データベース・システム 82－2
（1991. 3. 11）

階層型トランザクションの解除不能なデッドロック

兵頭 章子　大内 拓磨　滝沢 誠

東京電機大学理工学部経営工学科
e-mail hyo@takilab.k.dendai.ac.jp
e-mail ouchi@takilab.k.dendai.ac.jp
e-mail taki@takilab.k.dendai.ac.jp

本論文では、階層型トランザクションのデッドロックの解除方法について述べる。従来のデータベースシステムでは、デッドロックが生じると、デッドロックサイクル内のある一つのトランザクションを選び、ログにある更新前の状態をデータベースに戻すことで、そのトランザクションをアボートしていた。本論文では、階層型トランザクションの中で、デッドロックを解除するのに必要最低限の部分のみを、補償演算によりアボートする方法を考える。補償演算もまた、トランザクションであり補償演算を実行するために、オブジェクトにロックを要求する。本論文では、補償演算を用いたデッドロックの解除および、補償演算の実行により生じる解除不能なデッドロックについて述べる。また、この解除方法と、解除不能デッドロックの生じない安全なシステムを示す。

UNRESOLVABLE DEADLOCK IN COMPENSATING NESTED TRANSACTIONS

Akiko HYOHDOU, Takuma OUCHI, Makoto TAKIZAWA
Dept. of Information and Systems Engineering
Tokyo Denki University
Ishizaka, Hatoyama, Hiki,
Saitama 350-03, Japan

In this paper, we discuss how to resolve deadlock in the interleaved execution of nested transactions. In conventional database systems, when some deadlock occurs, one deadlocked transaction T is selected and is wholy aborted by using the log which includes the old states. In our method, only a part of T which is necessary to resolve the deadlock is tried to be aborted by executing the compensate operations. The compensate operations are also transactions and they require locks on the objects. In this paper, we show that there exists deadlock named unresolvable deadlock, which can not be resolved by executing the compensate operation. Also, we show a method for resolving the unresolvable deadlock and a safe system no unresolvable one occurs.

1. はじめに

トランザクションを同時実行することによってデッドロックが生じる問題がある。本論文では、階層型トランザクション[BERRN89, LYNC86, MOSS85, 86]の同時実行により生じたデッドロックの解除方法について述べる。

デッドロックが発生すると、待ちグラフ[HOLT72, KNAP87]に有向サイクルができる。従来の方法では、デッドロックを解除するために、サイクル内のあるトランザクションを一つ選び、そのトランザクションの全体がアボートされる。しかし、*CAD*等のシステムでは、実行時間が長く、多くのオブジェクトを利用するトランザクションをアボートし、再実行させることは、計算量が大きくシステムの性能を低下させてしまう。本論文では、補償演算[KORT90, TAKI90]を利用することによりトランザクションを全体ではなく、デッドロックの解除に必要となる部分のみをアボートする問題を考える。各演算 opに対して、任意の状態 sで $op^{-}(op(s)) = s$となる様な、補償演算 op^{-}が存在するものと仮定する。補償演算も通常のトランザクションと同様に、オブジェクトのロックを必要とする。よって、デッドロックを解除するために、任意の補償演算を実行することで、新たなデッドロックが生じ、デッドロックを解除できなくなる場合がある。本論文では、このような補償演算の実行によって解除不能なデッドロックの存在を示す。また、最大拡張系列という補償演算の系列によりデッドロックを解除する方法について述べる。

2章では、システムのモデルについて述べる。3章では、デッドロックを定義する。4章では、補償演算により、トランザクションを部分的にアボートして、デッドロックを解除する方法を示す。5章では、補償演算によって解除不能なデッドロックを示す。6章にて、解除不能デッドロックの生じない安全なシステムを定義する。7章で、非安全なシステムにおけるデッドッロクの解除方法について述べる。

2. システムモデル

2.1 システム構造

分散型システム Mは通信網で結合されたオブジェクトの集まりである。個々のオブジェクト oは抽象データ型[LISK75]である。これは、データ構造 Dと Dを操作するための演算の集合 Pにより与えられる。オブジェクトの各演算の実行は原子的であり、他のオブジェクトの演算を呼び出すことが可能である。オブジェクト oの各演算 opは、演算固有のローカル状態 L（ローカル変数の値やプログラムカウンタ）を持ち、Dの状態を D'に変化させると同時にローカル状態を L'に変化させる。ここで、Dと Lを合わせてオブジェクト oのシステム状態と呼ぶ。また、Dと Lを入力状態、D'と L'を出力状態と呼ぶことにする［図1］。

演算 opの実行によって、入力状態 Lと Dは、無矛盾な状態（i.e.インテグリティ制約 Iを満足した状態）L'と D'に変化する。

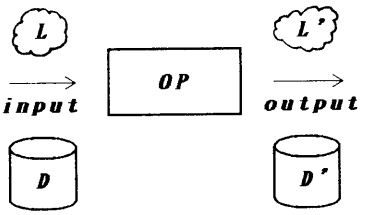


図1 オブジェクトの演算

2.2 トランザクション

トランザクション Tはオブジェクトの演算より構成される。各々の演算は、他のオブジェクトの演算を呼び出すことが可能である。これは、木構造によってモデル化できる。この様なトランザクションを階層型トランザクションとする。図2に階層型トランザクションの例 Tを示す。Tは、演算 aと bを呼び出し、aは a_1, ..., a_4を、a_3は a_{31}, a_{32}を、bは b_1, b_2を、b_1は b_{11}, b_{12}を、それぞれ呼び出している。ここで、根を根トランザクション

{T}、葉でない節を副トランザクション{a, a_3, b, b_1}、葉をアクション{a_1, a_2, a_{31}, a_{32}, a_4, b_{11}, b_{12}, b_2}と呼ぶ。枝は、枝の親が示す演算が、子の示す演算を呼び出すことを示している。

トランザクションT内でbを現在実行中の演算、aをb以前に実行された演算とする。lca(a, b)を、Tを示すトランザクション木の中で、aとbの共通の先祖の中で、一番根から遠い演算とする。例えば、図2で、a_{32}とa_4のlca(a_{32}, a_4)は、副トランザクションaである。

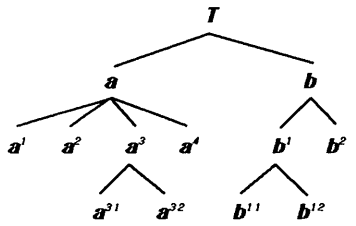


図2 階層型トランザクション T

トランザクション木内の節を縦型に探索した順序は、演算の実行順序を示す。トランザクションT内の演算aとbに対して、$a \rightarrow_T b$は、aがbよりも前に実行され先行することを示すとする。例えば、図2のトランザクションにおいて、$a_1 \rightarrow_T a_2$である。つまり、演算a_1は、a_2に先行する。

トランザクションT内で、演算aが演算a_1, ..., a_nを呼ぶとする。これを<[a; a_1; ...; a_n; a]>と書く。ここで[aは演算の開始、a]は演算の終了である。小文字のアルファベットa, b, ... は演算を示し、大文字のアルファベットA, B, ... は演算の系列を示すとする。演算系列A, Bに対して, <A; B> はそれらの連結を示す。 λは、空系列を示す。またトランザクションの部分系列を示すために、以下のような表記方法を用いる。

$A|_a{}^b$ = 先頭がbで最後がaの中置。

例えば、系列 A = <a; b; c; d; e>に対して、$A|_d{}^b$ = <b;c;d>である。

[定義] 以下の条件を満足するならば、A^iは、Aのiレベル拡張である。

(1) $A^i = A$ if Aが開始、または終了演算であるか、または、i = 0 である。
(2) $A^i = \langle [A; a_1{}^{i-1}; \ldots; a_m{}^{i-1}; A] \rangle$ if Aが、$a_1, \ldots, a_m$を呼び出す。
(3) $A^i = \langle a^i; A_2{}^i \rangle$ if $A = \langle a; A_2 \rangle$。□

演算系列内には、$A^i = A^j$ (j > i) となる固定点がある。この固定点を最大拡張系列とする。また、最大拡張によって得られる演算の系列を、最大拡張演算系列と呼び、A^*と書くことにする。

2.3 同期方法

複数のトランザクションがオブジェクトを操作するとき、システムの一意性を保つための同期方が必要となる。本論文では、オブジェクトの演算間の伴立性をもとにした同期方法を用いる。各演算aは、演算を実行する前にoをモードmode(a)でロックしなければならない。

[定義] a、bを演算、oをオブジェクトとする。m_1、m_2をmode(a)、mode(b)のモードとする。m_1がm_2と伴立であるとは、bがoをm_2でロックしているときに、aがm_1でoをロックできることである。m_1がm_2と非伴立であるとは、m_1がm_2と伴立でないことである。□

$MODE_o$をオブジェクトo上の演算によって要求されるロックのモードの集合とする。$MODE_o$は、モード間の半順序関係⊆について束となる。以下の二つの条件を満足するときモードm_1はm_2よりも排他的ではない($m_1 \subseteq m_2$)[KORT83]とする。即ち、(1) m_1がm_3と伴立であるならば、m_2はm_3と伴立である。(2) m_3がm_2と伴立であるならば、m_3はm_1と伴立である。また、∪を⊆上の最小上界(lub)とする。

次に、トランザクションTでのロックの方式を示す。Lock(a)を、演算aがコミットした後に演算aによってロックされているオブジェクトの集合、Omode(o)をoにロックを行っている演算のロックモードの集合とする。

[実行スキーマ]

(1) if aが根トランザクションであるならば、$Lock(a) = \phi$とする。

(2) if aが根トランザクションでなく、かつoが他の演算によってロックされていないならば、oを$mode(a)$でロックし、$Omode(o) := \{mode(a)\}$とする。

(3) if aが根トランザクションでなく、かつoが既に他の演算によってロックされている
then if $Omode(o)$内の全てのモード$mode(m)$と$mode(a)$が伴立である
then aはoのロックを獲得し、$Omode(o) := Omode(o) \cup \{mode(a)\}$とする。
else 条件(2)が満たされるまで、ブロックする。

(4) if aが直接呼び出した$a_1, \ldots, a_m$がコミットしている。
then if aが根トランザクションでない
then $Lock(a) = Lock(a) \cup \bigcup_{i=1,\ldots,m} Lock(a_i)$
else $Lock(a)$内の全てのロックを解除する。□

以上の方式は、ロックが根トランザクションが終了するまで保持されるので2相ロック方式である。このことから以下の定理が成り立つことは明かである。

[定理] 階層型トランザクションのインターリーブ実行は、直列可能である。■

また、各トランザクションは、副トランザクションを実行するときに、ローカル変数をトランザクションスタックSに退避し、その実行が終了した後で、Sからローカル変数を取り出し実行を続けるものとする。また、階層型トランザクションは、基本的には2相ロック方式なのでこれから作られるログでは、連鎖的アボート[BERN87]は起きない。

3. デッドロック

システムMの状態sを拡張待ち(*EWF*)グラフと呼ばれる有向グラフによって示す。グラフの節点はトランザクションの演算を示し、有向辺は演算間の待ち関係を示す。拡張待ち(*EWF*)グラフは、従来の待ちグラフ[KNAP87]を演算間の実行順序も考慮できるように拡張したものである。

[定義] a、bをオブジェクトoの演算とし、bはoをロックしているとする。このとき、aがbに直接的に依存するとは、$mode(a)$が$mode(b)$と非伴立であることである。□

[定義] 次のいずれかの条件を満足するならば、aは、bに依存している($a \rightarrow b$)とする。(1) aが、bに直接的に依存している。(2) あるトランザクションT内においてbがaに先行して実行されている。(3) $a \rightarrow c \rightarrow b$なる演算$c$が存在する。□

[定義] システムの状態sを示す拡張待ち(*EWF*)グラフGは、各節点が演算、各有向辺が、演算間の依存関係を示す有向グラフである。□

即ち、*EWF*グラフGが有向サイクルを含むならば、sはデッドロック状態である。

次にデッドロック状態の定義を行う。

[定義] $a \rightarrow a$ならば、aは直接デッドロック状態である。また、次のいずれかの条件を満足するとき、aはデッドロック状態である。(1) aは直接デッドロック状態である。(2) aはデッドロック状態の演算に依存している。□

本論文では、各状態sにおいて*EWF*グラフGを得られるものとする。

$Current(T)$をトランザクションTの現在実行中である演算とする。また、$FirstD(T)$をT内の直接デッドロック状態である演算であり、かつ最も先に実行された演算とする。

［例］図3に示すトランザクション T_1、T_2を含むシステムMについて考える。$[e$、$[n$がそれぞれ$[k$、$[c$に直接的に依存しているとする。Mの状態sを示すEWFグラフG［図4］が、有向サイクル$[a \rightarrow [a$を含むので、sはデッドロック状態である。また、$Current(T_1) = [e$、$FirstD(T_1) = [c$である。□

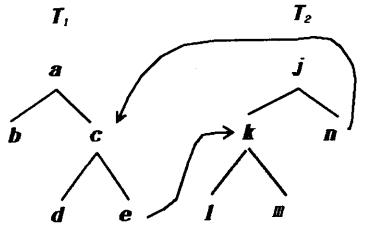


図3 トランザクション

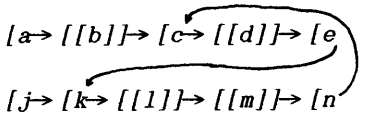


図4 EWFグラフG

4. 部分的アボート

デッドロックが検出されたとき、直接デッドロック状態のトランザクションをアボートすることによって、デッドロックを解除する。本論文では、デッドロックの解除に必要となる演算のみを、補償演算によって部分的にアボートする。

4.1 補償演算

各オブジェクトは、部分的アボートを実行するために個々の演算の結果を補償する補償演算を備えているものとする。オブジェクトoのデータ構造の状態をDとする。Dの状態で演算opを実行すると、別の状態D'となる。oの状態集合をEとする。E内のDにおいて、$op(D)$を状態Dで演算opを適用した状態とする。

［定義］演算$op^{\sim}$がopの補償演算であるとは、これらの演算は、同一のオブジェクトoによって提供されていて、E内のすべての状態Dにおいて、$op^{\sim}(op(D)) = D$である［図5］。□

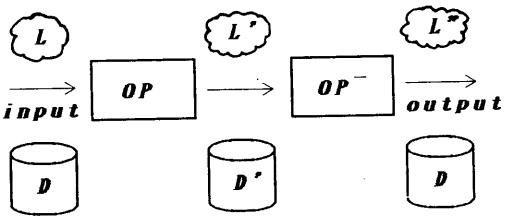


図5 補償演算

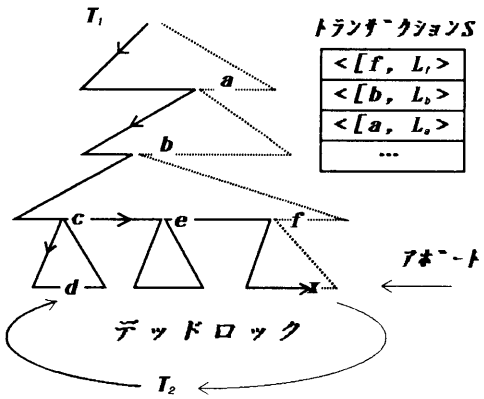
例えば、$B\text{-}tree$において、追加演算$append$で、あるオブジェクトkを追加し、続けて削除演算$delete$でkを削除すれば、追加前と同値の状態とできる。このように、$B\text{-}tree$オブジェクトでは追加（削除）演算は削除（追加）演算の補償演算である。各補償演算はそれぞれの演算の意味によって定義される。補償演算に関して、以下の仮定を設ける。

［仮定］(1) 各演算に対して、補償演算が存在する。
(2) オブジェクトo上の基本演算の実行$[[a]]$に対して、$[[a]]^{\sim}$もo上の基本演算の実行である。
(3) $mode(a^{\sim})$は$mode(a)$と伴立である。
□

4.2 デッドロックによる部分的アボート

［例1］図6に示すトランザクションT_1とT_2を考える。T_1において、xを現在実行中の演算$Current(T_1)$とし、$a, \ldots, f$をaに先行して実行された演算とする。$a, b, c, \ldots$の実行前のローカル状態を各々$L_a, L_b, L_c, \ldots$、データベースの状態を$D_a, D_b, D_c, \ldots$とする。今、演算xからdまでをアボートするとする。このとき、T_1を示すトランザクション木の中で、dとxの共通の先祖の中で、一番近い先祖の演算$lca(d, x)$はbである。また、bはaによって呼ばれているものとする。aがbを呼び出したときのaのローカル状態L_aは、トランザクションスタック内にある。従って、bとbが呼び出しているす

べての演算<[b; c; e; [f; [[x]]>を補償演算<[[x]]~; [f~; e~; c~; [b~>によりアボートして、スタックからbのローカル状態まで取り出すならば、ローカル状態L_bとデータベースの状態D_bを、bを呼び出す前の状態に復旧できる。従って、この後にbからTを再実行できる［図6］。



T_1: <...; [a; [b; [c; ...; [[d]]; ...; c]; [e; ...; e]; [f;

図6 演算dからxまでの部分的アボート

［定義］bをトランザクションTの現在実行中の演算とする。T内で、aをbの先祖の演算とし、$T'_{b}{}^{a}$をTの部分系列とする。このとき、$T'_{b}{}^{a}$の最大縮退系列$(T'_{b}{}^{a})^{-*}$とは、$T'_{b}{}^{a}$から以下の手続きにより得られる系列である。

［最大縮退系列(GRS)の検出手続き］
(1) 空のスタックMを用意し、$L = T'_{b}{}^{a}$とする。
(2) 系列Lの最後尾の演算xを一つづつ取り出す。
(3) $x = [[a]]$ または、$[a$ ならば、$[[a]]$または、$[a$をMにプッシュダウンする。
(4) $x = a]$ ならば、Lの最後尾から読んで$[a$を見つける。
$[a$が見つかったならば、aをMにプッシュダウンする。
(5) $L = \lambda$ならば、手続きを終了し、Mをポップアップして得られた演算の系列がMRAとなる。
else 手続き(2)から繰り返す。□

最大縮退系列には、トランザクションをアボートするための最上位の演算が含まれている。例えば、図6の演算xからbまでの最大縮退は、$(T'_{x}{}^{b})^{-*}$ = <[b; c; e; [f; [[x]]>である。

直接デッドロック状態のトランザクションTにおいて、FirstD(T) = a、Current(T) = bであるとする。ここでGRS(T)を、lca(a,b)からbまでの最大縮退系列とする。デッドロックを解除するために、直接デッドロック状態の演算をアボートし再実行する。つまり、GRS(T)内の演算を右から取り出し、各々の補償演算を実行し、トランザクションスタックからローカル状態を取り出すことにより、データベースとローカル状態を復旧することができる。従って、この後、aからTを再実行することができる。ここで、GRS(T)の最大拡張演算系列は、$T'_{b}{}^{lca(a,b)}$であることを注意しておく。

4.3 部分的アボートの手続き

本節では、ログから補償演算の系列を構成する方法について述べる。

トランザクションTでbを現在実行中の演算、aをb以前に実行された演算とする。また、演算aとbのlca(a, b)をcとする。演算aからbをアボートするとき、以下の手続きにより部分的にアボートされる。

［部分的アボート手続き］
(1) キューQを用意し、$L = T'_{b}{}^{c}$とする。
(2) 系列Lの最後尾の演算xを一つづつ取り出す。
(3) if $x = a]$ならば、Lを左に調べ$[a$を探し、aの補償演算$a^{\sim}$をQに入れる。
(4) if $x = [a$ならば、トランザクションスタックSから、演算aの実行前のローカル状態L_aを取り出し、演算xの補償演算$[a^{\sim}$をQに入れる。
(5) if $x = [[a]]$ならば、演算xの補償演算$[[a]]^{\sim}$をQに入れる。
(6) if $L = \lambda$ならば、手続き(7)に処理を進める。

else 手続き(2)から繰り返す。
(7) Qより一つづつ取り出して、補償演算を実行する。
(8) Tを演算bより再実行する。□

[例2] 通信販売でビデオの注文をし、その代金を自分の銀行口座Aからデパートの口座Bに振り込むトランザクション$T_{shopping}$があるとする。今、このトランザクションがデッドロック状態になり、アボートされるとする。

トランザクション$T_{shopping}$内で$[[r(B)]]$は現在の演算、$[[w(A)]]$から$[[r(B)]]$までをアボートする系列とする。$lca([[w(A)]], [[r_B)]])$は$Transfer^{AtoB}$である。よって、トランザクション$T_{shopping}$が、副トランザクション$Transfer^{AtoB}$の実行を開始する以前の状態まで、補償演算を実行し、スタックから$Transfer^{AtoB}$のローカル状態を取り出せば、$Transfer^{AtoB}$から再実行することができる。□

$GRS([Transfer^{AtoB}, [[r(B)]]) = <[[r(B)]]^{\sim}; [Deposit_B{}^{\sim}; Withdrawal_A{}^{\sim}$
$[Transfer^{AtoB}{}^{\sim}>$

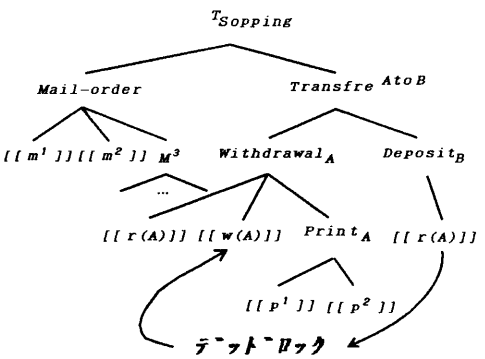


図7 トランザクション$T_{shopping}$

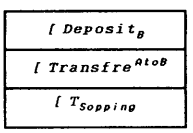


図8 トランザクションスタック

5. 解除不能デッドロック

デッドロックが検出された場合、デッドロックサイクル内のトランザクションTの$GRS(T)$の補償演算系列を実行して、トランザクションをアボートすることによってデッドロックを解除する。しかし、$GRS(T)$の補償演算系列もトランザクションであるために、オブジェクトのロックを要求することがある。このために、以下に示す解除不能なデッドロックが発生する可能性がある。

[例3] 図9に示される2つのトランザクションT_1、T_2について考える。T_1の演算系列$T_1 = <[a;b;[c;e;f;c];[d;g>$が実行され、かつ$T_1$がデッドロック状態であると仮定する。デッドロックを解除するために、T_1が選択されたならば、$GRS(T_1) = <[a;b;c;[d;g>$の補償演算の系列が実行される。また、T_2の演算系列$<[k;l;[m;o;p;m];[n;q;q^{\sim};([n)^{\sim};m^{\sim}>$が実行され、$m^{\sim}$が$b$によってロックされているオブジェクトに非伴立なモードでロックを要求しているとする。また、$c^{\sim}$がT_1によってロックされているオブジェクトに非伴立なモードでロックを要求しているとする。このとき、トランザクションT_1、T_2は、デッドロック状態である。デッドロックが検出され、$GRS(T_2) = <k[;l;m;[n;q;q^{\sim};([n)^{\sim};m^{\sim}>$の補償演算系列が実行されたならば、$m$の補償演算$m^{\sim}$が実行され、再びデッドロック状態となってしまう。同様にT_1をアボートしたならば、cの補償演算$c^{\sim}$が実行され、デッドロック状態となる。このように最大縮退系列$GRS(T)$の補償演算系列を実行して、トランザクションをアボートすることによって解除できないデッドロックを、解除不能デッドロックと呼ぶ。□

解除不能デッドロックを定義するために、以下に示す記号を用いる。

$I(T)$ = T内で最後にある補償演算。
$L(T)$ = T内で$I(T)$に先行している演算の集合。

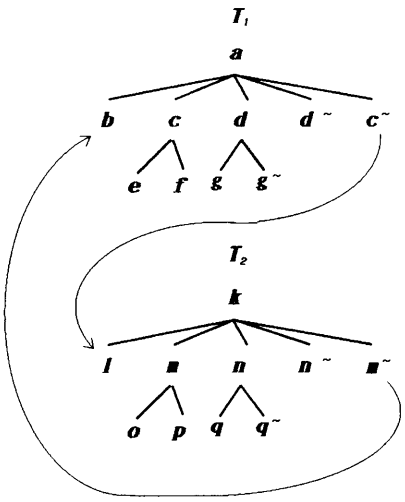


図9 トランザクション

[定義] トランザクション U内に $Current(T) \rightarrow op$なる演算 opが $L(U)$内に存在するとき、トランザクション Tは、Uに非安全に依存しているとする。□

[定義] トランザクション Tが Tに非安全に依存しているならば、Tは解除不能デッドロック状態である。□

6. 安全なシステム

図9に示すように、$op^{\sim}$が実行されるとき、必ず先行して opが実行されている。補償演算 $op^{\sim}$が opによって獲得されているオブジェクト以外のオブジェクトを利用することなく、かつ待ちなしでオブジェクトを利用することができるとする。このような $op^{\sim}$は安全であるという。$op^{\sim}$が安全ならば、新たなデッドロックを起こすことなく $op^{\sim}$を実行することができる。

[KORT83]によって、ロック xから yへの転換 U_x^yが提案されている。本論文では、このロックの転換の考えを基にして、以下に示すような伴立性を持つ転換 C_x^yを新たに提案する。

[C_x^yの伴立性]
(1) zが xと伴立であり、かつ yが zと伴立であるならば、zは C_x^yと伴立である。
(2) yが zと伴立であるならば、C_x^yは zと伴立である。
(3) xが vと伴立であり、かつ yが wと伴立であるならば、C_x^yは C_v^wと伴立である。□

opと $op^{\sim}$がオブジェクト o上の演算であり、ロックのモードがそれぞれ x、yであると仮定する。また T内で $op^{\sim}$に先行して opが実行されるとする。

[転換条件] (1) $y \subseteq x$ならば、opは xモードで oをロックし、$op^{\sim}$は xで oを利用することができる。
(2) $x \subseteq y$ならば、opは U_x^yで oをロックし、$op^{\sim}$はモードを yに転換する。
(3) (1)(2)以外の場合、opはモード C_x^yで oをロックし、$op^{\sim}$はモード $x \cup y$に転換する。□

[安全条件] $op^{\sim}$が次の二つの条件を満足するならば、$op^{\sim}$は安全である。
(1) $op^{\sim}$と $op^{\sim}$により呼ばれた演算によって利用されるオブジェクトが、T内で $op^{\sim}$に先行して実行された演算によって既にロックされている。
(2) $op^{\sim}$によって呼ばれた各演算 op_1に対して、$mode(op_1) \subseteq mode(op_2)$を満足する opによって呼ばれた演算 op_2が存在する。□

$op^{\sim}$が転換条件を満足し、かつ安全条件を満足するならば、$op^{\sim}$は安全である。システム内の全演算が安全ならば、解除不能デッドロックについて、システムは安全である。安全でないシステムを非安全なシステムとする。

[定理] システムが安全であるならば、解除不能なデッドロックは発生しない。
[証明] 定義より、$op^{\sim}$によって呼ばれた各演算は、opによって既に獲得されたロックによってオブジェクトを利用することができる。従って、補償演算を実行する際にデッドロックが発生することはない。■

安全なシステムでは、解除不能なデッドロックを発生させることなく、補償演算を実行してトランザクションをアボートすることができる。

[例4] 例2について考える。例2では $Withdrawal_A$の補償演算$Withdrawal_A^{-1}$は$Withdrawal_A$と同一の口座オブジェクト上の演算であるため、新たなロックを必要としない。また、要求するロックのモードも同一である。$Deposit_B$についても同様である。従って、このシステムは安全であり、$GRS(T)$の補償演算の実行によって、解除不能なデッドロックが生じることはない。□

7. 非安全なシステムでのデッドロック解除法

次に、非安全なシステムでの解除不能なデッドロックの解除方法について述べる。もし、補償演算が新たなオブジェクトのロックを要求しないならば、デッドロックが新たに発生することはない。$GRS(T)$の補償演算は新たなロックを要求することがあるが、最大拡張演算系列の補償演算系列は、既に獲得しているオブジェクト以外のロックを要求しない。直接デッドロック状態のトランザクションT_iにおいて、$FirstD(T) = a$、$Current(T) = b$であるとする。ここで、$TRS(T)$を$lca(a,b)$から、bまでの最大拡張演算系列とする。解除不能デッドロックは、$TRS(T)$内の演算を、右から先行している演算から取り出し、各々の補償演算を実行することによって、解除される。

デッドロックが検出されたならば、ある直接デッドロック状態のトランザクションTが選択される。次に、デッドロックが解除不能であるかどうか調べる。解除可能ならば、$GRS(T)$の補償演算系列を実行することによってTをアボートする。デッドロックが解除不能ならば、Tの演算系列（最大拡張系列）の補償演算系列を実行することによってTをアボートする。

[例5] 図10のEWFグラフGについて考える。Gは、デッドロックサイクルを含むので、直接デッドロック状態のトランザクションT_1を選択する。検出されたデッドロックは、解除不能デッドロックである。$GRT(T_1) = \langle [a; b; [c; e; f; c]; [d; g; g^{-1}; [d^{-1}; c^{-1}\rangle$の補償演算系列を実行する。この補償演算系列は、新たなロック要求を行わないため、デッドロックを起こすことなく実行される。従って、T_1のアボートにより、解除不能デッドロックは解除される。□

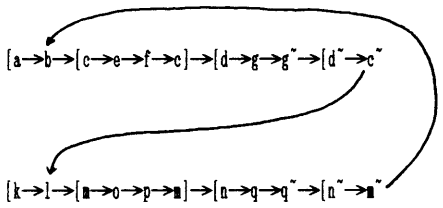


図10　拡張待ちグラフG

8. まとめ

本論文では、階層型トランザクションの同時実行によって生じるデッドロックの解除方法について述べた。本方式では従来の方式と異なり、補償演算を実行することによってトランザクションの一部分をアボートする。システムには、安全なシステムと非安全なシステムがある。非安全なシステムにおいて、解除不能デッドロックと呼ばれるデッドロックが生じることを示し、その解除方法を提案した。現在、従来の方法と補償演算の実行コストの比較を、システムの実現を通して行っている。

参考文献

[BERN87] Bernstein, P. A., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," *ADDISON-WESLEY*, 1987.

[ESWA76] Eswaren, K. P., Gray, J., Lorie, R. A., and Traiger, I. L., "The Notion of Consistency and Predicate Locks in Database Systems," *CACM*, Vol.19, No.11, 1967, pp.624-637.

[HOLT72] Holt, R. C., "Some Deadlock Properties on Computer Systems," *ACM Computing Survey*, Vol.14, No.3, 1972, pp.179-196.

[KNAP87] Knapp, E., "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, Vol.19, No.4, 1987, pp.303-328.

[KORT83] Korth, H. F., "Locking Primitives in a Database System," *JACM*, Vol.30, No.1, 1983, pp.55-79.

[KORT90] Korth, H. F., Levy, E., and Silberschatz, A., "A Formal Approach to Recovery by Compensating Transactions," *Porc. of the 16th VLDB Conf.* 1990, pp.95-106.

[LISK75] Liskov, B. H. and Zilles, S. N., "Specification Techniques for data abstractions," *IEEE Trans. on Software Engineering*, Vol.1, 1975, pp.294-306.

[LYNC86] Lynch, N. and Merritt, M., "Introduction to the Theory of Nested Transactions," *MIT/LCS/TR 367*, 1986.

[MOSS85] Moss, J. E., "Nested Transactions: An Approach to Reliable Distributed Computing," *The MIT Press Series in Information Systems*, 1985.

[MOSS86] Moss, J. E., Griggeth, N. D., and Graham, M. H., "Abstractionin Concurrency Control and Recovery Management (revised)," TR COINS 86-20, Univ. of Massachusetts, 1986.

[TAKI90] Takizawa, M. and Deen, S. M., "Synchronization Problem of Compensate Operations in the Object-Model," *Proc. of International conf. on Cooperating Knowledge Based Systems, Keele, England*, 1990.