Transactions in Relational Databases

(Preliminary Report)

Serge Abiteboul
Institut National de Recherche en Informatique et en Automatique
Domaine de Voluceau-Rocquencourt, B.P. 105, 78153
Le Chesnay Cedex, FRANCE

Victor Vianu(*)
Department of Electrical Engineering and Computer Sciences,
University of California, San Diego, MC C014
La Jolla, California 92093, USA

ABSTRACT

A large class of relational database update transactions is investigated with respect to equivalence and optimization. Several basic results are obtained. It is shown that transaction equivalence can be decided in polynomial time. A number of optimality criteria for transactions are then proposed, as well as two normal forms. Polynomial algorithms for transaction optimization and normalization are exhibited. Also, an intuitively appealing system of axioms for proving transaction equivalence is introduced. Finally, a simple, natural subclass of transactions, called 2-acyclic, is shown to have particularly desirable properties.

1. Introduction.

Static aspects of databases have been extensively studied using the formal framework of the relational model [C, M, Ul]. More recently, some dynamic aspects have been considered in several investigations [AH, BS, BG, Br, CCF, CW, DeAZ, FUV, HK, PBR, R, Uh, Ve, Vi1, Vi2]. However, there have been very few theoretical studies of database updates and transactions. Indeed, most previous investigations of transactions have focused on concurrency issues [BG, PBR]. In the present paper, we introduce a formal model of transactions in relational databases and present several basic results on transaction equivalence and optimization.

Transactions are viewed here as sequences of elementary operations forming a semantic unit. In this paper, we focus on a widely accepted class of transactions. Specifically, these transactions consist of sequences of insertions, deletions and updates, where the selection of tuples (to be deleted or updated) involves the inspection of individual attribute values for each tuple. Most of our results concern transaction equivalence and optimization. Indeed, our investigation can be regarded as the analogue for updates of fundamental investigations on query equivalence and optimization.

With respect to equivalence, two techniques for proving transaction equivalence are exhibited. The first one is based on a graphical, non-procedural representation of a transaction, and leads to a polynomial time algorithm for deciding transaction equivalence. The second one is based on a system of axioms, and highlights the interaction between insertions, updates and deletions.

With regards to transaction optimization, several optimality criteria are discussed and formalized (for instance, one criteria is the length of the transaction). It is shown that each given transaction can be optimized with respect to all the proposed criteria. A polynomial time optimization algorithm is then presented.

The paper is divided into six sections. The second section contains preliminary concepts. In Section 3 the formal model for our transactions is presented. A graphical, non-procedural way of representing the effect of transactions is introduced in Section 4. This is then used as an independent measure for the power of our transactions, and for showing that transaction equivalence is decidable. Section 5 is devoted to transaction optimization. Finally, in Section 6 the system of axioms for proving transaction equivalence is introduced.

Due to space limitations, some results and definitions are presented informally and others are omitted. (In particular, no proofs are included.)

2. Preliminaries.

In this section we briefly present some well-known concepts used throughout the paper.

We first present some basic concepts of relational databases. We assume the existence of an infinite set of symbols, called *attributes*, and for each attribute A, of an infinite set of *values*, denoted dom(A), called the domain of A. A *relational schema* is a finite set of attributes. Let U be a relational schema. A *tuple* t *over* U is a mapping from U such that for each A in U, t(A) is in dom(A). The set of tuples over U is denoted Tup(U). A *relation* over U is a finite set of tuples over U.

A *database schema* is a pair (U,S) where U is a finite set of attributes, and S a set of subsets of U such that $U = \cup\{X \mid X \text{ in } S\}$. An *instance* I of a database schema (U,S) is a mapping from S such that I(X) is a relation over X for each X in S. The set of all the instances of a database schema (U,S) is denoted Inst(U,S).

We now present some other concepts and notation used in the paper. A (*directed*) *graph* is a pair (V,E) where V is a finite set of elements, and E is a subset of V × V. An element in V is called a *vertex*. An element in E is called an *edge*. Let

(*) This work was partly performed while author was a Visiting Scientist at I.N.R.I.A.

G = (V,E) be a graph and p a vertex of G. The *in-degree* of p, denoted id(p), is the number of edges incident *into* p. The *out-degree* of p, denoted od(p), is the number of edges incident *from* p.

Let V be a set of elements. Let E and E′ be subsets of V × V. The *product* of E and E′, denoted $E \circ E'$, is defined by $E \circ E' = \{<x,z> \mid \text{for some } y, <x,y> \text{ in } E \text{ and } <y,z> \text{ in } E'\}$.

Finally, $\vdash_A B$ means that B can be proven using the set of axioms A.

3. A model for transactions.

Informally, a transaction is a sequence of instructions viewed as a semantic unit. Most commercial database management systems provide three types of atomic instructions which are used to build up transactions.

The three types of atomic instructions are:

(1) insertions	(appending to a relation a tuple or a set of tuples),
(2) deletions	(suppressing from a relation all tuples satisfying a given condition), and
(3) updates	(modifying in a relation all tuples satisfying a given condition).

In some dbms's the conditions used to select from a relation the set of tuples to be deleted or updated can be quite complex. Generally, these conditions may use the full power of tuple relational calculus. As a consequence, many basic questions about transactions (such as when two transactions have the same effect) are undecidable in an unrestricted framework. In this investigation, we focus on a tractable and widely used class of transactions. Specifically, we consider the important class of "domain-based" transactions, where the selection of tuples only involves the inspection of each individual attribute value of a tuple, independently of other attribute values in the tuple and of other tuples in the instance.

The following is a simple example of a domain-based transaction in INGRES [INGRES].

Example. Suppose the relation emp (employee) has been defined (its attributes are *name*, *depart*, and *rank*). The following transaction fires the manager of the parts department, transfers the manager of the sales department to the parts department, and hires a new manager for the sales department:

```
range of e is emp
delete e where e.depart = "parts"
          and e.rank = "manager"
replace e(depart = "parts") where
     e.depart = "sales"
replace e(depart = "parts")
     and e.rank = "manager"
append to emp (name = "Moe", depart = "sales",
          rank = "manager").
```

We now formally define the notions of a "condition" and satisfaction of a condition by a tuple.

Definition. Let U be a set of attributes. A *condition* over U is an expression of the form A=a or A≠a, where $A \in U$ and $a \in dom(A)$. A tuple u over U *satisfies* a condition A=a (A≠a) iff u(A) = a (u(A) ≠ a). The fact that a tuple u satisfies condition c is denoted by $u \models c$.

We will not explicitly use logical connectors to build up complex conditions (it can be easily seen that this would not add power to our transactions). However, we will define satisfaction of a *set* of conditions, which is analogous to satisfaction of the conjunction of all conditions in the set.

A set of conditions with no mutually exclusive conditions is called "meaningful." We next define formally meaningful sets of conditions. First though, we need the following:

Definition. Let C be a set of conditions over U, and $X \subseteq U$. The *restriction* of C to X is the set $C|_X = \{c \in C \mid c \text{ is a condition over } X\}$.

We now have:

Definition. A set C of conditions over U is *meaningful* if for each A in U, $A=a \in C$ implies $C|_A=\{A=a\}$.

Thus, the sets of conditions {A=5, A≠5} and {A=5, A=6} are not meaningful. The set of conditions {A= 5, B≠7} is meaningful.

A tuple u *satisfies* a meaningful set C of conditions (denoted $t \models C$) if $t \models c$ for each c in C. Note that a set of conditions is meaningful if and only if it is satisfied by some tuple. All sets of conditions considered from here on will be meaningful, unless otherwise specified.

A set of conditions over U is used to specify a set of tuples over U (those satisfying the conditions). Due to the form of our conditions, we use the intuitively suggestive term "hyperplane" to identify such sets of tuples:

Definition. The *hyperplane* H(U,C) defined by a (meaningful) set C of conditions over U is the set $\{t \in Tup(U) \mid t \models C\}$.

As mentioned earlier, $H(U,C) \neq \emptyset$ if C is meaningful. Also, $H(U,C_1) = H(U,C_2)$ implies that $C_1 = C_2$. For simplicity, we sometimes use the same notation for a set C of conditions over U and for the hyperplane H(U,C) defined by C. Thus, we say "hyperplane C" instead of "hyperplane H(U,C)", whenever U is understood.

We shall next define the atomic instructions used to build our transactions. These will be called "elementary transactions." The syntax and semantics of elementary transactions are defined next. The semantics of an elementary transaction is described by a mapping associating the old instances and new instances. Such mappings are called "actions". Formally, we have:

Definition. An *action* over a schema (U,S) is a mapping from Inst(U,S) to Inst(U,S).

The syntax and semantics of elementary transactions are now defined as follows:

Definition. Let (U,S) be a schema.

1) An *insertion* over the schema (U,S) is an expression of the form $i_X(C)$, where $X \in S$ and C is a condition which specifies a complete tuple(*) H(X,C) over X. The *effect* of an insertion $i_X(C)$ is the action $eff[i_X(C)] : Inst(U,S) \rightarrow Inst(U,S)$ defined by

$$eff[i_X(C)]\ (I)\ (Z) = \begin{cases} I(Z) & \text{if } Z \neq X \\ I(X) \cup H(X,C) & \text{if } Z = X . \end{cases}$$

In the following we sometimes write $i_X(<a_1,...,a_n>)$ instead of $i_X(\{A_1= a_1,...,A_n= a_n\})$.

(*) If $X = A_1...A_n$, then $C = \{A_1 = a_1, ..., A_n = a_n\}$ for some $a_i \in dom(A_i)$, $1 \leqslant i \leqslant n$.

2) A *deletion* over the schema (U,S) is an expression of the form $d_X(C)$ where $X \in S$ and C is a set of conditions. The effect of $d_X(C)$ is the action $\text{eff}\,[d_X(C)] : \text{Inst}(U,S) \to \text{Inst}(U,S)$ defined by

$$\text{eff}\,[d_X(C)]\,(I)\,(Z) = \begin{cases} I(Z) & \text{if } Z \neq X \\ I(X) - H(X,C) & \text{if } Z = X\,. \end{cases}$$

3) An *update* over the schema (U,S) is an expression $u_X\,(C_1;\,C_2)$ where $X \in S$ and for each $A \in X$ either $C_1|_A = C_2|_A$ or $A{=}a \in C_2$ for some $a \in dom(A)$. For each tuple t_1 in $H(X,C_1)$, the updated version $u_X(C_1;C_2)\,(t_1)$ of t_1 under $u_X(C_1;C_2)$ is the tuple $t_2 \in H(X,C_2)$ where

$$t_2(A) = \begin{cases} t_1(A) & \text{if } C_1|_A = C_2|_A \\ a & \text{if } A{=}a \text{ is in } C_2\,. \end{cases}$$

The effect of $u_X(C_1;C_2)$ is the action $\text{eff}\,[u_X(C_1;C_2)] : \text{Inst}(U,S) \to \text{Inst}(U,S)$ defined by

$$\text{eff}\,[u_X(C_1;C_2)]\,(I)\,(Z) = \begin{cases} I(Z) \;\; \text{if } Z \neq X \\ (I(X) - H(X,C_1)) \cup \{u_X(C_1;C_2)\,(t) \\ \qquad t \in H(X,C_1) \cap I(X)\} \text{ if } Z = X. \end{cases}$$

An *elementary transaction* over the schema (U,S) is an insertion, a deletion, or an update over (U,S). □

Informally, an update $u_X(C_1;C_2)$ is specified using two sets of conditions. The set C_1 is used to specify the tuples over X to be updated. The set C_2 describes the hyperplane obtained after applying the update to the hyperplane C_1. The equalities present in C_2 but not in C_1 indicate how tuples in $H(X,C_1)$ have to be modified. (All inequalities present in C_2 are "inherited" from C_1. Thus, if $A{\neq}a$ is in C_2, it is also in C_1 and the value of A remains unchanged in the update.)

Example. Consider a database consisting of a single relation EMPLOYEE with attributes U = {NAME, DEPARTMENT, RANK, SALARY}. The following are elementary transactions over U:

1) i_U (<MOE, PARTS, MANAGER, 30K>),
2) d_U (NAME ≠ MOE, DEPARTMENT = PARTS, RANK = MANAGER)
(this deletes all managers in the parts department whose names are not Moe),
3) u_U (DEPARTMENT = PARTS, RANK ≠ MANAGER; DEPARTMENT = SERVICE, RANK ≠ MANAGER, SALARY = 20K)

This transfers all employees who are not managers from the parts department to the service department. The rank remains unchanged. The new salary is 20K.

In the following we sometimes omit the subscripts in writing elementary transactions. For instance, we write i(C) instead of $i_x(C)$, if X is understood.

We can now formally define the notion of transaction and its effect on the database:

Definition. Let (U,S) be a database schema. A *transaction* over (U,S) is a finite sequence of elementary transactions over (U,S). (The empty transaction is denoted by ϵ.) The *effect* of a transaction $t = e_1 \ldots e_n$ $(n > 0)$ is the action $\textit{eff}(t) = \textit{eff}(e_1) \circ \ldots \circ \text{eff}(e_n)$. (The effect of ϵ is the identity mapping.)

Thus, the effect of a transaction is the composition of the effects of the elementary transactions that make it up. Two transactions are equivalent if they have the same effect:

Definition. Two transactions t_1 and t_2 over a given schema are *equivalent*, denoted $t_1 \approx t_2$, iff $\text{eff}(t_1) = \text{eff}(t_2)$.

Our first result (Proposition 3.1) indicates that in the present context only unirelational schemas need to be considered. First we need the following:

Notation. Let t be a transaction over a schema (U,S). For each $X \in S$, let $t|_X$ be the transaction over (X,{X}) obtained by erasing from t all elementary transactions which are not over (X,{X}).

3.1. ***Proposition.*** Let t_1 and t_2 be transactions over a schema (U,S). Then $t_1 \approx t_2$ if $t_1|_X \approx t_2|_X$ for each $X \in S$.

In view of the above, we only consider transactions over unirelational schemas from here on.

4. Transitions and Realizability.

In this section we introduce a non-procedural method for describing the effect of a transaction on a database. The effect is described at the tuple level using the notion of a "transition." Transitions can be specified in an intuitively appealing manner and are useful in several respects. First, they will be used to measure the power of our transactions. Second, transition specifications will be used to study the equivalence and optimization of transactions. Intuitively, the relation between transactions and transition specifications is somewhat similar to that between relational algebra expressions and tuple calculus expressions. (Indeed, transition specifications offer a non-procedural, graphical alternative to transactions which may be more appealing to certain users.)

Informally, a transition describes at the tuple level a change in the database state. For each tuple, a transition indicates whether the tuple is deleted or, if not, how it is updated. In addition, a transition gives a finite set of inserted tuples. Since there are an infinite number of tuples to be considered, only certain transitions can be effectively specified. Here, we only need to consider transitions which can be specified using our conditions. Thus, a transition will be specified by first partitioning the space of tuples into sufficiently many hyperplanes. The choice of hyperplanes will ensure that all tuples in each hyperplane of the partition are either deleted or updated to yield another hyperplane in the partition. This is specified using a "transition graph" whose vertices are the hyperplanes in the partition. If H_1 is updated to H_2, there is an edge from H_1 to H_2. If H_1 is deleted there is no edge leaving H_1.

The set of inserted tuples cannot be conveniently specified using the graph, and is given separately.

We now formally define a transition specification.

Definition. Let U be a finite set of attributes. A *transition specification* over U is a couple (G,Insert) where Insert is a finite set of tuples over U and G is a graph (V_G,E_G), where:

(i) V_G is a finite set of disjoint hyperplanes over U such that $\bigcup_{C \in V_G} C = \mathrm{Tup}(U)$,

(ii) if $(C_1,C_2) \in E_G$ then for each A in U either $C_1|_A = C_2|_A$ or $A=a \in C_2$ for some $a \in \mathrm{dom}(A)$,

(iii) for each C in V_G, $\mathrm{od}(C) \leqslant 1$.

(iv) Insert $\subseteq V_G$.

The graph G is called the ***transition graph*** of the transition specification. Insert is called the ***insert set***.

Note that condition (ii) implies that C_1 can be updated to C_2 (or, in other words, that $u(C_1;C_2)$ is a legal update). Condition (iii) follows from the assumption that the result of updating a hyperplane is a single hyperplane in V_G. If $\mathrm{od}(C) = 0$ then all tuples in C are deleted.

We now give a simple example of a transition specification, followed by a more complex one.

4.1. *Examples.* a) Let U = AB and G be the transition graph represented below (Figure 4.1)

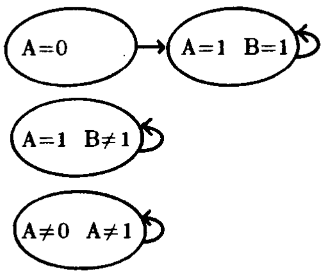


Figure 4.1

Let Insert = {{A=1,B=1}}. Then (G,Insert) is a transition specification over AB. The transition specified by (G,Insert) consists of replacing all tuples t where t(A) = 0 by the tuple <1,1>. All other tuples remain unchanged. The tuple <1,1> is inserted.

b) Let U = AB, Insert = ∅, and G be the transition graph represented(*) in Figure 4.2. Then (G,Insert) is a transition specification.

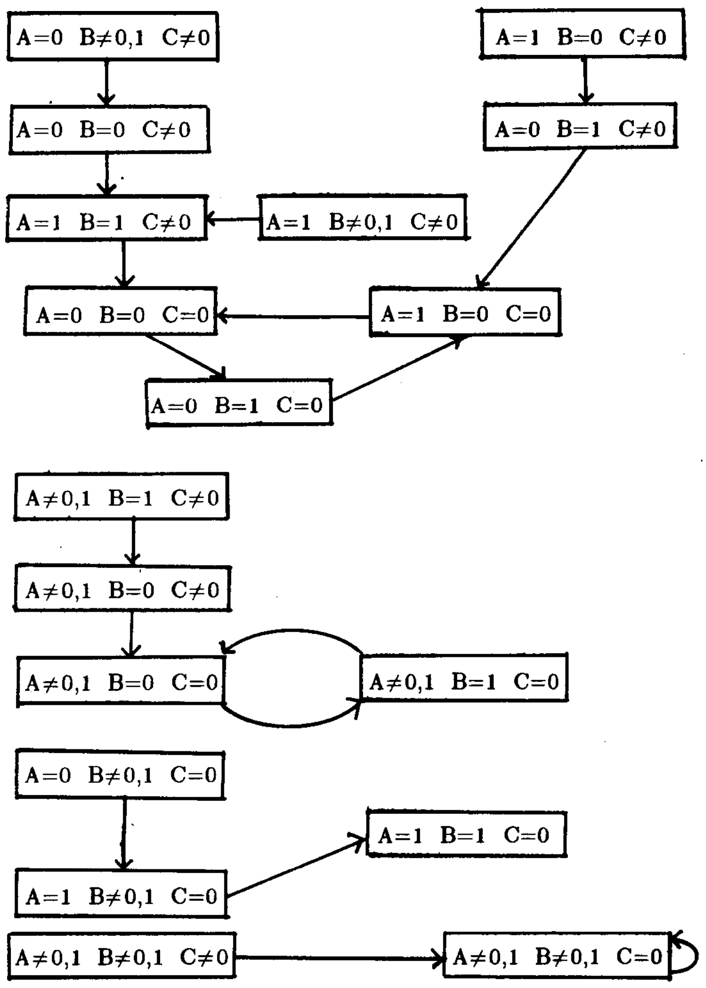


Figure 4.2

We now look at the connection between transition specifications and actions. Given a transition specification and a database state, one can obtain a new state by applying the transition to each tuple in the database. Therefore, each transition specification generates an action in a natural manner. However, transition specifications are more "refined" than actions, since they are tuple oriented rather than global. Indeed, two different transition specifications can generate the same action. This is illustrated by the following:

Example. Let U = AB and G be the transition graph represented below (Figure 4.3).

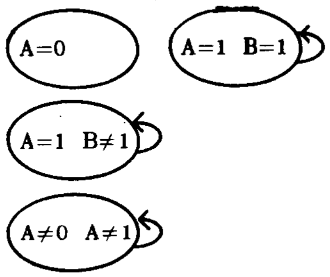


Figure 4.3

Let Insert = {{A=1, B=1}}. It is easily seen that the transition specification (G,Insert) and that of Example 4.1(a) generate the same action. (Since <1,1> is inserted, the resulting action is the

(*) We abbreviate the conditions $A \neq a_1$, $A \neq a_2$, ..., $A \neq a_n$ by $A \neq a_1,a_2,\ldots,a_n$.

same whether the hyperplane $\{A=0\}$ is deleted or updated to $<1,1>$.)

Following is a characterization of when two transition specifications generate the same action.

4.2. ***Proposition.*** Two transition specifications $(G_1, Insert_1)$ and $(G_2, Insert_2)$ over a given set of hyperplanes[*] generate the same action iff the following conditions hold:

(i) $Insert_1 = Insert_2$

(ii) For each $B, C \in V_{G_i}$, and $C \notin Insert_i$ $(i = 1,2)$, $(B,C) \in E_{G_1}$ iff $(B,C) \in E_{G_2}$.

Intuitively, Proposition 4.2 shows that two distinct transition specifications can generate the same action only when their differences are "masked" by the insert set. In particular, we have:

4.3. ***Corollary.*** Two transition specifications $(G_1, \emptyset)$ and $(G_2, \emptyset)$ over the same set of hyperplanes generate the same action iff $G_1 = G_2$.

We next investigate the relation between transactions and transitions. We will show that for each transaction there exists a corresponding transition which represents the final effect of the transaction. Then we show that one can specify transitions which cannot be realized by any transaction. Finally, we characterize those transitions which are realizable. Intuitively, the set of realizable transitions is a measure of the power of our transactions.

An algorithm is next presented for constructing the transition specification corresponding to a given transaction. First, however, it is necessary to perform some "preprocessing" of the transaction. Specifically, the transaction is modified so that all hyperplanes corresponding to distinct sets of conditions occurring in the transaction are disjoint. A transaction having this property is said to be in First Normal Form (1NF). The 1NF property simplifies considerably our algorithm as well as other results. We next define 1NF and show how to construct, for each transaction, an equivalent 1NF transaction. First though, we need the following:

Notation. Let t be a transaction over U. For each A in U, let the *active domain of* A *with respect to* t be the set adom(A,t) of all constants in dom(A) occurring in t. We now associate with t a partition $\underline{H(t)}$ of Tup(U) into hyperplanes as follows.[**] Let

$$H(t) = \{ \bigcup_{A \in U} C_A \;:\; \text{for each } A \in U,$$
$$C_A = \{A=a\},\ a \in adom(A,t), \text{ or}$$
$$C_A = \{A \neq a \;:\; a \in adom(A,t)\}\} .$$

Clearly, H(t) covers Tup(U) and every two distinct hyperplanes in H(t) are disjoint.

With the above notation, we have

Definition. A transaction t over U is in *First Normal Form* (1NF) iff every set of conditions occurring in t is in H(t).

If a transaction t is in 1NF, then every two hyperplanes corresponding to distinct sets of conditions occurring in t are disjoint. We require the first normal form property, rather than simply the disjointness condition, since this will simplify the statements of some results.

The following illustrates the definition of 1NF:

Example. Consider the transaction

$$t_1 = d(A=0)\ u(A \neq 7;\ A=5).$$

Then $H(t_1) = \{\{A=0\}\ \{A=7\}\ \{A=5\}\ \{A \neq 0,5,7\}\}$. Since $\{A \neq 7\}$ is not in $H(t_1)$, t_1 is not in 1NF. Consider next the transaction

$$t_2 = d(A=0)\ u(A=0;\ A=5)\ u(A \neq 0,5,7;\ A=5).$$

Then $H(t_2) = H(t_1)$, t_2 is equivalent to t_1, and t_2 is in 1NF.

Each transaction can be transformed into an equivalent 1NF transaction by "splitting" each hyperplane occurring in it into sufficiently small hyperplanes. For instance, consider transaction t_1 and t_2 from the previous example. One can obtain t_2 from t_1 by splitting the hyperplane $\{A \neq 7\}$ into $\{A=0\}$, $\{A=5\}$, $\{A \neq 0,5,7\}$. The update $u(A \neq 7;\ A=5)$ is split accordingly and becomes $u(A=0;\ A=5)\ u(A=5;\ A=5)\ u(A \neq 0,5,7;\ A=5)$ (the second update leaves the hyperplane $\{A=5\}$ unchanged and can be ignored). Thus, t_2 is obtained. We next show how each transaction can be transformed into an equivalent 1NF transaction using two simple transformation rules, called "SPLIT" axioms.

Definition. Let U be a set of attributes. The following two rules are the *SPLIT axioms* for transactions over U, where $A \in U$, $a \in dom(A)$, and C is a condition over U such that $A \neq a \notin C$ and $A=b \notin C$ for all b:

SPLIT1.

$$d(C) \approx d(C \cup \{A \neq a\})\ d(C|_{U-A} \cup \{A=a\}).$$

SPLIT2.

$u(C;C') \approx u(C \cup \{A \neq a\};\ C_1)$
$u(C|_{U-A} \cup \{A=a\}; C_2)$, where $C_1 = C_2 = C'$ if $A=b \in C'$ for some b, and $C_1 = C' \cup \{A \neq a\}$, $C_2 = C'|_{U-A} \cup \{A=a\}$ otherwise.

Intuitively, hyperplane H(U,C) is split into the hyperplanes $H(U,C) \cap H(U,\{A=a\})$ and $H(U,C) \cap H(U,\{A \neq a\})$. The update and deletion operations are then applied to the resulting hyperplanes. (Note that all resulting sets of conditions are meaningful.)

It can be easily seen that the SPLIT axioms are sound. Furthermore, they can be used to bring any transaction to First Normal Form. Formally, we have:

4.4. ***Theorem.*** For any transaction t there exists an equivalent 1NF transaction t′, such that $\vdash_{SPLIT} t \approx t'$.

We are now ready to outline the algorithm to construct from a given transaction t a corresponding transition specification TS(t) = (G,Insert). Let $t = e_1 \ldots e_n$, $n \geq 1$, be a 1NF transaction over U, where each e_i $(1 \leq i \leq n)$ is an elementary transaction. Let Insert $= t(\emptyset)$. We next define the transition graph $G = (V_G, E_G)$. Let $V_G = H(t)$. It is left to construct E_G. For each elementary transaction e occurring in t, let E(e) be the set of edges defined as follows: If $e = d_U(C)$, let

[*] (G,Insert), where $G = (V_G, E_G)$, is over the set of hyperplanes V_G.

[**] Here again we blur the distinction between a set of conditions and the hyperplane corresponding to it.

$E(e) = \{<H,H> \mid H \in H(t), H \neq C\}$. If $e = u_U(C_1;C_2)$, let $E(e) = \{<H,H> \mid H \in H(t), H \neq C_1\} \cup \{<C_1,C_2>\}$. Finally, if If $e = i_U(C)$, let $E(e) = \{<H,H> \mid H \in H(t)\}$. Now let $E_G = E(e_1) \circ \dots \circ E(e_n)$. The transaction specification $TS(t) = (G,\text{Insert})$ is now completely defined.(*) The transaction t and the transition specification TS(t) define the same transition and therefore the same action.

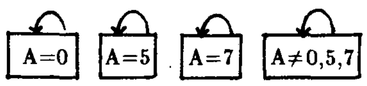
Example. Consider the following transaction over A :

$t = i(A=5)\ u(A=0;\ A=5)$

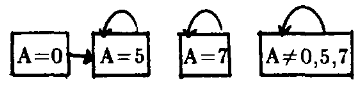
$u(A=5;\ A=7)\ u(A\neq 0,5,7;\ A=5)\ d(A=5)$.

Note first that t is in 1NF, and $H(t) = \{\{A=0\}\ \{A=5\}\ \{A=7\}\ \{A\neq 0,5,7\}\}$. We now construct $TS(t) = (G,\text{Insert})$, where $G = (V_G,E_G)$. Now $\text{Insert} = t(\emptyset) = \{<7>\}$. The set V_G is H(t), and $E_G = E_1 \circ E_2 \circ E_3 \circ E_4 \circ E_5$, where:

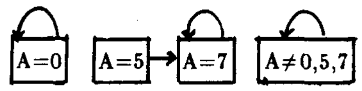
(i) $E_1 = E(i(A=5))$ is represented by:



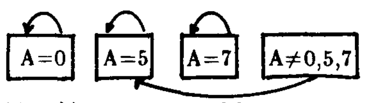
(ii) $E_2 = E(u(A=0;\ A=5))$ is represented by:



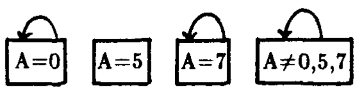
(iii) $E_3 = E(u(A=5;\ A=7))$ is represented by:



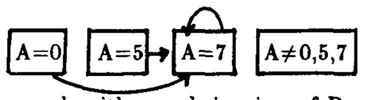
(iv) $E_4 = E(u(A\neq 0,5,7;\ A=5))$ is represented by:



(v) $E_5 = E(d(A=5))$ is represented by:



Finally, $E_G = E_1 \circ E_2 \circ E_3 \circ E_4 \circ E_5$ is represented by:



With the above algorithm and, in view of Proposition 4.2, we now have:

4.5. ***Theorem.*** It is decidable whether two transactions are equivalent. □

It can be shown that transaction equivalence can be decided in polynomial time (in the size of the transactions).

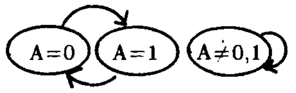
We have shown earlier how to obtain from each transaction t a transition specification TS(t) which defines the same transition as t. Consider now the converse. If the action defined by the transition specification (G,Insert) can be implemented by some transaction, then (G,Insert) is called "realizable." Thus, we have:

(*) Note that TS(t) is undefined for $t = \epsilon$.

Definition. A transition specification (G,Insert) is *realizable* if there exists a transaction defining the same action as (G,Insert).

As shown by the next example, not all transition specifications are realizable.

Example. Let (G,Insert) be the transition specification over A, where $\text{Insert} = \emptyset$ and G is represented by:



It can be shown that (G,Insert) is not realizable.

It is useful to distinguish between realizable transition specifications and those that can be obtained directly from some transaction via the algorithm TS. This motivates the following.

Definition. A transition specification (G,Insert) is *directly realizable* if $(G,\text{Insert}) = TS(t)$ for some 1NF transaction t.

Some transaction specifications are realizable without being directly realizable. For instance, it can be shown that the transition specification $(G_1,\emptyset)$, where G_1 is represented in Figure 4.4 is realizable but not directly realizable.

Intuitively, the set of realizable transition specifications constitutes a measure for the power of our transactions. The main results of the section characterize realizable and directly realizable transition specifications. Before presenting them, we need two definitions, a lemma, and some notation.

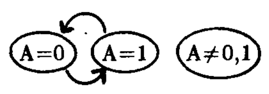


Figure 4.4

Definition. Let C be a set of conditions over U. The *support* of C is the set $\text{Supp}(C) = \{A \in U \mid A = a \in C \text{ for some } a \in \text{dom}(A)\}$.

It is easy to verify the following:

4.6. ***Lemma.*** *Let (G,Insert) be a transition specification over U. If C_1 and C_2 are nodes of G which belong to the same cycle of G, then $\text{Supp}(C_1) = \text{Supp}(C_2)$.*

In view of Lemma 4.6, we can extend the definition of support to a cycle:

Definition. Let (G,Insert) be a transition specification and c a cycle of G. Then the support of c is $\text{Supp}(c) = \text{Supp}(C)$ for some node C belonging to c. (Supp(c) is well-defined, by the above lemma.)

Notation. For each transition specification (G,Insert), let $\text{2-Cycles}(G) = \{c \mid c \text{ is a cycle of G of length at least } 2\}$.

With the above, we now have the following characterization of directly realizable transition specifications.

4.7. ***Theorem.*** A transition specification (G,Insert) is directly realizable iff for each cycle $c \in \text{2-Cycles}(G)$ there exists a vertex v(c) of G which does not belong to any cycle of G, such that $\text{Supp}(c) = \text{Supp}(v(c))$.

As we have seen earlier, a transition specification (G,Insert) can be realizable without being directly realizable. In such a case, however, (G,Insert) can be easily transformed into an equivalent, directly realizable transition specification. There are two types of transformations involved. The first consists of "splitting" G, i.e. splitting some of the vertices of G into smaller hyperplanes (the new edges are those induced by the old edges, by rules analogous to the SPLIT rules). For instance, consider again the transition specification (G_1,∅) (Figure 4.4) which is realizable but not directly realizable. The transition specification (G_2,∅) (Figure 4.5) is directly realizable and G_2 is obtained from G_1 by splitting the vertex {A≠0,1} into {A=2} and {A≠0,1,2}.

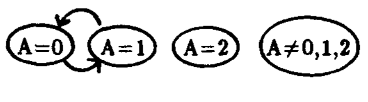
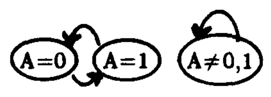


Figure 4.5

The second type of transformation involves the elimination from G of all edges made unnecessary by the insertions. For example, consider the transition specification (G_1,Insert), where Insert = {A=1} and G_1 is represented in Figure 4.6(a). Now (G_1,Insert) is realizable but not directly realizable. Since the tuple <1> is inserted, the tuple <0> can be deleted rather than updated to <1>, without changing the final effect. Thus, the edge ({A=0}, {A=1}) can be deleted from G_1 yielding G_2, represented in Figure 4.6(b). Clearly (G_2,Insert) is directly realizable.



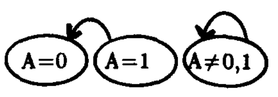


Figure 4.6(a) **Figure 4.6(b)**

The previous discussion is summarized by the following result characterizing realizable transition specifications.

4.8. ***Theorem.***(*) A transition specification (G_1,Insert) is realizable iff there exists a directly realizable transition specification (G_2,Insert) obtained by splitting G and by removing all edges of the form (C_1,C_2) where $C_2 \in$ Insert.

Consider again the characterization of directly realizable transition specifications. Intuitively, the role of the node v(c) associated with cycle c is that of a temporary storage variable. This is needed in order to permute the content of two or more hyperplanes. The need for temporary storage would disappear if one could perform two updates u(C_1;C_2) and u(C_2;C_1) simultaneously. Thus, suppose we added to the set of elementary transactions the instruction "switch (C_1;C_2)", whose effect would be the same as performing u(C_1;C_2) and u(C_2;C_1) simultaneously. Then it can be shown that every transition specification would be directly realizable.(**)

5. Transaction optimization.

In this section we focus on the problem of transaction optimization. We propose three intuitively appealing optimization criteria for transactions over a given set of hyperplanes, and show that they can be satisfied simultaneously. Then we show how an equivalent optimal transaction can be obtained from each given transaction. Furthermore, we show that optimal transactions can be obtained which have a certain desirable form that we call Second Normal Form.

We now discuss the three factors we will consider when optimizing a transaction. The first factor is the length(*) of the transaction (e.g., d(A=0) is preferred over d(A=0) d(A=0)). The second factor is the maximum number of times a tuple is modified by the transaction (e.g., d(A=0) d(A=1) is preferred over u(A=0; A=1) d(A=1) since in the second transaction <0> is unnecessarily updated to <1> before being deleted). Finally, the third factor is the complexity of the elementary transactions composing the transaction. We propose the following increasing order of complexity among elementary transactions:

(0)	u(C,C)	(C remains unchanged)
(1)	i(C)	
(2)	d(C)	
(3)	u(C_1,C_2),	where $C_1 \neq C_2$.

While the proposed ordering is intuitively appealing, it may clearly be invalid for certain specific implementations of the elementary transactions. However, it is likely that the ordering will be compatible with most reasonable implementations.

Notation. For every elementary transactions e and f, e ⩽ f denotes that e is of lesser or equal complexity than f according to the above ordering.

We now formally define optimal transactions with respect to the criteria discussed above.

Definition. A transaction $t = e_1...e_n$, $n \geqslant 0$, over the set of hyperplanes H(t) is *optimal* (with respect to H(t)) if for every transaction t′ over H(t) which is equivalent to t, the following hold:

(i) t′ is at least as long as t,

(ii) the maximum number of times t′ modifies a tuple is at least the maximum number of times t modifies a tuple, and

(iii) if t′ and t have the same length then there exists a permutation $e'_1...e'_n$ of t′ such that(**) $e_i \leqslant e'_i$, $1 \leqslant i \leqslant n$.

We next outline an algorithm that constructs, for each given transaction, an equivalent, optimal transaction over the same set of hyperplanes. First though, we need two definitions, one technical lemma, and some notation.

Definition. Let G be a transition graph. A ***storage assignment*** for G is a mapping v : 2-Cycles(G) → V_G such that for each c ∈ 2-Cycles(G), Supp(c) = Supp(v(c)) and v(c) does not belong to any cycle of G.

By Theorem 4.7, (G,Insert) is directly realizable if and only if there exists a storage assignment for G. A storage assignment is "safe" if it does not give rise to deadlock. Formally, we have:

Definition. A storage assignment v for a transition graph G is *safe* if there exists an enumeration $C_1,...,C_n$ of all connected

(*) A more formal statement of this theorem is given in the full paper.

(**) This observation is due to W. Lipski

(*) A transaction $t = e_1...e_n$, $n \geqslant 0$, has length n (e_i, $1 \leqslant i \leqslant n$, are elementary transactions).

(**) The transaction $e'_1...e'_n$ is not necessarily equivalent to t′.

components of G with at least two nodes, such that for each i ($1 \leqslant i \leqslant n$) and c in 2-Cycles ($C_i$),

(i) $v(c) \notin C_j$ for any j, $i < j \leqslant n$, and

(ii) if v(c) is in C_i, then it is adjacent to c (i.e., $(v(c),C) \in E_G$ for some vertex C of c).

The following shows that a safe storage assignment can be found for each directly realizable transition specification.

5.1. *Lemma.* If (G,Insert) is a directly realizable transition specification then there exists a safe storage assignment for G.

Finally, we need the following:

Notation. If T is a finite set of transactions, let $\underset{t \in T}{\circledast} t$ and $\circledast$ T denote a transaction $t_1 \ldots t_n$, where $t_1,\ldots,t_n$ is some enumeration of the elements of T. (If $T = \emptyset$, let $\circledast\, T = \epsilon$.)

We now outline the optimization algorithm for transactions.

Algorithm OPT.

```
Input:   a transaction t in 1NF.
Output:  a transaction OPT(t).
    1.   Construct TS(t) = (G,Insert) $(G = (V_G,E_G))$.
    2.   $E_G := E_G - \{(C_1,C_2) \mid C_1 \in V_G, C_2 \in \text{Insert}\}$.
    3.   $D := \{C \in V_G \mid od(C)=0\} \cap \text{Insert}$.
    4.   If there is c in 2-Cycles(G) with Supp(c) = U
         then
             5.   If there is no $C \in V_G - D$ such that
                  Supp(C) = U and C does not belong to
                  any cycle of G then
                      6.   Remove one vertex from D.
    7.   $E_G := E_G \cup \{(C,C) \mid C \in D\}$.
    8.   Compute a safe storage assignment v for G.
    9.   Construct(*) an enumeration $C_1 \ldots C_n$ of all con-
         nected components of G with at least two ver-
         tices, such that for each $C_i$ and cycle c in 2-
         Cycles($C_i$), $v(c) \notin C_j$ $(1 \leqslant i < j \leqslant n)$.
   10.   Let  $t_d = \circledast \{d(C) \mid C \in V_G, od(C)=0\}$,
              $t_i = \circledast \{i(C) \mid C \in \text{Insert}\}$.
   11.   For each cycle
         $c = \{(C_1,C_2),\ldots,(C_{n-1}, C_n),(C_n,C_1)\}$ of G(n ($\geqslant$ 2)
         let
         $t_c = u(C_n;v(c))\, u(C_{n-1};C_n)$
         $\ldots u(C_1;C_2)\, u(v(c);C_1)$ , where
         $(v(c),C_1) \in E_G$    if v(c) is adjacent to c.

   12.   Let $t_0 = t_d \circledast \{t_c \mid od(v(c)) = 0, id(v(c)) = 0\}$.
   13.   Remove all cycles from $C_1,\ldots,C_n$ (only edges are
         removed).
   14.   For i := 1 to n do
             15.   MAX := $\{C \in C_i \mid od(C)=0, id(C) > 0\}$.
             16.   While there are edges left in $C_i$ do
                       17.   $t_o := t_o \circledast \{t_c \mid v(c) \in$ MAX
                             $\circledast \{u(C';C) \mid C \in$ MAX, $(C',C) \in C_i$,
                             C′ is not v(c′) for
                             $c' \in$ 2-Cycles($C_j$)}
                             $\circledast \{t_{c'} \mid (v(c'),C) \in C_i$ for
                             some $C \in$ MAX, and $id(v(c'))$
                             $= 0\}$.
                       18.   Remove from $C_i$ all edges
                             $(C',C)$ where $C \in$ MAX.
                       19.   MAX: = $\{C \in C_i \mid od(C) =$
                             $0, id(C) > 0\}$.
             20.   $t_0 := t_0 t_i$.
             21.   Output $t_0$.
```

The following can now be shown:

5.2. ***Theorem.*** For each transaction t in 1NF, the transaction OPT(t) constructed by Algorithm OPT is equivalent to t, and optimal (with respect to H(t)). Furthermore, the algorithm is polynomial in the length of t.

Note that, technically, the optimality of a transaction was defined with respect to a given partition of the tuple space into hyperplanes. However, if an optimal transaction with respect to a given partition is split according to a different partition, the resulting transaction remains optimal with respect to the new partition. Formally, we have:

5.3. ***Proposition*** Let t be an optimal transaction with respect to H(t). If $\vdash_{\text{split}} t \approx t'$, where t′ is a 1NF transaction, then t′ is optimal with respect to H(t′).

The following illustrates the effect of Algorithm OPT on a simple transaction.

5.4. ***Example.*** Consider the transaction over AB:

$t =$ u(A≠0,B=1; A≠0,B=2) i(<0,1>) i(<3,2>)
u(A=0,B=1; A=0,B=2) u(A≠0,B=0; A≠0,B=1)
u(A=0,B=0; A=0,B=1) u(A≠0,B=2; A≠0,B=0)
u(A=0,B=2; A=0,B=0) d(A≠0,B=0)

The transition specification of t is TS(t) = (G,Insert), where Insert = {<0,0>} and G is represented in Figure 5.1.

The transaction output by Algorithm OPT is

OPT(t) = d(A≠0,B=1) d(A≠0,B=2) u(A=0,B=1; A=0,B=2)
u(A=0,B=0; A=0,B=1) u(A=0,B=2; A=0,B=0)
u(A≠0,B=0; A≠0,B=1) i(<0,0>)

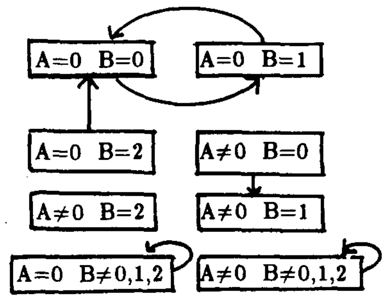


Figure 5.1

(*) Step (9) is possible by Lemma 5.1.

Remark. In general, a tuple in a database can be modified any number of times in the course of a transaction. However, it can be seen that an optimal transaction does not modify any tuple in the database more than twice. Thus, the total number(*) of tuple updates and deletions performed by an optimal transaction is at most twice the size of the database.(**)

Consider once more the transactions t and OPT(t) from Example 5.4 and let I be the instance over AB represented in Figure 5.2.

A	B
1	1
2	2

Figure 5.2

Applying transaction t to I results in the performance of(***) 6 tuple updates, 3 deletions, and 2 insertions. On the other hand, applying the optimal transaction OPT(t) to the same instance results in only 0 tuple updates, 2 deletions, and 1 insertion. The fact that OPT(t) is optimal accounts for part of the difference, but not all. For instance, consider
$t_1 = d(A \neq 0, B=1)\ d(A \neq 0, B=2) u(A=0, B=1;\ A=0, B=2)$
$u(A=0, B=0;\ A=0, B=1) i(<0,2>)\ u(A=0, B=2;\ A=0, B=0)$
$u(A \neq 0, B=0;\ A \neq 0, B=1)$. Then $t_1 \approx t$ and t_1 is also optimal. However, t_1 performs one more update than t when applied on I. This is so because t_1 inserts the tuple <0,2> first and then updates it to <0,0>, whereas t directly inserts the tuple <0,0>. Thus, the *relative order* of updates, deletions and insertions in a transaction affects the total number of operations performed. Specifically, the example suggests that all insertions should be performed last. Similarly, all deletions should be performed first (if not, some tuples may be updated first, and then deleted). Thus, it is preferable that a transaction consist of deletions, followed by updates, followed by insertions. A transaction having this property is said to be in "Second Normal Form." Formally, we have:

Definition. A transaction t is in Second Normal Form if it is in First Normal Form and $t = d_1 \ldots d_k\ u_1 \ldots u_m\ i_1 \ldots i_n$ where the d_j are deletions ($1 \leqslant j \leqslant k$), the u_j are updates ($1 \leqslant j \leqslant m$), and the i_j are insertions ($1 \leqslant j \leqslant n$).

Note that the transaction OPT(t) output by Algorithm OPT is in Second Normal Form. Thus, we have:

5.5. ***Theorem.*** For each transaction t there exists an equivalent, optimal transaction in Second Normal Form.

Remarks. (a) Let us briefly look at an alternative notion of optimality based on the number of tuple operations performed by a transaction. For each transaction t over U and relation r over U, let NOPS(t,r) be the total number of tuple operations (i.e., tuple deletions, updates and insertions) performed when t is applied to r. It would be appealing to define a notion of "strong" optimality as follows. A transaction t is strongly optimal if for each t′, $t' \approx t$, and relation r over U, NOPS(t,r) $\leqslant$ NOPS(t′,r) (i.e., t does at least as well as any other equivalent transaction on *all* databases). Unfortunately, it can be seen that, in general, there are no strongly optimal transactions equivalent to a given transaction. (In fact, a strongly optimal transaction equivalent to t exists iff the transition graph corresponding to t is 2-acyclic.(*) And, in this case, OPT(t) is strongly optimal.) A weaker but more promising notion is that of "weak" optimality. A transaction t is weakly optimal if for every equivalent t′, NOPS(t,r) > NOPS(t′,r) for some r implies that NOPS(t,r′) < NOPS(t′,r′) for some r′. In other words, if there is a transaction t′ which does better than t on some database, it does worse than t on another database. It can be seen that if t is optimal and 2NF then t is weakly optimal. In particular, OPT(t) is weakly optimal for every t.
(b) The optimization criteria considered so far do not take into account the number of hyperplanes occurring in a transaction. (Indeed, the way the space is split into hyperplanes does not affect the number or type of tuple operations performed by a transaction.) However, the cost of a transaction may also depend on the number of hyperplanes involved in the transaction. Given a transaction t, one can use the SPLIT rules to find an equivalent transaction t′ with a minimum number of hyperplanes. Unfortunately, it can be seen that it is not always possible to find a transaction with a minimum number of hyperplanes which also satisfies the other optimality criteria. The choice between minimizing the number of hyperplanes and satisfying the other optimality criteria has to be made depending on the particular implementation.

6. Axiomatization of tran saction equivalence.

In the previous two sections, we provided algorithms for deciding whether two transactions are equivalent, and for optimizing a given transaction. However, the algorithms do not provide much insight into why two given transactions are equivalent, or why a given transaction is (or not) optimal. In this section, we introduce some intuitively suggestive axioms for proving transaction equivalence. The axioms are based on simple transformation rules which highlight the interaction between deletions, updates and insertions. Due to space limitations, we only present here a brief, informal overview of our results.

We first introduce a system Ax of axioms for proving the equivalence of two transactions over the same set H of disjoint hyperplanes.(**) Ax consists of nineteen axioms grouped as follows ($C_1, C_2 C_3 C_4$ are hyperplanes in H):

Update-update axioms:

1) $u(C_1;C_2)\ u(C_3;C_4) \approx u(C_3;C_4)\ u(C_1;C_2)$
   $(C_2 \neq C_3, C_1 \neq C_4$ and $C_1 \neq C_3)$,
2) $u(C_1;C_2)\ u(C_2;C_3) \approx u(C_1;C_3)\ u(C_2;C_3)$.
3) $u(C_1;C_2)\ u(C_1;C_3) \approx u(C_1;C_2) \qquad (C_1 \neq C_2)$.
4) $u(C_1;C_2)\ u(C_1;C_2) \approx u(C_1;C_2)$.

(*) Updates and deletions of newly inserted tuples were not counted.

(**) Actually, a smaller, more accurate bound can be given.

(***) We include here updates and deletions of tuples inserted by the transaction.

(*) A graph is 2-acyclic if it has no cycles of length at least 2.

(**) Note that, if two transactions are over different sets of hyperplanes, they can be transformed into equivalent transactions over the same set of disjoint hyperplanes using the SPLIT axioms of Section 4.

Delete-delete axioms:

5) $d(C_1)\ d(C_2) \approx d(C_2)\ d(C_1)$,

6) $d(C_1)\ d(C_1) \approx d(C_1)$.

Insert-insert axioms:

7) $i(C_1)\ i(C_2) \approx i(C_2)\ i(C_1)$.

8) $i(C_1)\ i(C_1) \approx i(C_1)$.

Update-delete axioms:

9) $u(C_1;C_2)\ d(C_3) \approx d(C_3)\ u(C_1;C_2)$ $(C_3 \neq C_1,\ C_3 \neq C_2)$,

10) $u(C_1;C_2)\ d(C_2) \approx d(C_1)\ d(C_2)$,

11) $d(C_1)\ u(C_1;C_2) \approx d(C_1)$,

12) $u(C_1;C_2)\ d(C_1) \approx u(C_1;C_2)$ $(C_1 \neq C_2)$.

Update-insert axioms:

13) $u(C_1;C_2)\ i(C_3) \approx i(C_3)\ u(C_1;C_2)$ $(C_1 \neq C_3)$,

14) $i(C_1)\ u(C_1;C_2) \approx i(C_2)\ u(C_1;C_2)$,

15) $u(C_1;C_2)\ i(C_2) \approx d(C_1)\ i(C_2)$.

Delete-Insert axioms:

16) $d(C_1)\ i(C_2) \approx i(C_2)\ d(C_1)$ $(C_1 \neq C_2)$,

17) $d(C_1)\ i(C_1) \approx i(C_1)$,

18) $i(C_1)\ d(C_1) \approx d(C_1)$, and

Identity axiom:

19) $u(C_1,C_1) \approx \epsilon$.

Unfortunately, it turns out that the set of axioms Ax $\cup$ SPLIT is not complete. (In fact, we conjecture that there is no proper finite axiomatization for transaction equivalence.) Following is an example of two equivalent transactions whose equivalence cannot be proven using Ax $\cup$ SPLIT.

Example. Consider the trasactions over A:

$$t_1 = d(A=3)\ u(A=4;A=3)\ u(A=1;A=4)\ u(A=2;A=1)\ u(A=4;A=2), \text{ and}$$

$$t_2 = d(A=3)\ u(A=1;A=3)\ u(A=2;A=1)\ u(A=3;A=2)\ u(A=4;A=3)$$

It can be seen that $TS(t_1) = TS(t_2) = (G,\emptyset)$, where G is the transition graph represented in Figure 6.1. Thus, $t_1 \approx t_2$. (Intuitively, the only difference between t_1 and t_2 is that t_1 uses $\{A=4\}$ to realize the cycle in the transition graph, while t_2 uses $\{A=3\}$.) However, it can be shown that the equivalence cannot be proven using Ax $\cup$ SPLIT.

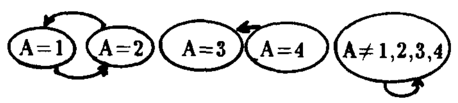


Figure 6.1

As shown above, the set of axioms Ax $\cup$ SPLIT is not complete. However, it is easy to see that Ax $\cup$ SPLIT is complete within the large subclass of 2-acyclic transactions. Furthermore, Ax is sufficiently powerful to essentially allow the optimization of *all* given 1NF transactions and to bring them to Second Normal Form. Thus, Ax $\cup$ SPLIT is sufficient for most practical purposes.

Although we do not exhibit a complete set of axioms for proving transaction equivalence, we present in [AV] a mechanism axiomatization. Informally, some "imaginary" hyperplanes are introduced as temporary storage, and Ax is extended to these imaginary hyperplanes.(*) Then we show that two transactions t_1 and t_2 are equivalent iff tt_1 and tt_2 can be proved equivalent using Ax, where t consists of a sequence of deletions of imaginary hyperplanes.

In addition, a second method is presented in [AV] for proving transaction equivalence using Ax. Specifically, it is shown that proving the equivalence of two arbitrary transactions can be reduced to proving the equivalence of several pairs of 2-acyclic transactions.

Remark. The results in this and the previous sections have shown that the large class of 2-acyclic transactions has particularly desirable properties. First, strong optimality can always be attained for 2-acyclic transactions. (Furthermore, the OPT algorithm always yields a strongly optimal transaction when applied to a 2-acyclic transaction.) Second, the system of axioms Ax $\cup$ SPLIT is complete for proving 2-acyclic transaction equivalence. Therefore the introduction of imaginary hyperplanes is not required in the 2-acyclic case. (Also, proving the equivalence of arbitrary transactions can be reduced to proving the equivalence of 2-acyclic transactions.) Finally, note that all 2-acyclic transition specifications are realizable.

Acknowledgement. The authors would like to thank François Bancilhon, Richard Hull, and Witold Lipski for their comments and helpful suggestions.

(*) Intuitively, the imaginary hyperplane corresponding to hyperplane C can be thought of as consisting of "marked" tuples of C, with values outside the domains of the attributes.

References

[AH] Abiteboul, S., R. Hull. IFO – A formal semantic database model. In preparation.

[AV] Abiteboul, S., V. Vianu. Transactions in relational databases. In preparation.

[BS] Bancilhon, F., N. Spyratos. Update semantics in relational views. *ACM Transactions on Database Systems*, Dec. 1981, 557-575.

[BG] Bernstein, P., N. Goodman. Concurrency control in distributed database systems. *Computing Surveys*, Vol. 13, No. 2, June 1981.

[Br] Brodie, M. On modelling behavioral semantics of databases. *Proc. 7th Int. Conf. on Very Large Databases* (1981), 32-42.

[C] Codd, E.F. A relational model for large shared data banks. *Communications of ACM* 13:6 (1970), 377-387.

[CCF] Castillo, I.M.V., M.A. Casanova, A.L. Furtado. A temporal framework for database specifications. *Proc. 8th Int. Conf. on Very Large Databases* (1982), 280-291.

[CW] Clifford, J., D.S. Warren. Formal semantics for time in databases. *ACM Transactions on Database Systems*, Vol. 8, No. 2, June 1983, 214-254.

[DeAZ] DeAntonellis, V., B. Zonta. Modelling events in database applications design. *Proc. 7th Int. Conf. on Very Large Databases* (1981), 23-31.

[FUV] Fagin, R., J. Ullman, M. Vardi. On the semantics of updates in databases. *Proc. Second ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1983), 352-365.

[HK] Hecht, M., L. Kerschberg. Update semantics for the functional data model. Database research report no. 4, January 1981, Bell Laboratories, Holmdel, New Jersey.

[INGRES] Woodfill, J., P. Siegal, J. Ranstrom, M. Meyer, E. Allman. INGRES version 7 reference manual (April 8, 1981).

[M] Maier, D. *The theory of relational databases.* Computer Science Press, 1983.

[PBR] Papadimitriou, C.H., B.A. Bernstein, J.B. Rothnie. Computational problems related to database concurrency control. *Proc. Conf. on Theoretical Computer Science*, Waterloo, Ontario, Canada (1977).

[R] Rolland, C. Event driven synchronization in REMORA. *Third Scandinavian Symp. on Information Modelling*, Tampere, Finland, 1984.

[Ul] Ullman, J. *Principles of database systems.* Computer Science Press, 1980.

[Uh] Ulrich, S. An abstract introduction to the temporal-hierarchical data model. *Proc. 9th Int. Conf. on Very Large Databases* (1983), 322-330.

[Ve] Verroust, A. Characterization of well-behaved database schemata and their update semantics. *Proc. 9th Int. Conf. on Very Large Databases* (1983), 312-321.

[Vi1] Vianu, V. Dynamic constraints and database evolution. *Proc. Second ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1983), 389-399.

[Vi2] Vianu, V. Object projection views in the dynamic relational model. *Proc. Third ACM SIGACT-SIGMOD Symp. on Principles of Database Systems* (1984).