specification are logical languages. Thus, the database is consistent if and only if it satisfies certain sentences in the language. An example of such a language is the language of *functional dependencies* [C2].

Traditionally, the logic used for semantic specification languages was *first-order logic*. The reason for that is probably the fact that this is the logic that most researchers and practitioners were familiar with. Recently, however, researchers in the area of semantic specification realized that there does not seem to be a straightforward way of specifying semantics of databases with *incomplete information* by means of first-order logic [Ho].

The situation is as follows. In principle, there is a conceptual database with complete information, called *weak instance* in database terminology, that completely describes reality. The semantics of this idealized database is given in first-order logic. In practice, however, we very often do not have all the information needed to describe reality. . That is, the actual database does not contain enough information to uniquely determine the conceptual database. How we do know whether our partial description of reality is semantically meaningful? The intuitive answer is that it is semantically meaningful if it can be completed to a full description of reality. This is the justification for the definition in [Ho] that an actual database, which may have incomplete information, is consistent if it can be completed to a consistent database with complete information.

[†] To be presented in the 3rd ACM Symp. on Principles of Database Systems, Waterloo, April 1984

[‡] The research reported here was done while this author was at Stanford University and supported by a Weizmann Post-Doctoral Fellowship and AFOSR grant 80-0212.

While this definition was readily adopted by researchers and triggered numerous investigations of its implications (e.g., [GMV. Sa. MUV]), its logical aspects were not yet investigated.

A logic consists of three essential components: a *language*, a class of *structures* and a *satisfaction relationship* between structures and sentences in the language. The notion of structure in database theory is well understood: databases are essentially *finite* relational structures. What we are interested here is in the language and satisfaction relationship components. Specifically, we try to answer the two following questions:

(1) What is the *complexity* of testing consistency?

(2) What is the language required to *axiomatize* consistency?

More formally, we are given a set Σ of first-order sentences that the conceptual database (with complete information) is supposed to satisfied. Let $CONS(\Sigma)$ be the class of actual database (with incomplete information) that can be completed to satisfy Σ. We try to find out what is the complexity of recognizing databases in $CONS(\Sigma)$ and whether we can axiomatize it, that is, construct a set (preferably finite) Σ' of sentences in some language such that $CONS(\Sigma)$ is exactly the class of actual databases that satisfy Σ'. We are interested here in the case where the conceptual database is required to satisfy first-order sentences of a special form, the so called *data dependencies* [BV1. Fa2]. This class of sentences is considered to be appropriate to semantic specification of databases with complete information.

Our first finding is that there exists a set Σ of eid's such that $CONS(\Sigma)$ is not recursive! We are hence forced to restrict ourselves to the subclass of *total* (or *full*) dependencies [BV1. Fa2]. In this case we show that $CONS(\Sigma)$ is in PTIME. The degree of the polynomial can, however, be arbitrarily high!

With this in mind we turn to the issue of axiomatizability. By using classic model-theoretic techniques, we show that consistency is axiomatizable by first-order logic and even by dependencies, but is not *finitely* axiomatizable by first-order logic. The fact that consistency can be tested in polynomial time, and the strong connection between polynomial time computation and *fixpoint logic* shown in [Im, Var2], suggest that fixpoint logic might be the right logic to axiomatize consistency. Indeed, the deepest result in the paper is that consistency is *finitely* axiomatizable by fixpoint logic.

We discuss some "philosophical" aspects of our work in the concluding part of the paper.

2. Basic Definitions

2.1. Tuples, Relations, and Databases

Attributes are symbols taken from a given finite set U called the *universe*. We use the letters $A, B, C, \cdots$ to denote attributes and $X, Y, \cdots$ to denote sets of attributes. Sets of attributes are also called *relation schemes* for reasons to become clear shortly. As a convention, we do not distinguish between the attribute A and the set $\{A\}$, and we denote the union of X and Y by XY.

With each attribute A is associated an infinite set called its *domain*, denoted $DOM(A)$. The domain of a set X of attributes is $DOM(X) = \bigcup_{A \in X} DOM(A)$. An X-*value* is a mapping $w: X \rightarrow DOM(X)$, such that $w(A) \in DOM(A)$ for all $A \in X$. A *tuple* is an X-value for some X. A *relation* on a relation scheme X is a finite set of X-values. We use $a, b, c, \cdots$ to denote elements of the domains, $s, t, \cdots$ to denote tuples, and $I, J, \cdots$ to denote relations.

A *database scheme* is a sequence $\mathbf{R} = (R_1, \ldots, R_k)$ of relation schemes such that $U = \bigcup_{i=1}^{k} R_i$. We will occasionally consider U as a database scheme, meaning (U). A sequence $\mathbf{I} = (I_1, \ldots, I_k)$ of relations on $R_1, \ldots, R_k$, correspondingly, is called a *database* on $\mathbf{R}$. Let $\mathbf{I} = (I_1, \ldots, I_k)$ and $\mathbf{J} = (J_1, \ldots, J_k)$ be databases on $\mathbf{R}$. We say that $\mathbf{I}$ is contained in $\mathbf{J}$, denoted $\mathbf{I} \subseteq \mathbf{J}$, if $I_m \subseteq J_m$

for $m=1,\ldots,k$.

For an X-value w and a set $Y\subseteq X$ we denote the restriction of w to Y by $w[Y]$. We do not distinguish between $w[A]$, which is an A-value, and $w(A)$, which is an element of $DOM(A)$. Let I be a relation on X. Then its *projection* on Y, denoted $I[Y]$, is a relation on Y, $I[Y]=\{w[Y]:w\in I\}$. Let R be a database scheme. We associate with R a projection map $\pi_{\mathbf{R}}$, defined as follows. Let I be a relation on U. Then $\pi_{\mathbf{R}}(I)$ is the sequence $(I[R_1],\ldots,I[R_k])$, which is a database on R. The set of all attribute values in a relation I is $VAL(I)=\bigcup_{A\in X} I[A]$, and the set of values in a database I is $VAL(\mathbf{I})=\bigcup_{j=1}^{k} VAL(I_j)$. The database I is *nonempty* if $VAL(\mathbf{I})\neq\emptyset$.

2.2. Dependencies

A *valuation* is a partial mapping $\alpha:DOM(U)\rightarrow DOM(U)$ such that for all $A\in U$ and $a\in DOM(A)$ we have $\alpha(a)\in DOM(A)$. We say that α is a valuation on a tuple w (resp., relation I, database I) if it is defined on $VAL(w)$ (resp., $VAL(I)$, $VAL(\mathbf{I})$). Let α be a valuation on a tuple w, then $\alpha(w)$ is the tuple $\alpha\circ w$ (i.e., α composed with w). Valuations are defined on relations and databases in the natural way, i.e., they are defined on relations tuple-wise, and they are defined on databases relation-wise.

For any given application only a subclass of all possible databases is of interest. This subclass is defined by semantic constraints that are to be satislied by the databases of interest. A family of constraints that was extensively studied in the literature is the family of *dependencies*.

A *tuple generating dependency* (abbr. tgd) says that if some tuples, satisfying certain equalities exist in the database, then some other tuples (possibly with some unknown values), must also exist in the database. Formally, a tgd on a database scheme R is a pair <I,J> of nonempty databases on R. It is satisfied by a database K on R if for every valuation α on I, such that $\alpha(\mathbf{I})\subseteq\mathbf{K}$, there exist a valuation β on I and J that agrees with α on $VAL(\mathbf{I})$ such that $\beta(\mathbf{J})\subseteq\mathbf{K}$. If $VAL(\mathbf{J})\subseteq VAL(\mathbf{I})$ then <I,J> is a *total* tgd (abbr. ttgd).

An *equality generating dependency* (abbr. egd) says that if some tuples, satisfying certain equalities exist in the database, then some values in these tuples must be equal. Formally, an egd on a database scheme R is a pair $\langle \mathbf{I}, a_1=a_2\rangle$ where I is a database and $\{a_1,a_2\}\subseteq VAL(\mathbf{I})$. It is satisfied by a database K on R if for every valuation α on I such that $\alpha(\mathbf{I})\subseteq\mathbf{K}$ we have $\alpha(a_1)=\alpha(a_2)$. A *functional dependency* (abbr. fd) is a statement $X\rightarrow Y$. It is satisfied by a relation I on U if for every two tuples u and v in I, if $u[X]=v[X]$ then $u[Y]=v[Y]$. It is equivalent to an egd on U.

We will use the term *dependencies* or *embedded dependencies* to refer to the class tgd's and egd's, and we will use the term *total* to refer to the class of ttgd's and egd's. We note that dependencies are equivalent to first-order sentences of a special syntax [Fa2].

2.3. Satisfaction and Consistency

If we are given a database scheme R and a set Σ of dependencies on R, then it is quite obvious how to define the class of semantically meaningful databases on R. It is just the collection

$$SAT(\mathbf{R},\Sigma)=\{\mathbf{I}:\mathbf{I}\text{ is a database on R that satisfies }\Sigma\}.$$

However, a basic idea in database theory is that of universal relation interface [MUV]. According to this approach, conceptually the database is a single relation on U, and consequently the semantic specification has to be given as a set of dependencies on U. In practice, however, information is often given to us not as tuples on U but in smaller units, tuples on subsets of U, and some information may even be missing. The database scheme $\mathbf{R}=\{R_1,\ldots,R_k\}$ describes the actual database, and its relations reflects parts of the bigger conceptual database.

Such a database on R is semantically meaningful if indeed it reflects a meaningful conceptual relation on U.

This lead Honeyman [Ho] to the following definition[1]. Let Σ be a set of dependencies on U, and let $\mathbf{I}=(I_1,\ldots,I_k)$ be a database on a database scheme $\mathbf{R}=(R_1,\ldots,R_k)$. We say that I is *consistent* with respect to Σ, if there exists a relation I on U, such that $I \in SAT(U,\Sigma)$ and $\mathbf{I} \subseteq \pi_{\mathbf{R}}(I)$. I is called a *weak instance* for I. Note that I does not reflect exactly the breakdown of the information in I to smaller units of information, but rather it reflects a subset of that information, since I_j can be a proper subset of $I[R_j]$. We denote the set of databases on R that are consistent with respect to Σ by $CONS(\mathbf{R},\Sigma)$.

We now define a condition on database schemes that will play an important role when it comes to axiomatizability of consistency. A set Σ of dependencies over U is said to be *m-bounded* with respect to a database scheme R, for some natural number m, if for every database I on R, we have that I is in $CONS(\mathbf{R},\Sigma)$ if and only if for all $\mathbf{J} \subseteq \mathbf{I}$ with $|VAL(\mathbf{J})| \leq m$, we have that J is in $CONS(\mathbf{R},\Sigma)$. We say that Σ is *bounded* with respect to R if it is m-bounded with respect to R for some m.

3. Complexity

Several researchers investigated the complexity of testing satisfaction and consistency [BV2,GMV,MSY,Y]. What they tried to do is to find the complexity of the set $\{\langle U,I,\Sigma\rangle : I \in SAT(U,\Sigma)\}$ and the set $\{\langle \mathbf{R},\mathbf{I},\Sigma\rangle : \mathbf{I} \in CONS(\mathbf{R},\Sigma)\}$. In this context several lower bounds were shown. We find these lower bounds somewhat misleading. In a specific application the database administrator has a specific universe U_0, a specific database scheme $\mathbf{R}_0$, and a specific set Σ_0 of dependencies that describe the semantics of the application. Thus, he has no interest in the complexity of the above mentioned sets, but rather he is interested in the complexity of the sets $SAT(U_0,\Sigma_0)$ and the set $CONS(\mathbf{R}_0,\Sigma_0)$. Thus, what seems to be of interest in general is the complexity of the sets $SAT(U,\Sigma)$ and $CONS(\mathbf{R},\Sigma)$ for fixed U, R, and Σ. In the terms of [Var2] we are interested here the the *data complexity* rather then the *expression complexity* or the *combined complexity*.

Let us consider first satisfaction.

Lemma 1. [Cha] Let R be a database scheme and let Σ be a finite set of dependencies on U. Then $SAT(\mathbf{R},\Sigma)$ is in LOGSPACE. ■

Unlike satisfaction, the complexity of consistency depends on the kind of dependencies we have in Σ.

Theorem 1.

(1) Let R be a database scheme, and let Σ be a finite set of embedded dependencies on U. Then $CONS(\mathbf{R},\Sigma)$ is recursively enumerable.

(2) There exist a universe U and a finite set Σ of embedded dependencies on U such that $CONS(U,\Sigma)$ is not recursive.

(3) The set of pairs $(\mathbf{R},\Sigma)$, where Σ is a finite set of embedded dependencies on U and $CONS(\mathbf{R},\Sigma)$ is recursive, is not recursive.

Idea of Proof.

(1) Given a database on R, we just have to enumerate all relations on U and check whether any of them is a weak instance for the database.

(2) First, by reduction from the word problem for finite semigroups [Gu], we construct a universe U and a finite set Σ of dependencies on U such that the set $\{\sigma : \sigma \text{ is an egd and } \Sigma \text{ logically implies } \sigma\}$ is not recursive. Then, we show that this set is Turing-reducible to $CONS(U,\Sigma)$. The reduction involves exponentially (in the length of the given egd) many tests for consistency.

(3) The claim follows from a general characterization of undecidable properties of sets of dependencies in [Var1]. ■

[1] We use the generalization in [GMV] of the original ideas in [Ho].

Theorem 1 strengthens the results in [GMV] that the set

$$\{\langle R,I,\Sigma\rangle : \Sigma \text{ is a set of embedded dependencies and}$$

$$I \in CONS(R,\Sigma)\}.$$

is not recursive. Both results indicate very strongly that the weak instance approach is not practical when embedded dependencies are necessary to specify the semantics of the application. When all dependencies in Σ are total, the situation is radically different.

Theorem 2.

(1) Let R be a database scheme, and let Σ be a finite set of total dependencies on U. Then $CONS(R,\Sigma)$ is in PTIME.

(2) There is a universe U and a finite set Σ of total dependencies on U such that $CONS(U,\Sigma)$ is logspace complete in PTIME.

(3) For every natural number k, there exist a universe U_k and finite set Σ_k of total dependencies on U_k, such that $CONS(U_k,\Sigma_k)$ can not be accepted in DTIME(n^k).

Idea of Proof.

(1) In [GMV,Ho] there is an algorithm to test for consistency. Given a database, the algorithm tries to construct a weak instance. It either succeeds, demonstrating consistency, or it fails, proving that there does not exists a weak instance. The complexity of the algorithm is $O(n^l)$, where n is the size of the database and l is the size of R and Σ.

(2) Hardness for PTIME is proven by reduction from the path system problem of [JL].

(3) By a generic reduction from deterministic polynomial time Turing machines. ■

Theorem 2 strengthens the result in [GMV] that the set

$$\{\langle R,I,\Sigma\rangle : \Sigma \text{ is a set of total dependencies and}$$

$$I \in CONS(R,\Sigma)\}.$$

is logspace complete in EXPTIME. It shows that testing consistency of I with respect to Σ is polynomial in the size of I and exponential in the size of Σ.

It is interesting to note in connection with Theorem 2, that if Σ consists of fd's, then $CONS(R,\Sigma)$ can be accepted in time $O(n \log n)$ and linear space, by computing the closure of some congruence relation as in [DST].

Let us now consider bounded sets of dependencies. Intuitively, it seems that it should be easier to test consistency with respect to bounded sets than for general ones.

Theorem 3. Let R be a database scheme, and let Σ be a set of dependencies on U, such that Σ is bounded with respect to R. Then $CONS(R,\Sigma)$ is in LOGSPACE.

Idea of Proof. Assume that Σ is m-bounded with respect to R. To check that $I \in CONS(R,\Sigma)$ it suffices to check that $J \in CONS(R,\Sigma)$ for all $J \subseteq I$ such that $|VAL(J)| \leq m$. It is easy to verify that checking each J requires space logarithmic in the size of I. ■

4. Axiomatizability

A subject of great interest in mathematical logic is that of *axiomatizability*. Given a class Ω of structures, the logician tries to axiomatize it by defining a logic Λ, which consists of a language L and a satisfaction relationship between structures and sentences in L. Ω is *axiomatizable* by Λ if there exists a set Σ of sentences of Λ, such that a structure M is in Ω if and only if M satisfies all sentences in Σ. If Σ is finite, then Ω is *finitely axiomatizable* by Λ. This notion of axiomatizability enables us to classify the expressive power of logics according to the classes of structures that they can axiomatize or finitely axiomatize.

We first try to axiomatize consistency by first-order logic. We have to bear in mind, however, that every class of databases is axiomatizable by first-order logic. This follows from the fact that every database can be described, up to isomorphism, by a single first-order sentence. The axioms for the class are the negations of the descriptions of all databases not in the class. In fact, one can show that every class of databases is even axiomatizable in a proper subset of first-order logic. This subset, which we call *universal-existential* logic, is the set of all first-order sentences whose prefix consists of a string of universal quantifiers followed by a string of existential quantifiers. Thus, axiomatizability results for first-order logic are not interesting, unless they talk about finite axiomatizability or about a proper subset of universal-existential logic.

The proof of next theorem uses *disjunctive equality-generating dependencies*. A disjunctive equality-generating dependency (abbr. degd) on a database scheme R is a pair <I,δ>, where I is a finite database and δ is a sequence of equalities $a_1=b_1, \ldots, a_k=b_k$ with $\{a_1, \ldots, b_k\} \subseteq VAL(\mathrm{I})$. It is satisfied by a database K on R if for every valuation α on I such that $\alpha(\mathrm{I}) \subseteq \mathrm{K}$ we have that either $\alpha(a_1)=\alpha(b_1)$, or ... or $\alpha(a_k)=\alpha(b_k)$. Observe that an egd is a degd where the sequence of equalities is of unit length.

Theorem 4. Let R be a database scheme, and let Σ be a set of dependencies on U. Then $CONS(\mathrm{R},\Sigma)$ is axiomatizable by egd's.

Idea of Proof. The proof goes in three steps. First, using the method of *diagrams* [CK] we show that $CONS(\mathrm{R},\Sigma)$ is axiomatizable by degd's. That is, there exists a set Σ' of degd's on R such that $CONS(\mathrm{R},\Sigma)=SAT(\mathrm{R},\Sigma')$. Now, using the fact that Σ is a set of dependencies, which are *Horn* sentences, we show that $CONS(\mathrm{R},\Sigma)$ is closed under *direct products*. Finally, using the last fact, we prove by McKinsey's technique [McKi] that we can assume without loss of generality that all the degd's in Σ' are actually egd's. ■

The above result is interesting theoretically, but does not really have practical significance because the set of egd's promised by the theorem can be non-recursive! What we would like to have is finite axiomatizability by first-order logic, because then we would be able to apply Lemma 1, and get logarithmic space complexity. Now, Theorem 3 gives us a case where consistency can be tested in logarithmic space, namely, when the given set of dependencies is bounded with respect to the database scheme. Can it be that Theorem 3 is just a corollary of Lemma 1? The answer is positive.

Theorem 5. Let R be a database scheme, and let Σ be a set of dependencies on U. Then $CONS(\mathrm{R},\Sigma)$ is finitely axiomatizable by egd's if and only if Σ is bounded with respect to R.

Idea of Proof. If $CONS(\mathrm{R},\Sigma)$ is finitely axiomatizable by egd's, then $CONS(\mathrm{R},\Sigma)=SAT(\mathrm{R},\Sigma')$ for some finite set Σ' of egd's. Let $m=\max\{k : \langle \mathrm{I}, a_1=a_2\rangle \in \Sigma' \text{ and } |VAL(\mathrm{I})|=k\}$. Then Σ is m-bounded with respect to R. Conversely, if Σ is m-bounded with respect to R, then $CONS(\mathrm{R},\Sigma)$ is axiomatizable by egd's $\langle \mathrm{I}, a_1=a_2\rangle$ with $|VAL(\mathrm{I})|=m$. ■

Theorem 5 leaves open the possibility that consistency is finitely axiomatizable by first-order logic though not by egd's. However, since first-order satisfaction can be tested in logarithmic space, finite axiomatizability of consistency by first-order logic will entail, by Theorem 2, that PTIME=LOGSPACE! This suggests the following result.

Theorem 6. There is a universe U, a finite set Σ of total dependencies on U, and a database scheme R, such that $CONS(\mathrm{R},\Sigma)$ is not finitely axiomatizable by first-order logic.

Idea of Proof. Let $U=\{A,B,C\}$, $\mathrm{R}=\{AB,AC\}$, and $\Sigma=\{A \rightarrow C, B \rightarrow C\}$. We now show by an *ultraproduct* argument[2] [CK] that $CONS(\mathrm{R},\Sigma)$ is not finitely axiomatizable by first-order logic. ■

In view of the last two theorems, we would like to be able to tell, given a database scheme R and a set of dependencies Σ, whether Σ is bounded with respect to R. Unfortunately, there is no effective test for boundedness.

Theorem 7. The following set of pairs (U,Σ), where Σ is a finite set of dependencies on U and Σ is bounded with

[2] Thus we have to go to infinite structures in order to prove a claim about finite structures.

respect to U, is not recursive.

Idea of Proof. The claim follows from a general characterization of undecidable properties of sets of dependencies in [Var1]. ■

We do not know whether boundedness is decidable when we restrict ourselves to total dependencies. We believe that if we restrict ourselves to functional dependencies, then it is decidable.

Since we can not finitely axiomatize consistency by first-order logic, we try to do it by higher-order logics. Studying the definition of consistency we observe that essentially it consists of existentially quantifying over weak instances, which are relations over a possibly extended domain. The logic of such definition is called in mathematical logic *many-sorted projective logic* [Fe]. It is a very powerful logic, whose satisfaction relationship is not necessarily recursive (by Theorem 1; see also [Ha]). One can try to bound the size of the extended domain in order to make the satisfaction relationship recursive [MZ], but Theorem 1 implies that when the given dependencies are embedded this can not be done.

Let us now consider the case that the given dependencies are total. As we shall see in this consistency can be finitely axiomatized by the *fixpoint logic* of [AU,CH].

Let P be a new n-ary relation name, and let $L(\mathrm{R},P)$ be the language obtained by adding P to $L(\mathrm{R})$. The fixpoint sentences of $L(\mathrm{R})$ are of the form $LFP(\varphi)$, where φ is a first-order formula of $L(\mathrm{R},P)$ with free variables $x_1, \ldots, x_n$, where P occurs positively. Let M be a structure of $L(\mathrm{R})$ with domain D. Let Q be the minimal n-ary relation on the domain of M, such that the sentences $\forall x_1 \cdots x_n(P(x_1, \ldots, x_n) \equiv \varphi)$ is satisfied in the structure (M,Q) of the language $L(\mathrm{R},P)$. The relation Q is the *least fixpoint* of φ in the structure M. We now define the satisfaction relationship: M satisfies $LFP(\varphi)$ if $Q = D^n$. The following facts hold for fixpoint logic.

(1) Any class of databases that is finitely axiomatizable in fixpoint logic is in PTIME [CH].

(2) There is a class of databases that is finitely axiomatizable in fixpoint logic and is logspace complete in PTIME [Var2].

(3) Let Ω be a class of databases that include a linear order relation, such that Ω is in PTIME. Then Ω is finitely axiomatizable by fixpoint logic [Im,Var2]. (The linear order seems to be essential in order to simulate Turing machines.)

There are two reasons to suspect that consistency with respect to total dependencies can be finitely axiomatized by fixpoint logic. The first reason is, in view of the aforementioned facts, that consistency with respect to total dependencies can be tested in polynomial time. The second reason is that from the algorithm for testing consistency of [GMV,Ho] it follows that consistency with respect to total dependencies can be axiomatized by fixpoint logic over extended domains. Both observation show that with some "extra" tool, either a linear order or an extended domain, we can finitely axiomatized consistency by fixpoint logic. The question is whether we can do it without the "extra" tool. The answer is positive.

Theorem 8. Let R be a database scheme, and let Σ be a finite set of total dependencies on U, then $CONS(\mathrm{R},\Sigma)$ is finitely axiomatizable by fixpoint logic.

Idea of Proof. It turns out that the extended domain is not essential. The information conveyed by the new elements can be captured by relations over the old elements. These relations can be defined by fixpoint logic. The construction, however, is very involved. The length of the fixpoint sentence needed to axiomatize $CONS(\mathrm{R},\Sigma)$ is exponential in the length of Σ! ■

5. Philosophical Remarks

Another use of logical languages in relational database management system is as query languages. The result of applying a formula of the language to a database is the set of all tuples that satisfy the formula. An example of such a language is the relational calculus [C3]. The logic used for query languages was also traditionally first-order logic. However, in the last few years, it was realized that first-order logic does not have a sufficient expressive power as a query language. This was realized first by Aho and Ullman [AU], who observed that transitive closure is not first-order definable (this fact was originally proven in [Fa1]). Following that observation, several works investigated higher-order logics for query languages, e.g., [CH, MZ, Var2].

One can also object to the exclusive use of first-order logic in database theory on an "ideological" basis. The reason for the prominence of first-order logic in mathematical logic is that first-order logic is mathematically tractable and has very rich proof and model theories, e.g., we have completeness and compactness theorems. However, mathematical logic usually deals with general structures, either finite or infinite. In database theory, one usually wishes to consider only finite structures. Under this restriction many of the nice properties of first-order logic evaporate. In particular, we do not have completeness and compactness. Thus, there is no a priori reason to prefer first-order logic to other logics, and one should base his preference on practical considerations, such as ease of use and computational complexity.

First-order logic has the advantage of almost being a "lingua franca". It is a logic with which many practitioners are familiar, unlike the more esoteric higher-order logics. On the other hand, if one takes polynomial time as a yardstick for computational tractability, then there is evidence that fixpoint logic is the natural logic for finite structures [Im, Var]. Our results strengthen this evidence by showing that fixpoint logic rather than first-order logic is the adequate logic to specify semantics of databases with incomplete information. We believe that fixpoint logic should be given far more attention than it has been given in the past.

References

[AN] ANSI/X3/SPARC Interim Report 75-02-08, FDT-SIGMOD 7(1975).

[AU] Aho, A.V., Ullman, J.D.: Universality of data retrieval languages. Proc. 6th ACM Symp. on Principles of Programming Languages, San Antonio, 1979, pp. 110-117.

[BV1] Beeri, C., Vardi, M.Y.: A proof procedure for data dependencies. Technical Report, Dept. of Computer Science, The Hebrew University of Jerusalem, 1980.

[BV2] Beeri, C., Vardi, M.Y.: On the complexity of testing implications of data dependencies. Technical Report, Dept. of Computer Science, The Hebrew University of Jerusalem, 1980.

[C1] Codd., E.F.: A relational model for large shared data banks, Comm. of ACM 13(1970), pp. 377-387.

[C2] Codd, E.F.: Further normalization of the database relational model. In *Data Base Systems* (R. Rustin, ed.), Prentice-Hall, 1972, pp. 33-64.

[C3] Codd, E.F.: Relational completeness of database sublanguage. In *Data Base Systems* (R. Rustin, ed.), Prentice Hall, 1972, pp. 65-98.

[CH] Chandra, A.K., Harel, D.: Structure and complexity of relational queries. J. of Computer and Systems Sciences 25(1982), pp. 99-128.

[Cha] Chandra, A.K.: Programming primitives for database languages. Proc. 8th ACM Symp. on Principles of Programming Languages, 1981, pp. 50-62.

[CK] Chang, C.C., Keisler, H.J.: Model Theory. North-Holland, 1977.

[DST] Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. of ACM 27(1980), pp. 758-771.

[Fa1] Fagin, R.: Monadic generalized spectra. Zeitschr. f. math. Logik und Grundlagen d. Math. 21(1975), pp. 89-96.

[Fa2] Fagin, R.: Horn clauses and database dependencies. J. of ACM 29(1982), pp. 252-285.

[Fe] Feferman, S.: Two notes on abstract model theory - Properties invariant on the range of definable relations between structures. Fundamenta Math. 82(1974), pp. 153-165.

[Gu] Gurevich, Y.: The word problem for certain classes of semigroups (in Russian). Algebra and Logic 5(1966), pp. 25-35.

[GMV] Graham, M.H., Mendelzon, A.O., Vardi, M.Y.: Notions of dependency satisfaction. Technical Report STAN-CS-83-979, Stanford University, August 1983.

[Ha] Hajek, P.: Some remarks on observational model-theoretic languages. Proc. 2nd Conf. on Set Theory and Hierarchy Theory, 1975. Lecture Note in Mathematics - Vol. 537, Springer-Verlag, 1976, pp. 335-345.

[Ho] Honeyman, P.: Testing satisfaction of functional dependencies. J. ACM 29(1982), pp. 668-677.

[Im] Immerman, N.: Relational queries computable in polynomial time. Proc. 14th ACM Symp. on Theory of Computing, San Francisco, 1982, pp. 147-152.

[JL] Jones, N.D., Laaser, W.T.: Complete problems in deterministic polynomial time. Theoretical Computer Science 3(1977), pp. 105-117.

[McKi] McKinsey, J.C.C.: The decision problem for some classes of sentences without quantifiers. J. of Symbolic Logic 8(1943), pp. 61-76.

[MSY] Maier, D., Sagiv, Y., Yannakakis, M.: On the complexity of testing implications of functional and join dependencies. J. of ACM 28(1981), pp. 680-695.

[MZ] Makowsky, J.A., Zvieli, A.: Definable queries. Unpublished manuscript, Technion-Israel Inst. of Technology, 1982.

[MUV] Maier, D., Ullman, J.D., Vardi, M.Y.:The revenge of the JD. Proc. 2nd ACM Symp. on Principles of Database Systems, Atlanta, 1983, pp. 279-287.

[Sa] Sagiv, Y.: Can we use the universal instance assumption without using nulls? ACM Symp. on Management of Data, 1981, pp. 108-120.

[Var1] Vardi, M.Y.: Global decision problem for relational databases. Proc. 22nd IEEE Symp. on Foundation of Computer Science, Nashville, October 1981, pp. 198-202.

[Var2] Vardi, M.Y.: The complexity of relational query languages. Proc. 14th ACM Symp. on Theory of Computing, San Francisco, 1982, pp. 137-146.

[Y] Yannakakis, M.: Algorithms for acyclic databases. Proc. Int'l Conf. on Very Large Data bases, Nice, 1981, pp. 82-94.

84/22

Algorithms for Providing Reliability in Unreliable Database Systems

Marc Graham

Georgia Institute of Technology
Atlanta, Georgia 30332

Nancy Griffeth

Georgia Institute of Technology
Atlanta, Georgia 30332

Barbara Smith-Thomas

University of North Carolina at Greensboro
Greensboro, North Carolina 27412

ABSTRACT

Transactions are the atomic units of work in a database system. This implies, in the most general case, that concurrent transactions must be run serializably and that a transaction be run to completion before it affects long-term system state. In this paper, we consider the latter property. Transaction schedulers are described which guarantee that incomplete transactions do not affect long-term system state: this property is proven for each scheduler. Also, the interaction with serializability is considered.

1. Introduction.

A database management system is expected to provide "atomicity" for the transactions which use it. The system provides the illusion that the transactions are atomic operations without internal structure. If we assume that every transaction submitted to the system successfully completes, serializability theory is sufficient for understanding atomicity.

Of course, real world computing is subject to failure. Beyond that, all popular concurrency control techniques are capable of aborting transactions, i.e., causing them to fail. Locking policies use aborts to prevent or resolve deadlock; nonlocking policies use it as their basic tool for providing serializability. Atomicity requires that aborted transactions appear to have never executed. The system must be able to recover from aborted transactions; this capability is called *recoverability*. The idea of *recoverable schedules* was defined in [H83]. In a recoverable schedule, a transaction does not commit until there is no possibility that it will be rolled back. If the underlying hardware or software is unreliable, this must not occur until all transactions which have written values read by the transaction have themselves committed.

A common solution for guaranteeing that a schedule is recoverable is to "hold write-locks to commit point," or more generally, to prohibit any access to a data-item which has been changed by a transaction until the transaction commits. Such a policy guarantees that no other transaction can ever read or overwrite a data-item value until the previous transaction writing it has committed. This prevents any transaction from committing before the values it has read have been committed by the writing transaction.

This policy fits well with two-phase locking, since write-locks are held to "lock point" anyway, but does not fit so well with timestamping protocols or even with locking protocols which allow earlier release of locks. Also, this policy may not fit at all if serialization is not required. We call this policy "pessimistic" and describe three other policies which appear to be worth considering: an "optimistic" policy, which does not block but will abort a transaction trying to commit if it has read data written by an aborted transaction, a "realistic policy," which blocks reads but not writes of uncommitted data, and a "paranoid policy" which aborts a transaction which attempts to access uncommitted data. These four policies can be compared with respect to their effect on concurrency, on the number of aborts, and on preservation of membership in serializability classes.

The effect on concurrency can be summarized as follows: The pessimistic policy allows the least concurrency of the four. In contrast, the optimistic policy does not reduce concurrency at all, the realistic policy reduces concurrency somewhat, and the paranoid policy seems, on the basis of simulations [GM84], to fall between realism and pessimism. This contrasts with the number of aborts a policy introduces. The optimistic policy will cascade aborts. The pessimistic policy may introduce many aborts due to deadlocks but it will not cascade aborts. The paranoid policy will introduce aborts but is deadlock free. The realistic policy will not introduce aborts if we can assume that all reads of a transactions precede all writes.

Finally, with respect to preservation of serializability classes, the pessimistic and realistic policies preserve membership in the class of DSR schedules (schedules that can be serialized by swapping non-conflicting adjacent operations). The optimistic and paranoid policies preserve membership in a somewhat larger class of schedules, but it do not preserve membership in SR. The realistic policy preserves membership in SR, the class of all serializable schedules, assuming only that all reads in a transaction precede all writes. The primary practical implication of these facts is that the realistic scheduler appears better in all cases than the pessimistic scheduler. Simulation and analysis comparing the various policies are ongoing. The simulations indicate that, as expected, the optimistic scheduler performs better than the other schedulers if few transactions abort, and, surprisingly, the paranoid scheduler has better performance than the pessimistic scheduler in some cases [GM84].

The recovery schedulers described here can be used in a wide variety of systems, with many different criteria for correctness of a schedule. In particular, the properties of recoverability and serializability have been considered separately, so that the recovery policies can be used even for transaction systems which do not require serializability, such as described by several authors [L82, G81, A83].

2. Definitions and Preliminaries.

We mostly follow Hadzilacos [H83]. However, we alter the notion of the meaning of a database operation to reflect the intention that aborted transactions should not affect subsequent transactions.

Let $D = \{ x,y,z,... \}$ be a set of data items. Database operations are $R_t[x]$, $W_t[x]$, C_t, A_t, $t \in T$, $x \in D$. These symbols are intended to represent, respectively, a read of data item x by transaction t, a write of x by transaction t, a commit of transaction t, and an abort of transaction t. We use the notation $D_t[x]$ to mean either $W_t[x]$ or $R_t[x]$, and E_t to mean either A_t or C_t. Two operations *conflict* if they are read or write operations accessing the same data item and at least one of them is a write, or if one of them is a commit or abort.

A *transaction*, t, is a partially ordered set $(op_t, <_t)$ where

$$op_t \subseteq \{ R_t[x] , W_t[x] : x \in D \}$$

and satisfying the following:

i) $A_t \in op_t$ iff $C_t \notin op_t$;
ii) if $A_t \in op_t$, then $\forall a \in op_t - \{ A_t \}, a <_t A_t$;
iii) if $C_t \in op_t$, then $\forall a \in op_t - \{ C_t \}, a <_t C_t$;
iv) any two conflicting operations are ordered by $<_t$.

Our definition of a transaction is slightly more general than is usual: for instance, we allow a transaction to write twice to the same data item, or to read a data item it has previously written. We do not require all reads to precede all writes. Observe that $<_t$ is not restricted to the pairs required by i)-iv) above; additional pairs will generally be of the form ($R_t[x]$, $W_t[y]$) and should be taken to imply that the value written to y depends on the value read from x.

Let T be a set of transactions; let $OP = U_{t \epsilon T}\, op_t$. A log L is a pair (OP , <) where < is a a partial order on OP which respects the transaction orders $<_t$. We further require that < order each pair of conflicting operations in OP (and possibly other pairs as well). We will write OP(L) and $<_L$ when necessary to avoid ambiguity.

A transaction, t, is *committed* in a log L if $C_t \epsilon OP$; t is *aborted* in L if $A_t \epsilon OP$; t is *active* if it is neither committed nor aborted. We let COM(L) denote the set of committed transactions in L, ABO(L) the set of aborted transactions in L, and T-ABO(L) the set of non-aborted transactions.The projection of a log, L, onto a subset τ of its transactions, denoted by $\Pi_\tau(L)$, is the restriction of L to $U_{t \epsilon \tau} op_t$. We will be most interested in the projection of a log onto its committed transactions: $\Pi_{COM(L)}(L)$.

For mathematical simplicity we extend all logs with (fictitious) initializing and terminating transactions t_0 and t_f. The initializing transaction t_0 writes all database items and then commits before any other transaction begins execution. The terminating transaction t_f reads all database items after all other transactions have committed and then commits. Thus, any read of a data item is preceded by a write to that data item, and any write of a data item is followed by a read of that data item.

We define an *immediately preceding write* relation on OP determined by <, as follows. For $D_t[x] \epsilon OP$, we write $W_u[x] \lessdot D_t[x]$ if $W_u[x] < D_t[x]$ and for every $W_v[x]$ with $W_u[x] < W_v[x] < D_t[x]$ we have $W_v[x] < A_v < D_t[x]$. We often write $W_u \lessdot D_t$ when the data item involved is immaterial; however, by definition W_u and D_t operate on the same item. When $W_u[x] \lessdot R_t[x]$, we say t "reads x from" u. We now define a meaning function M_L, in the standard way, for data accesses:

$$\begin{aligned} &M_L(R_t) = M_L(W_u) \text{ where } W_u \lessdot R_t \\ &M_L(W_t[x]) = g_{tx}((M_L(R_t^1), \ldots M_L(R_t^k)) \end{aligned}$$

where $\{ R_t^i \mid 1 <= i <= k \}$ is the set of all reads by t with $R_t <_t W_t[x]$ and g_{tx} is an uninterpreted function. Although our meaning function is defined in the standard way, the relation $\lessdot$ is non-standard as it takes aborts into account. This contrasts with the treatment given by Hadzilacos in [H83].

A log L is *recoverable* if for every prefix L' of L the meaning of each committed read and write in L' is the same as its meaning in $\Pi_{COM(L')}(L')$. The major implication of this definition is that if we want recoverability we must not allow a transaction to commit unless all transactions whose values it has read have previously committed.

We can also define serializable logs or logs having any other properties not involving the commit and abort operations. We say that a log L belongs to such a class of logs if $\Pi_{COM(L)}(L)$ belongs to the class. The classes which will be discussed in this paper are SR, DSR, and 2PL, á lá Papadimitriou [P79], and two other classes introduced here. A log is *serial* if there is no pair $D_t[x]$, $D_t[y]$ of data accesses of transaction t such that for some data access $D_u[z]$ with $u \neq t$, $D_t[x] < D_u[z] < D_t[y]$. Logs L and L' are *equivalent* if $M_L(R_f[x]) = M_{L'}(R_f[x])$ for all data items x in the database. A log is *serializable* if it is equivalent to a serializable log. Logs L and L' are *D-equivalent* ($L \sim L'$) if L and L' are identical up to an interchange in < of two non-conflicting data accesses. Define $\approx$ to be the transitive closure of $\sim$. A log L is *D-serializable* (DSR) if $L \approx L'$ for some serial log L'. We use p_L to denote position in

L; thus $\rho_L(W_i[x]) = n$ means that $W_i[x]$ is the nth element in $<_L$. A log L is *two-phase locked* if there exist distinct real numbers $l_1, \ldots, l_n$ and a partition of each $op_t \subseteq OP$ into g_t (the growing phase) and s_t (the shrinking phase) such that the following conditions hold:

i) for all $D_t \in g_t$, $\rho_L(D_t) < l_t$ and for all $D_t \in s_t$, $l_t < \rho_L(D_t)$;

ii) for conflicting operations $D_t \in g_t$ and D_u, if $\rho_L(D_t) < \rho_L(D_u)$ then $l_t < \rho_L(D_u)$ and $l_t < l_u$;

iii) for conflicting operations $D_t \in s_t$ and D_u, if $\rho_L(D_t) < \rho_L(D_u)$ then $\rho_L(D_t) < l_u$.

Finally, we introduce two new serializability classes. We say that a set S of transactions is *restricted* in a log L if $t \in S$ and $W_u[x] <_L R_t[x]$ implies that $u \in S$. A log is *restricted project serializable* (RPSR) if the projection of the log onto any restricted set is serializable. A log is *project serializable* (ΠSR) if the projection of the log onto an arbitrary subset of the set of transactions is serializable. Clearly, $\Pi SR \subseteq RPSR \subseteq SR$. It is a consequence of Lemma 3.1, proved below, that $DSR \subseteq \Pi SR$. All containments are proper, as the following examples show:

A log which is SR but not RPSR: $R_2[x]W_2[x]W_1[x]W_3[x]$

A log which is RPSR but not ΠSR: $R_1[z]W_1[z]R_2[z]R_3[x]W_2[x]W_3[x]W_1[x]$

A log which is ΠSR but not DSR: $R_1[z]W_2[z]R_3[z]W_3[x]W_1[x]$

3. Recovery Schedulers.

The transaction system consists of a collection of transaction processes and two online schedulers (a *serializer* and a *recovery scheduler*) which communicate with each other by passing a log back and forth. The transactions submit operations (C_t, A_t, $R_t[x]$, and $W_t[x]$) asynchronously to the serializer. The serializer's role is to guarantee that the operations interleave acceptably. (We call it a serializer since serializability is frequently required, but the model used here does not actually require that the so-called serializer produce serializable logs. In fact, the serializer need not have *any* effect, simply adding operations to its output log as they arrive.)

The serializer output is a sequence of logs. Each log in this output is input for one step of the recovery scheduler. The recovery scheduler modifies its input log I_j, returns the new log (its output log O_j) to the serializer, and passes a log X_j which contains a subset of the operations in O_j to the system for execution. The serializer needs to check that O_j is still an acceptable interleaving of the operations. It uses O_j and any new operations submitted by the transactions to produce another input I_{j+1} for the recovery scheduler. Figure 1 illustrates this behavior.

A transaction submits operations one at a time to the serializer. The recovery scheduler will often process operations in its input log in the log order. Operations which have been entered in the execution log are not processed a second time, but all other operations may be processed. The scheduler can make one of the following choices for each operation it processes:put the operation in the execution log X and the output log O immediately; defer the operation, usually adding it to the end of O; or abort the submitting transaction.

The behavior of the transaction system is constrained by the following rules:

(1) Transactions are well-formed, that is, each transaction is a partially ordered set of operations terminated by either a commit or an abort;

(2) $OP(I_j) - \{C_t : t \in T\}$ is contained in $OP(O_j)$ and $OP(O_j) - \{A_t : t \in T\}$ is contained in $OP(I_j)$;

(3) X_j is the restriction of O_j to a subset of its operations;

(4) X_j is the restriction of $I_{j+1}$to a subset of its operations;

(5) X_j is a prefix of X_{j+1}, that is, if $D_t[x] \in X_j$ and $D_u[y] < D_t[y]$ in X_{j+1} then $D_u[y] \in X_j$; and

(6) The serializer does not change any prefix of the log if the prefix could have been serializer output; similarly, the recovery scheduler does not change any prefix of the log if the prefix could have been recovery scheduler output (see Papadimitriou, ...).

The first rule requires only that no transaction submits an operation after it has terminated. The second rule requires the recovery scheduler to honor the data accesses and aborts submitted by the serializer. It is not allowed to make a log recoverable by throwing any operation other than a commit away. In other words, it can discard whole transactions but not individual operations. The second part of the rule prohibits adding any operations other than aborts. The third through fifth rules prohibit the serializer and the recovery scheduler from submitting an operation for execution in one step and taking it back in the next. (The nature of time makes such behavior hard to realize.)

We desire to limit the amount of communication between the serializer and the recovery scheduler. If the serializer does not allow all possible interleavings, then it may need to recheck all of O_j for serializability and it can happen that the serializer and the recovery scheduler take several steps to agree on a log. Consider, for example, the recovery scheduler which defers writes of a transaction until it sees a commit for the transaction, at which time it outputs the write and the commit. Suppose the serializer uses two-phase locking. Let

$$I_1 \quad = R_1[x]W_1[x]W_1[y]R_2[y]W_2[y]C_1C_2$$

Then, assuming the transactions do not submit more operations, the sequence of logs might be:

$$O_1 = R_1[x]R_2[y]W_1[x]W_1[y]C_1W_2[y]C_2$$

$$I_2 = R_1[x]R_2[y]W_1[x]W_2[y]C_2W_1[y]C_1$$
$$O_2 = R_1[x]R_2[y]W_2[y]C_2W_1[x]W_1[y]C_1$$

$$I_3 = O_2$$

In contrast, a recovery scheduler which fit well with a serializer might always produce an O_j that is a possible output of the serializer. In this case, the serializer will not change O_j, by rule (5). Thus we will look for positive results of the form:

> If I_j is in class X of logs and the recovery scheduler has property Y then O_j is in class X.

and negative results of the form:

> No recovery scheduler having property Y produces an output log O_j in class X for every I_j in class X.

We define four potential "property Y"s for recovery schedulers here, three of which are semantic and one syntactic. In all definitions we assume that I_j includes the fictitious initial and a terminal transaction. The first property defined here allows operations to be moved in the log, but only if the move has no effect on meaning (in particular, the scheduler must not move $W_t[x]$ and $R_u[x]$ relative to each other if $W_t[x] \lessdot_{I_j} R_u[x]$). It does not allow introduction of new aborts:

> A recovery scheduler is *strong meaning-preserving* if for every log I_j
> (1) $OP(I_j) = OP(O_j)$ and
> (2) for every D_t in $OP(O_j)$, $M_{I_j}(D_t) = M_{O_j}(D_t)$.

A second property allows a recovery scheduler to introduce aborts and rearrange operations as long as the meaning of non-aborted operations is unchanged:

> A recovery scheduler is *meaning-preserving* if for every log I_j, every t not in ABO(O_j), and every D_t in op_t, $M_{I_j}(D_t) = M_{O_j}(D_t)$. It is *fully meaning-preserving* if it is meaning-preserving and the final transaction is never aborted.

Using a fully meaning-preserving recovery scheduler guarantees that the final database values will be the same in O_j as in I_j. Any meaning-preserving scheduler must cascade aborts if any transaction reads data which was written by a transaction that later aborts. To avoid excessive abortment, we define two less restrictive properties for recovery schedulers, both of which allow the scheduler to pick a different writer for a reader to use when the writer used in the input log I_j aborts. First, we consider a property which refers only to meaning-preservation: we allow meaning to be changed as long as the meaning in O_j is what it would have been in I_j if none of the transactions aborted by the recovery scheduler had been present in I_j.

> A recovery scheduler is *weak meaning-preserving* if the final transaction is never aborted and for every log I_j, every t in T-ABO(O_j), and every D_t in op_t, $M_{I_j'}(D_t) = M_{O_j}(D_t)$, where $I_j' = \Pi_{T\text{-}ABO(O_j)}(I_j)$.

Second, we define a syntactic property which requires the overall order of operations to remain the same.

> A recovery scheduler is *order-preserving* if for every log I_j, every t and u in T-ABO(O_j), and every pair of operations D_1 and D_2 in t and u, respectively, if $D_1 <_{I_j} D_2$ then $D_1 <_{O_j} D_2$.

We will show that the first property is too strong for any situation, the third is too weak, and that while the second property provides some interesting information about recovery schedulers it does not guarantee that it preserves any class of interest, i.e., any class contained in DSR. On the other hand, the property of order-preservation does imply preservation of DSR.

Theorem 3.1. Every strong meaning-preserving scheduler is fully meaning-preserving and every meaning-preserving scheduler is weak meaning-preserving.

Proof: The first part of the theorem follows immediately from the definitions. To show that a meaning-preserving scheduler is weak meaning-preserving, take any u $\in$ T-ABO(O_j) and $D_u \in op_u$. If $D_u = R_u[x]$, then $M_{O_j}(R_u[x]) = W_t[x]$ where $W_t[x] \lessdot_{O_j} R_u[x]$. Meaning-preservation implies that $M_{O_j}(R_u[x]) = M_{I_j}(R_u[x]) = W_t[x]$. Any writes W_v such that $W_t[x] <_{I_j} W_v[x] <_{I_j} R_u[x]$ are aborted in I_j before $R_u[x]$ and therefore they are also aborted in O_j (since O_j must honor aborts). Thus in I_j', which is the restriction of I_j to the transactions in T-ABO(O_j), $W_t[x] <_{I_j'} R_u[x]$ and there are NO writes W_v such that $W_t[x] <_{I_j'} W_v[x] <_{I_j'} R_u[x]$. Therefore $W_t[x] \lessdot_{I_j'} R_u[x]$ and $M_{I_j'}(R_u[x]) = W_t[x] = M_{O_j}(R_u[x])$. It follows immediately that all writes of unaborted transactions also have the same meaning in O_j and I_j'.

[]

Meaning-preservation and order preservation do not have any simple relationship in general. None of the meaning-preservation properties imply order-preservation, as the following example shows:

$$\begin{aligned} I_j &= R_2[x]W_2[x]W_1[x]W_3[x]C_1C_2C_3 \\ O_j &= R_2[x]W_1[x]W_2[x]W_3[x]C_1C_2C_3 \end{aligned}$$

The above transformation from I_j to O_j is perfectly legitimate for any of the meaning-preserving schedulers but it does not preserve the order of $W_1[x]$ and $W_2[x]$. Similarly, order-preservation does not imply either strong meaning-preservation or meaning-preservation as the following example shows:

$$\begin{aligned} I_j &= R_1[x]W_1[x]R_2[x]W_2[x]C_2A_1 \\ O_j &= R_1[x]W_1[x]A_1R_2[x]W_2[x]C_2 \end{aligned}$$

Although the order of the data accesses is unchanged from input to output, the meaning of $R_2[x]$ is $W_1[x]$ in the input and $W_0[x]$ in the output. We note that, in the absence of any aborts in the input, order preservation implies strong meaning-preservation, but this is a particularly uninteresting case when we are studying recovery schedulers. Order-preservation also implies weak meaning-preservation, but we will see below that weak meaning-preservation is too weak a property to be of interest.

We claim that the property of strong meaning-preservation is so strong that any "interesting" class of logs contains at least one log which cannot be made the final transaction reads $W_3[x]$ in I_j and $W_2[x]$ in O_j. Even though the final transaction is a fiction, it is a reasonable fiction, since we might not want to consider meaning to be preserved unless the final results written in the database were the same for both logs.

Theorem 3.4. Every order-preserving recovery scheduler preserves the serializability classes ΠSR and DSR.

Proof: It follows immediately from the definition that $\Pi_{T\text{-}ABO(O_j)}(I_j) \approx .\Pi_{T\text{-}ABO(O_j)}(O_j)$ Preservation of ΠSR follows from this.

In order to show that our schedulers preserve the property of being DSR we establish the following variant of Papadimitriou's Corollary 2 [P79]:

Let $S(W_i)$, resp $S(R_i)$, denote the set of data items written by, resp. read by, transaction t_i. Recall that ρ_L denotes position in the temporal sequencing of L; thus $\rho_L(W_i[x]) = n$ means that $W_i[x]$ is the nth element in $<_L$. We write $\rho_L(W_i) < \rho_L(W_j)$ if there is an $x \epsilon S(W_i) \cap S(W_j)$ with $\rho_L(W_i[x]) < \rho_L(W_j[x])$. This is not, in general, a partial order.

Lemma 3.1: A log, L, on transactions $t_1, t_2, \ldots, t_n$ is DSR iff there exist real numbers $S_1, S_2, \ldots, S_n$ such that $S_i < S_j$ if any of the following hold:

a) $S(W_i) \cap S(W_j) \neq \phi$ and $\rho_L(W_i) < \rho_L(W_j)$ (t_i writes some x after which t_j also writes x);

b) $S(R_i) \cap S(W_j) \neq \phi$ and $\rho_L(R_i) < \rho_L(W_j)$ (t_i reads some x after which t_j writes x);

c) $S(W_i) \cap S(R_j) \neq \phi$ and $\rho_L(W_i) < \rho_L(R_j)$ (t_i writes some x after which t_j reads x).

Proof: Suppose L is DSR; let L′ denote a serialization of $L \cap \{R_t[x], W_t[x] : t \epsilon N, x \epsilon D\}$ obtained by interchanging non-conflicting operations. For each transaction t_i let S_i be the position in L′ of the first operation on t_i's behalf. The S_i's have the required property because if either a), b), or c) holds, then in L′, and therefore in L, all of t_i's operations occur before t_j's operations.

For the converse we show that if numbers $S_1, S_2, \ldots, S_n$ as described exist, then all operations of each transaction can be collected into adjacent locations by interchanging non-conflicting operations in $L \cap \{R_t[x], W_t[x]\}$. Suppose that for some pair $D_j[y]$, $D_i[z]$ of operations, $S_i < S_j$. Take the last operation $D_j[y]$ between $D_i[x]$ and $D_i[z]$ satisfying:

$$\rho_L(D_j[y]) < \rho_L(D_i[z]) \text{ and } S_i < S_j. \qquad (*)$$

Then we claim that $D_j[y]$ does not conflict with any operation $D_k[w]$ with

$$\rho_L(D_j[y]) < \rho_L(D_k[w]) < \rho_L(D_i[z])$$

or with $D_i[z]$ itself. But $S_i < S_j$ means that there is no $x \in D$ such that $\rho_L(D_i[x]) > \rho_L(D_j[x])$ for conflicting operations D_i and D_j. Therefore, $D_i[z]$ and $D_j[y]$ do not conflict. Similarly, since $S_k < S_i < S_j$, $D_k[w]$ and $D_j[y]$ do not conflict. As a result, we can construct a D-equivalent log by moving $D_j[y]$ immediately to the right of $D_i[z]$

Repeating this procedure for every such pair will eventually produce a serial log. The procedure terminates because we reduce the number of pairs satisfying (*) at each step.

[]

The following immediate consequence of Lemma 3.1 will prove most useful.

Corollary: Let L be a log; let ι be any subset of the transactions of L. If L is DSR then $\Pi_\iota(L)$ is DSR.

In particular if L is DSR and recoverable then $\Pi_{COM(L)}(L)$ is a DSR log in which every $R_t[x]$ has the same meaning it had in L. Thus $\Pi_{COM(O)}(O)$ is DSR, that is, an order-preserving recovery scheduler preserves serializability.

[]

We conclude this discussion of meaning-preservation by showing that weak meaning-preservation does not preserve any serializability classes of interest and meaning-preservation does not preserve any of the "syntactically" defined classes.

Theorem 3.5. A weak meaning-preserving recovery scheduler need not preserve SR or RPSR. No meaning-preserving scheduler need preserve any serializability class contained in ΠSR.

Proof: We use counterexamples to establish the theorem. First, consider the input log:

$$I_j \quad = W_4[z]R_1[z]W_1[z]R_2[z]R_3[x]W_2[x]W_3[x]W_1[x]C_1C_2C_3A_4$$

We show first that this log is RPSR and then display an output log which is not SR but which would be permitted by some weak meaning-preserving recovery scheduler. To see that the log is RPSR, note that the restricted sets are {1}, {3}, {1,2}, {1,3}, and {1,2,3} since 2 reads z from 1. The projection onto 1 and 2 is serializable in the order 21 (2 is dead). The projection onto 1 and 3 is DSR (the order is 31). The entire log is serializable in the order 321, with 2 dead. The output log

$$O_j \quad = W_4[z]R_1[z]W_1[z]A_4W_1[x]A_1R_2[z]R_3[x]W_2[x]W_3[x]C_2C_3$$

satisfies the weak meaning-preservation property, since the projection of I_j onto 2 and 3 is equal to the projection of O_j onto 2 and 3. But this projection is

$$R_2[z]R_3[x]W_2[x]W_3[x]C_2C_3$$

which is not serializable.

To establish the claim that no meaning-preserving scheduler (strong, weak or ordinary) need preserve any serializability class which is contained in ΠSR, consider the serial input log:

$$I_j = R_1[x]W_1[x]W_2[x]W_3[x]C_1C_2C_3$$

A strong meaning-preserving scheduler could output the log:

$$O_j = R_1[x]W_2[x]W_1[x]W_3[x]C_1C_2C_3$$

but this log is not ΠSR, since the projection onto transactions 1 and 2 is the nonserializable log:

$$R_1[x]W_2[x]W_1[x]$$

[]

Thus the property of meaning-preservation does not tell us when a recovery scheduler will preserve the classes of logs most likely to represent output from an online serializer, that is, those logs contained in DSR.

Most of the algorithms developed in the next few sections are order-preserving. Two of them are also meaning-preserving. We assume that each transaction starts with a 'begin transaction t' message to the scheduler, which has the effect of causing the scheduler to allocate and initialize any required data structures.

4. Pessimism.

A pessimistic scheduler blocks all data accesses to uncommitted data. It is the policy used by a 2PL system in which locks are held to commitment. It proceeds as follows:

Data Structures--

```
lock data:
    for each data item x ∈D
        committed [x]:  Boolean, initialized to true
        if not committed [x] then
            owner [x]:       transaction id
            waiting for [x]: queue of (transaction id, operation) pairs

transaction data for each active transaction:
    t.locked:             set of data items
    t.waiting_on:         data item

global data:
    waits_for_graph:      graph whose node set is T, initially empty.
    blocked:              queue of operations, initially empty
```

Proof is by induction on the number of writes of data item x. If there are only two writes to x they must be $W_{t_0}[x]$ and $W_t[x]$. By assumption C_{t_0} appears in O before any "real" operation and after all the "imaginary" initializing writes.

If there are $j > 2$ writes to x in O let the last two be $W_t[x]$ and $W_u[x]$. If the request write (u ,x) found committed[x] = true then release_locks(t) must have previously been executed; release_locks(t) is only executed following the entering of C_t or A_t in O. If the request write (u ,x) found committed[x] = false then (u, write) was added to the queue waiting_for [x]. In this case $W_u[x]$ is entered in O by release_locks(t) after entering either C_t or A_t and zero or more $R_v[x]$ into O. []

Observe that the Pessimistic Scheduler preserves the temporal order of requests on behalf of a transaction, and preserves the order of conflicting reads and writes.

Theorem 4.1: The Pessimistic Scheduler is a recovery scheduler.

Proof: That O = I if I could have been an output of the Pessimistic Scheduler follows almost immediately from Lemma 4.1. The prefix of I that is equal to X has this property follows from Lemma 4.1. Consider the remaining operations. These operations are accesses to uncommitted data and will therefore be entered in the queue *blocked* when the Pessimistic Scheduler first encounters them. When it has reached the end of the log, it will copy these operations from the queue to O, without changing their order. Thus O = I.

To show that O is recoverable, we actually prove a stronger assertion, namely that for all reads $R_t[x]$ in O, if O' is the initial segment up to and including $R_t[x]$, and if lcw [x] = "last committed writer of x" in O', then either

$$M_O(R_t[x]) = M_{O'}(W_{lcw(x)}[x])$$

or

$$M_O(R_t[x]) = M_{O'}(W_t[x]).$$

It follows that for every prefix O' of the output O and $t \in COM(O)$, $M_{O'}(R_t[x]) = M_{\Pi_{COM(O')}(O')}(R_t[x])$. Since the scheduler respects the order of the requests on behalf of t, it also follows that $M_{O'}(W_t[x]) = M_{\Pi_{COM(O')}(O')}(W_t[x])$.

So let $R_t[x]$ be in O; write O as $O_1 R_t[x] O_2$. $R_t[x]$ can have been entered in O by read or by release_locks. If $R_t[x]$ was entered by read then at the time $R_t[x]$ was entered either owner [x] = t or committed [x] = true. In the first case the preceding write to x was by t and $M_O(R_t[x]) = M_{O'}(W_t[x])$; in the second case the last preceding write, $W_v[x]$, in O' must be followed by a C_v or an A_v. By lemma 1 any $W_u[x]$ preceding $W_v[x]$ is either committed or aborted. Thus $M_O(R_t[x]) = M_{O'}(W_{lcw(x)}[x])$.

If $R_t[x]$ was entered in O by release_locks then $R_t[x]$ is one of group $R_{t_1}[x] \dots R_{t_k}[x]$ immediately preceded by either a C_v or an A_v, which is preceded by a $W_v[x]$. Again we see from Lemma 1 that all writes $W_u[x]$ which precede $R_t[x]$ are either committed or aborted and so $M_O(R_t[x]) = M_{O'}(W_{lcw(x)}[x])$.

[]

Pessimism is not meaning preserving. To illustrate this claim, consider the input

$$W_1[x]R_2[x]A_1C_2$$

In the output, transaction 2 reads from the initial transaction. Pessimism is meaning preserving in the absence of aborts. However, as we show in the following theorem, pessimism is order-preserving.

Theorem 4.2: The pessimistic scheduler is order-preserving.

Proof: Let D_1 and D_2 be conflicting data accesses with $D_1 <_I D_2$. The Pessimistic Scheduler processes D_1 and D_2 in the order they appear in the input log. Thus if D_1 is not deferred we will have $D_1 <_O D_2$. If D_1 is blocked, then some conflicting write precedes D_1. If the latest write is committed or aborted before D_2 is processed then D_1 is output before D_2 is processed. Otherwise D_2 is queued behind D_1.

[]

The following example shows that the Pessimistic Scheduler does not preserve 2PL. Let

$$I_j = W_1[x]R_2[y]R_3[x]W_2[x]W_4[y]$$

and

$$X_{j-1} = W_1[x]R_2[y]$$

I_j is a 2PL log. (Locks can be inserted as follows:

$WL_1[x]W_1[x]U_1[x]RL_2[y]R_2[y]RL_3[x]R_3[x]U_3[x]WL_2[x]W_2[x]U_2[x,y]WL_4[y]W_4[y]$.)

The output of the Pessimistic Scheduler corresponding to this input is

$$O_j = W_1[x]R_2[y]W_4[y]R_3[x]W_2[x]$$

and

$$X_j = W_1[x]R_2[y]W_4[y]$$

O_j is not 2PL. Transaction 2 must release its lock on y before $W_4[y]$, but it cannot lock x until after $R_3[x]$. Therefore transaction 2 cannot be two-phase in a legal log.

Examination of the log with the locks will show that the Pessimistic Scheduler should have blocked $W_4[y]$, because transaction 2 still held a read lock on y. If the Pessimistic Scheduler operates on locks and unlocks instead of reads and writes, however, we find that it then enforces the policy of holding write-locks to commit point. The required changes to the Pessimistic Scheduler are summarized here:

i) Any operation requested by a blocked transaction is deferred and appended to the end of the blocked queue;

ii) An operation requested by a transaction which is not blocked is dealt with as follows:

a) Reads, writes, and unlocks of read locks are output immediately ;

b) Unlocks of write locks are deferred and the requesting transaction is blocked;

c) Read and write locks of locked data items are deferred and the requesting transaction is blocked; read and write locks of unlocked data are output immediately;

d) Aborts and commits are output immediately after all the unlocks of write locks held by the transaction being terminated. After the commit or abort has been output, subsequent locks on the newly unlocked data items are output, the corresponding transactions are unblocked, and execution of the other operations in the blocked queue is reconsidered.

This example illustrates that the serializer and the recovery scheduler may not "fit" together well unless they use they are scheduling the same operations. Thus a recovery scheduler which fits well with *all* serialization policies is unlikely.

5. Optimism

The Pessimistic Scheduler enforces recoverability by severely limiting concurrency. If there is little likelihood of spontaneous aborts by transactions, the reads and writes can be allowed as they come in as long as commits are reordered by the scheduler to enforce recoverability.

Optimism. Allow arbitrary reads and writes; hold commits until all values read are committed. Abort transactions which read data which is later aborted.

Data Structures --

```
local wait data:
    for each x ∈D
        committed [x]:            Boolean, initialized to true
        pending_writers [x]:      queue of transactions, initially nil

transaction data:
    for each active transaction a record
        t.commit_requested:       Boolean, initialized to false
        t.read_from:              list of transactions, initially nil
        t.wrote_to:               list of transactions, initially nil
        t.items_written:          list of database items, initially nil

global data:
    waits_for_graph:   graph with node set T, initially empty edge set
    abort_queue:  queue of transactions, initially empty
```

Procedures -- in response to database requests the Optimistic Scheduler takes the following actions; processing of aborts is deferred until all other operations have been processed. After all operations in the log have been processed, process_aborts is called:

```
read (t,x) =
          (enter $R_t[x]$ in X,O
          if not committed [x]
          then add last(pending_writers [x]) to t.read_from
               add t to last(pending_writers [x]).wrote_to)

write (t,x) =
          (enter $W_t[x]$ in X,O
          add x to t.items_written
          committed [x] ← false
          add t to pending_writers [x])

commit (t) =
          (if t.read_from = nil
          then put t on commit_list
               commit (commit_list) (* see below *)
          else (* someone t read from is not yet committed *)
               t.commit_requested ← true
               for each u ∈ t.read_from
                    add < t,u > to waits_for graph)

commit (commit_list) =
          (while commit_list ≠ nil
               t ← first element of commit_list
               remove t from commit_list
               enter $C_t$ on X,O
               for each u ∈ t.wrote_to
                    remove t from u.read_from
                    if u.commit_requested
                    then remove < u , t > from waits_for_graph
                         if u.read_from = nil
                         then add u to commit_list
               for each x ∈ t.items_written
                    if t = last(pending_writers [x])
                    then committed [x] ← true
                    delete t from pending_writers [x]
               remove t from waits_for_graph)

abort (t) =
          (put t on abort_queue )

process_aborts (abort_queue) =
          (while abort_queue ≠ nil
               t ← first item on abort_queue
               remove t from abort_queue
               enter $A_t$ on S
               for each u ∈ t.wrote_to
```

```
                if u ∉ abort_queue
                then add u to abort_queue
            remove < u , t > from waits_for_graph
        for each u ∈ t.read_from
            remove t from u.wrote_to
            remove < u , t > from waits_for_graph
        for each x ∈ t.items_written
            remove t from pending_writers [x]
            if pending_writers [x] = nil
            then committed [x] = true
        remove t from waits_for_graph)
```

If a cycle occurs in the waits for graph the transactions may be *all committed* provided a means exists for atomically committing multiple transactions. We assume such a means exists.

Observe that the Optimistic Scheduler outputs reads and writes in the same order in which they are received. Also, if transactions may continue to submit requests after they have been aborted by the scheduler some means must be provided to ignore later requests.

Theorem 5.1: The Optimistic Scheduler is a recovery scheduler.

Proof: A transaction, t, is only allowed to commit if t.read_from = nil, that is, all data read by t has previously been committed. Thus O is recoverable.

Let l be a log which could have been output from the optimistic scheduler. Then all aborts appear at the end of the log and no commit is requested by a transaction until all of the data it has read from has been committed. Thus each commit will be output immediately. The aborts will be enqueued in the order they are encountered and then processed in exactly that order. No new aborts will need to be added to the queue since l could have been output; therefore exactly that set of aborts is processed.

□

Theorem 5.2: The Optimistic Scheduler is order preserving and meaning-preserving but not fully meaning-preserving.

Proof: Reads and writes are output immediately by the Optimistic Scheduler. Therefore it is order-preserving. To see that it is meaning-preserving, let $R_t[x]$ be any read operation of a transaction t in T - ABO(O). Suppose that $W_u[x] <_l R_t[x]$. Then transaction u is also in T - ABO(O). This is true because, at the time t tries to read x, either u has committed or data item x is still uncommitted, putting t in v.read_from for some transaction v. If v ≠ u then there must be an abort of v in l and therefore also in O, so that t would be aborted by process_aborts. Therefore v = u. If u aborted during process_aborts, then t would also have been aborted.

To see that the Optimistic Scheduler is not fully meaning-preserving, consider the log:

$$l_j \quad = R_1[x]W_1[x]A_1R_f[x]C_f$$

The abort of transaction 1 is postponed to the end of the log, forcing an abort of the final transaction.

□

The importance of postponing processing of aborts until the end of the log is illustrated by the following log:

$$I_j \quad = R_1[x]W_1[x]R_2[x]W_2[x]C_2A_1R_3[x]W_3[x]$$

If aborts were processed immediately, the Optimistic Scheduler would abort transactions 1 and 2 but not 3. The meaning of $R_3[x]$ would then be changed from $W_2[x]$ to $W_0[x]$.

Let $RW = \{R_t[x], W_t[x] \mid x \in D \text{ and } t \in T\}$

Theorem 5.3: Optimism has the property that

$$\Pi_{COM(O)}(O) \cap RW = \Pi_{COM(O)}(I) \cap RW$$

and therefore preserves 2PL.

Proof: Reads and writes are output immediately and unchanged.

□

6. Realism

Although the Optimistic Scheduler is "correct" it has the potentially troublesome property that aborts can cascade. The problem is caused by undisciplined reading of uncommitted data. In our Realistic Scheduler we hold up reads until committed values are available. Since we wish not to postpone overwrites we must either tolerate the possibility of starvation or keep a list of values for overwritten data items. We choose to do the latter, so that the realistic scheduler requires multiple versions of the data items.

Whenever a read is output (in O_j or X_j), the data item is subscripted with the transaction whose write it should read. Thus $R_t[x_u]$ means that transaction t reads data item x from transaction u. We require that $W_u[x]$ precede $R_t[x_u]$ in the log. To make this work correctly, we alter the meaning function so that $M_L(R_t[x_u]) = W_u[x]$. As a result, even though reads are blocked, it will not be necessary to change their position in O_j, although of course they will appear as "late reads" in X_j. Consider, for example, the input log:

$$I_j \quad = R_1[x]W_1[x]W_1[y]R_2[x]R_3[y]W_2[y]W_3[x]$$

Pessimism would block $W_2[y]$ and $W_3[x]$ until transaction 1 has been committed, but the realistic scheduler allows it to proceed. The output log corresponding to this is:

$$O_j \quad = R_1[x_0]W_1[x]W_1[y]R_2[x_1]R_3[y_1]W_2[y]W_3[x]$$

and the execution log is:

$$X_j \quad = R_1[x_0]W_1[x]W_1[y]W_2[y]W_3[x]$$

$R_2[x_1]$ and $R_3[y_1]$ cannot be output yet because the most recent write could be either committed or aborted. If transaction 1 commits, then the recovery scheduler outputs $C_1R_2[x_1]R_3[y_1]$ in the execution log. If it aborts, either because the serializer output an abort or because the recovery scheduler found a deadlock and chose to abort transaction 1, the recovery scheduler outputs $A_1R_2[x_0]R_3[y_0]$ in the execution log. In no case do we change the order of data operations from I_j to O_j.

Realism -- Allow arbitrary writes; hold reads until committed data is available.

Data Structures --

```
local wait data:
          for all $x \in D$
               ops[x]:doubly linked list of triples (op,t, status)
                    where op $\in$ {R,W}
                         status $\in$ {pending, committed}
               initialized to ((W, $t_0$, committed))

transaction data:
          for each active transaction
               t.writes:          set of data items, initially nil
               t.waiting__on:data item, initially nil

global data:
     waits__for__graph:   graph with node set T, initially empty edge set
```

Procedures -- in response to database requests the Realistic Scheduler takes the following actions:

```
read (t,x) =
          (if last(ops [x]) = (W,u, committed)
          then enter $R_t[x_u]$ on X,O
          else if last(ops [x]) = (W, u, pending) (* last unaborted write is pending *)
                then enter $R_t[x_u]$ on O
                      add (R,t, pending) to ops [x]
                      t.waiting_on ← x
                      add < t, u > to waits_for_graph where (W,u, pending) is
                      the last write triple on ops [x])

write (t,x) =
          (enter $W_t[x]$ on X,O
          add (W,t, pending) to ops [x]
          add x to t.writes)

commit (t) =
          (enter $C_t$ on X,O
          for each $x \in t$.writes
                in ops [x] change (W,t, pending) to (W,t, committed)
                cleanup (x, (W,t, committed)) (* see below *)
          remove t from waits_for_graph)

cleanup (x, (W,t, committed)) =
          (while (R,u, pending) is next item after (W,t, committed) on ops [x]
                enter $R_u[x_t]$ on X
                delete (R,u, pending) from ops [x]
                u.waiting_on ← nil
                remove < u , t > from waits_for_graph
          if next item on ops [x] is (W, v, committed) then
                delete (W,t, committed) from ops [x]
          if previous item on ops [x] is (W,v, committed) then
                delete (W, v, committed) from ops [x])

abort (t) =
          (enter $A_t$ on X,O
          for each $x \in t$.writes
                delete (W, t, pending) from ops [x]
                if preceding item in ops [x] = (W,u, committed)
                then cleanup (x, (W,u, committed))
          remove t from waits_for_graph)

deadlock_recovery =
          (while there is a cycle in waits_for_graph
                choose t on a cycle
                for x = t.waiting_on
                      enter $R_t[x_u]$ on X
                      delete (R,t, pending) from ops [x]
                abort (t))
```

The pairs $R_t[x_u]$ place in the execution log by the Realistic Scheduler are intended to instruct the database manager to return to t the value of x written by u; this instruction is issued only after u has committed and x may have been overwritten many times in the interval between the requests read(t,x) and commit(u).

To examine the meaning-preserving and order-preserving properties of the Realistic Scheduler, we construct an order on a single-version log which gives the same meaning to each non-aborted operation as the logs O and X. If one tries to construct an order on all the operations, the simultaneous restrictions of respecting the internal order of the transactions and reordering reads so that only eventually committed data is read leads to cycles. Fortunately, such cycles must involve transactions which are aborted. Since we wish to eliminate the effects of aborted transactions anyway, we will only construct our order on $\Pi_{COM(O)}$ O.

Definition: Let $C(O) = \Pi_{COM(O)}(O)$; let $\ddot{O} = C(O) - \{ R_t[x_u] \}$. Define <' on C(O) by

a) for all t $<' |_t = <_t$

b) $<' |_{\ddot{O} \times \ddot{O}} = < r_T |_{\ddot{O} \times \ddot{O}}$, where $< r_T$ is the restriction of the order on O to conflicting operations

c) for each $R_t[x_u] \in C(O)$

i) $W_u[x] <' R_t[x_u]$

ii) for each $W_v[x] \in C(O)$ with $W_u[x] < W_v[x]$, $R_t[x_u] <' W_v[x]$. That is, $R_t[x_u]$ falls 'logically' between $W_u[x]$ and any subsequent committed writes to x

iii) for each $C_v \in C(O)$ let $C_v <' R_t[x_u]$ if $C_v <_I R_t[x]$ and let $R_t[x_u] <' C_v$ if $R_t[x] <_I C_v$

Let $<_O$ be the reflexive and transitive closure of <'.

Lemma 6.1: If $a_1 <' a_2$ then $a_1 <_I a_2$.

Proof: If neither of a_1 and a_2 is a read then we let $a_1 <' a_2$ using condition b. above, that is $a_1 <_I a_2$. If both a_1 and a_2 are reads then they must both be from the same transaction and thus $a_1 <' a_2$ iff $a_1 <_t a_2$ iff $a_1 <_I a_2$. Condition c. iii) correctly orders reads and commits so it only remains to verify that reads and writes are ordered correctly. But this follows from the way read(t,x) and cleanup(x, (W,t, committed)) interact with ops[x]. If cleanup emits $R_t[x_u]$, $W_u[x]$ must precede $R_t[x]$ in ops[x] and must therefore have preceded it in the input. Further any $W_v[x]$ which falls between $W_u[x]$ and $R_t[x]$ must have aborted before $R_t[x_u]$ was emitted.

[]

Corollary: $<_O$ is a partial order on C(O) and $< C(O), <_O >$ is a log.

Theorem 6.1: The Realisitic Scheduler is a recovery scheduler.

Proof: By reasoning similar to that of Lemma 4.1 shows that between any write operation in X and a subsequent read, there must be either a commit or an abort of the writing transaction. Therefore the log X is recoverable.

If the input log I could have been output from the Realisitic Scheduler, then !#@$%¢¢

□

Theorem 6.2: Realism is order-preserving.

Proof: This follows immediately from Lemma 6.1.

□

7. Paranoia

The Paranoid Scheduler uses abort as its only tool for maintaining recoverability. It operates much like Pessimism except instead of blocking it aborts the transaction. (A variant similar to Realism in which only read operations may cause aborts is possible.) Paranoia is the simplest of all the schedulers; to our surprise, ongoing simulations show it occasionally to have better throughput than Pessimism [GM84].

Paranoia -- Abort a transaction which attempts to read or overwrite uncommitted data.

Data Structures --

```
local lock data:
        for each $x \in D$
            committed [x]: Boolean, initialized to true

transaction data:
        for each active transaction
            t.locked: set of data items, initialized to empty
```

Procedures -- in response to database requests the Paranoid Scheduler takes the following actions:

```
read (t,x) =
        (enter $R_t[x]$ in X,O
        if not committed [x] and $x \notin$ t.locked
        then abort (t)) (* see below *)

write (t,x) =
        (enter $W_t[x]$ in X,O
        if committed [x] or $x \in$ t.locked
        then committed [x] $\leftarrow$ false
            t.locked $\leftarrow$ t.locked $\cup$ { x }
        else abort (t))

abort (t) =
        (for each $D_t \notin O$
            enter $D_t$ in X,O
        enter $A_t$ on O
        for each $x \in$ t.locked
            committed [x] $\leftarrow$ true)

commit (t) =
        (enter $C_t$ on O
        for each $x \in$ t.locked
            committed [x] $\leftarrow$ true)
```

Theorem 7.1: The Paranoid Scheduler is a recovery scheduler.

Proof: For all $R_t[x] \in OP(O)$ if $W_u[x] <_O R_t[x]$ then $C_u <_O R_t[x]$. Therefore O and X are recoverable. If the input log I could have been produced by the Paranoid Scheduler, then there is no attempt to read or write uncommitted data and therefore no extra aborts will be inserted. Since this is the orly change that the scheduler can make, there will be no changes.

[]

Theorem 7.2: The Paranoid Scheduler is meaning-preserving but not fully meaning-preserving.

Proof: Any read which is output read from data in I which was already committed in I and which was also committed in O. Therefore the meaning of reads is unchanged and the Paranoid Scheduler is meaning-preserving. However, the final transaction may be aborted, for example in the following input log:

$$I_j \quad = R_1[x]W_1[x]R_f[x]C_f$$

In fact, any time there are uncommitted transactions in the input log the final transaction is aborted.

[]

Since the Paranoid Scheduler is non-blocking and non-deferring $\Pi_{COM(O)}(O) = \Pi_{COM(O)}(I)$ and hence

Theorem 7.3: The paranoid scheduler is order-preserving and preserves 2PL.

REFERENCES

[BG81] Bernstein, P. A. and Goodman, N., "Concurrency control in distributed database systems," Computing Surveys, 13:2 (1981), pp.\% 185-221.

[E76] Eswaran, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. "The notions of consistency and predicate locks in a database system," \f2Commun. ACM\f1, Vol.\ 19, No.\ 11, November\ 1976, pp.\%624-633.

[G79] Gifford, D. K. "Weighted voting for replicated data," \f2Proc. 3rd Berkeley Workshop Distributed Databases and Computer Networks\f1, August 1978.

[GM84] Griffeth, N. and Miller, J., in preparation.

[H83] Hadzilacos, V. "An operational model for database system reliability," \f2Proc. of the 1983 Conference on Principles of Distributed Computing\f1, (1983), pp.\%244-257.

[KR81] Kung, H. T. and Robinson, J. T. "On optimistic methods for concurrency control," \f2ACM Trans. Database Syst.\f1, Vol.\ 6, No.\ 2, June 1981, pp.\%213-226.

[P79] Papadimitriou, C. H. "The serializability of concurrent database updates," \f2JACM\f1, Vol.\ 26, No.\ 4 (1979), pp.\%631-653.

[78b] Reed, D. P. "Naming and synchronization in a decentralized computer system," Ph.D. dissertation, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., September 1978.

[R78a] Rosen, E. C. "The updating protocol of the ARPANET's new routing algorithm: A case study in maintaining identical copies of a changing distributed database," \f2Proc. 4th Berkeley Workshop Distributed Data Management and Computer Networks\f1, August 1979. .LE

GIT-ICS-84/12

Committing and Aborting Subactions of Nested Transactions[†]

by

Marc H. Graham*

Nancy D. Griffeth*

Capt. J. Eliot B. Moss**

March 1984

*School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

**Box 483
U. S. Army War College
Carlisle Barracks, PA 17013

[†]This research was supported by the National Science Foundation under grant no. MCS-8200854, and by the Office of Naval Research under contract no. N00014-79-C-0873.

This document was prepared with a Xerox Star 8010 system.

Introduction

Nested transactions as described by Moss in [3] were motivated in part by the hope of improving system reliability by insulating an action from failure of a nested subaction. However, maintaining atomicity of nested transactions may require blocking access to some objects for long periods of time. In this paper, we consider the possibility of committing a nested subaction of a transaction immediately on its completion. It is a surprising and very useful result that this technique can be considered correct if there is an inverse action to the action being committed. This is a weak sense of correctness. That is, the state at the level of abstraction of the top-level actions can be guaranteed, under certain conditions, to be the same as would be reached by running only the top-level committed actions. However, the final state actually reached may differ, at the lowest level, from the final state that would result from running only the top-level committed actions. We also develop the conditions on the interleaving of actions and on the inverse actions which guarantee this type of correctness. This work extends the wok of Beeri et al. in [2] on serializing nested transactions in that the additional concurrency gained using their methods is preserved rather than reduced by the recovery technique.

Some frequently-cited nested actions which cause undesirable and apparently unnecessary blocking include actions which allocate pages and actions which manipulate indexes, such as B-trees. (Many other examples can be found in [1].) A pessimistic recovery scheme would require blocking allocation of new pages until a transaction which has been allocated a page has been committed. Of course, entirely different pages could be legitimately allocated, but because both transactions manipulate the same object--the page table--one of them is blocked. (We recognize that it is always possible to bypass a recovery scheme and use clever coding of the page allocation algorithms instead, but we are concerned here with automatic recovery schemes.)

A similar problem, which we will use an an example throughout the paper, arises when actions manipulate indexes. In Figure 1 transaction T_1 is adding keys X and Y to the index. Transaction T_2, meanwhile, is interleaved in time with T_1, and trying to add key Z to the same index. Suppose that the index is a B-tree and that adding key X has required that several nodes be split. Some time after key X has been added, let us suppose that T_1 will be aborted--due to software failure, concurrency control deadlock, or act of God. When T_1 aborts, we would invalidate T_2 if we restored before-images of pages. It appears that we must either block T_2 when it tries to Add(Z,B) or risk a cascaded abort. But instead, as Figure 1 shows, we can simply delete the keys added by T_1.

In this paper, we generalize this solution to arbitrary actions and address some of the issues raised, such as state-dependence of the above undo operations. (For example, if X was already in the database when add(X,B) was performed, then the undo is a no-op rather than a delete(X,B)). The major contributions of the work are the technique of committing individual actions immediately, instead of waiting for the top-level commit, the characterization of allowable interleavings of nested transactions to allow each incomplete action to be either committed or aborted, and characterization of conditions allowing the undos to be safely applied. For an action to be committed immediately, there must be an undo which will restore the prior state for any result state. (We assume that the "undos" are coded along with the "dos". Obviously, it may not be practical or even possible to supply such undos for all actions. The issue of how to determine which actions should be undo-able is left for future work.)

Basic Definitions

We assume a system partitioned into a finite number of levels of abstraction, 0, 1, 2, ... , n, where 0 is the lowest level, corresponding to the intuitive notion of primitive objects and actions. The approach to levels of abstraction is based loosely on the work of Schaffert [4]. Unlike Schaffert, we do not allow creation of new objects; instead we assume an infinite supply of objects from which we pick "new" ones as necessary.

At each level i there is a set of abstract states S_i, a collection A_i of actions and a collection O_i of objects. Each object has a state, given by the mapping

$$\omega_i\colon S_i \times O_i \rightarrow Obj(S_i)$$

where $Obj(S_i)$ is the union over all $o \in O_i$ of the possible states of o:

$$Obj(S_i) = \cup_{o \in O_i} States(o)$$

Conversely, we assume that the state at a level of abstraction is uniquely determined by the states of the objects at that level. System states are identical if they differ only in the identification of objects. For example, a consistent renumbering of disk blocks and references to disk blocks would not change the state.

An action $a \in A_i$ defines a partial function f_a from S_i to S_i. An action a touches an object o if it either changes it or looks at it, that is,either $o_i(s,o) \neq o_i(f_a(s),o)$ or for some s, t, if $p \neq o$ then $o_i(s,p) = o_i(t,p)$, but $f_a(s) \neq f_a(t)$.The value of the function f_a is determined by the states of the objects touched by a.

We may compose two actions a and b in A_i by applying first one and then the other. Thus $f_a \circ f_b (s) = f_b (f_a (s))$. Actions a and b conflict if there is some state s such that $f_a \circ f_b (s) \neq f_b \circ f_a (s)$. Otherwise they commute. If a and b conflict, they must both touch some object o. The converse does not necessarily hold, as illustrated by the operations of increment and decrement on the set of positive and negative integers.

Abstract states at adjacent levels (i–1) and i are related by a representation mapping $\rho_i\colon S_{i-1} \rightarrow S_i$ which is a partial function from S_{i-1} onto S_i. Thus some states in S_{i-1} may not correspond to valid abstract states, but every state in S_i is represented by some state of S_{i-1}. Also, several lower level states may correspond to a single higher level state. By composing representation mappings $\rho_{j+1} \circ \rho_{j+2} \circ \ldots \circ \rho_i$ we get a representation mapping from S_j to S_i.

An action trace is a set of actions together with a partial order which orders all pairs of conflicting actions. Each action at level $i > 0$ is implemented in terms of nested subactions at level i and below. For each action a at level i $(i \geq 1)$ we may imagine an associated program text, which determines the action trace to be used when the action is executed in state s. For a correct implementation, executing the actions in the action trace on any state $t \in S_0$ which represents state $s \in S_i$ should result in a state $u \in S_i$ which represents $f_a(s)$.

The program text also specifies terminating actions to be used when execution of the action is terminated, either because it is complete or because it cannot be completed. If it is complete, the action is called a commit action. If it cannot be completed, the action is called an abort action. Commit actions conflict with all other actions. Abort actions do not conflict with any other actions. The trace of an abort includes an "inverse" action, called an "undo", to reverse the effect of each committed nested subaction of the action being aborted. The partial order relating the undos will be the reverse of the order of the actions being undone. Thus undos a^{-1} and b^{-1} of conflicting actions a and b, with $a < b$, will be ordered $b^{-1} < a^{-1}$. The set of action traces implementing a given action will be called Imp(a,s). This set includes the complete trace terminated by a commit action and a set of aborted traces each of which is a prefix of the complete trace terminated by an abort action.

To describe the effect of running an action trace, we introduce a special kind of action trace, called an action tree, in which a set of actions is structured as a tree and conflicting actions are ordered by a partial order. The tree structure reflects the nesting of actions and the partial order reflects the order of execution. An action tree is thus an action trace with additional structure.

We say that an action in an action tree has been committed if it has a child which is a commit action. We say that an action in an action tree has been aborted if it has a committed child which is an abort action. Otherwise, the action is incomplete. Since we want actions to be insulated from failures of their children, an action may have aborted children without being aborted itself. An abort action may even be aborted. Similarly, aborted actions may have committed children (all of which must be undone by the abort action, which is the last child of the aborted action). In Figure 2, T_1 has nested subactions A and B. A is committed, B is aborted, and T_1 is incomplete. At this point, T_1 could be either committed by adding a commit subaction, or aborted by adding an abort subaction.

Given an action and an initial state, we can describe execution of the action using an action tree. The root of the tree is the original action. Its children are the actions belonging to some action trace in Imp(a,s), ordered as in the trace. This process continues recursively down to the primitive (level 0) actions to define the entire tree. The partial order in each action trace is retained in the tree and if actions are ordered then their descendants inherit the order, i.e., if parent(a) < parent (b) then a < b. There may be additional conflicting actions which are not ordered; the partial order is extended arbitrarily to include these. The set of action trees obtained in this manner is called Ex(a,s).

The effect of executing an action tree T is determined by the leaves. If the initial state is $s \in S_0$ then the result state is determined by applying the composition of the functions defined by the leaves in an order consistent with the partial order. Denote the function so defined by $f_T^{(0)}$. We say that actions trees T and U are <u>equivalent</u> if $f_T^{(0)}(s) = f_U^{(0)}(s)$ for all states s.

We now consider when an action tree defines a mapping at level i. Define an equivalence relation $\equiv_i$ on states of S_0 where $s \equiv_i t$ if $\rho_1 \circ ... \circ \rho_i(s) = \rho_1 \circ ... \circ \rho_i(t)$. We say that $f_T^{(0)}$ is a level i homomorphism if $f_T^{(0)}(s) \equiv_i f_T^{(0)}(t)$ for $s \equiv_i t$. In this case, there is a well-defined function $f_T^{(i)}: S_i \rightarrow S_i$ where $f_T^{(i)}(\rho_1 \circ ... \circ \rho_i(s)) = \rho_1 \circ ... \circ \rho_i(f_T^{(0)}(s))$. If a is an action at level i ($a \in A_i$) and T is an action tree which implements a in state s, then T is a correct implementation if and only if the function $f_T^{(0)}$ is a level i homohorphism and $f_T^{(i)}(s) = a(s)$.

<u>Inverse Actions</u>

Let $f: \Sigma \rightarrow \Sigma$ be a function mapping a set Σ onto itself. Define range(f) = $\{ f(x) \mid x \in \Sigma \}$. f is <u>surjective</u> if range(f) = Σ. f is <u>injective</u> if whenever $x, y \in \Sigma$ and $f(x) = f(y)$ then $x = y$. f is <u>bijective</u> if it is both surjective and injective.

<u>Lemma A</u>. If $f: \Sigma \rightarrow \Sigma$ and $g: \Sigma \rightarrow \Sigma$ are bijective then they have inverses (f^{-1} and g^{-1}) and if f and g commute with each other then f and g^{-1}, g and f^{-1}, and f^{-1} and g^{-1} also commute.

We will call actions which are one-to-one (injective) <u>invertible actions.</u> The inverse of action a is a^{-1}.

An action trace consisting entirely of invertible surjective actions can be run backwards almost as easily as forward. The following lemma characterizes the degree of freedom we are allowed in rolling back. First, we need a few definitions. A <u>prefix</u> P of an action trace T is an action trace containing some set of actions of T and having the property that if $b \in P$ and $a < b$ for some $a \in T$ which conflicts with b, then $a \in P$. The partial order on the prefix is the restriction of the partial order on the original trace to the actions in the prefix.

A set X of actions of a trace T is <u>final</u> in the partial order of T if T−X is a prefix of T. If X is ordered by the restriction to X of the partial order on T, then X is also an action trace. The <u>inverse trace</u> X^{-1} is the set $\{a^{-1} \mid a \in X\}$, together with the partial order $<_I$ in which $a^{-1} <_I b^{-1}$ if and only if $b < a$. The subtrace X of T is said to be <u>reversible</u> if the action trace $T \cup X^{-1}$ ordered by $< \cup <_I \cup \{(t, x^{-1}) \mid t \in T \text{ and } x \in X\}$ is equivalent to the action trace T−X.

Lemma B. If T is an action trace consisting entirely of invertible surjective actions then a subtrace X of T is reversible if and only if it is final in the partial order of T.

Thus to roll an action back we need only run its inverse and the inverses of all subsequent conflicting actions. Alternatively, we can avoid "cascading" rollback by blocking conflicting actions.

If actions do not define bijective functions, however, apparently non-conflicting actions on the same object may not be so easily reversible. Consider two successive adds of the same key to an index which does not contain the key initially. The first add action adds the key; its inverse deletes it. The second add action is identity on any state in which the tree already contains the key; its inverse is also the identity. The add actions commute, so the singleton set consisting of either add by itself is final in an action trace. However, the singleton set consisting of the first add, by itself, is not reversible, since running the inverse trace would remove the key.

In general, consider two non-conflicting invertible actions a and b. Take any state $s \notin range(a)$ such that $b(s) \in range(a)$. Then

$a^{-1} \circ b(s) = b(a^{-1}(s))$ is undefined

but

$b \circ a^{-1}(s) = a^{-1}(b(s))$ is defined.

Even if we extend a^{-1} to be the identity where it is undefined, it is still true that

$a^{-1}(b(s)) \neq b(a^{-1}(s)) = b(s)$,

So that a^{-1} and b still conflict.

The conditions we require to guarantee that the inverses commute with the actions and each other are: (1) $s \in range(a) \Leftrightarrow b(s) \in range(a)$ for all s, and (2) $s \in dom(a) \Leftrightarrow b(s) \in dom(a)$ for all s. Actually, we are only interested in these conditions when a and b commute, and in this case we already have half of them, because if $s \in range(b)$ then $a(s) = a(b(t)) = b(a(t)) \in range(b)$ and if $b(s) \in dom(a)$ then $a(b(s)) = b(a(s))$ is defined, so that $s \in dom(a)$. However, the converses do not necessarily hold. Consider $b(i) = i + 1$ for integers i in the set $\{0, \ldots, n\}$. $b(0) \in range(b)$ but $0 \notin range(b)$, so that b does not respect itself. In fact, no action respects itself if it is defined on a value outside its range (this applies to allocate, free, and adding a key to an index). We say that action a respects action b if $a(s) \in range(b) \Rightarrow s \in range(b)$ and $s \in dom(b) \Rightarrow a(s) \in dom(b)$. We say that actions a and b r-commute if a respects b, b respects a, and $a \circ b = b \circ a$. Otherwise, a and b r-conflict.

Lemma C. If actions a and b r-conflict, then they touch some common object.

This lemma follows from the observation that if they don't touch any common object, then clearly they commute, and whenever $a(s) = b(t)$, a state u can be constructed in which all objects touched by the action a have the same state as they do in s and objects touched by the action b have the same state as they do in t. But then s = b(u), so that a and b commute and respect each other, i.e., they r-commute.

We assume that actions which touch the same object will be ordered when an action tree is actually executed (by the order in which they touch the object), whether or not they are ordered in the partial order of the action tree. We call the execution order the r-order and say that a set X is r-final in an action trace T if T–X is a prefix of T in the r-order of T. Because abort actions are the last actions in the implementation of an action and do not conflict with any other action, a set consisting only of abort actions is always r-final.

Lemma D. If T is an action trace consisting entirely of invertible actions then a subtrace X of T is reversible if and only if it is r-final in the r-order of T.

We now consider non-invertible actions. The standard way of dealing with such actions in a DBMS is to log enough information (e.g., the prior state or some part of it) to make the action one-to-one. Thus the modified action on the augmented state is invertible. Suppose that we have non-invertible non-conflicting actions a and b. Augmenting the state by recording the prior state will not affect their mutual respect. We will henceforth assume that all actions are invertible.

An undo may be either state-based, such as restoring the before image of the pages involved, or general, such as a delete of a key which has been added to a B-tree. In the latter case, the undo applies to any state in the range of the add. Thus any action which r-commutes with the add may proceed without causing a dependency of its parent actior on the parent of the add.

Aborting Actions

A completed invertible action a can be undone by running the inverse action, as long as every action which has run subsequently r-commutes with a. If, however, an r-conflicting action has run, its effects must be reversed before the original action can be undone.

If an action has begun but not completed, then it is possible that some but not all of the nested subactions in the action trace have been run. In this case the nested "abort" action consists of an undo for each nested subaction in the trace of the original action, with the order the reverse of the forward actions. That is, if $a<b$ in the trace, then $b^{-1}<a^{-1}$ in the trace of the abort.

If no subsequent actions r-conflicted with the nested subactions in the trace of an action, then as a consequence of Lemma D the effect of the trace with the abort will be as if neither the action nor the abort had run. We now consider conditions on a trace that allow us to either commit or abort any incomplete action, knowing that the state achieved by the trace is the same as would have been achieved if only the committed actions had run. First, we define wo partial orders on actions, one reflecting the order in which they are executed and the other reflceting dependencies.

An action a immediately precedes an action b in a trace if a and b r-conflict; $a<b$ in the r-order of the trace; and there is no action c which r-conflicts with both a and b such that $a<c<b$ in the r-order. An action a precedes b if there is a sequence $a_1, \ldots a_n$ of actions such that $a=a_1$, $b=a_n$, and a_i immediately precedes a_{i+1}. The relation "precedes" is a partial order on actions.

An action b depends on an action a in an action tree if there is a child action c of a and another child action d of b such that c immediately precedes d. This definition generalizes the notion of a dependency between transactions T_1 and T_2, where T_2 depends on T_1 if T_2 reads a data-item written by T_1. We will say that an action tree is r-serializable if the relation which is the closure of "depends on" is a partial order. If this is the case, we then denote this partial order $<_D$. This a much weaker condition than serializability, saying only that $<_D$ separates (in the sense of [2]) any two-level forest consisting of a set of actions from the action tree and their children. If this is the case, we then denote this partial order $<_D$. Note that if a and b are aborted actions and $b<_D a$ then $<_D$ also orders abort(a) and abort(b). We assume henceforth that all action trees are r-serializable.

An action tree is revokable if every abort of an action follows (in $<_D$) the abort of any action which depends on the action. Suppose that a trace is revokable. Then the following proposition states that the set of aborted actions is r-final in the action trace, and therefore reversible. (That the undos have been performed in such a way as to reverse the actions will also have to be verified.)

Proposition 1. The set of aborted actions is r-final in a revokable action trace.

A trace is recoverable if every commit of an action precedes (in $<_D$) the commit of any action which depends on it. Suppose that a trace is recoverable. Then the following proposition states that any uncommitted actions can be aborted without requiring the abort of any committed actions.

Proposition 2. The set of uncommitted actions is r-final in a recoverable action trace.

In an action trace which is recoverable, uncommitted actions can be either committed or aborted. In an action trace which is revokable, the set of aborted actions is a set whose effects can be reversed.

Proposition 3. The action trace containing just the committed actions of an action tree T is an action forest F. Define an action a whose normal implementation is the set of roots of F. Then there is an action trace $U \in Ex(a,s)$ whose projection onto $U-\{a\}$ is equal to F.

Proof: This follows because all leaves of F are committed actions, so that all actions are initiated in the same state in F as in T and therefore the implementation is the same.□

There are a number of things that we would like to say but cannot say about the state after aborting all incomplete actions. For example, we would like to say that the result state is the same as it would be if we ran only the committed top-level actions. We would also like to say that the state at initiation of a committed action depends only on the previously committed actions. Neither of these is quite true.

Consider the example in Figure 1. The state in S_0 depends not only on the committed top-level action T_2 but also on the committed actions Add(X,B), Add(Y,B), Delete(Y,B), and Delete(X,B). The effect on the index and the actual pages allocated may be different for the sequence:

Add(W,B) Add(Z,B)

than for the sequence:

Add(W,B) Add(X,B) Add(Z,B) Add(Y,B) Delete(Y,B) Delete(X,B)

The following theorem says, however, that at the highest level of abstraction the state resulting is the same as the state resulting from running only the top-level actions. We interpret a in the theorem as an action which initiates the actual top-level actions. In fact, we can take any set of actions at a single level of abstraction in the tree, and considering the these as the top-level actions, we find that the theorem still applies.

Recovery Theorem. If an action tree T = Ex(a,s) for some action a is recoverable and revokable and if the children of a are all at a single level of abstraction i then the action trace S expanding all of the committed actions in T has $f_S^{(i)}(s) = f_U^{(i)}(s)$ for some action trace U which expands only the committed children of a.

Proof. Consider the action forest F which is the set of trees whose roots are the committed actions. This includes committed top-level actions as well as some actions which are committed children of aborted actions. We need to verify that these children of aborted actions have no effect on the final state.

By the definition of revokable, an aborted action is aborted only after all actions which depend on it.Consider some last aborted action a in $<_D$ ($<_D$ is acyclic). Any action b which occurs between a and abort(a) in the execution order will see the effect of a on the state and therefore its action trace may be different from the action trace generated if it had occurred before a or after abort(a). However, since all of its children r-commute with all children of a, we see that we can move the children either before a or after abort(a) and the result state should be the same, although the expansion of the children may be different.

Repeating this procedure recursively, we bring together a and abort(a) in the trace. The result is equivalent to a trace in which these have been deleted. After repeating this for all aborted actions, we have only the top-level actions left (possibly with entirely different traces).□

Conclusions

The standard criterion for correctness of transactions is atomicity, that is, only the effects of complete transactions are visible to other transactions. The recovery theorem states that transactions are atomic at the highest level of abstraction if the actions trees are r-serializable, recoverable, and revokable. However, this is not true at lower levels of abstraction. Lower-level actions may be run to completion and even committed even though the top-level action requesting them has been aborted. But the properties mentioned above do ensure that any effects which could be visible at the highest level are eventually reversed.

We argue that, from the application point of view, the only atomicity that is appropriate as a criterion of correctness is atomicity at the highest level of abstraction. The lower-level states are only used to implement the top-level state, thus we should be satisfied as long as the state reached by executing an action tree is equal to the state that would have been reached had only the top-level committed actions been executed. Furthermore, to require atomicity at lower levels of abstraction seems far too strong, since a nested transaction may be sufficiently time-consuming that its lack of atomicity may become apparent in any case.

Other issues to be investigated in future work include investigation of the impact of immediate commitment on the number of log operations and the size of the log required to recover from transaction aborts; techniques for recovering from system crashes as well as from transaction aborts; and special-case techniques for undoing actions which conflict with later actions, without having to undo the later actions.

Bibliography

[1] Allchin, J. E. "An Architecture for Reliable Distributed Ssytems," Ph.D. dissertation, Georgia Institute of Technology, GIT-ICS-82/23, 1983.

[2] Beeri, C., Bernstein, P. A., Goodman, N., Lai, M. Y., and Shasha, D. E. "A Concurrency Conrol Theory for Nested Transactions," *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, Montreal, Aug. 17-19, 1983, pp. 45-62.

[3] Moss, J. E. B. "Nested Transactions: An Approach to Reliable Distributed Computing," Ph.D. dissertation, MIT Laboratory for Computer Science, 1981.

[4] Schaffert, J. Craig. "Specification and Verification of Programs Using Data Abstraction and Sharing," Ph.D. dissertation, MIT Laboratory for Computer Science, 1981.

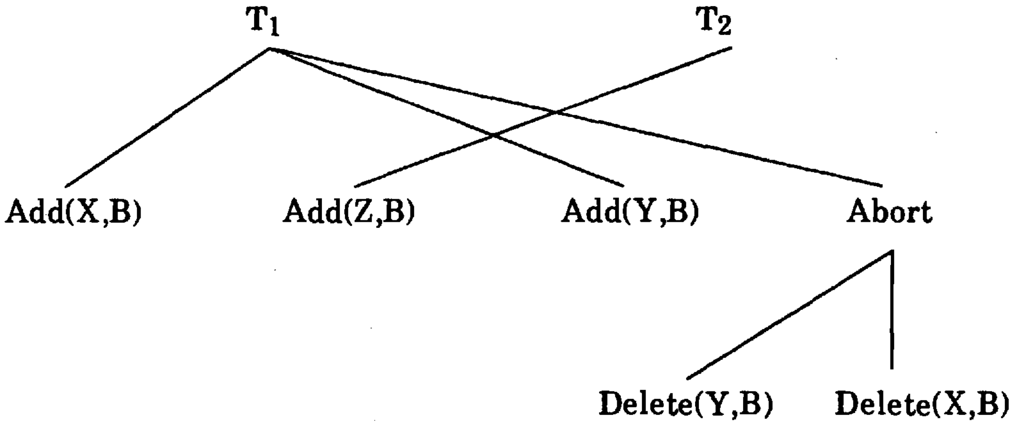


Figure 1. T_1 has been aborted without aborting T_2.

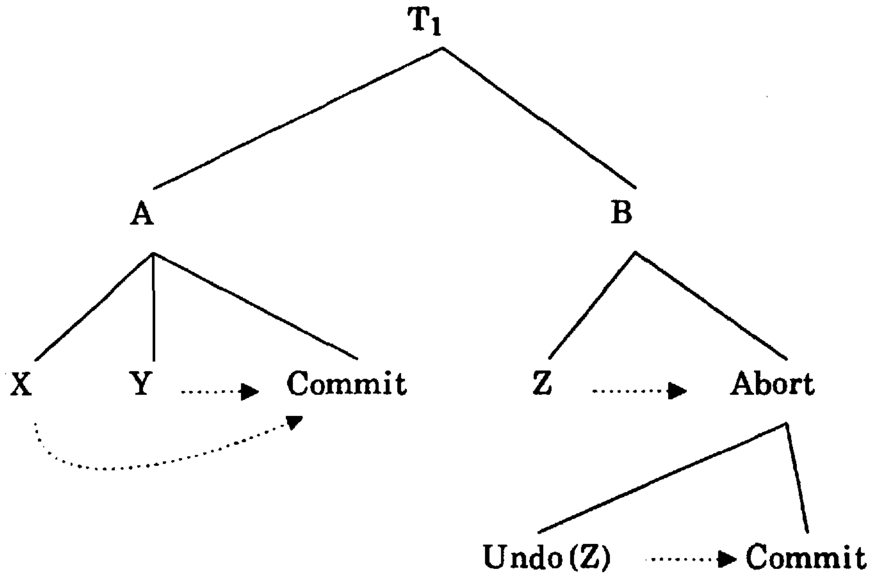


Figure 2. Dotted lines reflect the partial order on actions.

GIT-ICS-83/15

Reliable Scheduling of Database Transactions for Unreliable Systems

by

Marc H. Graham*

Nancy Griffeth*†

Barbara Smith-Thomas**

EXTENDED ABSTRACT

October 1983

*School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

**Department of Mathematics
University of North Carolina at Greensboro
Greensboro, North Carolina 27412

†Research supported by the National Science Foundation under Grant No. MCS-8200854, and by the Office of Naval Research under contract no. N00014-79-C-0873.

This document was prepared with a Xerox Star 8010 system.

Introduction.

A database transaction is usually required to have an "all-or-none" property, that is, either all of its changes to the database are installed in the database or none of them are. This property can be violated by a hardware or software failure occurring in the middle of the execution of a transaction or even by a decision to abort a transaction to avoid an inconsistent (non-serializable) schedule. In such a situation, we must recover from the failure by restoring the database to a state in which the failed transaction never executed. The idea of recoverable schedules was defined in [H83]. In a recoverable schedule, a transaction does not commit until there is no possibility that it will be rolled back. If the underlying hardware or software is unreliable, this must not occur until all transactions which have written values read by the transaction have themselves committed .

The standard solution for guaranteeing that a schedule is recoverable is to "hold write-locks to commit point," or more generally, to prohibit any access to a data-item which has been changed by a transaction until the transaction commits. Such a policy guarantees that no other transaction can ever read or write a data-item value until the last transaction writing it has committed. This prevents any transaction from committing before values it has read have been committed by the writing transaction(s).

This policy fits well with two-phase locking, since write-locks are held to "lock point" anyway, but does not fit so well with timestamping protocols or even with locking protocols which allow earlier release of locks. Also, this policy may not fit at all if serialization is not required. We call this policy the "pessimistic" policy and describe two other policies--the "optimistic" policy, which does not block but will abort a transaction trying to commit if it has read data written by an aborted transaction--and the "realistic policy," which blocks reads but not writes of uncommitted data. These three policies can be compared, using the results of this paper, with respect to their effect on concurrency, on the number of aborts, and on preservation of membership in serializability classes.

The pessimistic policy allows the least concurrency of the three. In contrast, the optimistic policy does not reduce concurrency at all and the realistic policy reduces concurrency somewhat. The optimistic policy will, however, cascade aborts. The pessimistic policy will introduce aborts due to deadlocks by it but it will not cascade aborts. The realistic policy will not even introduce aborts if we can assume that all reads of a transaction precede all writes and that the schedules are serializable.

Finally, the pessimistic policy preserves membership in the class DSR of schedules (schedules that can be serialized by swapping adjacent operations). The optimistic policy preserves membership in a somewhat larger class of schedules, PSR (schedules all of whose prefixes are also serializable). The realistic policy preserves membeship in SR, the class of all serializable schedules, assuming only that all reads of a transaction precede all writes. The primary practical implication of these facts is that the realistic scheduler appears better in all cases than the pessimistic scheduler. Simulation and analysis comparing the various policies are ongoing.

There are several contributions in this abstract in addition to the new policies for producing recoverable schedules. In order to discuss schedulers for recoverable sequences of operations as well as the sequences themselves, we have had to distinguish between the "temporal order," in which the scheduler sees the operations, and the "log order," which determines the interpretation of the operations. This dis'inction is of interest independently of the results; for example, it applies nicely to describing the behavior of multi-version timestamping described in [R78]. The properties of recoverability and serializability have been considered separately, so that the recovery policies can be used even for transaction systems which do not require serializability, such as described by several authors [L82, G81, A83].

Definitions.

We mostly follow Hadzilacos [H83]; however, we alter the notion of the meaning of database transaction to reflect the intention that aborted transactions should not affect subsequent database transactions. We use the term log as in [H83] to mean schedule, or history of transaction operations.

Let $D = \{x, y, z, ...\}$ be a set of data items. Transaction operations are $R_t[x]$, $W_t[x]$, B_t, C_t, A_t, $t \in N$, $x \in D$. These symbols are intended to represent, respectively, a read of x by transaction t, a write of x by transaction t, the 'begin' of transaction t, commit of transaction t, and an abort of transaction t. We also use $D_t[x]$ (access to data-item x by transaction t) to mean an operation which may be either a read or a write. Two operations conflict if they are read or write operations accessing the same data item and at least one of them is a write, or if one of them is a begin, commit, or abort.

A transaction, t, is a partially ordered set $(op_t, <_t)$ where

$$op_t \subseteq \{R_t[x], W_t[x] : x \in D\} \cup \{B_t, C_t, A_t\}$$

satisfying the following:

i) $B_t \in op_t$;

ii) $A_t \in op_t$ iff $C_t \notin op_t$;

iii) $\forall o \in op_t - \{B_t\}$, $B_t <_t o$;

iv) if $A_t \in op_t$, then $\forall o \in op_t - \{A_t\}$, $o <_t A_t$;

v) if $C_t \in op_t$, then $\forall o \in op_t - \{C_t\}$, $o <_t C_t$;

vi) any two conflicting operations are ordered by $<_t$.

Our definition is more general than some, since transactions may contain concurrent subtransactions and we do not require all reads to precede all writes.

A complete log, L, is a triple $(OP, <_L, <_T)$ where

1) $OP = \cup_t op_t$;

2) for each t, $<_t \subseteq <_L$, any two conflicting data accesses are ordered by $<_L$, and $<_L$ is the smallest partial order satisfying these two conditions;

3) $<_T$ is a total order on OP such that for all transactions t, u and data items x,

   a) $<_t \subseteq <_T$ and

   b) if $W_t[x] <_L R_u[x]$ then $W_t[x] <_T R_u[x]$

$<_L$ is the log order of L and reflects the intended meaning; $<_T$ is the temporal order of L and reflects the order in which operations are submitted to a scheduler. We say that any order on OP which satisfies 2) is a log order for OP and any order on OP which satisfies 3) is a temporal order on OP. A log is a temporal order prefix of a complete log. A transaction, t, is committed in a log L if $C_t \in OP$; t is aborted in L if $A_t \in OP$; t is active if it is neither committed nor aborted. We let $COM(L)$ denote the set of committed transactions in L.

The temporal order of a log reflects the order in which it is submitted to the scheduler, i.e., the actual interleaving of transaction requests on the system. The log order reflects the order in which it is interpreted (defined formally below). Ordinarily, the temporal order is a topological sort of the log order, but in concurrency control methods such as multiversion timestamping we may wish to allow some independence between the two orders. We need to make this distinction between log order and temporal order not only to allow input to the recovery scheduler from a multiversion timestamping method, but also to describe the output in the case of the realistic policy. It is not necessary for the other policies.

The projection of a log, L, onto a subset τ of its transactions, denoted by $\Pi_\tau(L)$, is the restriction of L to $\cup_{t\in\tau} op_t$. We will be most interested in the projection of a log onto its committed transactions: $\Pi_{COM(L)}(L)$. For mathematical simplicity we extend all logs with a fictitious initializing transaction t_0, which begins, writes all database items, and then commits. Any access to a data item is preceded by a write to that data item by t_0, where "preceded" refers to both log order and temporal order.

The semantics of read and write operations in a log L is defined by a function M_L (the meaning in L), satisfying

1) $M_L(R_t[x]) = M_L(W_u[x])$ where $W_u[x] <_L R_t[x]$ and for any v such that $W_u[x] <_L W_v[x] <_L R_t[x]$ we have $A_v <_T R_t[x]$. That is, the meaning of a read is the meaning of the most recent unaborted write to the same data item. This is different from Hadzilacos' definition, where a read operation may use the value from an aborted write.

2) $M_L(W_t[x]) = g_{tx}(M_L(R_t[y_1]), \ldots, M_L(R_t[y_n]))$ where $R_t[y_1], \ldots, R_t[y_n]$ are all the reads by t which $<_t$ precede $W_t[x]$ and g_{tx} is a function which computes the new value of x to be wrtten by transaction t from the values previously read by transaction t.

A log L is recoverable if for every prefix L' of L the meaning of each committed read and write in L' is the same as its meaning in $\Pi_{COM(L')}(L')$. The major implication of this definition is that if we want recoverability we must not allow a transaction to commit unless all transactions whose values it has read have previously committed.

A scheduler takes a log as input, one operation at a time in temporal order, and outputs a new log. It may do one of three things with the operation:

1) output it;

2) abort the requesting transaction;

3) take either action after reading more operations from the input and choosing one of these three actions for each.

An operation which has appeared in the scheduler's input log but not in its output log is said to be blocked. If an operation o_1 of a transaction t is blocked then t may not request any operation o_2 such that $o_1 <_L o_2$.

Since the operations are input to a scheduler and output from a scheduler in temporal rather than log order, the "position" of each operation in the log order must also be part of the input and output. To do this, we define a function POS_L mapping reads and writes of L in $(OP - OP_{t_0})$ to OP as follows:

$$POS_L(D_u[x]) = D_t[x]$$

where

1) $D_u[x]$ and $D_t[x]$ are conflicting operations;

2) $D_t[x] <_L D_u[x]$;

3) $D_t[x] <_T D_u[x]$; and

4) if $D_v[x] <_L D_u[x]$ for $v \neq t$ then either $D_v[x] <_L D_t[x]$ or $D_u[x] <_{T(L)} D_v[x]$

We can reconstruct a log order containing $<_L$ fom OP_L, $<_T$, and POS_L by taking the transitive closure $<_*$ of $<$, defined as follows:

1) $D_t[x] < D_t[y]$ if $D_t[x] <_T D_t[y]$; and

2) $D_t[x] < D_u[x]$ if $D_t[x] = POS_L(D_u[x])$

or recursively if $POS_L(D_t[x]) < D_u[x]$ and $D_u[x] <_T D_t[x]$

Each read or write will be preceded, in the reconstructed log order, by any operation which preceded it in the original log order. In fact, the immediately preceding conflicting operation must be the same, so that the interpretation of each read will depend on the same write. Writes, however, may be preceded by additional reads of the same transaction. If we make the reasonable assumption that these additional reads have no effect on the outcome of the writes, then we may assume that the interpretations are equivalent, in the sense of returning the same values to a terminal transaction of all reads, under the two log orders.

Formally, a scheduler is a 4-tuple $<\mathbf{A}, O, \sigma, \omega>$ where $\mathbf{A}$ is a set of logs without commits or aborts; O is a set of pairs $<o, POS(o)>$, where o is an operation; σ is a state transition mapping

$$\sigma: \mathbf{A} \times O \rightarrow \mathbf{A}$$

and ω is an output function

$$\omega: \mathbf{A} \times O \rightarrow O^*$$

Extend σ to σ' on $\mathbf{A} \times O^+$ by defining

$$\sigma'(A, o_1) = \sigma(A, o_1)$$

$$\sigma'(A, o_1 \ldots o_n) = \sigma(\sigma'(A, o_1 \ldots o_{n-1}), o_n)$$

and extend ω to ω' on $\mathbf{A} \times O^+$ by defining

$$\omega'(A, o_1) = \omega(A, o_1)$$

$$\omega'(A, o_1 \ldots o_n) = \omega(\sigma'(A, o_1 \ldots o_{n-1}), o_n)$$

We will say that $S = (OP_S, <_S, <_{T(S)})$ is the output of a scheduler if OP_S is the set of operations output, $o_t <_{T(S)} o_u$ if o_t is output before o_u, where o_t and o_u are arbitrary operations of transactions t and u, and $<_S$ is constructed from POS_S as described above.

A scheduling policy restricts the choices a scheduler can make. We will describe policies somewhat informally by stating (1) the types of operations which may be blocked while other operations are processed, under what conditions they will be blocked, and under what conditions they will be output; and (2) the conditions which require a transaction to be aborted. For a given scheduling policy, we can define a mapping P from input logs to sets of output logs, such that each output log could have been produced from the corresponding input log. A scheduler implements a scheduling policy if for each input log $L=(OP, <_L, <_T)$, if $o_1 \ldots o_n$ is the set of operations in OP ordered by $<_T$, then $\omega'(\varnothing, o_1, \ldots, o_n)$ is in $P(\{o_1, \ldots o_n\} <_{POS}, <_T)$.

Scheduling Policies and Schedulers.

If $S=(OP_S, <_S, <_{T(S)})$ is the output produced from an input log $L=(OP_L, <_L, <_{T(L)})$, then for all schedulers we will require the following:

(L1) for each $o_t \in OP_S$ either $o_t = A_t$ or $o_t \in OP_L$;

(L2) if $A_t \in OP_S$ then $C_t \notin OP_S$ and for all $o_t \in OP_S$, $o_t <_{T(S)} A_t$;

(L3) if $o_t <_L o'_t$ and $o'_t \in OP_S$ then $o_t \in OP_S$ and $o_t <_{T(S)} o'_t$;

(L4) $POS_S(D_t[x]) <_{T(S)} D_t[x]$ and there is no conflicting read $R_u[x]$ with $POS_S(D_t[x]) <_{T(S)} R_u[x] <_{T(S)} D_t[x]$.

Condition L1 states that each operation output by the scheduler was either an operation input by the scheduler or an abort. Condition L2 states that no operations of a transaction are output after an abort of the transaction has been output. Condition L3 states that the order of operations in a transaction is preserved and that no operation is output until and unless all previous operations of the transaction have been output. Condition L4 states that knowledge of the future is not required. It follows from these conditions that S is a log.

We also require that if the input log is a complete log, (that is, all transactions have either committed or aborted), then

(C1) if $C_t \in OP_L$ then either $C_t \in OP_S$ or $A_t \in OP_S$; and

(C2) if $A_t \in OP_L$ then $A_t \in OP_S$.

(C1) and (C2) clearly imply that if the input log is complete then so is the output log.

We first describe optimistic policies. A scheduler implementing an optimistic policy blocks commits of transactions which have read uncommitted data. All other operations are output immediately. If a transaction abort A_t is output, then so must be aborts A_u of all transactions which have read data ($R_u[x]$) last written by the aborting transaction ($W_t[x] = POS_L(R_u[x])$). POS_S is equal to POS_L.

Proposition $\varnothing$. If S is the output log from an optimistic scheduler and if $W_t[x] <_S R_u[x]$ and if for each $W_v[x]$ such that $W_t[x] <_S W_v[x] <_S R_u[x]$ there is an operation A_v such that $A_v <_{T(S)} R_u[x]$ then $C_u \in OP_S \Rightarrow C_t \in OP_S$ and $C_t <_{T(S)} C_u$.

Proposition Ø states that commits are blocked until all transactions which have written data read by the committing transaction have committed.

Theorem 1. Any output of an optimistic scheduler is a recoverable log if the input is a log. If the input log is complete then so is the output log. If S is the output log produced from input log L by a scheduler which implements an optimistic scheduling policy then

$$\Pi_{com(S)}(S) = \Pi_{com(S)}(L)$$

One scheduler which implements an optimistic policy behaves as follows: Every operation which is input is output immediately unless it is a commit. If it is a commit, then the commit can also be output immediately unless the transaction being committed has read data written by an uncommitted transaction. In that case, the commit must be blocked until all transactions from which it has read data have committed. If a transaction aborts, then all transactions which have read data from it must also abort. It is possible that a cycle of transaction dependencies may form. In this case, the scheduler is free to either abort all involved transactions or to commit all involved trasnactions, in a single atomic operation. The latter course can be chosen only if provision has been made.

Next, we describe pessimistic policies. A scheduler implementing a pessimistic policy must block all data accesses to uncommitted data. Otherwise, operations are output immediately. $POS_S(D_t[x])$ is equal to the most recent non-aborted write, as ordered by $<_{T(S)}$.

Proposition P1. If L is an input log to a pessimistic scheduler and if $W_u[x] <_L D_t[x]$ then for every operation o_t such that $D_t[x] <_L o_t$, either $C_u <_{T(L)} o_t$ or $A_u <_{T(L)} o_t$.

Proposition P1 states that operation $D_t[x]$ is blocked until transaction u has committed.

Proposition P2. If S is the output log from a pessimistic scheduler and if $W_t[x] \in OP_S$ and $D_u[x] \in OP_S$ and $W_t[x] <_S D_u[x]$ then either $C_t <_{T(S)} D_u[x]$ or $A_t <_{T(S)} D_u[x]$.

Proposition P2 states that no operations are output until any preceding conflicting operations are either committed or aborted.

Theorem 2. Any output of a pessimistic policy is a recoverable log if the input is a log. If the input log is a complete log then so is the output log. If for every $A_t \in OP_S$ we also have $A_t \in OP_L$ and $<_S \subseteq <_L$ then

$$\Pi_{COM(S)}(S) = \Pi_{COM(S)}(L).$$

One way to implement a pessimistic scheduler would be to set a lock when a data-item is written. The lock is not released until the writing transaction is either committed or aborted. If cycles develop among transactions awaiting completion (by commit or abort) of other transactions, then a victim should be selected to be aborted. Waiting operations may be dispatched in any order, but to guarantee that $\Pi_{COM(S)}(S) = \Pi_{COM(S)}(L)$ it is necessary either to dispatch them in temporal order consistent with log order or to specify the immediately preceding write operation with POS_S.

Finally, we describe what we call a "realistic" scheduling policy. A scheduler implementing a realistic policy must block all reads of uncommitted data. A read $R_u[x]$ may be output as soon as some write $W_t[x]$ which $<_L$ preceded it in the input log has committed and all other writes $W_v[x]$ with $W_t[x] <_L W_v[x] <_L R_u[x]$ have been aborted. $POS_S(R_u[x])$ is defined as $W_t[x]$. If a data access $D_t[x]$ was never blocked, then $POS_S(D_t[x]) = POS_L(D_t[x])$.

<u>Proposition R1.</u> If L is an input log to a realistic scheduler then for every $R_t[x]$, if $W_{u_1}[x] <_L W_{u_2}[x] <_L \ldots <_L W_{u_k}[x]$ is the sequence of all writes preceding $R_t[x]$ then there is some $i \leq k$ such that for all o_t with $R_t[x] < o_t$, $C_{u_i} <_{T(L)} o_t$, and for all $j > i$, $A_{u_j} <_{T(L)} o_t$.

Proposition R1 states that a transaction issuing a read operation is blocked until some preceding write has been committed and all subsequent writes preceding $R_t[x]$ have been aborted.

<u>Proposition R2.</u> If S is an output log from a realistic scheduler and if $W_t[x] \in OP_S$ and $R_u[x] \in OP_S$ and $W_t[x] <_S R_u[x]$ then either $C_t <_{T(S)} R_u[x]$ or $A_t <_{T(S)} R_u[x]$ or there is a $W_v[x]$ with $W_t[x] <_S W_v[x] <_S R_u[x]$ and $C_v <_{T(S)} R_u[x]$.

Proposition R2 states that a read may not be output until the most recent non-aborted write has been committed.

<u>Theorem 3</u>. Any output of a realistic scheduler is a recoverable log if the input is a log. If the input log is complete then so is the output log. If $A_t \in OP_S$ implies $A_t \in OP_L$ and if $<_S \subseteq <_L$ then

$$\Pi_{COM(S)}(S) = \Pi_{COM(S)}(L).$$

To implement a realistic scheduling policy, the scheduler could behave similarly to the pessimistic scheduler, except that writes are output immediately in all cases. A read is output as soon as the most recent non-aborted write is committed. As with the pessimistic scheduler, if a cycle of waits develops, then a victim must be selected to be aborted. In a realistic scheduler, $<_S$ may be distinct from $<_{T(S)}$. Suppose, for example, that we have a sequence of operations

$$W_t[s] \cdots R_u[s] \cdots W_v[s] \cdots C_t \cdots$$

Then $W_v[s]$ is output without blocking whereas $R_u[x]$ is not output until after $W_v[s]$. Thus $R_u[s] <_S W_v[x]$ but $W_v[x] <_{T(S)} R_u[x]$.

Serializability and scheduling policies.

We state three general results about scheduling policies and classes of logs. A class Λ of logs has the prefix property if every log-order prefix of a log in Λ also belongs to Λ . The class DSR of logs which can be serialized by swapping non-conflicting operations has the prefix property; the class SR does not. The largest class of serializable logs having the prefix property is strictly larger than DSR. We will call this class PSR.

Theorem 4. If an input log L belongs to a class Λ of logs having the prefix property, then an output log S produced by a scheduler which implements the optimistic scheduling policy also belongs to Λ.

A class Λ of logs has the projection property if every projection of a log onto a subset of the transactions in the log also belongs to the class Λ. The class DSR of serializable logs has the projection property. The classes PSR and SR do not.

Theorem 5. If an input log L belongs to a class Λ of logs having the projection property, then an output log S produced by any scheduler satisfying L1-L4 and C1-C2 belongs to Λ.

It follows immediately from theorem 5 that the class DSR of logs is preserved by any scheduler satisfying L1-L4 and C1-C2. We get a stronger result for the realistic scheduling policy if we require all reads of a transaction to precede all writes of the transaction. In this case, it is not possible for cycles of waiting transactions to form (since only reads are blocked) and therefore it is possible to avoid having any aborts in the output log that do not also appear in the input log.

Theorem 6. There is a realistic scheduler which preserves membership of the class SR, if all reads of each transaction precede all writes.

REFERENCES

[A83] Allchin, J.E. "An architecture for reliable decentralized systems," Ph.D. dissertation, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, September 1983.

[G81] Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. Dept. EE & CS, Princeton University, April, 1981. Technical Report 285.

[H83] Hadzilacos, V. "An operational model for database system reliability", *Proc. of the 1983 Conference on Principles of Distributed Computing*, (1983), pp. 244-257.

[L82] Lynch, Nancy. Multilevel atomicity. Proc. 1982 ACM SIGACT-SIGMOD Symp. on Principles of Database Systems., pp. 63-69.

[P79] Papadimitriou, C.H. "The serializability of concurrent database updates," *JACM*, vol. 26, no. 4 (1979), pp. 631-653.

[R78] Reed, D. P. "Naming and synchronization in a decentralized computer system," Ph.D. dissertation, Dept. of Electrical Engineering, M.I.T., Cambridge, Mass., September 1978.

GIT-ICS-82/04

FUNCTIONS IN DATABASES*

MARC H. GRAHAM

MAY 1982

* To appear in ACM Transactions on Database Systems

Functions in Databases

Marc H. Graham

School of Information and
Computer Science

Atlanta, Georgia 30332

ABSTRACT

We discuss the objectives of including functional dependencies in the definition of a relational database. We find two distinct objectives. The appearance of a dependency in the definition of a database indicates that the states of the database are to encode a function. A method based on the chase of calculating the function encoded by a particular state is given and compared to methods utilizing derivations of the dependency. A test for deciding whether the states of a schema may encode a non-empty function is presented as is a characterization of the class of schemas which are capable of encoding non-empty functions for all the dependencies in the definition. This class is the class of dependency preserving schemas as defined by Beeri et al. and is strictly larger than the class presented by Bernstein.

The second objective of including a functional dependency in the definition of a database is that the dependency be capable of constraining the states of the database; that is, capable of uncovering input errors made by the users. We show that this capability is weaker than the first objective; thus, even dependencies whose functions are everywhere empty may still act as constraints. Bounds on the requirements for a dependency to act as a constraint are derived.

These results are founded on the notion of a weak instance for a database state which replaces the Universal Relation Instance Assumption and is both intuitively and computationally more nearly acceptable.

April 12, 1982

Functions in Databases

Marc H. Graham

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

1. Introduction

It is common when designing a relational database schema to include in its description elements of a class of statements called functional dependencies. These statements serve two purposes. They act as constraints on the database states for the schema, declaring some of these states to be illegal. They also indicate that the database states for the schema represent, *inter alia*, functions whose descriptions, i.e., domain and codomain sets, are given by the dependencies. These two purposes are not identical, although they cannot be divorced. We will be taking the convenient view that the illegal states do not represent any functions.

The subject of representation of functions in database states of a schema has been addressed before. In the 1978 Very Large Database Conference, Bernstein et. al. [BBG] presented an excellent summation of the state of dependency theory as it existed at that time. They declared that a schema represents a set, F, of functional dependencies exactly when it embeds a cover of F. (The condition is called Rep2 in [BBG].) They were lead to this idea as a consequence of the Universal Relation Instance Assumption (URIA). The URIA states that for any database state there exists an instance of the "universal" relation, the relation on all the attributes of the schema, such that the relation instances of the state are its projections. There are many objections to this assumption which combine to make it untenable. Indeed the authors of [BBG] objected to the assumption themselves. The first of these objections, and by no means the least, was established by Honeyman et. al. in [HLY]. That paper showed that the decision problem: Does an arbitrarily chosen database state satisfy the URIA?; is NP-complete. Thus maintaining the URIA as a database constraint is an impossibly difficult task in general. A second objection was put forward by Bernstein and Goodman [BG] to the effect that maintaining the URIA reintroduces globally the "update anomalies" first mentioned by Codd [C] for single relations. A third objection, perhaps the most crucial even though unprovable, is the reasonable belief that almost no database state arising in practice will satisfy the URIA.

We will present an alternative to the URIA, called the weak instance, originally due to Honeyman [H] and independently to Vassiliou [V]. We will present new definitions of functional representation by states and by schemas which rest on it. We will show that a dependency may be represented even when it is not embedded and may act as a constraint even when it is not represented.

A secondary result of this research touches on the derivation of a dependency from a set of dependencies. In [B], Bernstein describes an application of the inference rule 'pseudo-transitivity' (defined in section 2, below) as functional composition. He also gives a graphical formulation of a derivation, called a derivation tree. A functional dependency can be interpreted as a table-lookup, i.e., as an expression in the relational algebra. We will give a method which uses a derivation tree as a guide to the composition of such expressions. We point out that two distinct derivation trees may result in inequivalent expressions, even when applied to a single satisfying instance. We also show that the function represented by a multi-relation database state is not necessarily calculated by any of its derivations.

In the next subsection, we present the basic definitions of relational theory which we will need. Definitions of particular concern to our investigation are given in the body of the paper when needed. In section 2 we describe the functions corresponding to functional dependencies in the context of single relations. In section 3 we do the same for multi-relation databases. In section 4 we present a new definition of representation by a schema of the function associated with a functional dependency and give a characterization of those schemas which represent all given dependencies. In section 5 we return to dependencies as database constraints and give bounds (i.e., sufficient and necessary conditions) on the requirements for a dependency to constrain a database. Section 6 reviews related work of other authors and makes some comments on the practical significance of this work. Section 7 recapitulates the results established.

1.1. Basic Defintions

The *scheme* of a relation is a set R of elements called ***attributes***. Associated with each attribute A there is a set of values called the domain of A and denoted $dom(A)$. Following standard notational convention, we use an attribute name A to represent either itself or the set $\{A\}$. Also, we elide the operator when taking unions of sets of attributes; i.e., $XY \equiv X \cup Y$.

An *instance* of the relation R is a set of tuples. A ***tuple*** is a function t from R to $\bigcup_{A \in R} dom(A)$ such that $t(A) \in dom(A)$. It is customary to use square brackets instead of parentheses when evaluating a tuple for an attribute; thus $t[A] \in dom(A)$. By extension, for all $X \subseteq R$, $t[X] \equiv t \mid X$; that is, the restriction of t to the attributes in X. $t[X]$ is also called an X-value. We will often ignore, both in English and in symbols, the distinctions among the constant value $a \in dom(A)$, the A-value which is the function taking attribute A to value a, and the relation instance on tne scheme A whose sole element is the A-value a. We may do the same for sets X of attributes, X-values and singleton X relation instances.

We will need four of the operators of relational algebra: projection, selection, natural join and union.

If I is an instance of the relation R and $X\subseteq R$, the projection of I onto X is given by

$$\pi_X(I)\equiv\{t[X] \mid t\in I\}$$

Let C be a conjunction of atomic formulae of the form $X=x$ where $X\subseteq R$ and x is an X-value. Then $\sigma_C(I)$ is the subset of I each of whose elements makes C true. For example,

$$\sigma_{X=x\wedge Y=y}(I)\equiv\{t \mid t\in I\wedge t[X]=x\wedge t[Y]=y\}$$

If r is an instance of R and s is an instance of S, then the natural join of r and s is a relation on scheme RS given by

$$r^*s\equiv\{t \mid t[R]\in r\wedge t[S]\in s\}$$

If $r_1, r_2, \ldots, r_n$ are all instances of the scheme R, then we may take their union as

$$\bigcup_{i=1}^{n} r_i=\{t \mid (\exists\, 1\leq j\leq n) t\in r_j\}$$

For more details on these operations, see the text by Ullman [U].

2. Functional Dependencies Within One Relation

We begin our investigation by considering the behaviour of functional dependencies within a relation. We first recall the definition of functional dependencies in their role as constraints.

(Satisfaction-1) Let R be a relation scheme and F a set of functional dependencies. If I is an instance for R, then we say I satisfies F (or I is a legal instance for R) if the following holds: For each dependency $X\rightarrow Y$ in F, for

every pair of tuples t, u in I, $t[X]=u[X]$ implies $t[Y]=u[Y]$. ▪

Now let us consider the functions described by the dependencies in F. The primary means whereby a relation instance relates attribute values is by including a tuple in which the values appear. With this in mind, assign each dependency in F a unique label. Let f be the label of $X \rightarrow Y$. Now define[†]

$$\varphi_f \equiv \lambda I.\, \lambda x.\, \pi_Y(\sigma_{X=x}(I))$$

where I is an instance and x is an X-value. Thus φ_f, a *table lookup* as decribed by Arora and Carlson [AC], is a function from instances for R to mappings from X-values to Y-values. Clearly if I is a satisfying instance then $\varphi_f(I)$ is a function. A key observation of this work is that the instance I may have more information about the function f than is given by $\varphi_f(I)$. In particular, X-values not in $\pi_X(I)$ may nonetheless have Y-values assigned to them by I. We demonstrate this by way of an example.

Example 1. Let $F=\{g:X \rightarrow Y,\ h:YW \rightarrow Z,\ f:XW \rightarrow Z\}$. Consider the instance

I=

X	Y	W	Z
x_1	y_1	w_1	z_1
x_2	y_1	w_2	z_2

We claim that I assigns the Z-value z_2 to the XW-value x_1w_2. To justify the claim, note that any tuple added to I containing XW-value x_1w_2 must have Y-value y_1 (by $(\varphi_g(I))(x_1)=y_1$) and Z-value z_2 (by $(\varphi_h(I))(y_1w_2)=z_2$) if it is to be satisfying. Of course, $(\varphi_f(I))(x_1w_2)$ is undefined. ▪

We propose a definition of the function f_I represented in a satisfying instance, I, for the dependency $f:X \rightarrow Y$.

[†] The λ-operator is the abstraction operator of the lambda calculus. Say we have some expression x^2. Now if it is defined at all, x^2 is some number which we know exactly when we know x. The expression $\lambda x.x^2$ is the squaring function, which may be thought of as a set of ordered pairs.

The completion, or chase, of I_x, denoted I_x^*, is the result, $\chi(I_x)$, of the application of any sequence of transformations $\chi=\tau_1, \tau_2, \ldots, \tau_p$ such that

- τ_1 is enabled in I_x
- τ_{i+1} is enabled in $\tau_i(\tau_{i-1}(\cdots(\tau_1(I_x))\cdots))$
- no transformation applied to $\chi(I_x)$ results in a change to it.

That I_x^* is well-defined for I_x is proven in [G].

Example 2. Reconsider example 1. The augmented instance, $I_{x_1w_2}$ is

X	Y	W	Z	
x_1	y_1	w_1	z_1	1
x_2	y_1	w_2	z_2	2
x_1		w_2		3

where the blanks represent the non-distinguished symbols and the tuples have been numbered for convenience. The transformation $<X \rightarrow Y, \{1, 3\}>$ is enabled and when applied produces:

X	Y	W	Z	
x_1	y_1	w_1	z_1	1
x_2	y_1	w_2	z_2	2
x_1	y_1	w_2		3

This causes the transformation $<YW \rightarrow Z, \{2, 3\}>$ to become enabled, which when applied produces:

X	Y	W	Z	
x_1	y_1	w_1	z_1	1
x_2	y_1	w_2	z_2	2
x_1	y_1	w_2	z_2	3

No further changes can be made so this is $I^*_{x_1w_2}$. We note, recalling example 1, that $z_2=f_f(x_1w_2)=(\varphi_f(I^*_{x_1w_2}))(x_1w_2)$. We will now show that, whenever a function is defined, it may be calculated in this way. ■

It has been shown [H] [G], that for any satisfying instance containing I and a tuple with X-value x, there exists a valuation function [ASU], a function from constants and non-distinguished symbols to constants, which function is the identity on constants and which takes tuples of I_x^* to tuples of the instance.

Lemma 1. Let I be an instance of R; let $f:X\to Y$ be a dependency and x an X-value. Then $f_I(x)$ is defined iff $I_x^*\neq\phi$ and no non-distinguished symbol appears in $(\varphi_f(I_x^*))(x)$. Further

$$f_I(x)=(\varphi_f(I_x^*))(x)$$

when $f_I(x)$ is defined.

Proof. Let t be any tuple with $t[X]=x$. If $I_x^*=\phi$, it is apparent that $I\bigcup\{t\}$ is not a satisfying instance. Thus no Y-value may be found to satisfy the definition, and $f_I(x)$ is undefined. Otherwise, assume $(\varphi_f(I_x^*))(x)$ contains no non-distinguished symbol, but $t[Y]\neq(\varphi_f(I_x^*))(x)$. But then there is no valuation function from I_x^* to $I\bigcup\{t\}$; therefore, $I\bigcup\{t\}$ is not satisfying. Finally assume $(\varphi_f(I_x^*))(x)$ contains a non-distinguished symbol in some column; say $(\varphi_f(I_x^*))(x)[Y_i]=b$. Assume $I\bigcup\{t\}$ is satisfying and let g be a valuation function from I_x^* to $I\bigcup\{t\}$. Let g' be defined such that $g'(a)=g(a)$ for $a\neq b$ and $g'(b)=c$ where c is a Y_i-value not appearing in $I\bigcup\{t\}^{\dagger}$. Then $g'(I_x^*)$ is a satisfying instance, implying $f_I(x)$ is undefined. ▪

The crux of examples 1 and 2 is that f may be derived from g and h. Sound and complete inference rules for deriving functional dependencies from a set of such dependencies have been known since the work of Armstrong [A]. The closure of a set, F, of dependencies, denoted F^+, is the smallest set containing F which is closed under the inference rules. Two sets of dependencies, F, G are equivalent, written $F\equiv G$, if $F^+=G^+$. G is said to be a cover of F when $F\equiv G$. G is a non-redundant cover of F when no proper subset of G is a cover of F.

A particular sound and complete set of rules is given by Bernstein [B]. The

†In the presence of domain constraints [F2], such an Y_i-value may not be available.

rules are reflexivity, augmentation and pseudo-transitivity.

$$(reflexivity)\ \ \phi \vdash X \rightarrow X$$
$$(augmentation)\ \ \{X \rightarrow Y\} \vdash XW \rightarrow Y$$
$$(pseudo-transitivity)\ \ \{X \rightarrow Y,\ YW \rightarrow Z\} \vdash XW \rightarrow Z$$

Bernstein has shown that any derivation of any dependency having a single attribute on the right from a set of such dependencies can be done using pseudo-transitivity as the sole inference rule followed by at most one application of augmentation. In [B], he presented a labelled graph construction which models derivations using pseudo-transitivity as the sole inference rule. The graphs are called derivation trees and they are defined recursively.

i) If A is an attribute, a single vertex labelled A is a derivation tree.

ii) If T is a derivation tree, $B_1B_2 \cdots B_p \rightarrow C$ is a dependency and C labels a leaf of T, then the tree formed from T by adding p leaves labelled $B_1, B_2, \ldots, B_p$ as descendants of C is a derivation tree.

iii) Nothing else is a derivation tree.

A derivation tree built with respect to a set of dependencies, F, the leaves of which tree are labelled by the set X and the root of which is labelled, A, is called an F-based derivation tree of X→A. Such an object need not be unique.

Bernstein gives the following justification of psuedo-transitivity. "[Let f be $X \rightarrow Y$; g be $YW \rightarrow Z$. If h is $XW \rightarrow Z$,] we can say $h(xw)$ is defined to be $g(f(x)w)$." We intend to explore the consequences of this idea. We will interpret each dependency in the given set as the appropriate table lookup expression, φ. We will compose these expressions using a given derivation tree as a guide. The result is called a *derivation expression*. Having done this, we will compare the derivation expressions for a given dependency to each other and to the corresponding function as defined above.

We proceed in stages. The first stage produces an expression over projection and a modified form of selection in which the selection formula may involve expressions. The second stage transforms the expression into one over selection, projection and join, demonstrating that the modified selection operator adds no new power to the relational algebra.

Stage 1. Let Dt be a derivation tree. The expression constructed by stage 1, denoted $\gamma(Dt)$, is defined by recursion on the height of Dt as follows:

$\gamma(Dt)=$

i) if *height*$(Dt)=0$, then l, where l is a formal variable associated with the label of the single node in Dt;

ii) if *height*$(Dt)>0$ and letting the degree of the root of Dt be m, then

$$\pi_B(\sigma_{X_1=\gamma(Dt_1)\wedge\cdots\wedge X_m=\gamma(Dt_m)}(I))$$

where B is the label of the root of Dt; X_i is the label of the root of Dt_i, the ith subtree of Dt. As we are concerned with expressions computing functions, each evaluation of $\gamma(Dt_i)$ returns at most one value (formally, at most a singleton, unary relation instance).

The modified selection expression generated by part *ii* of the definition is meant to return the subset of the relation comprised of tuples whose X_i-value is the value returned by the expression $\gamma(Dt_i)$.

Stage 2. Let $e=\gamma(Dt)$. Define a function δ recursively on the depth of expression nesting of e (equivalently, the height of Dt) as follows:

if the depth of e is 0, then $\delta(e)=e$, which is some formal variable l; otherwise, e is $\pi_B(\sigma_C(I))$ where C may be written

$$C_1\wedge\cdots\wedge C_k\wedge D_1\wedge\cdots\wedge D_l$$

for $k,l\geq 0$, where C_i is a simple condition of the form "attribute = formal variable" and D_j is of the form "attribute = stage 1 expression". In this case $\delta(e)$ is given by

$$\pi_B(\sigma_{C_1\wedge\cdots\wedge C_k}(\delta(e_1)*\cdots*\delta(e_l)*I))$$

where e_j is the stage 1 expression for i_j. The join portion of this expression is the subset of the relation comprised of tuples whose values are given by the values of the converted stage 1 expressions. Thus, informally, $\delta(\gamma(DT))$ is as desired.

The expression formed from a derivation tree, Dt, for an fd $f:L_1\cdots L_k \rightarrow A$ is denoted ψ_f and is given by

$$\psi_f \equiv \lambda I.\,\lambda l_1\cdots l_k.(\delta(\gamma(Dt)))$$

where $\{l_i \mid 1\leq i\leq k\}$ is the set of formal variables for the leaves of Dt. For $g:L_1\cdots L_kL_{k+1}\cdots L_{k+m}\rightarrow A$ derived by augmentation from f, the corresponding expression is

$$\psi_g \equiv \lambda I.\,\lambda l_1\cdots l_k l_{k+1}\cdots l_{k+m}.(\delta(\gamma(Dt)))$$

This effectively ignores the values of the attributes added by augmentation. Therefore we will feel free to denote by ψ_f the expression for the derivation of any fd from f by augmentation.

A derivation expression will be called *trivial* if the tree which generates it is trivial; i.e., is of height 0. If the label on the only vertex of a trivial derivation tree is A, the dependency derived is $A\rightarrow A$ (by reflexivity). The expression for this tree is

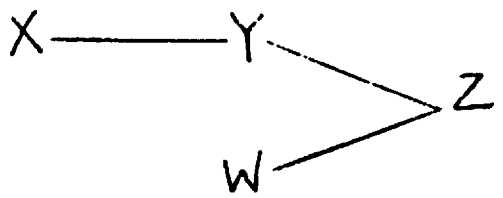
$$\lambda I.\,\lambda a.a$$

which for every instance J is the identity on $dom(A)$. For consistency, we define, where v is an A-value,

$$((\lambda I.\,\lambda a.a)(J))(v)\equiv\{v\}$$

This convention allows us to replace the selection operator with a join. This form facilitates any proof by induction over the complexity of the expression. Note that some trivial dependencies may have non-trivial expressions.

Example 3. Reconsider example 1. The derivation tree for f from g and h is



The first stage expression for this tree is

$$\gamma(Dt)=\pi_Z(\sigma_{W=w\wedge Y=\pi_Y(\sigma_{X=x}(I))}(I))$$

The output of stage two is

$$\delta(\gamma(Dt))=\pi_Z(\sigma_{W=w}((\pi_Y(\sigma_{X=x}(I)))*I))$$

Rewritten without selection, this becomes

$$\pi_Z(\{w\}*((\pi_Y(\{x\}*I))*I)) \quad ■$$

The following property of derivation expressions is basic.

Lemma 2. If I is an instance satisfying a set of functional dependencies F and ψ is a derivation expression for $X \to A$ wrt F, then for all X−values x, either $(\psi(I))(x)=\phi$ or $|(\psi(I))(x)|=1$.

Proof. A simple induction on the depth of nesting in ψ, which is omitted. ■

If $(\psi(I))(x)=\phi$, we say ψ is undefined at x in I. If $(\psi(I))(x)=\{a\}$, we will write $(\psi(I))(x)=a$. Lemma 2 states that, if ψ is a derivation expression for $X \to A$ built with respect to a set of dependencies F, then ψ is a mapping from instances satisfying F to functions from X-values to A-values.

Since a dependency may be derived from a set of dependencies in more than one way, ψ_f as defined is uniquely determined only with respect to a given derivation tree. Say that there are n distinct derivation trees for a given dependency.[†] Assign the integers $1 \cdots n$ to these trees in any way. Denote by ψ_f^i the

[†] There may be infinitely many distinct derivation trees.

expression generated by the i^th^ derivation tree for f. We will show, by way of an example, that instances may exist for which not only $\psi_j^i(I) \neq \psi_j^j(I)$, but indeed domain values exist for which these functions return distinct results.

Example 4. Let $F=\{X \rightarrow A,\ X \rightarrow B,\ YA \rightarrow Z,\ YB \rightarrow Z\}$. Then $F \vdash XY \rightarrow Z$ in two different ways. Consider the instance, which satisfies F

J=

X	A	B	Y	Z
x_1	a_1	b_1	y_2	z_1
x_2	a_1	b_2	y_1	z_2
x_3	a_2	b_1	y_1	z_3

The two stage 1 expressions associated with the two derivations of $XY \rightarrow Z$ are

$$\pi_Z(\sigma_{Y=y \wedge A=\pi_A(\sigma_{X=x}(I))}(I))$$

which uses $X \rightarrow A$, and

$$\pi_Z(\sigma_{Y=y \wedge B=\pi_B(\sigma_{X=x}(I))}(I))$$

which uses $X \rightarrow B$. The derivation expressions are:

$$\psi^1 \equiv \lambda I.\ \lambda xy.\ \pi_Z(\sigma_{Y=y}((\pi_A(\sigma_{X=x}(I))) * I))$$
$$\psi^2 \equiv \lambda I.\ \lambda xy.\ \pi_Z(\sigma_{Y=y}((\pi_B(\sigma_{X=x}(I))) * I))$$

Partially evaluating each of these expressions in J at x_1y_1 we have that

$$(\psi^1(J))(x_1y_1)=\pi_Z(\sigma_{Y=y_1}(J'))$$

and

$$(\psi^2(J))(x_1y_1)=\pi_Z(\sigma_{Y=y_1}(J''))$$

where

J'=

X	A	B	Y	Z
x_1	a_1	b_1	y_2	z_1
x_2	a_1	b_2	y_1	z_2

and

$J''=$

X	A	B	Y	Z
x_1	a_1	b_1	y_2	z_1
x_3	a_2	b_1	y_1	z_3

Which is to say, $(\psi^1(J))(x_1y_1)=z_2$ and $(\psi^2(J))(x_1y_1)=z_3$.

Consider calculating $XY \to Z$ at x_1y_1. $J_{x_1y_1}$ is given by

$J_{x_1y_1}=$

X	A	B	Y	Z	Nbr
x_1	a_1	b_1	y_2	z_1	1
x_2	a_1	b_2	y_1	z_2	2
x_3	a_2	b_1	y_1	z_3	3
x_1			y_1		4

(the blanks representing non-distinguished symbols) and we apply the transformation sequence

$$
\begin{gathered}
<X \to A,\ \{1,4\}> \\
<X \to B,\ \{1,4\}> \\
<YA \to Z,\ \{2,4\}> \\
<YB \to Z\{3,4\}>
\end{gathered}
$$

the last transformation being contradicted. So $I_x^*=\phi$ and the function is undefined at that point. ▪

In light of example 4, we define for any functional dependency f, a mapping Ψ_f from instances to functions. For $f:X \to A$, Ψ_f is defined for an instance I and X-value x as

$$(\Psi_f(I))(x)=(\bigcup_i(\psi_f^i(I))(x))$$

where the union is taken over all derivation expressions for f. This definition is sensitive to the particular set of dependencies with respect to which the derivations are carried out. A consequence of this sensitivity is pointed out in a subsequent section. We say that $\Psi_f(I)$ is undefined[†] at a value x where $|(\Psi_f(I))(x)|\neq 1$. As above, if $(\Psi_f(I))(x)=\{a\}$, we will write $(\Psi_f(I))(x)=a$.

Finally, we add dependencies with multi-attribute right hand sides. For

[†] Our use of the term 'undefined' to describe certain behaviour of the evaluation of expressions does not alter the fact that $(\Psi_f(I))(x)$ is well defined for all I, x. $(\Psi_f(I))(x)$ is in every case an instance of a relation whose scheme is given by the right hand side of f.

$f:X\to Y$ where $Y=Y_1\cdots Y_k$, let Ψ_{f_i} be the expression for $f_i:X\to Y_i$. Then $(\Psi_f(I))(x)=\overset{k}{\underset{i=1}{*}}(\Psi_{f_i}(I))(x)$. $\Psi_f(I)$ is defined exactly at those points where all of the $\Psi_{f_i}(I)$ are defined, as the reader may verify. For the most part, we will restrict ourselves to single attribute right hand sides for convenience.

Now we present some facts relating derivation expressions to functions in the setting of a single relation. Ψ_f may, for a given instance, be defined at more values than φ_f is defined. However, the two functions agree wherever both are defined.

Proposition 1. For any satisfying instance I, fd $f:X\to Y$

$$(\varphi_f(I))\subseteq(\Psi_f(I))$$

Furthermore if $f_I(x)$ is defined then

$$f_I(x)=(\Psi_f(I))(x)$$

Proof. see [G]. ▪

This proposition, with lemma 1, states that the value of a function at a point is computed by some derivation of that function. It suggests that the derivations may provide more information than is actually present in the relation, returning values where the function is undefined. This can in fact occur.

Example 5. Add to the set F in example 4, the dependency $Z\to C$. Let g be $XY\to C$ and let K be the instance

K=

X	A	B	Y	Z	C
x_1	a_1	b_1	y_2	z_1	c_1
x_2	a_1	b_2	y_1	z_2	c_1
x_3	a_2	b_1	y_1	z_3	c_1

Then $(\Psi_g(K))(x_1y_1)=c_1$, but $g_K(x_1y_1)$ is undefined, as in example 4. ▪

In the many relation case considered in the next section, we will see that the derivations may be less defined than the function.

We end this section by stating a fact about derivation trees. Whenever F allows derivation trees to be built within which an attribute may label more than one node on some root to leaf path, then F allows the construction of infinitely many derivation trees. (This occurs, for example, when $F=\{A\rightarrow B,\ B\rightarrow A\}$.) Those trees in which no attribute appears more than once on a root to leaf path are called bounded trees; there are clearly only finitely many of them.

Proposition 2. For any satisfying instance I and dependency $f:X\rightarrow Y$, if $f_I(x)$ is defined then $f_I(x)=(\psi(I))(x)$ where ψ is the expression for some bounded derivation tree for f.

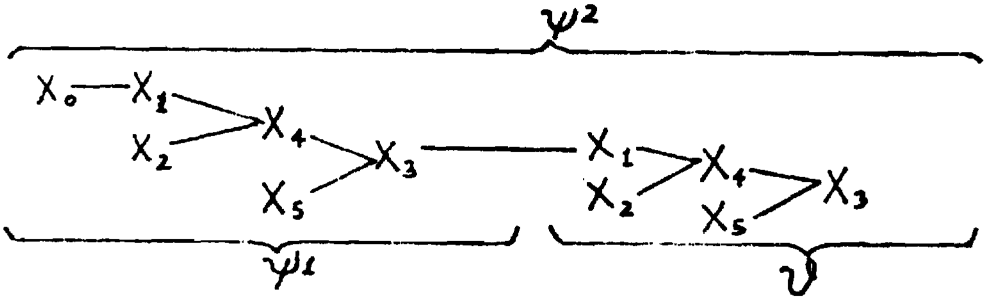
Proof. see [G]. ▪

Observe that at values of X for which f_I is not defined, it may be that a non-bounded expression may return a result not returned by any bounded expression. Sets of dependencies exist such that for any $k>0$, an instance satisfying the given set may be constructed in which k distinct results are returned by k different expressions for the same dependency.

Example 6. Let

$$R=X_0X_1X_2X_3X_4X_5$$
$$F=\{X_0\rightarrow X_1,\ X_3\rightarrow X_1$$
$$X_4X_5\rightarrow X_3,\ X_1X_2\rightarrow X_4\}$$

For ease of construction, for each i let $dom(X_i)$=N, the natural numbers. Let f be $X_0X_2X_5\rightarrow X_3$ one of whose non-height-bounded trees is



Consider the instance of R given by

I

X_0	X_1	X_2	X_3	X_4	X_5
0	0				
	0	0		0	
	1		1	0	0
	1	0		1	
	2		2	1	0
.					
.					
.					
	$k-1$	0		$k-1$	
	k		k	$k-1$	0

in which the blanks represent unique values. Therefore I has $2k+1$ tuples, the first of which contains 0 in X_0X_1. The $i+1^{st}$ contains

- for i odd $\left\lfloor \frac{i}{2} \right\rfloor$ in X_1X_4, 0 in X_2;
- for i even $\frac{i}{2}$ in X_1X_3 $\frac{i}{2}-1$ in X_4, 0 in X_5.

The reader may wish to verify that

$$(\psi^1(I))(000)=1; \ (\psi^2(I))(000)=2$$

and that, if ψ^l is the tree formed by adding $l-1$ copies of ϑ to ψ^1, then for $l \leq k$

$$(\psi^l(I))(000)=l \ ▪$$

3. Functional Dependencies in Multi-relation Databases

We now take up the behaviour of functional dependencies in databases containing more than one relation. The *schema* of such a database is of the form R$=\{R_1, \ldots, R_k\}$. A *state* of such a database is of the form $\rho=\{r_1, \ldots, r_k\}$, where for each i, r_i is an instance of R_i. We associate with such a database a universal relation scheme R where $R= \bigcup\limits_{1 \leq i \leq k} R_i$. An instance of R is called a *universal instance* for R. Let F be the set of functional dependencies which we wish all states to satisfy and represent. The set F_i is the subset of F^+ which mentions only attributes in R_i. We say that ρ is *locally satisfying* if for each i, r_i satisfies

F_i. We say that R is cover-embedding if $(\bigcup_{i \le k} F_i)^+=F^+$.

Consider a schema R and a state ρ for R which satisfies the Universal Relation Instance Assumption. That is, there exists an instance I of R such that for every instance r_i of ρ

$$\pi_{R_i}(I)=r_i$$

Such a state is said to be join consistent, since the join of such a state is such an instance. Now if R is cover-embedding and each instance r_i of ρ satisfies F_i, then I satisfies F. Thus for join consistent states of cover-embedding schemas, local satisfaction may be taken as a definition of satisfaction for the state. If ρ is not join consistent, then this definition is vacuous and another must be sought.

If ρ is a state and C is an instance of R, then C is said to be a containing instance for ρ if $\pi_{R_i}(C)\supseteq r_i$ for each instance r_i of ρ. If C satisfies F, then C is said to be a *weak instance*. Our definition of satisfaction is straight-forward.

(Satisfaction-2) A database state is satisfying if there exists a weak instance for it.

Say that W is a weak instance for a satisfying state, ρ. Then ρ may be expanded to a join consistent state through insertion of certain tuples, namely those tuples in $\pi_{R_i}(W)-r_i$ for each r_i, such that the resulting universal instance satisfies all the dependencies of interest. A non-satisfying state can not be modified so as to have a satisfying universal instance without some data being dropped. The intuition behind this definition is that a satisfying state is one for which it can not be proven that some dependency has been violated.

Even within cover-embedding schemas, local satisfaction is not sufficient for satisfaction. We demonstrate this through an example.

Example 7. Let $F=\{A\to C, B\to C\}$. Let **R**$=\{AB, AC, BC\}$. **R** is the output of the schema synthesis algorithm of Biskup et. al. [BDB] when given input F. Let ρ be the state

A	B
0	0

A	C
0	0

B	C
0	1

No weak instance exists for ρ. Any such instance must have a tuple with AB-value 00. The C-value of that tuple must be simultaneously 0 and 1. ▪

In [H], Honeyman gives an algorithm based on the algorithm of Downey et al. [DST] for deciding if a database state has a weak instance. The algorithm has time complexity $O(n \log n)$ where n is roughly the number of tuples in the state. Our interest is in the functions represented in a database state, to which we now turn.

(Representation-2) Let **R** be a schema and F a set of fd's. Let $f{:}X\to Y$ be a dependency in F or derivable from it. Let ρ be a state for **R**. Then for x any X-value and y a Y-value, $f_\rho(x)=y$ if and only if

i) there exists some weak instance, w, for ρ in which $f_w(x)=y$ *and*

ii) for any weak instance w' for ρ, either $f_{w'}(x)=y$ or $f_{w'}$ is undefined at x.

Otherwise f_ρ is undefined at x. ▪

The functions represented in a database state map values in their domains to results which are required by the information in the state. By this definition, non-satisfying states represent only empty functions.

This definition does not suggest an effective means of calculating the functions. The method used in the single relation case may be adapted for use in the multi-relation case through the use of the *tableau* for the state, which is a containing instance for the state in which certain places are occupied by variables.

(Tableau) For each tuple, t, of each relation instance, r_i, in a state ρ, there is a row, u, of the tableau T_ρ with $t[A]=u[A]$ for every attribute, A, in R_i. For every attribute, B, not in R_i, $u[B]$ is a non-distinguished symbol appearing nowhere else in T_ρ.

The chase procedure as described in section 1 may be applied to a tableau. The result of chasing a tableau T, denoted T^* and called a completed or chased tableau, does not depend on the order in which the transformations are applied [G] nor on the cover with respect to which the transformations are formed [MMS]. If $T_\rho^*=\phi$, where ρ is not the empty state, then ρ has no weak instance [H] [G]. As for augmented instances in section 1, there is a valuation function from the completed tableau into every weak instance for the database state. Lemma 1 can now be restated for multi-relation database states.

Lemma 3. Let R be a scheme; F be a set of fd's. Let ρ be a satisfying state for R and let $f:X \rightarrow A$ be a dependency in F or derivable from it. Let x_0 be an X-value. Let T be T_ρ to which a row is added containing X-value x_0 and new, distinct non-distinguished symbols everywhere else. Then

$$f_\rho(x_0)=(\varphi_f(T^*))(x_0)$$

where T^* is the completion of T with respect to F and $f_\rho(x_0)$ is undefined if $T^*=\phi$ or $(\varphi_f(T^*))(x_0)$ contains a non-distinguished symbol. ▪

Example 8. Consider the system defined by:

$$F=\{X_1 \rightarrow Y_1,\ X_2 \rightarrow Y_2,\ Y_1Y_2 \rightarrow X_1,\ Y_1Y_2 \rightarrow X_2,\ X_1X_2 \rightarrow Z\}$$
$$\mathrm{R}=\{X_1Y_1,\ X_2Y_2,\ Y_1Y_2Z\}$$

Let ρ be the state given by

X_1	Y_1
x_1	y_1

X_2	Y_2
x_2	y_2

Y_1	Y_2	Z
y_1	y_2	z

Let f be the dependency $X_1X_2 \rightarrow Z$. We may use lemma 3 to compute $f_\rho(x_1x_2)$. T is

	X_1	X_2	Y_1	Y_2	Z
t_1	x_1	b_1	y_1	b_2	b_3
t_2	b_4	x_2	b_5	y_2	b_6
t_3	b_7	b_8	y_1	y_2	z
t_4	x_1	x_2	b_9	b_{10}	b_{11}

where the subscripted b's are non-distinguished symbols. A transformation sequence for T with respect to F is

$$\begin{gathered} <X_1 \rightarrow Y_1, \ \{t_1, t_4\}> \\ <X_2 \rightarrow Y_2, \ \{t_2, t_4\}> \\ <Y_1Y_2 \rightarrow X_1, \ \{t_3, t_4\}> \\ <Y_1Y_2 \rightarrow X_2, \ \{t_3, t_4\}> \\ <X_1X_2 \rightarrow Z, \ \{t_3, t_4\}> \end{gathered}$$

which sets $b_{11}=z$. Therefore $(\varphi_f(T^*))(x_1x_2)=z=f_D(x_1x_2)$. This example justifies our interest in calculating functions on values not present in a database state. It does not seem reasonable to believe that any state of **R** "contains" any X_1X_2 value. Nonetheless, **R** has states in which f is defined. ▪

In proposition 1 we saw that for the single relation case, a function is no more defined than its derivation cxpressions. We now demonstrate that this is not true in the multi-relation case. The derivation expressions can only be applied to a satisfying single relation. The tableau T_ρ^* is the natural candidate. The prior example demonstrates the falseness of proposition 1 for the multi-relation case, as the only derivation expression for $X_1X_2 \rightarrow Z$ with respect to F is just the simple select, project expression, φ. However, this result depends upon our choice for F. The proposition gives an example insensitive to the choice of cover.

For the fd, $f:X \rightarrow Y$, an expression ψ_f will be considered undefined in T_ρ^* at x unless $(\psi_f(T_\rho^*))(x)=y$ for some constant Y-value. Similarly, $(\Psi_f(T_\rho^*))(x)$ is said to be undefined unless it contains exactly one constant Y-value and when defined will be said to be equal to that value. We say that a function f is *more defined* than a function g, if for all x at which $g(x)$ is defined, $f(x)=g(x)$, and some value x' exists at which $f(x')$ is defined and $g(x')$ is not.

Proposition 3. There exists a schema R having a state ρ such that a function g_ρ for $g \in F^+$ is more defined than $\Psi_g(T_\rho^*)$. This is insensitive to the choice of cover for F with respect to which Ψ_g is defined.

Proof. We give an example of such a system. Let $F=\{X_1\rightarrow X_4, X_2\rightarrow X_4, X_3\rightarrow X_4, X_4X_5\rightarrow X_7, X_4X_6\rightarrow X_7\}$. Let $R=\{X_3X_4X_5X_7, X_2X_5X_6, X_1X_2\}$ Let the state ρ be given by

X_3	X_4	X_5	X_7
0	0	0	0

X_2	X_5	X_6
0	0	0

X_1	X_2
0	0

T_ρ^* is given by

X_1	X_2	X_3	X_4	X_5	X_6	X_7
		0	0	0		0
	0		b_1	0	0	b_2
0	0		b_1			

where the blanks stand for non-distinguished symbols which are not repeated. Let g be $X_1X_3X_6\rightarrow X_7$. There are two derivations of g from F; one uses $X_1\rightarrow X_4$, the other $X_3\rightarrow X_4$. Using lemma 3, we can calculate $g_D(00)=0$. But $(\Psi_g(T_D^*))(00)$ is undefined. The derivation through $X_1\rightarrow X_4$ returns b_2. The derivation through $X_3\rightarrow X_4$ returns ϕ. Proofs of these contentions a.'e left to the reader.

We can prove the result to be insensitive to the choice of cover for F, by showing that F is unique; that is, for any set of fd's G such that the right hand side of each dependency in G is a single attribute, $G\equiv F$ and G non-redundant implies $G=F$.

So let G be non-redundant and equivalent to F. We know $G\vdash X_1\rightarrow X_4$. So there is some fd in G with left hand side X_1, since, by inspection, none of the inference rules decrease the left hand side of any fd. X_4 is the only attribute in the closure of X_1 wrt F (other than X_1 itself). So $X_1\rightarrow X_4\in G$. Similar arguments hold for $X_2\rightarrow X_4$, $X_3\rightarrow X_4$. Now note that each of X_4, X_5 determines only itself in F^+. Therefore the derivation of $X_4X_5\rightarrow X_7$ from G does not proceed by pseudo-transitivity from $X_4 \to W$, $X_5 W \to X_7$ for any W (and symmetrically). So there is an fd $X_4 X_5 \to Y$ in G. If Y is other than X_7, then $G \not\equiv F$. The same argument shows $X_4 X_6 \to X_7$ is in G.

Since $F \subseteq G$, $G \equiv F$ and G is non-redundant, $G = F$ as claimed. ▪

4. **Representation by a Schema**

In the previous section we gave a definition of function representation in the state of a mul'iple relation schema. We now discuss what it means for the schema to represent, or fail to represent, a function.

(Representation-3) Let R be a schema and F a set of fd's. Let f be a dependency in or derivable from F. We say that R represents f if there exists a state ρ for R such that f_ρ is not the empty function. ▪

Consider the contrapositive of this definition. If in every state ρ for R f_ρ is defined nowhere, then surely it is reasonable to state that R does not represent f. Therefore any reasonable definition must be at least as strong as this one and the results of this section are implied by any such definition.

The main result of this section is that a schema represents all the functions of interest exactly when it is dependency preserving with respect to them. Before we can present the result, we need to make some preliminary definitions.

Associated with any schema there is an expression called the projection-join mapping of the schema. If $\mathbf{R} = \{R_1, \ldots, R_k\}$ is a schema, the associated mapping, denoted $m_{\mathbf{R}}$, is given by

$$m_{\mathbf{R}} \equiv \lambda I.\ (\pi_{R_1}(I) * \cdots * \pi_{R_k}(I))$$

where I is an instance of the universal relation for R. Thus $m_{\mathbf{R}}$ is a mapping from universal instances to universal instances. R is a ***dependency preserving*** schema if for any satisfying instance, I, $m_{\mathbf{R}}(I)$ is a satisfying instance.

A tableau, $T_{m_{\mathbf{R}}}$, may be constructed for the projection-join mapping associated with a schema. Let J be a universal instance for $\mathbf{R}$ consisting of a single tuple. For uniqueness and conformity with [ASU] and others, a special constant called a *distinguished symbol* is used in each column of this tuple. Let σ be the state formed by projecting this instance onto the schemes of R. $T_{m_{\mathbf{R}}}$ is just T_{σ}, the tableau for this state.

A set of attributes W is said to be *embedded* in a tableau if there is some row t of the tableau such that for each $A \in W$, $t[A]$ is the distinguished symbol. A functional dependency $X \rightarrow Y$ is embedded in a tableau if the set XY is embedded in it. The following theorem is proven in [BMSU].

Theorem 1. The following are equivalent:

i) A schema $\mathbf{R}$ preserves a set of dependencies F

ii) $T^{*}_{m_{\mathbf{R}}}$ embeds some non-redundant cover of F

iii) $T^{*}_{m_{\mathbf{R}}}$ embeds every non-redundant cover of F. ▪

From this theorem we can immediately see that any schema which is either cover-embedding or a lossless decomposition is dependency preserving. (A schema is a lossless decomposition if $T^{*}_{m_{\mathbf{R}}}$ contains a row of only distinguished symbols. In this case for all satisfying universal instances I, $m_{\mathbf{R}}(I)=I$.)

The following states a basic relationship between an arbitrary state and the tableau $T_{m_{\mathbf{R}}}$.

Lemma 4. Let t, u be rows of $\chi(T_{\rho})$ for a state ρ of a schema $\mathbf{R}$ and some transformation sequence χ. Let t correspond to a tuple from relation T, u one from relation U. Let A be an attribute.

1) If $t[A]=u[A]$ then in $T^{*}_{m_{\mathbf{R}}}$ $T[A]=U[A]$ where the relation names are used to denote the rows of $T^{*}_{m_{\mathbf{R}}}$ representing them.

2) If $t[A]$ is constant then $T[A]$ is distinguished.

Proof. We show that any transformation sequence on a database state tableau can be carried out in the tableau $T_{m_{\mathbf{R}}}$. Let $<h, \{r,s\}>$ be a transformation of the sequence where r is from relation R, s from relation S. Convert this to $<h, \{R,S\}>$. Note that if $R=S$, this is a null transformation and if it makes the premise of 1 or 2 true the consequence will also hold.

We prove the lemma by induction on the length of the transformation sequence preceding the transformation making the premise of 1 or 2 true.

Basis The first transformation in the sequence involves an fd whose left hand side is a subset of one of the relation schemes of R.

Induction Assume the lemma holds for prefixes of length no greater than m. Assume a prefix of length $m+1$. We need to show that the $m+2$'nd transformation becomes enabled and if it makes the premise of property 1 or 2 true in T_ρ, then the consequence will hold in $T^*_{m_{\mathbf{R}}}$.

Let $r=<W \rightarrow B, \{r,s\}>$ be the $m+2$'nd transformation where $W=W_1 \cdots W_l$. Consider the transformation, if one exists, which set $r[W_i]=s[W_i]$ $1 \le i \le l$. If no such transformation exists then this equality holds in the T_ρ and therefore in $T_{m_{\mathbf{R}}}$ since $W_i \in R$, $W_i \in S$. Assuming such a transformation does exist, it preceded r, has been executed and $R[W_i]=S[W_i]$ by induction after its execution (and possibly before). So r becomes enabled in $T_{m_{\mathbf{R}}}$. Now assume r has the effect of setting $t[B]=u[B]$. Then, possibly after renaming, $r[B]=t[B]$ and $s[B]=u[B]$ before execution of r; that is, these equalities were established by a transformation which preceded r. Therefore, by induction, $R[B]=T[B]$ and $S[B]=U[B]$ and after r is executed in $T_{m_{\mathbf{R}}}$ $T[B]=U[B]$ (and possibly before).

Now assume r sets the B-value of some row to a constant. This constant appeared in one of $r[B]$ or $s[B]$. Thus $R[B]$ (or $S[B]$) is distinguished by induction. Thus property 2 holds after τ is executed in T_{m_R} (and possibly before). ▪

For a schema R, the tableau T_X is the tableau T_{m_R} to which a row is added containing only distinguished symbols in the X-columns and new, non-distinguished symbols everywhere else. This added row will be called the X-row.

Lemma 5. A schema R represents a dependency $f:X\to Y$ in or derivable from a set of fd's F if and only if in T_X^* the Y-value in the X-row is all distinguished symbols.

Proof. (Only if) This is an easy consequence of lemma 4. T_X is the tableau of the projection-join mapping of the schema $\mathbf{R}\cup\{X\}$. The procedure of lemma 3, which calculates the value of a function at a point, is the chase of a state of this schema.

(If) Consider a state ρ for R which is the set of projections of an instance of the universal relation containing a single tuple, t. We claim $f_\rho(t[X])=t[Y]$.

Let T be $T_\rho\cup\{t_x\}$ where $t_x[X]=t[X]$ as in lemma 3. T is (up to isomorphism) the tableau T_X. By hypothesis, T^* contains a row containing only constant values in the XY-columns. Since the only Y-value in T is $t[Y]$, the claim is established. ▪

It is not difficult to show that dependency preserving schemas satisfy the conditions of lemma 5 for all dependencies in or derivable from F. Such schemas represent any dependency appearing in any non-redundant cover, as a consequence of theorem 1. For a dependency $X\to Y$, not in any cover, representation is an immediate consequence of the following proposition and the fact that any superset of a dependency preserving schema is dependency preserving.

Proposition 4. ([BMSU], proposition 1) Let R be a dependency preserving schema. Let X be the set of attributes of a row r of T_{m_R} containing distinguished symbols. Then the row of $T_{m_R}^*$ corresponding to r has distinguished symbols in the attributes $X^+\equiv\{A \mid X\to A\in F^+\}$ and no symbol of r outside of X^+ repeats in

$T^*_{m_R}$. (X^+ is called the *closure* of X and is called a *closed* set of attributes.)

We will now show that dependency preserving schemas are in fact the only schemas which satisfy the conditions of lemma 5 for all dependencies.

Let $X \to A$ be a dependency in F not embedded in $T^*_{m_R}$, but embedded in T^*_X. We may assume without loss of generality, that no row of $T^*_{m_R}$ has only distinguished symbols in the X-columns. Assume otherwise. Note that T^*_X may be formed by chasing $T^*_{m_R} \cup \{t_X\}$ where t_X is the X-row. If some row of $T^*_{m_R}$ has all distinguished symbols in the X-columns, then $T^*_{m_R} \cup \{t_X\}$ contains $T^*_{m_R}$ in the tableau containment sense[†]. Since the chase preserves the containment relationship and $X \to A$ is embedded in T^*_X it must be embedded in $T^*_{m_R}$, a contradiction. Assume however that $X \to A$ is embedded in T^*_X.

We may begin the chase of T_X by first chasing the rows of T_{m_R}, transforming them to $T^*_{m_R}$. Let χ be any sequence of transformations with respect to F which completes the calculation of T^*_X from this point. We wish to distinguish two types of transformations.

i) A transformation is of type *i* if it equates a non-distinguished symbol in the X-row to some other non-distinguished symbol.

ii) A transformation is of type *ii* if it equates a non-distinguished symbol in a row of $T^*_{m_R}$ to a distinguished symbol.

Lemma 6. If χ makes any two symbols in rows of $T^*_{m_R}$ equal which were not equal in $T^*_{m_R}$, then χ contains a transformation of type *i* or *ii*.

Proof. Assume otherwise. Let $\tau = <Z \to A, \{t_1, t_2\}>$ be the first transformation of χ which makes two such symbols equal. Since there are no transformations of

[†] A tableau T_1 contains a tableau T_2, if a function exists from the symbols of T_1 to the symbols of T_2 which *i)* maps rows of T_1 to rows of T_2 and *ii)* preserves distinguished symbols. Such a function is called a containment mapping.

type *ii* in χ. $t_1[A]$, $t_2[A]$ must both be non-distinguished. Neither t_1 nor t_2 may be the X-row by the exclusion of type *i* transformations. τ must not have become enabled during the calculation of $T^*_{m_E}$, else $t_1[A]=t_2[A]$. Therefore it became enabled during χ. But then two of the symbols in t_1, t_2 became equated during χ, violating the choice of τ as the first transformation with this effect. ▪

Lemma 7. The dependency of the first transformation of type *i* or *ii* which appears in χ is not embedded in $T^*_{m_E}$ but its left hand side is embedded.

Proof. Let $\tau=<Z \to A, \{t_1, t_2\}>$ be this dependency, if it exists. By lemma 6, one of t_1, t_2 must be the X-row. Let t_2 be the other row, corresponding to a row of $T^*_{m_E}$. Since there are no prior transformations of type *i*, it must be that $t_2[Z]$ is all distinguished. Since there are no prior transformations of type *ii*, all these symbols must have been distinguished in $T^*_{m_E}$. But $t_2[A]$ is not distinguished, otherwise τ would not be of type *i* or *ii*. ▪

As we have seen before, if Z is embedded in some row of $T^*_{m_E}$ but $Z \to C$ is not, then T^*_Z will not embed $Z \to C$ and R will not represent it. Therefore we may assume that no transformation of type *i* or *ii* appears in χ. Further we may assume that every transformation of χ involves the X-row. Let F' be the subset of F appearing in transformations of χ. It can be shown that $F' \vdash X \to Y$ (see [G]). But $X \to Y$ can not be in F', since no row of $T^*_{m_E}$ will have X become embedded in it. Therefore $X \to Y$ is not in F or F is redundant.

We have shown that a non-dependency preserving schema fails to represent at least one functional dependency in every cover. The reader should note that it may represent some of the non-embedded dependencies in a non-redundant cover. In example 8, the dependency $X_1X_2 \to Z$ was shown to be represented, even though it is not embedded in $T^*_{m_E}$. Note that the dependencies $Y_1Y_2 \to X_i$ $(i=1, 2)$ are not represented in that example. We express this result as a theorem.

Theorem 2. A schema R represents all the functional dependencies in or derivable from F if and only if it preserves the dependencies in F. ▪

We can now prove a property of dependency preserving schemes which reinforces the belief that they are "good," namely that when an instance of the universal scheme is stored as the state of a dependency preserving schema, the functional associations of the decomposed state are exactly those of the instance.

Proposition 5. Suppose R preserves a set of dependencies F and ρ is the set of projections of a satisfying universal instance I. Then for every $f \in F^+$, $f_\rho = f_I$.

Proof. (Sketch) It can be shown that since ρ is join consistent, T^*_ρ contains a set of isomorphic images of $T^*_{m_{\mathbf{R}}}$, one image for each tuple of I. Since R is dependency preserving, every row of T^*_ρ has constants in a closed set of attributes and no non-distinguished symbol appears more than once. Suppose we wish to calculate f_ρ at a value x. The chase of $T^*_\rho \cup \{t_x\}$ will, unless and until it finds a contradiction, make no change to the rows of T^*_ρ. So every transformation will make a non-distinguished symbol of t_x constant. Exactly the same is true of any transformation sequence for I_x. From these facts it can be shown that for every transformation sequence on $T^*_\rho \cup \{t_x\}$ there is one on I_x with the same effect on the row t_x and conversely. The proposition follows from that. A complete proof may be found in [G].

5. Functional Dependencies as Constraints

We have shown that a schema may represent functional dependencies which are not embedded in any relation of the schema. This has salutory effects on schema design. It has long been known that there exists sets of dependencies for which no cover-embedding schema may be found each of whose relations is in Boyce-Codd Normal Form (BCNF)[†]. It has also been known that lossless

[†] A scheme R is in BCNF if for every functional dependency $X \to A$ embedded in R, X is a key of R, i.e., $X \to R$

decompositions in arbitrarily stringent normal forms can always be found for any set of dependencies. (BCNF is the strongest normal form when only functional dependencies are considered.) Theorem 2 suggests that a lossless decomposition is "good enough". We present further evidence for this belief.

Example 9. Consider $R=ABC$, $F=\{AB \rightarrow C, C \rightarrow B\}$. No BCNF, cover-embedding decomposition of R exists. However the schema $\mathbf{R}=\{AC, BC\}$ is lossless and therefore dependency preserving. Therefore **R** represents the dependency $AB \rightarrow C$ which it does not embed. States for **R** may be constructed which are not satisfying precisely because they violate $AB \rightarrow C$.

A	C
a_1	c_1
a_1	c_2

B	C
b_1	c_1
b_1	c_2

This state is locally satisfying. However the reader may convince himself using the techniques of the prior sections, that no weak instance exists for it. The state is the projection of a universal instance containing the two tuples $a_1b_1c_1$ and $a_1b_1c_2$. This instance does not satisfy F. ▪

Expanding on example 9, we would like to know under what circumstances a dependency has the power to act as a constraint on database states. If a schema represents the dependency $X \rightarrow Y$, then it is capable of assigning a Y-value to any X-value. It seems reasonable to believe that such a schema is capable of assigning more than one distinct Y-value to a given X-value. A state of the database in which this occurs would be illegal, were this X-value to appear in the database. Moreover, we can show that dependencies which are not represented may still act to constrain the set of satisfying database states.

Example 10. Let F be $\{A \rightarrow C, B \rightarrow C, CD \rightarrow E\}$. Let **R** be $\{AB, BDE, C\}$.Let a state ρ be

A	B
0	0
0	1

B	D	E
0	0	0
1	0	1

with the instance for C empty. ρ is locally satisfying but not satisfying as the reader may verify. The reader may also verify that no dependency in F is represented by R. However, if any dependency is removed from F, ρ has a weak instance satisfying the remaining dependencies. ▪

If R is a relation scheme, it is customary to define

$$SAT(R,F)\equiv\{r \mid r \text{ is an instance of } R \text{ satisfying } F\}$$

By analogy, if R is a database scheme we define

$$SATW(\mathbf{R},F)=\{\rho \mid \rho \text{ is a state of } \mathbf{R} \text{ having a weak instance wrt } F\}$$

We may now define what is meant by a dependency acting as a constraint.

(Constraint.) Let R be a database scheme; F a set of fd's. Let f be a functional dependency. We say that f acts as a constraint on R wrt F if

$$SATW(\mathbf{R},F\cup\{f\})\neq SATW(\mathbf{R},F-\{f\})$$

Note that $SATW(\mathbf{R},F\cup\{f\})\subseteq SATW(\mathbf{R},F-\{f\})$ holds for all R, F, f. The definition implies that, when f constrains R, chasing states of R using all the dependencies in $F\cup\{f\}$ gives a different *yes/no* result for some states than is given by not using f. Thus if f does not act as a constraint on R it does not affect the determination of satisfaction for any state of R.

From the definition, we see immediately that if $F-\{f\}\vdash f$, f does not act as a constraint. This is an application of the easily proven fact that for sets of dependencies F and C, $F\equiv C$ implies $SATW(\mathbf{R},F)=SATW(\mathbf{R},C)$ for any R. The converse of this statement is false, as we show by way of example.

Example 11. Let F be $\{A\rightarrow C\}$. Let R be $\{AB, BC\}$. From lemma 5 we deduce that R does not represent $A\rightarrow C$. Therefore the chase of the tableau for any state of R never produces a tuple with a constant AC-value. Thus no such state contradicts $A\rightarrow C$. In short

$$SATW(\mathbf{R}, \{A \to C\}) = SATW(\mathbf{R}, \phi)$$

and every state of **R** is satisfying. ▪

We now present a sufficient condition for a dependency to act as a constraint. We will then present a necessary condition. The two conditions are similar but not identical. We end the section by demonstrating that acting as a constraint is a "cover-sensitive" property of dependencies.

Assume that F is non-redundant and $f \in F$. Define the contribution of f wrt F by

$$contr_F(f) = F^+ - (F - \{f\})^+$$

An element of $contr_F(f)$ is a dependency all of whose derivations use f. Since F is non-redundant, we always have $f \in contr_F(f)$. Let $\vec{d}$ be a vector of distinguished symbols; the length of the vector may be deduced from the context in which it appears.

Theorem 3. Let **R** be a schema and F a non-redundant set of fd's. A dependency $f \in F$ acts as a constraint on **R** if there exists a dependency g in $contr_F(f)$ such that $(\Psi_g(T^*_{m_\mathbf{R}}))(\vec{d}) = a$.

Proof. Let $g \in contr_F(f)$ be $L \to B$ and let $(\Psi_g(T^*_{m_\mathbf{R}}))(\vec{d}) = a$. Construct w a two tuple universal instance. Let these tuples be t_1, t_2 and let $t_1[A] = t_2[A]$ precisely when $L \to A \in (F - \{f\})^+$. Let ρ be the set of projections of w onto the schemes of **R**. We show that ρ witnesses the fact that f acts as a constraint on **R**. That is, we show $\rho \in SATW(\mathbf{R}, F - \{f\}) - SATW(\mathbf{R}, F)$.

Part 1. $\rho \in SATW(\mathbf{R}, F - \{f\})$.

We show w is in $SAT(R, F - \{f\})$. Let $h: M \to C$ be any fd in $F - \{f\}$. If $t_1[M] = t_2[M]$ then $L \to M \in (F - \{f\})^+$ so $L \to C \in (F - \{f\})^+$ and $t_1[C] = t_2[C]$ by construction.

Part 2. $\rho \not\in SATW(\mathbf{R}, F)$

Assume to the contrary, that ρ satisfies F. We note that T_ρ contains two isomorphic images of $T_{m_{\mathbf{R}}}$, namely, the projections of each of t_1 and t_2.[†] Let these two images be T_1 and T_2. Since $T_\rho^* \neq \phi$, T_ρ^* contains, in the set theoretic sense, the two images T_1^*, T_2^* of $T_{m_{\mathbf{R}}}^*$. Since $(\Psi_g(T_{m_{\mathbf{R}}}^*))(\vec{d})=a$ then for some derivation we know $(\psi_g^i(T_{m_{\mathbf{R}}}^*))(\vec{d})=a$. So $(\psi_g^i(T_1^*))(t_1[L])=t_1[B]$ and $(\psi_g^i(T_2^*))(t_2[L])=t_2[B]$. That is to say, $|(\psi_g^i(T_\rho^*))(t_1[L])|>1$ contradicting lemma 2. It must be that ρ is not satisfying. ▪

Theorem 4. For database schema $\mathbf{R}$, F a non-redundant set of fd's, if a dependency $f \in F$ acts as a constraint, then there is a dependency $g \in F^+$ and some derivation tree Dt_i constructed using f, such that $(\psi_g^i(T_{m_{\mathbf{R}}}^*))(\vec{d})=a$.

Proof. If f acts as a constraint, then there exists a witness to that fact, i.e., an element ρ of $SATW(\mathbf{R},F-\{f\})-SATW(\mathbf{R},F)$. Let $\chi(T_\rho)=\phi$. We analyze χ with the goal of finding a derivation tree in which f is used and then show, using lemma 4, that the expression for this tree returns the distinguished symbol on the vector of distinguished symbols.

Let ζ be some proper prefix of χ. Consider two rows r, s of $\zeta(T_\rho)$ such that for some attribute A, $r[A]=s[A]$. The set of transformations $\Gamma(''r[A]=s[A]'')$ is the subset of χ directly responsible for this equality. If the equality holds in T_ρ, then $\Gamma(''r[A]=s[A]'')=\phi$. Otherwise, there is a unique transformation $\nu=<X \to A, \{t,u\}>$ in ζ such that before its execution $r[A] \neq s[A]$ and after its execution $r[A]=s[A]$. This requires, possibly after renaming, that $r[A]=t[A]$ and $s[A]=u[A]$ hold before execution of ν in ζ. Then

$$\Gamma(''r[A]=s[A]'')=\{\nu\} \cup \Gamma(''r[A]=t[A]'') \cup \Gamma(''s[A]=u[A]'')$$

Let $\tau \equiv <Z \to C, \{v,w\}>$ be the last transformation of χ; i.e., τ is a contradiction. Let $v[C]=c_1$, $w[C]=c_2$ just before execution of τ. The sets $\Gamma(''v[C]=c_1'')$,

[†] Compare proposition 5.

$\Gamma(''w[C]=c_2'')$ can be defined. We can now superimpose a directed graph on χ.

The directed, arc-labelled acyclic graph $C_\tau(\chi)$ has a node for each transformation of χ and the nodes $\bot$ and $\top$. The arcs of $C_\tau(\chi)$ are given by

- an arc labelled C is directed from $\top$ to τ and to each transformation in $\Gamma(''v[C]=c_1'') \cup \Gamma(''w[C]=c_2'')$.
- from a transformation $<Y_1 \cdots Y_k \rightarrow B, \{x,y\}>$ in χ, for each $i \in \{1, \ldots, k\}$, an arc labelled Y_i is directed to each element of $\Gamma(''x[Y_i]=y[Y_i]'')$. If this set is empty, then an arc labelled Y_i leads to $\bot$.

The acyclicity of $G_\tau(\chi)$ is apparent: no arc leads to $\top$ nor from $\bot$ and if $<\nu, \xi>$ is an arc then ξ precedes ν in χ.

One can establish by an easy induction that the set of all paths from $\top$ to $\bot$ contains that subset of χ which is necessary to reduce T_ρ to ϕ. Since ρ is a witness for f, one of these transformations must use f.

Now consider any set H of paths from $\top$ to $\bot$ in $G_\tau(\chi)$ which satisfies the following criteria:

i) every path in H begins with the same arc

ii) for every $\eta \equiv <Y_1 \cdots Y_k \rightarrow B, \{x,y\}>$ on some path in H, exactly k of the arcs leaving η, no two labelled the same, are on paths of H.

H corresponds in a natural way to a derivation tree: Let $\overline{H}$ be the graph formed from H by letting the nodes of $\overline{H}$ be the arcs of H, labelled accordingly, and an arc lead from node n_1 to node n_2 in $\overline{H}$ if the node which arc n_1 of H enters is the node from which arc n_2 of H leaves. $\overline{H}$ is the line digraph of H [Ha]. (For $\overline{H}$ to be a tree, node splitting may be necessary in H for those transformations with two incoming arcs.[†] Assume therefore H is a tree as well.) The dependencies used to

[†] To find all the derivation trees in $G_\tau(\chi)$, node splitting should be done before selecting the paths of H. However, here we are searching for only one tree.

form the derivation tree $\overline{H}$ are the dependencies used in the transformations of H. The left hand side of the dependency represented by $\overline{H}$ is the set of attributes labelling arcs incoming to $\bot$; the right hand side is the label on the arc leaving $\top$, C in this case.

As noted above, we may assume H contains a transformation on f; therefore $\overline{H}$ is a derivation tree utilizing f. Let ψ be the expression for $\overline{H}$. It remains to show that $(\psi(T^*_{m_R}))(\bar{d})=a$.

Each subexpression $\pi_A(\sigma_C(I))$ of ψ corresponds to a transformation of H. Call the set of tuples returned by $\sigma_C(I)$ during some evaluation of ψ the selected set for the transformation. Let the height of a transformation in H be the length of the longest path from it to $\bot$. By induction on the height of a transformation, we show that its selected set when ψ is evaluated in $T^*_{m_R}$ at $\bar{d}$ includes the rows of $T^*_{m_R}$ representing the schemas of the rows of T_ρ in the transformation.

The basis references those transformations of H enabled in T_ρ. For $<X \to A, \{r,s\}>$ such a transformation, clearly $X \subseteq R \cap S$ (R the scheme of r; S the scheme of s) so the hypothesis holds.

For the induction, let $\eta \equiv <Y_1 \cdots Y_k \to B, \{r,s\}>$ be at height m. If the Y_i-arc leads from η to $<Z \to Y_i, \{v,w\}>$, then by construction $r[Y_i]=v[Y_i]=w[Y_i]=s[Y_i]$ at the point during the execution of χ at which η appears. Therefore, $R[Y_i]=V[Y_i]=W[Y_i]=S[Y_i]$ in $T^*_{m_R}$ by lemma 4. If it leads to $\bot$, then $R[Y_i]=S[Y_i]=a$ in T_{m_R}. By the induction hypothesis and the expression for η, $\{R, S\}$ is a subset of the selected set for η. So the induction is established.

Now consider the selected set for the root of ψ, the expression for the sole descendant of $\top$ in H. The rows of this transformation are constant in $(\chi-\{\tau\})(T_\rho)$ on the label of the arc leaving $\top$ (C in our example). Therefore they are distinguished on that attribute in the selected set, by lemma 4. This completes the proof. ▪

We may use the techniques of the proofs of theorems 3 and 4 to narrow the gap between them somewhat. If the collection **S** is a subset of a schema **R**, the mapping $m_{\mathbf{S}}$ is a mapping from universal instances for **R** to universal instances for S. The rows of the tableau $T_{m_{\mathbf{S}}}$ form a subset of the rows of $T_{m_{\mathbf{S}}}$. The reader will note that the reasoning of lemma 4 applies to $T_{m_{\mathbf{S}}}$.

Theorem 5. Let **R** be a schema, F a non-redundant set of fd's, $f \in F$. Suppose there exists $\mathbf{S} \subseteq \mathbf{R}$, $g \in F^+$, such that

1) $(\Psi_g(T^{*}_{m_{\mathbf{S}}}))(\vec{d})=a$ and

2) for every derivation tree Dt_i such that $(\psi^i_g(T^{*}_{m_{\mathbf{S}}}))(\vec{d})=a$, f is used in Dt_i,

then f constrains **R** with respect to F.

Proof. Let X be a set of attributes. For each $B \in X^+-X$, say that f is *unnecessary for* $<B,X,\mathbf{S}>$, if there is some derivation tree Dt_i for $X \rightarrow B$ in which f is *not* used and $(\psi^i_{X\rightarrow B}(T^{*}_{m_{\mathbf{S}}}))(\vec{d})=a$. Define the set $\bar{X}$ by

$$\bar{X}=\{A \mid A \in X^+ \text{ and } f \text{ is unnecessary for } <A,X,\mathbf{S}>\}$$

We have $X \subseteq \bar{X} \subseteq X^+$ Form a two tuple relation over the universe which tuples agree exactly on the set $\bar{X}$. Let ρ be the projection onto **S** of these two tuples. We will show that ρ is a witness for f.

We can show that $\rho \notin SATW(\mathbf{R},F)$ in the manner used in the second part of the proof of theorem 3. To show that $\rho \in SATW(\mathbf{R},F-\{f\})$, we will prove that any sequence χ such that $\chi(T_\rho)=\phi$ contains a transformation using f.

Again we exploit the two images T_1, T_2 of $T_{m_{\mathbf{S}}}$ in T_ρ. Let us separate these images and chase each individually. (This may cause some duplication of rows, if some scheme of S is a subset of $\bar{X}$. If this occurs, ensure by renaming that T^*_1, T^*_2 share only values in $\bar{X}$.) Let ζ be any transformation sequence such that $\zeta(T^*_1 \cup T^*_2)=\phi$. By the methods of theorem 4, we construct the graph $G(\zeta)$. As in theorem 4, we can extract from this graph, derivation trees whose expressions are defined in $T^{\bullet}_{m_8}$ at $\bar{d}$ to be a. An attribute labelling the leaf of any such tree is the label on an arc of $C(\zeta)$ leading to $\perp$. The rows of the transformation from which this arc emanates agree on this attribute in $T^{\bullet}_1 \cup T^{\bullet}_2$. We claim that every such attribute is an element of X. We establish this claim by proving that every transformation in ζ involves a row of $T^{\bullet}_1$ and a row of $T^{\bullet}_2$. Such a pair of rows agree only on attributes in X, by construction.

During the execution of ζ, no two rows r, s of $T^{\bullet}_i$ ($i=1$ or 2) become equal on any attribute on which they were not already equal. They may not become equal on any attribute unless the corresponding rows of $T^{\bullet}_{m_8}$ are equal on that attribute, by lemma 4. As $T^{\bullet}_i$ is an isomorph of $T^{\bullet}_{m_8}$, they are equal on that attribute. Therefore no transformation on any two rows of $T^{\bullet}_i$ becomes enabled during the execution of ζ.

We have established our claim that every derivation tree which we may extract from $C(\zeta)$ has its leaf attributes within X. We therefore have expressions for dependencies of the form $Y \to B$ for $Y \subseteq X$ which expressions return a at $\bar{d}$ in $T^{\bullet}_{m_8}$. Clearly $B \in X^{+}-X$. Therefore, each of these expressions must use f. Therefore, f must be used in some transformation of ζ. ▪

We end this section by showing that the property of acting as a constraint is 'cover sensitive'.

Proposition 6. There exist non-redundant sets of dependencies $G \equiv F$ with $f \in F \cap G$ and a schema R such that f constrains R with respect to F but not with respect to G.

Proof. Let R and F be as given in example 8. Let G be F without $X_1X_2 \to Z$ and with $Y_1Y_2 \to Z$. Let f be $Y_1Y_2 \to X_1$. Now $Y_1Y_2 \to Z \in contr_F(Y_1Y_2 \to X_1)$ and $(\Psi_{Y_1Y_2 \to Z}(T^{\bullet}_{m_8}))(aa)=a$. Thus $Y_1Y_2 \to X_1$ acts as a constraint on R with respect to F

by theorem 3.

We claim that no transformation on $Y_1Y_2 \rightarrow X_1$ can be contradicted in any state of **R**. (This fact is insensitive to the choice of cover.) By lemma 4, only two tuples from an instance of Y_1Y_2Z may agree on Y_1Y_2 as only that scheme has Y_1Y_2 in the closure of its set of attributes. No tuple from Y_1Y_2Z may have a constant in X_1, again by lemma 4, so the claim is established.

If $<Y_1Y_2 \rightarrow X_1, \{t_1, t_2\}>$ appears in the graph described in the proof of theorem 4, it must appear in $\Gamma(\text{"}r[X_1]=s[X_1]\text{"})$ for some r, s and by the above reasoning all of r, s, t_1, t_2 come from Y_1Y_2Z and agree on Y_1Y_2. Thus application of $<X_1 \rightarrow Y_1, \{r, s\}>$ has no effect and may be omitted. But $X_1 \rightarrow Y_1$ is the only dependency in G with X_1 on the left. Thus, we have shown directly that $SATW(\mathbf{R}\, G)=SATW(\mathbf{R}, G-\{Y_1Y_2 \rightarrow X_1\})$. •

6. Other Approaches

Other researchers have concerned themselves with the functions represented in a database state for the functional dependencies of the schema. Both Ling and Tompa [LT] and Arora and Carlson [AC] define the function f_I for the single relation case to be $\varphi_f(I)$. They are both concerned with determining if the function represented by a multi-relation state is equivalent to the same function in a single relation. Arora and Carlson consider only join consistent states; that is, their work makes the universal relation instance assumption. Ling and Tompa offer no model of the database as a whole. Both sets of authors give methods of calculating a function from its derivations. Neither method is presented in the relational algebra and it is not apparent that their methods can be converted to such a presentation. Ling and Tompa are particularly concerned that all such calculations produce the same function, something we have shown not to be feasible even when the database consists of a single relation. Arora and Carlson specifically reject the method of calculation presented here after noticing that it results in functions more defined than the table lookup functions we denoted φ. Significantly, they do not apply their method of calculating derived functions to their counter-example (figure 1 in [AC]). It produces the same result as the equivalent ψ-function. This is not surprising, as all sound techniques must produce the same result, if any. Neither set of authors considers the technique of lemmas 1 and 3.

We have argued that restricting the function f_I to be $\varphi_f(I)$ is unnatural. We have demonstrated that a function is not necessarily calculated by its derivations. It is not clear, however, how the result of a function should be interpreted on a value not "present" in the state. If the functions in the state model functions in the world represented by the state, then these functions say something about that world, even at values not recorded in the database. In the single relation case, the user may wish to interpret the absence of a value as denoting the non-existence of some entity, relationship or whatever. Thus if a function is defined at such an absent value, it might be interpreted as stating that should such an entity, etc., come into existence, certain of its attributes are fixed by what is already known. On the other hand, if the database is thought to capture only partial information about the world, statements about existence in that world are less certain. As shown by example 8, in the multi-relation case it is much less certain which values are present or absent. In any case, these aspects of the user's interpretation are not captured by the theoretical model underlying this paper. That model attempts to derive statements which are true for any interpretation in which the dependencies are true.

The fact that distinct derivations of a given dependency may be nonequivalent when interpreted as relational expressions has long been known. The "uniqueness assumption" [B] requires that all such derivations calculate the same function. In the single relation case, proposition 1 verifies the uniqueness assumption at those values present in the instance. Some authors, in particu'ar Sciore [Sc], claim that the presence of nonequivalent derivation expressions indicates the "semantic overloading" of some attributes. It is questionable whether such an easy transition between user semantics and the syntax of dependencies is justifiable. Perhaps the user should be given the freedom to require nonequivalent derivations to be constrained to be equal while allowing provably equivalent derivations to disagree. This latter freedom requires "attribute renaming".

As a consequence of theorems 2 and 3, a schema designed by either a synthetic [BDB] or decompositional [F1] algorithm represents and is constrained by all the functional dependencies given. One may wonder whether any practical benefit is to be gained by closing the gap between theorems 4 and 5. Although these algorithms guarantee nice theoretical properties, it is not certain that they guarantee "good" designs in practice. Dependencies do not capture all the semantics inherent in the user's interpretation and they completely ignore performance considerations. There is much more to schema design than these algorithms capture. It is perhaps more useful to consider theoretical results such as these as providing schema analysis rather than design. Tsichritzis and Lochovsky [TL] present a fuller account of theoretical issues in this light. It seems that it is important to consider arbitrary designs, even though the class of practically useful designs is likely to be small, since that class has yet to be identified.

7. Summary

We have investigated the interrelationship of a schema, considered as a collection of subsets of the universe, and a set of functional dependencies. We have studied two properties of this interrelationship. A functional dependency may be interpreted as the description of a function. We have given the conditions under which a given schema represents a given dependency as a function and when it represents all of a given set of dependencies as functions. Interestingly, this last property is enjoyed by exactly the class of dependency preserving schemas as defined by [BMSU]. This class is strictly larger than the class of cover embedding schemas, which class has heretofore been considered the largest class representing all of a given set of dependencies.

We have studied the calculations of the functions described by the dependencies. We considered the derivation of a dependency as a blueprint for the construction of a relational algebra expression. This is in keeping with the description of Armstrong's rules given by Bernstein [B]. The expression produced in this way does not, we discovered, always calculate the function. In the single relation case, if the function is defined at a given value, then at that value it agrees with the collection of its derivation expressions. However, if the function is undefined due to the non-existence of the requisite weak instance, this may not be noticed by the derivation expressions. In the multi-relation case, we have shown by example that the derivation expressions may fail to return a result at a value at which the function is defined. In general, therefore, we have shown that the method of derivation expressions is incomparable to the method of the chase.

We noted that under certain circumstances a set of dependencies may allow for infinitely many derivations. This can be ignored when the existence of *any* derivation of a given dependency is being tested, as in [B]. However, we have shown that it is possible for each of an infinite set of derivations of a given dependency to correspond to a different mapping from database states to functions.

Functional dependencies are also meant to act as constraints on the states of a schema. Although we have not fully characterized this phenomenon, we have shown necessary and sufficient conditions for a given dependency to act as a constraint with respect to a schema and set of functional dependencies. In particular, a dependency may act as a constraint even though the function it describes is empty in every state. Before these investigations, the distinct properties of being represented as a function and acting as a contraint on states which a dependency may enjoy with respect to a schema had been confused by other researchers, as we have shown.

Acknowledgements. I am grateful to A. O. Mendelzon for the definition of constraint and many hours of conversation. I thank the referees for their comments. I particularly thank the first referee, who suggested example 5 and proposition 5.

References

[A] Armstrong,W.W.;"Dependency Structures of Data Base Relationships," *Proc. of IFIP 1974*, North-Holland, 580-583

[AC] Arora,A.K.;Carlson,C.R.;"The Information Preserving Properties of Relational Database Transformations," *Proc. of VLDB 1978* 352-359

[ASU] Aho,A.V.;Sagiv,Y.;Ullman,J.;"Equivalence Among Relational Expressions," *SIAM J. Computing 8,2 (1979)* 218-246

[B] Bernstein,P;"Synthesizing Third Normal Form Relations from Functional Dependencies," *TODS 1,4 (1976)* 277-298

[BBG] Beeri,C;Bernstein,P;Goodman,N;"A Sophisticates Introduction to Database Normalization Theory," *Proc. of VLDB 1978* 113-124

[BDB] Biskup,J;Dayal,U;Bernstein,P; "Synthesizing Independent Database Schemas," *SIGMOD 79* 143-151

[BG] Bernstein,P;Goodman,N;"What Does Boyce-Codd Normal Form Do?" *Proc. of VLDB 80* 245-259

[BMSU] Beeri,C;Mendelzon,A;Sagiv,Y;Ullman,J;"Equivalence of Relational Database Schemes," *TR-252 (1978)* Princeton University

[C] Codd,E.F.;"Further Normalization of the Data Base Relational Model," *Data Base Systems, R. Rustin, ed.;* Prentice-Hall (1972), 33-64

[DST] Downey,P.J.;Sethi,R.;Trajan,R.E.;"Variations on the Common Subexpression Problem," *JACM 27,4* (Oct. 19870), pp. 758-772

[F1] Fagin,R; "The decomposition versus synthetic approach to relational database design," *Proc of VLDB 77* 441-446

[F2] Fagin,R; "A Normal Form for Relational Databases that is Based on Domains and Keys," *IBM RJ2520* (1980)

[G] Graham,M.H.;"Satisfying Database States," Ph.D. Thesis, University of Toronto, (1981)

[H] Honeyman,P;"Testing Satisfaction of Funtional Dependencies," *to appear, JACM*

[Ha] Harary,F;*Graph Theory*, Addison-Wesley, (1969)

[HLY] Honeyman,P;Ladner,R.;Yannakakis,M;"Testing the Universal Instance Assumption," *Information Processing Letters 10:1 (1980)* 14-19

[LT] Ling,T;Tompa,F; "Implicit Constraints Within Relational Data Bases," DCS, University of Waterloo, *CS-78-35* 1978

[MMS] Maier,D.;Mendelzon,A.;Sagiv,Y.;"Testing Implications of Data Dependencies," *TODS 4,4 (1979)* 455-469

[Sc] Sciore,E.;"Improving Semantic Specification in a Relational Database," *SIGMOD 79* 70-78

[TL] Tsichritzis,D.;Lochovsky,F.;*Data Models*, Prentice-Hall, (1982)

[U] Ullman,J;*Principles of Database Systems*, Computer Science Press, (1980)

[V] Vassiliou,Y.;"A Formal Treatment of Imperfect Information in Database Management," Ph.D. Thesis, University of Toronto, (1980)