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 0014
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 \([C, M, \Pi]\). 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 \([BC, PBR]\). 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 formalised (for instance, one criteria is the length of the transaction). It is shown that each given transaction can be optimised with respect to all the proposed criteria. A polynomial time optimisation 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 optimisation. 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 \(\text{dom}(A)\), 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 \(\text{dom}(A)\). The set of tuples over \(U\) is denoted \(\text{Tup}(U)\). 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 \(U = \cup \{X | X \in S\}\). 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 \(\text{Inst}(U,S)\).

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 \(V \times V\). An element in \(V\) is called a vertex. An element in \(E\) is called an edge. Let
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 ⊆ U. The restriction of C to X is the set C|X = {c ∈ C | c 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 ∈ C implies ∃ C | 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 u ⊨ C) if t ⊨ 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 transactions, 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 ∈ Tup(U) | t ⊨ C\}.

As mentioned earlier, H(U,C) ≠ ∅ if C is meaningful. Also, H(U,C₁) = H(U,C₂) implies that C₁ = C₂. For simplicity, we sometimes use the same notation for a set 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 iₓ(C), where X ⊆ S and C is a condition which specifies a complete tuple (*X) H(X,C) over X. The effect of an insertion iₓ(C) is the action eff[iₓ(C)] : Inst(U,S) → Inst(U,S) defined by

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

In the following we sometimes write iₓ(<a₁,...,aₙ>) instead of iₓ(⟨A₁ = a₁,...,Aₙ = aₙ⟩).

(* … Continued...
2) A deletion over the schema \((U,S)\) is an expression of the form \(d_X(C)\) where \(X \subseteq S\) and \(C\) is a set of conditions. The effect of \(d_X(C)\) is the action
\[
eff(d_X(C)) : \text{Inst}(U,S) \rightarrow \text{Inst}(U,S) \text{ defined by}
\]
\[
eff(d_X(C))(t) = \begin{cases} 
I(Z) & \text{if } Z \neq X \\
I(X) - H(X,C) & \text{if } Z = X.
\end{cases}
\]

3) An update over the schema \((U,S)\) is an expression
\[
u_X(C, C') \quad \text{where } X \subseteq S \text{ and for each } A \in X \text{ either}
\]
\[C_1|_A = C_2|_A \text{ or } A=a \in C_2 \text{ for some } a \in \text{dom}(A).\]

For each tuple \(t\) in \(H(X,C_1)\), the updated version \(v_X(C_1,C_2)(t)\) of \(t\) under \(v_X(C_1,C_2)\) is the tuple \(t_2\) in \(H(X,C_2)\) where
\[
t_2(A) = \begin{cases} 
t_1(A) & \text{if } C_1|_A = C_2|_A \\
& \text{if } A=a \text{ is in } C_2.
\end{cases}
\]

The effect of \(v_X(C_1,C_2)\) is the action
\[
eff(v_X(C_1,C_2)) : \text{Inst}(U,S) \rightarrow \text{Inst}(U,S) \text{ defined by}
\]
\[
eff(v_X(C_1,C_2))(t) = \begin{cases} 
I(Z) & \text{if } Z \neq X \\
(t) & \text{if } Z = X.
\end{cases}
\]

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 \(A=a\) 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 \(U = \{\text{name}, \text{department}, \text{rank}, \text{salary}\}\). The following are elementary transactions over \(U\):

1) \(i_0(\text{name} = \text{Moe}, \text{department} = \text{parts}, \text{rank} = \text{manager}, \text{salary} = 30K)\),

2) \(d_0(\text{name} = \text{Moe}, \text{department} = \text{parts}, \text{rank} = \text{manager})\)

This deletes all managers in the parts department whose names are not Moe.

3) \(u_1(\text{department} = \text{parts}, \text{rank} \neq \text{manager}, \text{department} = \text{service}, \text{rank} \neq \text{manager}, \text{salary} = 20K)\)

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:

Proceedings of the Tenth International Conference on Very Large Data Bases.
(i) $V_G$ is a finite set of disjoint hyperplanes over $U$ such that $\bigcup_{C \in V_G} C = \text{Tup}(U)$.

(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 $a \in \text{dom}(A)$.

(iii) For each $C \in V_G$, $\text{od}(C) \leq 1$.

(iv) $\text{Insert} \subseteq V_G$.

The graph $C$ 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 $\text{od}(C) = 0$ 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](image)

Let $\text{Insert} = \{ \{A=1, B=1\} \}$. Then $(G, \text{Insert})$ is a transition specification over $AB$. The transition specified by $(G, \text{Insert})$ 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 $U = AB$, $\text{Insert} = \emptyset$, and $G$ be the transition graph represented (*) in Figure 4.2. Then $(G, \text{Insert})$ is a transition specification.

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](image)

Let $\text{Insert} = \{ \{A=1, B=1\} \}$. It is easily seen that the transition specification $(G, \text{Insert})$ and that of Example 4.1(a) generate the same action. (Since $<1,1>$ is inserted, the resulting action is the}
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 \((G_1, \text{Insert}_1)\) and \((G_2, \text{Insert}_2)\) over a given set of hyperplanes \((\star)\) generate the same action iff the following conditions hold:

1. \( \text{Insert}_1 = \text{Insert}_2 \)
2. For each \( B, C \in V_0 \), and \( C \not\in \text{Insert}_i \) \((i = 1, 2)\), \((B, C) \in E_i \) iff \((B, C) \in E_{i'}\).

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 transaction. 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 \( \text{adom}(A, t) \) of all constants in \( \text{dom}(A) \) occurring in \( t \). We now associate with \( t \) a partition \( H(t) \) of \( \text{Tup}(U) \) into hyperplanes as follows. Let \( H(t) = \{ \cup_{A \in U} C_A : \text{for each } A \in U,\) 
\[ C_A = \{ A=a \} \text{ if } a \in \text{adom}(A, t), \text{ or } C_A = \{ A\neq a \} \text{ if } a \not\in \text{adom}(A, t) \} \].

Clearly, \( H(t) \) covers \( \text{Tup}(U) \) 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 \( H(t_1) = \{ \{ A=0 \} \} \). \( A\neq 7 \) is not in \( H(t_1) \) and \( t_1 \) 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 \( u(A=7; A=5) \) 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 \( A \in U \), \( a \in \text{dom}(A) \), and \( C \) is a condition over \( U \) such that \( A\neq a \in C \) and \( A=b, b \in C \) for all \( b \):

**SPLIT1.**

\[
 d(C) \approx d(C \cup \{ A\neq a \}) d(C \cup \{ A=a \}).
\]

**SPLIT2.**

\[
 u(C; C') \approx u(C \cup \{ A\neq a \}; C') u(C \cup \{ A=a \}; C') \text{ if } A=b \in C \text{ or some } b, \text{ and } C_1 = C' \cup \{ A\neq a \}, C_2 = C' \cup \{ A=a \} \text{ otherwise.}
\]

Intuitively, hyperplane \( H(U, C) \) is split into the hyperplanes \( H(U, C) \cap H(U, \{ a \in C \}) \) and \( H(U, C) \cap H(U, \{ a \notin C \}) \). 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:

### Theorem

For any transaction \( t \) there exists an equivalent 1NF transaction \( t' \), such that \( t \approx t' \).

We are now ready to outline the algorithm to construct from a given transaction a corresponding transition specification \( T_S(t) = (G, \text{Insert}) \). 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 \( \text{Insert} = i(\emptyset) \). 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:

\[
 e = d(V, C), \text{ let }
\]

- If \( e = d(U, C) \), let
- If \( e = u(U, C) \), let
- If \( e = i(U, C) \), let

*Proceedings of the Tenth International Conference on Very Large Data Bases.*
E(e) = \{<H,H> \mid H \in H(t), H \neq C\}. If e = i_u (C; C_1), let
E(e) = \{<H,H> \mid H \in H(t), H \neq C \cup \{C_1,C_2\}\}. Finally, if
E(e) = i_u (C), let E(e) = \{<H,H> \mid H \in H(t)\}. Now let
E_G = E(e_1) \circ ... \circ E(e_u). The transition specification
TS(t) = (G,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 :

\[ t = \begin{cases}
(A=5) & (A=0)
\end{cases} \]
\[ u(A=5; A=7) \quad u(A#0,5,7; A=5) \quad d(A=5). \]

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

(i) \[ E_1 = E([A=5]) \] is represented by:

(ii) \[ E_2 = E([A=0; A=5]) \] is represented by:

(iii) \[ E_3 = E([A=0; A=7]) \] is represented by:

(iv) \[ E_4 = E([A#0,5,7; A=5]) \] 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

\[ \square \]
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:

\[ \text{Definition. A transition specification } (G,\text{Insert}) \text{ is realizable if there exists a transaction defining the same action as } (G,\text{Insert}). \]

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

Example. Let (G,Insert) be the transition specification over A,
where Insert = \{<7>\} and G is represented by:

\[ \text{Figure 4.4} \]

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.

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

Some transition specifications are realizable without being directly realizable. For instance, it can be shown that the transition specification (G_(c=0), where G_c 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.

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

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 Supp(C_1) = Supp(C_2).

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

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

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 2-Cycles(G) there exists a vertex v(c) of G which does not belong to any cycle of G, such that Supp(c) = 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 $(G_1, \text{Insert})$ (Figure 4.4) which is realizable but not directly realizable. The transition specification $(G_2, \text{Insert})$ (Figure 4.5) is directly realizable and $G_2$ is obtained from $G_1$ by splitting the vertex $\{A \neq 0,1\}$ into $\{A = 2\}$ and $\{A \neq 0,1,2\}$.

**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 $(G_1, \text{Insert})$, where Insert = $\{A = 1\}$ and $G_1$ is represented in Figure 4.6(a). Now $(G_1, \text{Insert})$ 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 $(G_2, \text{Insert})$ is directly realizable.

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

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

**Theorem.** A transition specification $(G, \text{Insert})$ is realizable iff there exists a directly realizable transition specification $(G', \text{Insert})$ obtained by splitting $G$ and by removing all edges of the form $(C_i, C_j)$ where $C_j \in \text{Insert}$.

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, we 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 $d(A=0) d(A=1)$ 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:

- $u(C, C)$ (C remains unchanged)
- $i(C)$
- $d(C)$
- $u(C_1, C_2)$, 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 $f$, $e \leq f$ 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 $t = e_1...e_n, n \geq 0$, 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:

1. $t'$ is at least as long as $t$,
2. the maximum number of times $t'$ modifies a tuple is at least the maximum number of times $t$ modifies a tuple, and
3. if $t'$ and $t$ have the same length then there exists a permutation $e'_1...e'_n$ of $t'$ such that $e_i \leq e'_i$ for $1 \leq i \leq n$.

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 $v : 2\text{-Cycles}(G) \rightarrow V_G$ such that for each $c \in 2\text{-Cycles}(G)$, $\text{Supp}(c) = \text{Supp}(v(c))$ and $v(c)$ does not belong to any cycle of $G$.

By Theorem 4.7, $(G, \text{Insert})$ 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.

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984
components of \(G\) with at least two nodes, such that for each \(i \leq i \leq n\) and \(c\) in 2-Cycles \((C_i)\),

(i) \(v(c) \in C_j\) for any \(j\), \(i < j \leq n\), 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 \((G, \text{Insert})\) 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 \(\otimes T \in T\) and \(\oplus T \in T\) 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 \(\oplus T = \epsilon\).)

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) = \((G, \text{Insert})\) \((G = (V_G, E_G))\).

2. \(E_G := E_G - \{(C_i, C_j) \mid C_i \in V_G, C_j \in \text{Insert}\}\).

3. \(D := \{C \in V_G \mid od(C) = 0\} \cap \text{Insert}\).

4. If there is \(c\) in 2-Cycles\((G)\) with \(\text{Supp}(c) = U\) then

   5. If there is no \(C \in V_G - D\) such that \(\text{Supp}(C) = U\) 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 \((\ast)\) an enumeration \(C_1 \ldots C_n\) of all connected components of \(G\) with at least two vertices, such that for each \(C_i\) and cycle \(c\) in 2-Cycles\((C_i)\), \(v(c) \notin C_j\) \((1 \leq i < j \leq n)\).

10. Let \(t_d = \oplus \{d(C) \mid C \in V_G, od(C) = 0\}\), \(t_i = \oplus \{i(C) \mid C \in \text{Insert}\}\).

11. For each cycle \(c = \{(C_1, C_2), \ldots, (C_{n-1}, C_n), (C_n, C_1)\}\) of \(G(n \geq 2)\) let

   \(t_c = u(C_i, v(c)) \cup u(C_i, -v(c)) \cup u(C_i, v(c)) \cup u(C_i, -v(c))\), where \((v(c), C_i) \in E_G\) if \(v(c)\) is adjacent to \(c\).

12. Let \(t_0 = t_d \oplus \{t_c \mid od(v(c)) = 0, id(v(c)) = 0\}\).

13. Remove all cycles from \(C_1, \ldots, C_n\) (only edges are removed).

14. For \(i = 1\) to \(n\) do

   15. \(\text{MAX} := \{C \in C_i \mid od(C) = 0, id(C) > 0\}\).

   16. While there are edges left in \(C\), do

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 \(t = t' /\text{split}\), 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 \(AB\):

\[t = u(A#0, B=1; A#0, B=2) \cup (A=0, B=1) \cup (A=0, B=2)\]

\[u(A=0, B=0; A=0, B=1) \cup (A=0, B=2) \cup (A=0, B=0)\]

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

The transaction output by Algorithm OPT is

\[\text{OPT}(t) = d(A#0, B=1) \cup d(A#0, B=2) \cup u(A=0, B=1; A=0, B=2)\]

\[u(A=0, B=0; A=0, B=1) \cup (A=0, B=2) \cup (A=0, B=0)\]

\[(A=0, B=0; A=0, B=1) \cup <0,0>\]

\[A=0 \quad B=0 \quad A=0 \quad B=1\]

\[A=0 \quad B=2 \quad A=0 \quad B=0\]

\[A=0 \quad B#0,1,2 \quad A#0 \quad B#0,1,2\]

Figure 5.1

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984
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 operations performed by an optimal transaction is at most twice the size of the database.

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

\[
\begin{array}{cc}
A & B \\
1 & 1 \\
2 & 2 \\
\end{array}
\]

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 \( \text{OPT}(t) \) to the same instance results in only 0 tuple updates, 2 deletions, and 1 insertion.

The fact that \( \text{OPT}(t) \) is optimal accounts for part of the difference, but not all. For instance, consider

\[
t_1 = d(A=0,B=1) d(A=0,B=2) u(A=0,B=0; A=0,B=1) u(A=0,B=2; A=0,B=0) u(A=0,B=1; A=0,B=1).
\]

Then \( t_1 \) and \( t_1 \) are 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 \( (1 \leq j \leq k) \), the \( u_j \) are updates \( (1 \leq j \leq m) \), and the \( i_j \) are insertions \( (1 \leq j \leq n) \).

Note that the transaction \( \text{OPT}(t) \) 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 \( \text{NOPS}(t,r) \) 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 \( U \), \( \text{NOPS}(t',r) \leq \text{NOPS}(t'',r) \) (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 transaction graph corresponding to \( t \) is 2-acyclic.) And, in this case, \( \text{OPT}(t) \) is strongly optimal.

(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 transaction 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) \( u(C_1 C_2) u(C_3 C_4) \approx u(C_5 C_6) u(C_7 C_8) \)
2) \( u(C_1 C_2) u(C_3 C_4) \approx u(C_5 C_6) u(C_7 C_8) \)
3) \( u(C_1 C_2) u(C_3 C_4) \approx u(C_5 C_6) u(C_7 C_8) \)
4) \( u(C_1 C_2) u(C_3 C_4) \approx u(C_5 C_6) u(C_7 C_8) \)

(*) 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.

Singapore, August, 1984
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) \( u(C_1;C_2) d(C_1) \approx d(C_2) u(C_1;C_2) \) \( )_{C_1 \neq C_2} \),
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) \( u(C_1;C_2) i(C_1) \approx i(C_2) u(C_1;C_2) \) \( )_{C_1 \neq C_2} \),
14) \( i(C_1) u(C_1;C_2) \approx i(C_2) u(C_1;C_2) \),
15) \( u(C_1;C_2) i(C_1) \approx d(C_1) i(C_2) \).

Delete-Insert axioms:
16) \( d(C_1) i(C_1) \approx i(C_1) d(C_1) \) \( )_{C_1 \neq C_2} \),
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 transactions over \( A \):

\[ t_1 = d(A=3) u(A=4;A=3) u(A=1;A=4) \]
\[ u(A=2;A=1) u(A=4;A=2), \quad and \]

\[ t_2 = d(A=3) u(A=1;A=3) u(A=2;A=1) \]
\[ u(A=5;A=2) u(A=4;A=5) \]

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](image)

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 if \( t_1 \) and \( t_2 \) 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


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