マルチメディア通信と分散処理 68－4
グループウェア 9－4
（1995. 1. 26）

分散型データベースシステムにおける意味的な同時実行制御

田中 勝也　滝沢 誠

東京電機大学理工学部経営工学科
E-mail {katsu, taki}@takilab.k.dendai.ac.jp

分散型システムは複数のオブジェクトから構成される。オブジェクトが異なった計算機内にある場合には、トランザクションは並行実行される。本論文では、トランザクションの並行実行時の同時実行方式について考える。従来のデータベースシステムでは、CAD 等のトランザクションのログに記憶するデータ量が増大し、復旧時間が長くなる。本論文では、ログの記憶量と復旧時間を削減する復旧方法として補償演算について考察する。トランザクションの並行実行時に生じるデッドロックを考察する。さらに、補償演算の実行によって解除できない補償不可能デッドロックの解決方法を提案する。

Semantic Concurrency Contorol in Distributed Database Systems

Katsuya Tanaka and Makoto Takizawa
Tokyo Denki University

Distributed systems are composed of multiple objects which are in different computers interconnected by communication networks. Transactions manipulating multiple objects are executed in parallel. In distributed applications like *groupwares* and *CADs*, ntransactions manipulate larger amount of complex objects for longer time than the coventional transactions. In the transactions, larger data have to be stored in the log - and it takes longer time for the transactions to recover from the failure. In this paper, only the part of the transactions to be undone is aborted by executing the compensate operations executed parallel in the part. We discuss how to resolve the deadlock occured in executing the compensate operations.

1 はじめに

企業等の組織体の情報システムでは、データベースシステムがその中核となってきている。データベースシステムは、クライアントとサーバから構成される。情報システムは、通信ネットワークで相互接続された複数のクライアントと複数のサーバから構成される機能分散型のシステムとなっている。利用者は、クライアントを通して、複数のサーバを利用できる。また、電子会議、電子医療、電子授業等の複数のサーバを用いる多様な応用が可能となってきている。

これまでのトランザクション [3] は、小量のデータを短時間扱うものであった。これに対して、分散型応用ではトランザクションは、複雑なデータ構造とマルチメディアデータ等の大量のデータを長時間扱うようになってきている。本研究は、分散した大量のオブジェクトを長時間扱うための、トランザクションの同時実行方式について考える。分散型システムは、複数の計算機に分散された複数のオブジェクトから構成される。各オブジェクトは、データ構造と操作演算を提供する抽象データ型である。オブジェクト o の各操作演算 op は、原子的に実行される。op は、o を操作するとともに、さらに他のオブジェクト o' の演算 op' を実行する。op' も o' で原子的に実行される。このように、トランザクションは階層的 [10] に構成される。トランザクションが操作するオブジェクトが異なった計算機にある場合には、これらのオブジェクト演算が並行に実行される場合がある。本論文では、トランザクションが並行実行される場合も考える。

デッドロック、システム障害等により、トランザクションがアボートされる場合がある。これまでのデータベースシステムのログには、トランザクションが更新したシステムの状態が記憶されている。システムの状態を、ログ内に記憶された過去の状態に戻すことによりトランザクションがアボートされる [1]。グループウェア、CAD 等の分散型応用のトランザクションは、大量の複雑なデータを長時間操作することから、トランザクションのログに記憶されるデータ量が増大し、復旧時間が長くなることが問題となる。このために、本論文では、トランザクションが実行した演算をログに記憶することにより、ログの記憶量の減少をはかる。ログ内の演算 op の補償演算 $op^{\sim}$ を実行することにより、トランザクションのアボートを行なう。$op^{\sim}$ はトランザクションとして実行されるために、$op^{\sim}$ の実行により、デッドロックが生じる場合がある。このデッドロックの解除方法について考える。

2章で、システムモデルを示す。3章で補償演算について述べる。4章で階層型トランザクションの概念を示す。5章で同期手法を示す。6章では、トランザクションの並列実行により生じるデッドロックについて論じる。7章では、補償演算の実行により生じるデッドロックについて論じる。

2 システムモデル

システム M は、通信網で相互接続された計算機内のオブジェクトから構成される。各オブジェクト o は、抽象データ型であり、データ構造 D_o と、D_o を操作するための操作演算の集合との組として与えられる。操作演算には、公開と基本の2種類がある。オブジェクト o の利用者は、公開操作演算を用いてのみ o を扱える。o の各公開演算 op の実行は原子的である。即ち、op は実行されるか、全く実行されないかのいずれかである。op の実行が完了したとき、op はコミットしたとする。op の実行を完了できないとき、op はアボートしたとする。もう1つの基本操作演算は、オブジェクトを直接扱う操作演算で、o の公開操作演算を実現するために利用される。

オブジェクト o は、公開演算 op の実行により状態が変化する。o の任意の状態 s に対して、$op(s)$ は、s に op を適用して得られた状態とする。op_1 と op_2 を、o の公開演算とする。$op_1 \circ op_2$ は、o に op_1 を適用し、ついで op_2 を適用することを示す。

[定義] オブジェクト o の任意の状態 s に対して、$op_1 \circ op_2(s) = op_2 \circ op_1(s)$ であるとき、op_1 と op_2 は交換可能である。交換可能でないとき、op_1 と op_2 は競合する。□

op_1 と op_2 が競合するとき、op_1 と op_2 がどの様な順序で実行されるかにより、得られる o の状態が異なる。ここで、o_1 と o_2 をオブジェクトとする。また、op_1 が、o_1 と o_2 のおのおのの公開演算 op_{11} と op_{21} を実行し、op_{11} と op_{12}、op_{21} と op_{22} が、おのおの o_1 と o_2 で競合するとする。直列可能性理論 [1] では、o_1 と o_2 で競合する演算は、おのおので op_1 または op_2 のいずれかが実行したものが先行しなければならない。例えば、o_1 で op_{11} が op_{12} に先行するならば、o_2 でも op_{21} が op_{22} に先行している必要がある。ここで、op_1 と op_2 が o で交換可能であるとする。このとき、op_1 と op_2 の実行順序がいずれでもよいことから、op_{11} と op_{12}、op_{21} と op_{22} の実行順序が問題とならなくなる。オブジェクトが階層構造を持つ場合の直列可能性については、[10] で論じられている。

公開演算 op は、o に適用される前に、o をモード $mode(op)$ でロックする。op_1 と op_2 が交換可能であるとき、$mode(op_1)$ と $mode(op_2)$ は伴立であるとする [4]。ここで、op_1 が o を $mode(op_1)$ でロックしているとき、op_2 が o を利用しようとしているとする。このとき、$mode(op_1)$ が $mode(op_2)$ と伴立であれば、op_2 は、o を $mode(op_2)$ でロックし、操作できる。伴立でないとき、op_2 はロックできるまで待つことになる。

公開演算 op がオブジェクト o をロックした後に、op を実現している基本演算と他のオブジェクトの公開演算が実行される。o の公開演算 op が、基本演算のみにより構成されているとき、op は単純であるとする。op が単純でないとき、即ち、公開演算を用いて実現されているとき、op を複合演算とする。o の全ての公開演算が単純であるとき、o を単純とする。単純でないオブジェクトを複雑とする。従来のデータベースサーバとファイルサーバは、単純オブジェクトの例である。これに対して、複数のサーバを用いた応用は、複雑オブジェクトの例である。

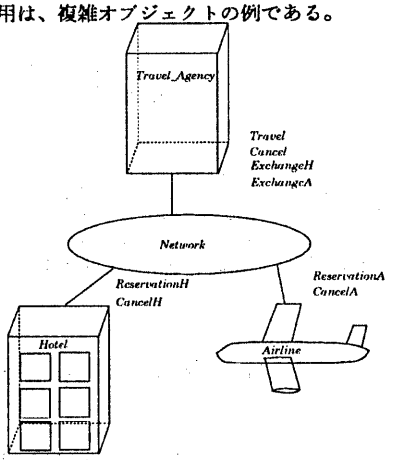

図 1: 分散型システム M

[例1] システム M は、3つのオブジェクト *Travel_Agency*, *Airline*, *Hotel* から構成される [図1]。*Travel_Agency* は、公開演算として *Travel*, *Cancel*, *ExchangeH*, *ExchangeA* を提供する。*Travel* は、旅行のためにホテルと飛行機の予約を行なう公開演算である。*Cancel* は、ホテルと飛行機の予約のキャンセルを行なう公開演算である。*ExchangeH* は、あるホテルに予約している人の宿泊期間、またはホテルを変更する公開演算である。*Hotel* と *Airline* は、それぞれ公開演算 *ReservationH*, *CancelH*, *ReservationH*, 及び *CancelH* を提供する。*ReservationH* と *ReservationA* はホテルと飛行機の予約を各々行なう公開演算である。*CancelH* と *CancelA* は、それぞれホテルと飛行機の予約をキャンセルする公開演算である。このように、*Travel_Agency* は複雑オブジェクトであり、*Airline* と *Hotel* は単純オブジェクトである。*ExchangeA* と *ExchangeH* は交換可能な操作演算であり、$mode(ExchangeA)$ と $mode(ExchangeH)$ は伴立である。□

各オブジェクト o は、公開演算 op の実行要求を受けたとき、op を実行できるならば、op の実行を行なう。o は op を実行し、結果を返す。このように、op は遠隔呼出し (RPC) に基づいて実行される。

3 補償演算

オブジェクト o の公開演算 op が、公開演算 $op_1, \ldots, op_m$ を実行するとする。各 op_i を、オブジェクト o_i の公開演算とする ($i = 1, \ldots, m$)。op は、$op_1, \ldots, op_m$ の全てが完了したときに、完了できる。$op_1, \ldots, op_m$ の1つでも完了できないときには、op はアボートする。op の実行により、o だけでなく、$o_1, \ldots, o_m$ の状態も変化するので、システム M の状態が更新される。ここで、以下を定義する。

[定義] op_1 と op_2 をオブジェクト o の演算とする。o の任意の状態 s に対して、$op_1 \circ op_2(s) = s$ であるとき、op_2 を op_1 の補償演算とする。□

op の補償演算を、$op^{\sim}$ と書くことにする。op の実行後に $op^{\sim}$ を実行することにより、o の状態を op の実行前の状態に戻すことができる $(op \circ op^{\sim}(s) = s)$。$op$ は、o 以外のオブジェクトも操作する。$op^{\sim}$ は、o の状態を op の実行以前の状態に戻すが、他のオブジェクトの状態を戻すとは限らない。システム M の状態 S に op を適用し、得られた状態を $op(S)$ とする、このとき、$op \circ op^{\sim}(S) = S$ となるとは限らない。

[例 2] 図 1のシステム M を考える。ここで、オブジェクト *Travel_Agency* が飛行機とホテルの各々の全予約人数 *amount* に関するデータ D_{TA} を持つとする。D_{TA} 内の全人数 *amount* は、公開演算 *Reservation* と *Cancel* により更新されるとする。*Reservation* が実行され、ついで *Reservation* の補償演算 *Cancel* が実行される場合を考える。*Reservation* は、*ReservationH* と *ReservationA* を実行し、*Cancel* は *CancelH* と *CancelA* を実行する。ここで、*Reservation* で予約した客と *Cancel* でキャンセルされる客が違っていても構わない。すなわち、宿泊日や飛行機の搭乗日を過ぎた予約者、あるいは仮予約の客を優先的にキャンセルすると仮定する。このとき、*Travel_Agency* の状態は、*Reservation* の実行以前に戻るが、*Hotel* と *Airline* の状態は戻らない。□

$op \circ op^{\sim}(S) = S$ となるとき、$op^{\sim}$ を op の逆演算とし、op^{-1} と書く。op の実行後に、$op^{\sim}$ が実行されたとき、op は補償されたとする。

4 階層型トランザクション

分散型システム M のトランザクション T は、操作演算の実行系列であり、原子的な実行単位である。ここで、T は、公開演算 $op_1, \ldots, op_m$ から構成されるとする。各 op_i はオブジェクト o_i に発行され、o_i で原子的に実行される。さらに、op_i は、公開演算 $op_{i1}, \ldots, op_{im_i} (m_i \geq 1)$ を実行する。各 op_{ij} は、オブジェクト o_{ij} で原子的に実行される。ここで、op_{ij} は op_i の子とし、op_i を op_{ij} の親とする。このように、T は、演算を節点とし、節点間の親子関係を示す木構造により示される。これをトランザクション木とする。トランザクション木で、葉は基本操作演算を示し、中間節点は公開演算を示す。根はトランザクションを示す。

次に、op_i が、$op_{i1}, \ldots, op_{im_i}$ を実行し、各々が、オブジェクト $o_{i1}, \ldots, o_{im_i}$ を操作するとする。これらのオブジェクトが異なった計算機内にある場合を考える。例えば、o_{ij} と o_{ik} が異なった計算機にあり、互いに独立に実行できるとき、op_{ij} と op_{ik} を並行に実行できる。op_i が実行する公開演算の集合を $\Omega(op_i) = \{op_{i1}, \ldots, op_{im_i}\} (m_i \geq 0)$ とする。$\Omega(op_i)$ 内の op_{ij} と op_{ik} について、op_{ij} が op_{ik} に先行して実行されるとき、op_{ij} は op_{ik} に先行するとし、$op_{ij} \rightarrow op_{ik}$ と書く。$op_{ij} \rightarrow op_{ih} \rightarrow op_{ik}$ であるとき、op_{ij} は、op_{ik} に推移的に先行するとする。$op_{ij} \rightarrow op_{ik}$ は基本的であるとする。op_{ij} と op_{ik} 間に先行関係が無いとき、op_{ij} と op_{ik} は独立であるとし、$op_{ij} \parallel op_{ik}$ と書く。$\parallel$ は、対称的である。このとき、op_{ij} と op_{ik} は並行に実行されている。$\Omega(op_i)$ 間で、$op_{ik} \rightarrow op_{ij}$ なる op_{ik} が、1つも存在しない op_{ij} を先頭演算とする。また、$op_{ij} \rightarrow op_{ik}$ なる op_{ik} が一つも存在しない op_{ij} を最終演算とする。

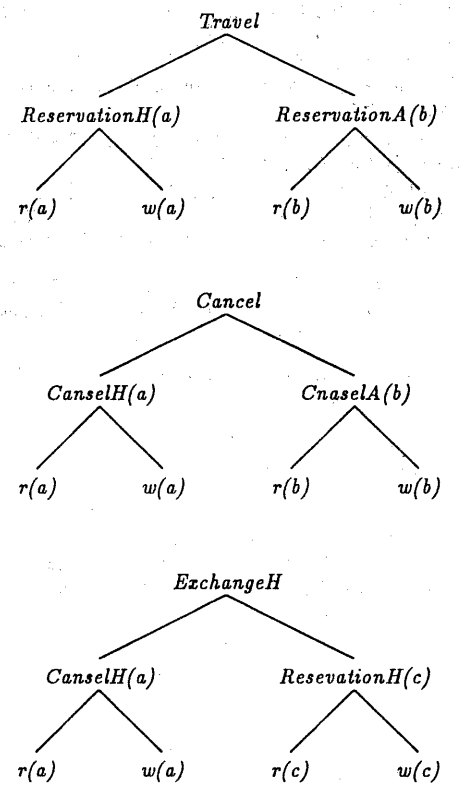

図 2: トランザクション木

[例 3] 図 2に図 1で示したトランザクション *Travel*, *Cancel*, *ExchangeH* のトランザクション木の例を示す。トランザクション *Travel* は、公開演算 *ReservationH(a)* と *ReservationA(b)* を実行している。ここで、a と b はおのおの予約したいホテルと飛行機を示す。さらに、*ReservationH(a)* は、基本操作演算 $r(a)$ と $w(a)$ を実行し、*ReservationA(b)* は、$r(b)$ と $w(b)$ を実行する。トランザクション *Cancel* は、*CancelH* と *CancelA* の 2つの公開演算を実行し、*ExchangeH* は、*CancelH* と *ReservationH* を実行する。*ReservationH(a)* と *ReservationA(b)*、*CancelH(a)* $\parallel$ *CancelA(b)*、そして *CancelH(a)* $\parallel$ *ReservationH(c)* であり、並行に実行されている。□

トランザクション T 内の各操作演算 op_i は、操作演算 $op_{i1}, \ldots, op_{im_i} (m_i \leq 0)$ を実行するとする。これを $[op_i, op_{i1}, \ldots, op_{im_i}, op_i]$ と記述する。$[op_i$ と $op_i]$ は、おのおの op_i の開始と完了を示す。図 2に示したトランザクション *Trtavel* は、[*Travel*, *ReservationH*, *ReservationA*, *Travel*] と書ける。これを *Travel* の 1レベル拡張とし、$Travel^1$ と書く。*ReservationH* と *ReservationA* は、さらに基本操作演算を実行していて、$ReservationH^1$ = [*ReservationH*, $r(a)$, $w(a)$, *ReservationH*] と $ReservationA^1$ = [*ReservationA*, $r(a)$, $w(a)$, *ReservationA*] となる。以上から *Travel* は、[*Travel*, [*ReservationH*

, $r(a)$, $w(a)$, $ReaservationH$], [$ReservationA$, $r(a)$, $w(a)$, $ReservationA$], $Travel$] となる。これを $Travel$ の 2 レベル拡張 $Travel^2$ とする。同様にして演算 op の i レベル拡張 op^i が定義できる。ここで、$op^i = op^j$ $(j > i)$ で、$op^i \neq op^k$ $(k < i)$ のとき、op^i を最大拡張とし、op^* と書く。$Travel^* = Travel^2$ である。

トランザクション T が直列に実行される場合には、木内の操作演算が縦型の順序で実行される。ここで、T 内の各演算 op_i の子の集合 $\Omega(op_i) = \{op_{i1}, \ldots, op_{im_i}\}$ と実行順序 $\rightarrow (\subseteq \Omega(op_i)^2)$ を考える。T 内の各節点 op_i について、以下のような有向辺を設ける。

- $\Omega(op_i)$ 内の任意の op_{ij} と op_{ik} に対して、$op_{ij} \rightarrow op_{ik}$ でこれが推移的でないとき、節点 op_{ij} から op_{ik} に有向辺を設ける $(op_{ij} \rightarrow op_{ik})$。
- op_{ij} から $\Omega(op_i)$ 内の先頭演算 op_{ij} に有向辺を設ける $(op_i \rightarrow op_{ij})$。
- $\Omega(op_i)$ 内の最終演算 op_{ij} から、op_i に有向辺を設ける $(op_{ij} \rightarrow op_i)$。

こうして得られた木を、拡張トランザクション木とする。op_i の子演算が直列実行される場合には、$\Omega(op_i)$ 内の演算は全順序づけられている。$\Omega(op_i)$ 内の op_{ij} と op_{ik} 間に有向辺がない場合は、op_{ij} と op_{ik} は並列実行される。

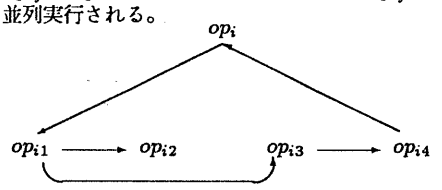

図 3: 拡張トランザクション木

[例 4] 公開演算 op_i が、操作演算 op_{i1}, op_{i2}, op_{i3}, op_{i4} を実行している例を考える。$\Omega(op_i) = \{op_{i1}, op_{i2}, op_{i3}, op_{i4}\}$。$op_{i1}$ は op_{i2} と op_{i3} に先行し、op_{i3} は op_{i4} に先行している。即ち、$op_{i1} \rightarrow op_{i2}$, $op_{i1} \rightarrow op_{i3} \rightarrow op_{i4}$。$op_{i2}$ と、op_{i3} 及び op_{i4} は、並行実行されている。op_{i1} は op_i の先頭で、op_2 と op_4 は、op_i の最終である。図 3に、op_i の拡張トランザクション木を示す。□

例 4 から分かるように、拡張トランザクション木内の有向辺は、演算の実行順序を示している。op_i は、op_{i2} と op_{i4} がともに完了したときに完了する。

図 3で、op_{i2} の実行に失敗し、op_{i2} がアボートされたとする。このとき、op_{i3} 及び op_{i4} は、op_{i2} と独立であり、アボートする必要はない。op_{i2} に続いて、op_{i1} がアボートされるとする。このときは、op_{i3} と op_{i4} をアボートする必要がある。このように、op_i が公開演算 op_{ij} の実行に失敗する場合がある。op_{ij} がアボートされても、op_{ik} と op_{ij} が独立ならば、op_{ik} はアボートされる必要がない。

2 つの演算 op_1 と op_2 に対して、op_1 と op_2 の lub、$op_1 \bigcup op_2$ $(=op)$ を以下のように定義する。

(1) $op_1 \rightarrow op$, $op_2 \rightarrow op$、
(2) $op_1 \rightarrow op'$, $op_2 \rightarrow op'$, $op' \rightarrow op$ なる op' が存在しない。

5 同期手法

オブジェクト o が、公開操作演算 op を実行するときのロック手法を考える。op は、オブジェクト o_1 に対する公開演算 $op_1, \ldots, o_m$ に対する op_m を実行するとする。

[ロック方法]
(1) o を $mode(op)$ でロックしてから、op を実行する。
(2) $op_1, \ldots, op_m$ を実行する。□

次に、オブジェクトのロックをいつ解放するかを考える。ロックの解放方法 [11] として以下がある。

[ロックの解放方法]
1. op がロックした全オブジェクトのロックを解放する。
2. op の子 $op_1, \ldots, op_m$ がロックしたオブジェクト $o_1, \ldots, o_m$ のロックを解放する。しかし、o のロックは解放しない。
3. ロックの解放を行なわない。T が全ての演算を実行し完了 (又はアボート) したときにのみ、T が獲得した全てのロックを解放する。□

1 番目の方法を開方式、2 番目を半開方式、3 番目を閉方式とする。閉方式は、厳格な 2 相ロック方式 [1, 3] である。ロックはトランザクションが終了するまで保持され、終了 (完了又はアボート) 時に解放される。従って、他の方式と比較して、一番多くのオブジェクトがロックされる。開方式は、オブジェクトに対する公開演算が完了したならば直ちに、オブジェクトを解放する。開方式のトランザクションは、一番少ないオブジェクトがロックされる。半開方式では、op の子演算のオブジェクトのみが解放され、o のロックは保持される。

演算 op が完了し、ロックを解放したとき、op はコミットしたとする。

6 デッドロック

6.1 並列実行デッドロック

図 4に示す二つのトランザクション T_1 と T_2 について考える。ここで、op_{122} と op_{23}、op_{13} と op_{252} は、互いに競合するとする。op_{122} は、op_{23} のロックの解放を待ち、op_{252} は、op_{13} のロックの解放を待つとする。このとき、T_1 と T_2 はデッドロックしている。各トランザクションが並列実行される場合に生じるデッドロックについて考える。

op_1 と op_2 を二つの演算とする。op_1 が、op_2 を待つとは、次の場合である。

(1) op_2 がロックしているオブジェクトの解放を、op_1 が待つ。
(2) op_1 は op_2 が完了しないとコミットできない。□

演算間の依存関係を定義する。

[定義] op_1 と op_2 を、同じトランザクション T 内の二つの演算とする。op_1 が、op_2 に T 内で依存する $(op_1 \Rightarrow_T op_2)$ とは、以下の条件のいずれかが充足されることである。

(1) $op_1 \rightarrow op_2$、または、
(2) $op_1 \| op_2$、$op = op_1 \bigcup op_2$ が未完了で、op_1

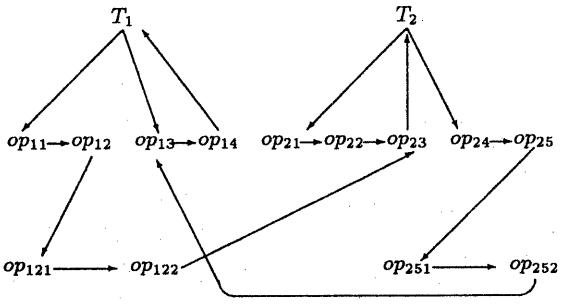

図 4: デッドロック

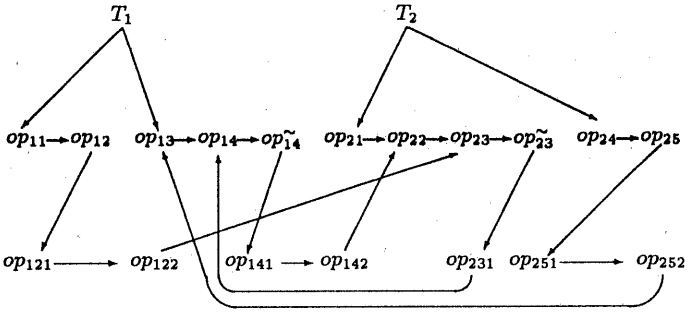

図 5: 補償不可能デッドロック

$\rightarrow op' \rightarrow op$ なる全ての演算 op' が完了していて、op_2 が完了していない。

(3) $op_1 \Rightarrow_T op' \Rightarrow_T op_2$ なる op' が存在する。□

例えば、図4で、$op_{11} \bigcup op_{14} = T_1$ であり、$op_{14} \Rightarrow_{T_1} op_{122}$、$op_{121} \Rightarrow_{T_1} op_{122}$ である。

次に、op_1 と op_2 を異なったトランザクションの演算とする。op_1 が op_2 を直接待つとは、op_1 が、op_2 のロックしたオブジェクトを待つことである。図4では、op_{122} が op_{23} を直接待つ。

[定義] op_1 が op_2 に依存する ($op_1 \Rightarrow op_2$) とは、以下の条件のいずれかが充足されることである。

(1) あるトランザクション T 内で、$op_1 \Rightarrow_T op_2$ である。

(2) op_1 は、op_2 を直接待つ。

(3) $op_1 \Rightarrow op \Rightarrow op_2$ なる演算 op が存在する。□

例えば、図4で、$op_{14} \Rightarrow op_{122}$, $op_{122} \Rightarrow op_{23}$, $op_{23} \Rightarrow op_{252}$, $op_{252} \Rightarrow op_{13}$, $op_{13} \Rightarrow op_{14}$ である。以上から、$op_{13} \Rightarrow op_{14} \Rightarrow op_{122} \Rightarrow op_{23} \Rightarrow op_{252} \Rightarrow op_{13}$ となる。

[定義] T_1 と T_2 をトランザクションとする。T_1 内の演算 op_1 と、T_2 内の演算 op_2 に対して、$op_1 \Rightarrow op_2$ であるとき、T_1 は T_2 に依存する。□

依存関係を用いて、デッドロックを定義する。

[定義] トランザクション T に対して、$T \Rightarrow T$ であるとき、T はデッドロックしている。□

即ち、図4で、$T_1 \Rightarrow T_2 \Rightarrow T_1$ であり、T_1 と T_2 はデッドロックしている。即ち、依存関係 $\Rightarrow$ について、有向閉路内の演算と全てのトランザクションがデッドロックしている。この有向閉路をデッドロック閉路とする。

op をトランザクション T の演算とする。このとき、op が以下の条件を充足するとき、op を要求されている演算とする。

(1) $op \Rightarrow op$、即ち op はデッドロックしていて、

(2) $op' \Rightarrow op$ なる演算 op' が他のトランザクションに存在し、

(3) $op' \rightarrow op$ なる全ての演算 op' に対して、$op' \Rightarrow op'$ でない。

ここで、(2) の op' を要求している演算とする。

6.2 補償不可能デッドロック

トランザクションがデッドロックした場合には、デッドロック閉路内のトランザクションをアボートする必要がある。例えば、図4で T_1 をアボートすることを考える。デッドロックを解除するためには、T_1 内の要求されている演算 op_{13} をアボートする必要があるが、op_{11}, op_{12} をアボートする必要はない。このように、T 内の全演算ではなく、その一部をアボートすることを、部分アボートとする。

T をトランザクションとし、op を T の完了した演算とする。T 内で $op \rightarrow op'$ なる演算を $C(op)$ とする。$C(op) = \{op' \mid op \rightarrow op'\} \cup \{op\}$。図4で $C(op_{13}) = \{op_{13}, op_{14}\}$ である。$C(op)$ は、デッドロック状態を解除するためにアボートされる演算の集合を与えている。本論文では、完了した演算 op の結果は、op の補償演算 $op^{\sim}$ を用いて無効とする。ここで、$C(op)$ 内の演算を補償するための補償演算集合 $C(op)^{\sim}$ を以下に与える。

(1) $C(op)^{\sim} = \{op^{\sim} \mid op \in C(op)\}$、

(2) $C(op)$ 内の任意の演算 op_1 と op_2 に対して、$op_1 \rightarrow op_2$ ならば、$C(op)^{\sim}$ で、$op_2{}^{\sim} \rightarrow op_1{}^{\sim}$ である。

図4で、$C(op_{13})^{\sim}$ とは、$op_{14}{}^{\sim} \rightarrow op_{13}{}^{\sim}$ である。即ち、$op_{13} \rightarrow op_{14} \rightarrow op_{14}^{\sim} \rightarrow op_{13}^{\sim}$ により、op_{13} と op_{14} が補償される。

演算 op の補償演算 $op^{\sim}$ も、op と同様に原子的かつ階層的に実行される。このために、$op^{\sim}$ の実行中にデッドロックとなる可能性がある。図4で、T_1 の $C(op_{13})$ を $C(op_{13})^{\sim}$ により補償するとする。ここで、まず $op_{14}^{\sim}$ が実行され、op_{141} と op_{142} が、$op_{14}^{\sim}$ の子演算で実行されるとする。ここで、op_{142} が、op_{22} を直接待つとするとデッドロックとなる。次に、このデッドロックを解除するために、$op_{23}^{\sim}$ が実行され、op_{231} が実行され、op_{231} が op_{13} を直接待つとする。再び、T_1 と T_2 はデッドロックする [図5]。デッドロックを解除するために T_1 を補償するとする。op_{142}, op_{141} が補償され、op_{14} を補償するために、再度 $op_{14}^{\sim}$ により、op_{142}, op_{141} が実行され、再びデッドロックとなる。T_2 についても、op_{231} を補償した後に、再度 $op_{23}^{\sim}$ が実行され、図5と同じ状態になる。このように、補償演算を実行しても解除できないデッドロックが存在する。これを補償不可能デッドロック [11] とする。トランザクションが並列に実行されない場合については、[11] で論じられている。ここでは、並列実行した場合を考える。トランザクションが並列実行された場合の補償不可能デッドロックを定義する。T_1 と T_2 をトランザクションとする。$op_1^{\sim}$ を T_1 で実行された補償演算とする。

(1) $op_1^{\sim} \rightarrow op_2$ に対して、$op_2 \Rightarrow op_3$ なる演算 op_3 が T_2 にあるとき、$T_1 \rightsquigarrow T_2$ である。
(2) $op_2 \Rightarrow_{T_1} op_1^{\sim}$ なる演算 op_2 に対して、$op_3 \Rightarrow op_2$ なる演算 op_3 が T_2 にあるとき、$T_1 \mapsto T_1$ である。 □

[定義] T_1 が、T_2 に非安全に従属する ($T_1 \vdash T_2$) とは、以下のいずれかが充足されることである。
(1) $T_1 \rightsquigarrow T_2$ で、$T_1 \mapsto T_2$ である。
(2) $T_1 \rightsquigarrow T_3$ で、$T_3 \vdash T_2$ なる T_3 が存在する。
(3) $T_1 \vdash T_3$ で $T_3 \mapsto T_2$ なる T_3 が存在する。 □

補償不可能デッドロックを以下に定義する。

[定義] トランザクション T に対して、$T \vdash T$ であるとき、T は補償不可能デッドロックしている。 □

6.3 解除法

分散型システム M 内の任意のオブジェクト o_1 と o_2 間の関係について考える。o_1 の演算 op_1 が、o_2 の演算 op_2 を実行するとき、$o_1 \succ o_2$ とし、o_1 は o_2 より上位とする。ここで、o_1 と o_2 について、$o_1 \succ o_2$ でも $o_2 \succ o_1$ でもないとき、o_1 と o_2 は同位 ($o_1 \parallel o_2$) とする。このとき、以下の条件を満足するシステム M を階層的であるとする。

(1) M 内の任意の o_1 と o_2 について、$o_1 \succ o_2$、$o_2 \succ o_1$、または $o_1 \parallel o_2$ である。
(2) M 内の任意の o_1, o_2, o_3 について、$o_1 \succ o_2 \succ o_3$ であるならば、$o_1 \succ o_3$ でない。

トランザクション T が以下の条件を満足するとき、T を正規形であるとする。
(1) T が直接実行する任意の演算のオブジェクトは同位である。

[定理] 分散型システム M が階層的で、全てのトランザクションが正規形ならば、補償不可能デッドロックは生じない。 □

7 おわりに

本論文では、同時実行性を高めるために、階層型トランザクションの概念を述べた。トランザクションが並列実行される場合のデッドロックを定義した。このとき、補償演算によりトランザクションの部分アボートを行なうと、解除できないデッドロックの存在を示した。さらに、階層型システムでは補償不可能デッドロックの生じないことを示した。

参考文献

[1] Bernstein, P. A., Hadzilacos, V., Goodman, N., "Concurrency Control and Recovery in Database Systems," *Addison-Wesley Publishing Company*, 1987.

[2] Garcia-Molina, H. and Salem, K., "Sagas," *Proc. of the ACM SIGMOD*, 1987, PP.249-259.

[3] Gray, J., "The Transaction Concept : Virtues and Limitations," *Proc. of VLDB*, 1981, pp.144-154.

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

[5] Korth, H. F., Levy, E., and Silberschalz, A., "A Formal Approach to Recovery by Compensating Transactions," *Proc. of the VLDB*, 1990, pp.95-106.

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

[7] Moss J. E., Griffeth, N. D., and Graham M. H., "Abstraction in Concurrency Contorol and Recovery Management (Revised)," *TR COINS 86-20, Univ. of Massachusetts*, 1986.

[8] 滝沢誠, "データベース入門技術解説," ソフト・リサーチ・センター, 1992.

[9] Takizawa, M. and Deen, S. M., "Lock-mode Based Resolution of Uncompensatable Deadlock in Compensating Nested Transaction," *Proc. of Fareast Workshop on Future Database Systems*, 1992, pp.168–175.

[10] Weikum, G. and Schek, H.-J., "Concepts and Applications of Multilevel Transaction and Open Nested Transactions," *Database Transacrtion Models for Advanced Applications*, 1992, pp.516–553.

[11] Yasuzawa, S. and Takizawa, M., "Uncompensatable Deadlock in Distributed Object-Oriented Systems," *Proc. of the International Conference on Parallel and Distributed Systems (ICPADS)*, 1992, pp.150–157.