マルチメディア通信と分散処理　83－3
（1997．6．6）

部分トランザクションの独立性を考慮した入れ子トランザクションモデル

内田 和之　多田 知正　樋口 昌宏　藤井 護

大阪大学大学院基礎工学研究科

e-mail: k-uchida@ics.es.osaka-u.ac.jp

オブジェクト指向データベースでは，処理時間が長いトランザクションが存在し，その障害時の再実行のオーバーヘッドが大きいという問題がある．この問題を解決するためのモデルとして，入れ子トランザクションモデルが提案されている．しかし従来の入れ子トランザクションモデルにおいて，スケジューリングは入れ子トランザクション全体を一つの単位として行なわなければならなかった．これに対し本研究では，スケジューリングにおいて各部分トランザクションを独立したトランザクションとして扱うような入れ子トランザクションモデルを提案した．さらに，提案モデルの下での逐次化グラフを用いたスケジューリングアルゴリズムを提案した．

A Nested Transaction Model Considering Independency of Subtransaction

Kazuyuki Uchida　Harumasa Tada　Masahiro Higuchi　Mamoru Fujii

Graduate School of Engineering Science
Osaka University

e-mail: {k-uchida,tada, higuchi, fujii }@ics.es.osaka-u.ac.jp

In object oriented database systems, when a transaction fails, long lifetime of the transaction causes a high overhead for rollback. Nested transacton has proposed to solve the problem. In the traditional nested transaction model, however, the whole nested transaction are dealt with indivisible one in scheduling. In this paper, we propose a new model of nested transaction in which each subtransaction can be scheduled as independent transaction. We also propose scheduling algolithm on our model, using serialization graph.

1 まえがき

オブジェクト指向データベースでは，存続期間の長いトランザクションが存在し，トランザクションが中断された時，再実行のオーバーヘッドが大きいという問題がある．そのような問題を解決するために提案されたモデルの一つに入れ子トランザクションモデルがある．入れ子トランザクションの並行制御に関する研究では，すでに二相ロック，逐次化グラフを用いたスケジューリング[1, 2]などが提案されている．逐次化グラフを使ったスケジューリングは，二相ロックを用いるよりも高い並行性が得られることが知られている．

しかし，従来の入れ子トランザクションモデルでは，部分トランザクションの独立性を考慮していなかった．そのため，並行な実行が制限され，高い並行性を得ることが出来なかった．本研究では，より高い並行性を得るため，各部分トランザクションの独立性を考慮した新しい入れ子トランザクションモデルを提案する．そして，新しいモデルにおける，逐次化グラフを用いたスケジューリングアルゴリズムを提案する．

以降，2. でトランザクションと逐次化可能性について説明し，3. では提案する入れ子トランザクションのモデルを，4でそのスケジューリングアルゴリズムを示し．5. でアルゴリズムの正当性について証明する．

2 準備

2.1 データベースシステム

データベースは，データ項目の集合からなり，それぞれのデータ項目は値を持っている．データ項目の値の集合がデータベースの状態を表す．データベースの状態集合のある部分集合は矛盾のない状態と定義される．

2.2 トランザクション

トランザクションとは，*Read*，*Write* 操作によってデータベースを操作するプログラムのことである．

トランザクションは実行を開始する際に *Start* 操作によってデータベースシステムにそのことを知らせる．トランザクションを終了した際には *Commit* または *Abort* によって知らせる．*Commit* 操作により，トランザクションが正常に終了したことを，*Abort* 操作により，トランザクションが異常な状態で終了したことを知らせる．トランザクションは，単独で実行された場合に，データベースの状態を矛盾のない状態から矛盾のない状態に移す (一貫性)．

2 つ以上のトランザクションを並行に実行する場合，それらのデータベース操作の相互作用 (interfere) により，データベースが矛盾した状態に陥ることがある．これは各トランザクションの実行が次の2つの条件をみたすように制御することによって，避けることが出来る．

1 複数のトランザクションが並行に実行された場合でも，それらが逐次的に実行されたのと同じ影響をデータベースに与える (独立性)．
2 トランザクションが正常に終了すればその影響はすべて保存され，そうでなければ全く影響を及ぼさない (アトミック性)．

2.3 実行履歴

複数のトランザクションの実行は、実行履歴と呼ばれるデータベース操作の半順序でモデル化される[3]．実行履歴 Hの半順序関係を$<_H$で表す．

定義 1 異なるトランザクション T_i,T_j中の操作 o_i,o_jが，同じデータ項目に対する操作で，少なくとも一方が *Write* である場合，o_i,o_jは衝突する (conflict) という．□

衝突する2つの操作 o_i,o_jには，常に $o_i <_H o_j$ 又は $o_j <_H o_i$が成り立つ．

定義 2 実行履歴 H において，*Commit*，*Abort* が H に含まれるならば，トランザクション T_i はそれぞれ commit している，abort しているという．commit も abort もしていないトランザクションは active であるという．□

定義 3 active なトランザクションT_iの操作 o_iとトランザクションT_j の操作 o_jに対して、o_iと o_jが衝突し、$o_i <_H o_j$が成り立つ場合、T_jは T_iに依存する，という．また，トランザクションTの *Commit* を除く全ての read,write 操作が Hに含まれる場合，Tは commit 待ちの状態である，という．□

2.4 逐次化可能性

実行履歴の正当性の基準として逐次化可能性 (serializability) と呼ばれる性質が一般に用いられている[3]．

定義 4 2 つの実行履歴 H_1，H_2 が以下の条件を満たす場合，H_1,H_2は等価であるという．

1 H_1,H_2に含まれる操作の集合が等しい．

2 abort していない任意のトランザクション T_i, T_jの 2 つの衝突する操作 o_i,o_jについて，$o_i <_{H_1} o_j$ならば $o_i <_{H_2} o_j$が成り立つ． □

定義 5 実行履歴 H 中の任意のトランザクション T_i , T_j について，以下の条件を満たす場合、Hは逐次的 (serial) であるという．

$$\forall o_i \in T_i \forall o_j \in T_j (o_i <_H o_j)$$
$$\vee \quad \forall o_i \in T_i \forall o_j \in T_j (o_j <_H o_i)$$

□

定義 6 実行履歴 Hは，それと等価な逐次的実行履歴 H'が存在する場合，逐次化可能 (serializable) であるという． □

2.5 逐次化グラフ

実行履歴が逐次化可能であるかどうかは，逐次化グラフ (serialization graph) を用いて確かめられる[4]．

定義 7 実行履歴 H についての逐次化グラフ $SG(H)$ は以下のように定義される有向グラフである．

- 頂点 $V = \{T | T \in H \wedge commit \in T\}$
- 辺 $E = \{(T_i, T_j) | o_i \in T_i, o_j \in T_j$ が衝突しかつ，$o_i <_H o_j\}$

□

定理 1 実行履歴 H が逐次化可能であるための必要十分条件は，$SG(H)$ が閉路を持たないことである[4]． □

2.6 逐次化グラフを用いたスケジューリングアルゴリズム

スケジューラはトランザクション T_iのデータ項目 x への操作 $p_i[x]$ を受けとると，まず初めにトランザクション T_i に対する頂点が逐次化グラフに存在するかどうか調べ，存在しなければそれを加える．次に $p_i[x]$ と衝突していてすでに実行されている操作 $q_j[x]$ を含む全てのトランザクション T_jに対して，T_jから T_i への辺を加える．これにより，閉路が出来ない場合に $p_i[x]$ を実行し，閉路が出来る場合は閉路を形成している活動中のトランザクションのうちいくつかを中断し，中断されたトランザクションについての頂点と隣接する辺を取り除くことで逐次化グラフを閉路のない状態にする．

3 拡張入れ子トランザクション

3.1 入れ子トランザクション

入れ子トランザクションはトップレベルトランザクションと部分トランザクションから成り立つ．トップレベルトランザクションは部分トランザクションを入れ子型に含む．部分トランザクションは一貫性を満たさない．

定義 8 トップレベルトランザクション又は部分トランザクション Tが，部分トランザクション T'を起動したとき，Tは T' の親，T'は Tの子 $(T \Rightarrow T')$ であるという．$T \overset{*}{\Rightarrow} T'$が成り立つとき Tを T'の先祖，T'を Tの子孫と呼び，T,T'は *directly related*$(T \Leftrightarrow T')$ であるという．また T',T''が $\exists T((T' \not\Leftrightarrow T'') \wedge (T \overset{*}{\Rightarrow} T') \wedge (T \overset{*}{\Rightarrow} T''))$ を満たす場合，*indirectly related*$(\rightleftharpoons)$ であるという．

部分トランザクション Tの親は，Tを起動すると自分自身の操作を中断し，Tからの操作の終了の message を待って操作を再開する．Tは全ての read,write 操作が終ると親に message を送る．部分トランサクションは commit は行なわない．トップトランザクションは全ての操作が終ると，commit を行なう．Tが abort した場合，親に伝え，親は Tを再実行する．トップレベルトランザクションまたは部分トランザクション Tが abort した場合，その子孫である T'は abort される．

入れ子トランザクションの特徴として，*finer grain recovery* が挙げられる．従来のトランザクション Tにおいては，ある操作に失敗すると，Tは最初から再実行される．オブジェクト指向データベースにおいて，トランザクションは比較的生存時間が長く，トランザクションが abort した場合，トランザクション全体を再実行すると再実行のオーバーヘッドが大きくなる．入れ子トランザクションを用いることにより，部分トランザクション T'が abort したとしても T'のみが再実行され，その他の部分には影響を与えないので，再実行のオーバーヘッドを小さくすることが出来る．

入れ子トランザクションの動作の説明のためにDR グラフを導入する．DR グラフは入れ子トランザクションの各部分トランザクションを頂点とし,directly related を二重線 (directly related edge) で,conflict edge を矢印で, 表したグラフ (図 1) である．

3.2 従来のモデルの問題点

従来の入れ子トランザクションモデルではトランザクション全体が終了した時にのみデータ

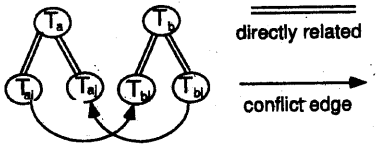

図 1: DR グラフ

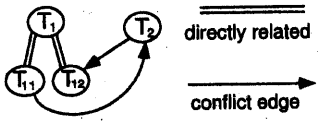

図 2: デッドロックの回避

ベースの一貫性を保存するモデルであった。各部分トランザクションはデータベースの無矛盾性を保存することは保証していなかった．そのためスケジューリングは入れ子トランザクション全体の逐次化可能性を考えなければならなかった．

ここで A 社，B社がある金額を複数の人の口座に振り込むという作業を考える．この作業は，トップレベルトランザクション T_a，T_bがそれぞれ人数分の子を起動し，それぞれの子が一人の振り込みを行う，として実現出来る。

この作業は，部分トランザクションが単体で行っても，データベースは無矛盾であり，部分トランザクションは一貫性を満たすので，図1のようなスケジューリングを行ってもデータベースは無矛盾である．しかし，今までのモデルでは T_aと T_bの逐次化可能性を考えていたので，図1のようなスケジューリングは行えなかった。

このようなスケジューリングを行なえるモデルを考えると，並行性を高めることが出来る．

3.3 独立入れ子トランザクション

本研究では，部分トランザクションが一貫性を満たす入れ子トランザクションモデルを考える．部分トランザクションを，従来のトランザクションとするモデルである．このモデルを独立入れ子トランザクションと呼ぶ．このモデルでは，部分トランザクションの逐次化可能性を考慮した，スケジューリングを行なうことが出来る．これによって，図1のようなスケジューリングが行えるので，並行性を高めることが出来る．なお，本研究では、directly related なトランザクションは同じ項目を操作することはなく，並行に動作できるものとする．

3.4 v-commit

通常，あるトランザクション Tがあるデータ項目 x に対する操作を行い，その後別のトランザクション T'が x に対する操作を行った場合，T'は Tが commit するまで commit 出来ない．図2の例では，T_{11}、T_2、T_{12}という順に commit することになる．しかし T_{11}は T_1によって abort される場合があるので commit できずデッドロックとなる。これは T_{11}が親トランザクションによって abort される可能性がない場合に，T_{11}を commit させると，デッドロックを回避できる．そこで，v-commit という概念を導入する．

定義 9 独立入れ子トランザクションの部分トランザクション Tは，以下の条件を全て満たす時かつその時のみ v-commit を行なう．

- Tが commit 待ちである．
- 他のトランザクションに依存しない．
- Tがトップレベルトランザクション以外の場合、Tの先祖が v-commit を行っている。□

この v-commit を用いて，独立入れ子トランザクションの commit の定義を行う．

定義 10 トランザクション Tは，以下の条件を全て満たす時かつその時のみ commit する．

- Tが commit 待ちである．
- Tの全ての先祖が v-commit を行っている．
- Tの全ての子孫が commit している．
- Tは他のトランザクションに依存しない．□

図2の例の場合，トランザクション T_1が commit 待ちになり，T_1の依存するトランザクションが commit すると，T_1が v-commit し，T_{11}が commit する．その後，T_2，T_{12}と commit して，T_1が commit 出来るようになる．

4 スケジューリングアルゴリズム

4.1 逐次化グラフの問題点

従来の逐次化グラフはトランザクションを頂点として，衝突するトランザクション間にのみ辺を引くものであった．しかし，入れ子トランザクションにおいては，従来の逐次化グラフで閉路が出来ない場合でも，デッドロックとなる場合がある。例として次のような実行を考える．T_{11}が write した値を T_{11}に directly related でない T_2が read した後，T_{11}の先祖である T_1が write を行う．この場合の DR グラフは図 3 のように

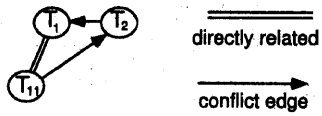

図 3: デッドロック

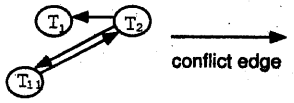

図 4: 拡張逐次化グラフ

なる．この場合，T_1が v-commit するまでT_{11}は commit できない．しかしT_1はT_2が commit しなければ v-commit できず，T_2はT_{11}が commit しなければ commit できない，というデッドロックになる．このような実行を避けるために逐次化グラフに変更を加える必要がある．

4.2 拡張逐次化グラフを用いたスケジューリング

2.5節のアルゴリズムで、スケジューラはトランザクションT_iの操作o_iを受けとった時に逐次化グラフに辺を加える手続きを，以下のように変更する．

- トランザクションT_iの操作o_iが，もしトランザクションT_jの命令o_jと衝突し，o_iよりもo_jが前にスケジューリングされているならば，$T_j \rightarrow T_i$を加える．T_iのすべての子孫T_kにも$T_j \rightarrow T_k$を加える．
- もしトランザクションT_iが子トランザクションT_jを新たに開始した場合，$T_k \rightarrow T_i$があれば，$T_k \rightarrow T_j$を引く．

図 3の DR グラフに対応する拡張逐次化グラフを図 4に示す。

5 正当性

5.1 デッドロックの回避

補題 1 拡張逐次化グラフを用いたスケジューリングはデッドロックフリーである．

[略証] 拡張逐次化グラフに閉路が存在しなければ，その部分集合である逐次化グラフにも閉路が存在しない．入れ子トランザクションでなければ，トランザクションTが待つのは，Tが依存しているトランザクションのみであり，依存関係にあるトランザクション間には，必ず conflict edge が存在するため，逐次化グラフで閉路を含まない場合，デッドロックを生じない[4]．しかし独立入れ子トランザクションにおいては，トランザクションTは commit する際，親が v-commit し，全ての子が commit するのを待つ必要があるので，逐次化グラフに閉路がない場合でもデッドロックが生じる場合がある．デッドロックになるのは入れ子トランザクションを１つの頂点としたとき，依存関係のサイクルが出来る場合であり，図 5の様な状態となる．図のT_iに入ってくる辺をT_iの in-edge, 出ていく辺をT_iの out-edge とする．このときそれぞれの頂点に対応する入れ子トランザクションTの拡張逐次化グラフで，Tの in-edge から Tの out-edge の conflict edge の経路が存在するか，Tの out-edge の始点のトランザクションT_{out}が commit するならば，デッドロックが回避できることを以下に述べる．

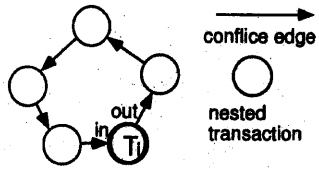

図 5: デッドロック

もし，T_{out}が commit するならば，そこからデッドロックが回避できる．デッドロックになるのは，全てのT_{out}が commit できない場合である．そのときには，全てのTで in-edge から out-edge への conflict edge の経路が存在し，全体で confict edge の閉路が出来ることになる．よって，そのようなスケジューリングはアルゴリズムによって回避される．

頂点に対応する入れ子トランザクションの内部の状態は、in-edge,out-edge の場所により、以下の 4 つに場合分け出来，それぞれの場合について，in-edge から out-edge への conflict edge の経路が存在するか，T_{out}が commit 出来るかのどちらかは成り立つことを証明する．以下略．

1 DR グラフにおいて，in-edge があるトランザクションTに入り，Tと *indirectly related* なトランザクションT_{out}から out-edge が出ている場合 (図 6(a))

2 DR グラフにおいて，あるトランザクションTへ in-edge が入り，Tの先祖であるトランザクションT_{out}から，out-edge が出てい

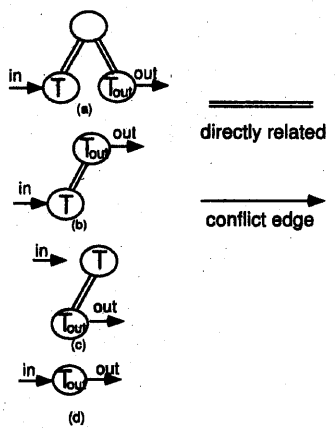

図 6: 内部の状態

る場合 (図 6(b))

3 DR グラフにおいて，あるトランザクション Tに in-edge が入り，Tの子孫であるトランザクション T_{out}から out-edge が出ている場合 (図 6(c))

4 トランザクション T_{out}に in-edge が入り，out-edge が出ている場合 (図 6(d))

5.2 データベースの無矛盾性の保存

補題 2 提案した手法を用いた時，実行履歴は部分トランザクション単位で逐次化可能である．

[証明] データベースにおいて初期状態では，逐次化グラフは部分トランザクションを表す頂点を1つも持たない．したがって，明らかに閉路を含まない．

提案した手法において，逐次化グラフに存在する辺は、拡張逐次化グラフにも常に存在するので、拡張逐次化グラフに閉路が存在しなければ、逐次化グラフも閉路を形成しない．また、補題1より、デッドロックが生じないので、定理1より，提案した手法により生成される実行履歴は常に部分トランザクション単位で逐次化可能である． □

定理 2 提案したアルゴリズムを用いた時，データベースの無矛盾性は保存される．

[証明] 3節 で提案した入れ子トランザクションモデルでは，各部分トランザクションの実行はデータベースを一貫した状態から一貫した状態に移すことを仮定した．よって補題 2 より提案した手法を用いた時，データベースの無矛盾性が保存されていることが示される． □

6 まとめ

入れ子トランザクションの並行制御において従来の方法では，本来並行に実行できる操作を並行に実行できない場合が存在した．本研究では，そのような並行実行を行なえるよう入れ子トランザクションの各部分トランザクションを独立なものとしてスケジューリングすることを検討し，新しい入れ子トランザクションモデルを提案した．さらに提案したモデル上での逐次化グラフを用いたスケジューリングアルゴリズムを提案した．提案したスケジューリング手法ではモデルの変更に伴って起こりうるデッドロックに対処するため v-commit の概念を導入している．提案した手法により、従来行えなかった実行を行えるようになり、高い並行性が得られた．

参考文献

[1] L. Daynes, O. Gruber and P. Valduriez: "Locking in OODBMS Client Supporting Nested Transaction", IEEE the 11th intl. conf. on Data Engeneering, pp.316-323(1995).

[2] S. Ben-Hassen and M. Rusinkkiewicz: "On Serializability of Distributed Nested Transaction", IEEE The 12th intl. conf. on Distributed Computing Systems, pp.152-159(Jun. 1992).

[3] Bernstein, P.A.,et al., "Formal Aspects of Serializability in Database Concurrency Control", IEEE trans. Softwre Eng., Bol.SE-5, No.3, pp.203-216, May 1979.

[4] P. A. Bernstein, V. Hadzilacos and N. Goodman: "Concurrency Control and Recovery in Database Systems", Addison-Wesley,(1987).