時制データベースによる長時間トランザクションの直列化方式

Serialization Method of Long-time Transaction by Temporal Database

工藤　司*
Tsukasa KUDO

Abstract: In business systems, databases are usually updated by two methods: the online entry is used for update from many terminals concurrently; the batch update is used to update a great deal of data in a lump-sum. To execute the latter as the transaction with maintaining the ACID properties, we had proposed the temporal update method utilizing the temporal database. However, in order to construct the serializable schedule, there was the problem that some online entry transactions are delayed. In this study, I propose a novel serialization method for this problem.

1．はじめに

業務システムではデータベースの更新は大きく 2 つの処理に分類される．第一はオンライン端末からの入力（以下，「オンライン入力」）であり，短時間のトランザクションにより入力は即時にデータベースに反映される．例えば，金融機関における ATM(Automatic Teller Machine)やネットショップの購買のように，多くのサービスはオンライン入力によって実行され，同時に多数のユーザへサービスを行うためロックによる同時実行制御が行なわれている．第二はバッチ処理による大量データの一括更新（以下，「バッチ更新」）であり，長時間に及ぶため半自動化されている．この事例としては，金融機関における大量の口座振替があり，トランザクションとして処理するためには長時間に渡りデータをロックする必要がある．これは，オンライン入力を長時間の待ち状態にするため，かつては 2 つの処理は別の時間帯に実行されていた．

ところが，ネットショップや ATM を始めとして，業務システムのノンストップサービスが普及しており，現在では両者を並行して実行することが必要になっている．このため，バッチ更新を短時間のトランザクションに分割して順次実行するミニバッチが広く使用されているが，この方式ではバッチ更新全体としてトランザクションの ACID 特性が維持できないという課題がある [1]．

この課題に対し，筆者らは先行研究において時制更新方式と呼ぶバッチ更新方式を提案した [15]．本方式はデータの履歴に関する時間の概念を未来に拡張し，未来の時刻を指定してバッチ更新を行うことで，現在時刻に対する更新であるオンライン入力との競合を避けるものである．これにより，バッチ更新を長時間のトランザクションとしてオンライン入力と同時に実行できることを示した．本技術は特許として登録されており [2]，さらに，相互に関連するデータの更新や，分散データベース環境への応用に有効であることを示してきた [3-4]．

一方で，バッチ更新をトランザクションとして実行するためには，オンライン入力のトランザクションとの間で直列化可能スケジュール [5] を構成する必要がある．従って，バッチ更新をオンライン入力と同時実行しても，その更新結果を有効にするコミットは，オンライン入力が行われていない状況で実行されることが必要になる．すなわち，コミットは先行する全てのオンライン入力の完了を待ち，さらにコミットに後続するオンライン入力はコミットの完了を待って実行を開始必要があるため，オンライン入力に遅延が発生するというという課題があった．

本研究の目的は，この課題に対して複数のオンライン入力が実行中であっても，バッチ更新のコミットに伴う待ち時間を発生させることなく，バッチ更新とオンライン入力を直列化可能スケジュールとして実行できる時制更新方式を提供することである．

本論文では，2 章で従来のバッチ更新方式の課題を示し，3 章で待ち時間を抑止した新たな時制更新方式を提案する．4 章で本方式の実装と評価を示し，5 章で評価結果について考察する．

2．関連研究

2.1　従来のバッチ更新方式と時制更新方式

トランザクションの同時実行制御に関しては様々な方法が提案されている [6-7]．現在，広く使用されているロックを使用する方法では，トランザクションがアクセスするデータを事前にロックし，競合するトランザクションを待ち状態にすることにより直列化可能スケジュールを構成する [5]．すなわち，実行結果が各々のトランザクションを直列に実行した場合と同様になるように制御する．従って，長時間のバッチ更新に適用した場合には，オンライン入力には長時間のロック解除待ちが発生するという課題があ

2017 年 2 月 3 日受理
*　総合情報学部　人間情報デザイン学科

る．逆に，ミニバッチとして短時間のトランザクションに分割し，短時間のロックにより順次更新を行う方式では，バッチ更新全体としてトランザクションの ACID 特性，すなわち原子性，一貫性，分離性，持続性が維持されないという課題がある[1),8)]．また，時刻印方式や，商用のデータベースで実装されているスナップショット分離レベルに基づく多版型同時実行制御がある[9)]．しかし，これらの方法では他のトランザクションが対象データを更新していた場合にはアボートするか，あるいはロック法と同様に明示的に排他ロックを行う必要があるため同様のロック解除待ちが発生するという課題がある．このように，長時間に及ぶバッチ更新をトランザクションとして安定的に実行できる方法は見当たらなかった．

この課題に対し，筆者らは時制更新方式を提案した[10),15)]．これは，時間の履歴を管理する時制データベースの 1 つであるトランザクション時間データベース[11)]の概念を使用している．トランザクション時間データベースは，ある事実がデータベースに存在した時間であるトランザクション時間によりデータの履歴を管理する．従って，データベースへのデータの追加，修正，削除の一連の操作が時間に関する履歴として保存される．本方式では，過去の履歴を対象としてきたトランザクション時間の概念を未来に拡張し，バッチ更新とオンライン入力の競合を回避する．なお，時制データベースはさまざまなデータベース管理システム（以下，「DBMS」）上での実装が提案されているが[14)]，本研究はリレーショナルデータベース管理システム（以下，「RDBMS」）上での実装を前提としている．

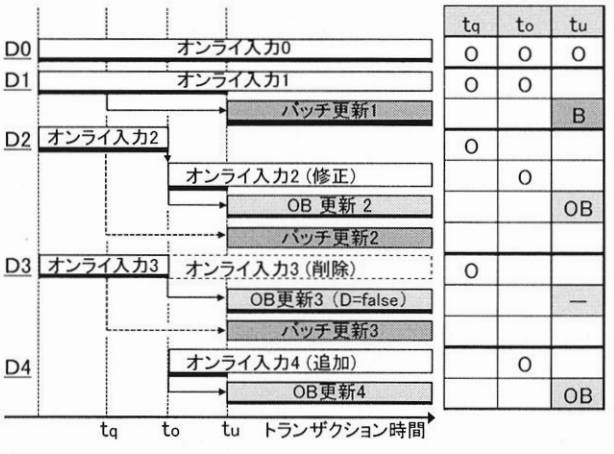


図 1　時制更新方式におけるデータ履歴

図 1 に，本方式により，現在時刻t_0でオンライン入力とバッチ更新を同時に実行した事例を示す．バッチ更新は過去の時刻t_q時点のデータを読み出し，未来の時刻t_uに更新結果を保存する．さらに，オンライン入力対象のデータに対しては，個別にオンライン入力の結果に対するバッチ更新を実行し，同じく時刻t_uに保存する．これをオンライン・バッチ更新（以下，「OB 更新」）と呼ぶ．すなわち，オンライン入力が現在時刻t_0時点のデータを更新するのに対し，バッチ更新は過去の時刻t_qと未来の時刻t_uのデータを対象とする．この結果，各々のトランザクションは競合せずに，時刻がt_uになった時点で，オンライン入力，バッチ更新，OB 更新の各データがデータベースに保存されることになる．図 1 の右の表は，時刻t_q，t_0，t_uの各々の時点で検索されたデータを示す．ここで，「O」，「B」，「OB」の表記は，各々，オンライン入力，バッチ更新，OB 更新の結果であることを示す．すなわち，有効なデータのみを検索することで，オンライン入力とバッチ更新を直列に実行した場合と同様の結果を得ることができる．

本方式では上記の通りトランザクション時間データベースを使用しているため，テーブルのリレーションは式(1)で表現される．

$$R(K, T_a, T_d, P, D, A) \qquad (1)$$

- K：主キー属性．これは，ある時点のスナップショットである属性集合 $\{K, A\}$ に射影した場合の主キーを構成する．
- T_a：追加時刻．データがデータベースに追加された時刻．
- T_d：削除時刻．データ削除された時刻であり，T_d に削除時刻を設定して論理的に削除することによりデータの履歴は残される．なお，データが削除されていない場合には，時間の推移とともに変化する現在時刻を表すnow[12)]が設定される．
- P：更新区分．データが OB 更新，オンライン入力，バッチ更新のいずれによって行われたかの区分を設定する．各々の属性値の集合を $\{p_{ob}, p_b, p_o\}$ で表現するとき，これらは全順序集合 $p_{ob} > p_b > p_o$ を構成し．図 1 に支援するように，この順序により検索の優先順位を決定する．
- D：削除区分．検索されたデータが検索対象か対象外かを示す区分を $\{true, false\}$ で示す．図 1 の「D3」に示すように，オンライン入力のデータ削除に伴う OB 更新では $D = false$ に設定され，バッチ更新結果が検索されない．
- A：その他の属性集合．

なお，以下の事例では金融機関における預金口座の残高を想定しており，この場合には式(1)のKは口座番号，Aは預金残高となる．

2.2　従来の時制更新方式の課題

時制更新でバッチ更新をトランザクションとして実行するためには，オンライン入力との間で同時実行制御を行い，直列化可能スケジュールを構成する必要がある．図 2 にオンライン入力とバッチ更新の直列化可能スケジュールの事例を示す．ここで，各々のT_iはオンライン入力のトランザクションでありオンライン入力O_i，OB 更新OB_iから

構成され最後にコミットが行われる．また，cはバッチ更新のコミットを示し，cの実行によりOB_i，およびバッチ更新の結果が有効になる．ここで，図2のスケジュールではオンライン入力とバッチ更新の競合するデータ操作の順序は変更されない．すなわち，このスケジュールは競合直列化可能スケジュールである．従って，バッチ更新のコミット前に実行されるバッチ更新と競合するオンライン入力をT_j，同じくバッチ更新後のオンライン入力をT_k，O_jとO_kのデータの読出しと書込みの操作を各々r_jとw_j，r_kとw_k，また，バッチ更新およびOB_iの，cによる書込みをw_bとすると，直列スケジュールは以下で表現できる．

$$S: r_j w_j w_b r_k w_k \qquad (2)$$

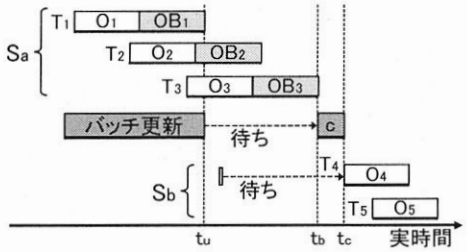


図2　時制更新における直列化可能スケジュール

ここで，図2の事例では，上記の通りw_bはバッチ更新のコミットcの実行で有効になる．すなわち，直列化可能スケジュールを構成するためには，cはバッチ更新と競合する全てのT_jが完了するのを待つ必要がある．逆に，T_kはcの結果を読込む必要があるため，T_jおよびcの完了を待つ必要がある．

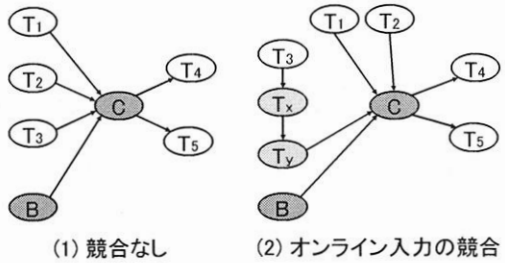


図3　バッチ更新のコミットに関する先行グラフ

次に，T_kで発生する待ち時間を検討するため，図2の先行グラフを図3の(1)に示す．この事例では，オンライン入力の間に競合がないと仮定する．ここでcは全ての先行するT_jの完了を待つため，ここにボトルネックが発生することが分かる．ただし，cが待つのはバッチ更新のデータ操作Bの完了以前に開始されたT_jの完了のみとなる．したがってT_k（図2ではT_4）はcの完了と同時に開始されるため，待ち時間の最大値は，オンライン入力トランザクションの最大実行時間を関数max_timeで示すと，これとバッチ更新のコミットの時間の合計，すなわち以下となる．

$$\max\_\text{time}(O_j + OB_j + c) \qquad (3)$$

一方で，実際のオンライン入力では相互に競合するオンライン入力のトランザクションが発生する．この事例として図3の(2)に，T_3がT_x，T_yと競合し，順次実行された場合を示す．この場合にはcは3つのトランザクション全ての実行完了を待つ必要があり，かつ，これらのトランザクションは直列に実行される．すなわち．あるデータに対して同時に発生する競合の最大数をNとするとT_kの待ち時間の最大値は以下となる．

$$\max\_\text{time}(O_j + OB_j) \times N + \max\_time(c) \qquad (4)$$

すなわち，この場合にはT_kは本来競合しないオンライン入力T_jの完了を待つだけでなく，T_jに競合に伴う遅延が発生した場合には待ち時間が増大するという課題がある．特に，バッチ更新において大量のデータを更新する場合には，そのうちの1つでも長時間のオンライン入力と競合するならば，バッチ更新以降の競合する全てのオンライン入力にも長時間の待ちが発生するという課題があった．

3．提案する直列化方式

3.1 バッチ更新に伴うオンライン入力の更新手順

2章の課題に対し，先行するオンライン入力の完了を待つことなく，バッチ更新のコミットを実行するための方式を提案する．本提案では，一般的に使用されている多版型同時実行制御[5]を対象とする．すなわち，分離レベルはスナップショット分離レベル[13]であり，読込みではトランザクションは開始時点でコミット済のデータを検索する．

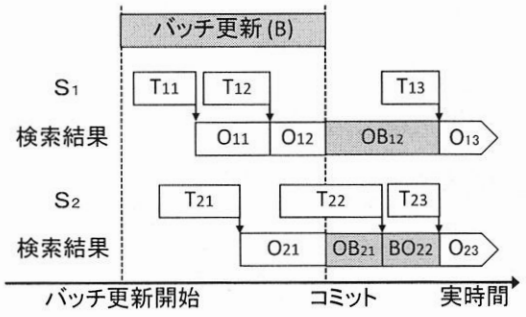


図4　同時実行に伴う検索結果

まず，時制更新に伴うスケジュールは，図4に示すようにコミット時点で先行しているオンライン入力が完了しているS_1と，オインライン入力が未完了であるS_2に分けられる．図4のS_1，S_2では上段にオンライン入力トランザクションT_{im}の実行のタイミングを，下段に時間の推移に伴って検索されるデータを示す．ここで，iはスケジュール番号を，O_{im}はトランザクションT_{im}のオンライン入力の結果を示す．なお，OB更新については，オンライン入力とバ

ッチ更新の実行順序により2つの結果を用いる．一方は図のOB_{im}（具体的には，OB_{12}，OB_{21}）でありオンライン入力の結果にバッチ更新が行われたことを示す．他方は図のBO_{im}（同じく，BO_{22}）であり，バッチ更新の結果に対してオンライン入力が行われたことを示す．

これらのスケジュールを式(2)と同様に表現すると以下となる．

$$S_1: r_{11}w_{11}r_{12}w_{12}w_b r_{13}w_{13} \quad (5)$$

$$S_2: r_{21}w_{21}w_b r_{22}w_{22}w_{23}w_{23} \quad (6)$$

すなわち，S_1ではT_{11}，T_{12}の実行後にバッチ更新，T_{13}が実行するのに対し，S_2ではT_{21}の実行後にT_{22}，T_{23}を実行するが，T_{22}の実行に先立ってバッチ更新を実行する．ここで，スケジュールS_1のデータ操作は従来の時制更新と同様であり，図1に示すようにバッチ更新の終了と同時にオンライン入力にバッチ更新を反映した結果を得ることができる．なお，T_{12}はスケジュールに示されるように，T_{11}のオンライン入力の結果を読込む．

一方で，S_2ではバッチ更新のコミット時点で，一旦，バッチ更新の結果を有効にする必要がある．従って，T_{21}のオンライン入力の結果O_{21}に対してバッチ更新を行ったOB更新の結果であるOB_{21}が有効になる．T_{22}はこのOB更新の結果を読込み，これに対するオンライン入力の結果を書込む必要がある．ところが，コミット時点では既にT_{22}は実行中であるためr_{22}が行われている．このため，従来の時制更新方式ではT_{22}の完了を待ってバッチ更新のコミットを行う方式を採用しており，この結果，後続のオンライン入力T_{23}に遅延が発生してしまうという課題があった．

そこで，本提案ではOB更新において，まずバッチ更新を行った上でオンライン入力を行うBO_{im}のデータ操作を追加することによりスケジュールS_2を構成し，実行中のオンライン入力T_{22}の完了を待たずにバッチ更新のコミットを行うことを可能にする．

表1　バッチとの競合に伴うオンライン入力の操作

区分	競合の有無（注）		更新結果
	実行中	終了時	
0	−	−	$O_{im}: O(d_c)$
1	有	−	$O_{im}: O(d_c) \rightarrow B: OB(O(d_c))$
2	有	有	$B: B(d_c) \rightarrow O_{im}: O(B(d_c))$

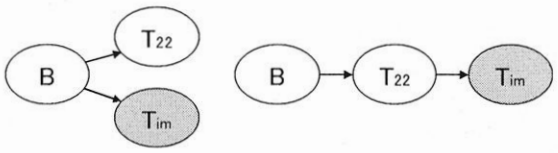
注：「実行中」，「終了時」はバッチ更新の実行中と終了時

表1にバッチ更新との競合に伴うオンライン入力のデータ操作を示す．区分0は，バッチ更新が行われていないか，あるいはバッチ更新対象外のデータを操作するオンライン入力のトランザクションであり，この場合にはオンライン入力のデータ操作O_{im}のみを行う．なお，関数$O(d_c)$はオンライン入力トランザクションが開始時点でコミットされていたデータを読込み，更新した結果を書込むことを示す．区分1はオンライン入力がバッチ更新と競合する場合のうち，バッチ更新の終了前にオンライン入力が終了するものであり，従来のOB更新を行う．ここで，$B: OB(O(d_c))$はバッチ更新のコミット時点でOB更新の結果を書込むことを示す．それまでは，オンライン入力$O(d_c)$の結果が有効になっている．

区分2が本提案で追加したデータ操作であり，オンライン入力がバッチ更新のコミット時点でも実行中の場合である．この場合には，このコミット時点で，バッチ更新開始時点すなわち図1のトランザクション時刻がt_q時点のデータに対するバッチ更新結果が有効になり，オンライン入力完了時点でこのバッチ更新結果に対してオンライン入力を行った結果が有効になる．なお，図4の事例では，バッチ更新中にオンライン入力が行われているため，バッチ更新の代わりにOB更新の結果が有効になる．

3.2 スケジュール直列化可能性

表1の「更新結果」は式(5)および(6)に示すスケジュールに沿って実行されている．すなわち，競合するデータ操作の順番が直列スケジュールと等価な競合直列化可能スケジュールになる．ここで，図4でバッチ更新のコミット後もT_{22}が継続して実行されている．このトランザクションについても競合直列化可能スケジュールに影響を与えないことを示す．対象のスケジュールが競合直列化可能スケジュールであることは，図3に示すような先行グラフを作成し，これに閉路が存在しないことで判定できる[5]．



(1) 読込みトランザクション　(2) 書込みトランザクション

図5　競合するトランザクションとの先行グラフ

まず，図5(1)に示すように，T_{22}と同時に実行されるオンライン入力のトランザクションのT_{im}がT_{22}と競合しないか，競合するデータの読込みのみであれば，多版型同時実行制御の特性からコミット済のデータ，すなわちバッチ更新Bの結果を読込む．従って，(1)に示すように先行グラフに閉路は存在しない．逆に，図5(2)に示すようにT_{22}と競合す書込みを行うトランザクションは．ロックによる待ちのためT_{22}のコミットを待ってから読込むことになる．すなわち，読込み対象は$O_{22}: O(B(d_c))$であり先行するバッチ更新，オンライン入力T_{22}が直列に実行された結果を読込むことになる．従って，図5に示すように先行グラフに閉路は存在しない．また，T_{22}の完了後に実行されるトランザクションについては，当然ながら図5(2)の先行グラフになる．以上のように，全ての場合について先行グラフ

に閉路が存在せず，競合直列化可能スケジュールが構成される．

3.3 バッチ更新の ACID 特性の維持

次に本提案方式でオンライン入力，バッチ更新の各々のトランザクションの ACID 特性が維持できることを示す．2.1 節に示した通り本方式は RDBMS 上での構築を前提としており，持続性は RDBMS の持続性によって保証される．さらに，各々のオンライン入力は 3.1 節に示すようにスナップショット分離レベルのトランザクションとして実行されており，RBMS の特性により原始性，分離性，整合性が維持される．

また，バッチ更新においても 3.2 節に示すようにオンライン入力との間に直列化可能スケジュールが構成されている．すなわち，図 3 に示すようにコミット時点でバッチ更新結果がオンライン入力の間で実行されたものと同様のスケジュールを構成しており，分離性が維持される．従って，整合性の維持を確認した上で，ロールバック，コミットを選択することができる．ここで，バッチ更新，OB 更新の結果はバッチ更新のコミットに伴い有効となる．すなわち，コミット操作を行わなければバッチ更新，OB 更新の結果は有効にならない．すなわち，ロールバックのデータ操作が可能になる，従って，原始性，整合性も維持することができる．

以上ように，本方式ではオンライン入力，バッチ更新の双方を，ACID 特性を維持したトランザクションとして実行することができる．

4．直列化方式の実装と評価

提案した直列化方式によりバッチ更新の ACID 特性が維持されることの検証と，図 3 に示す従来方式との効率の比較評価のため，リレーションの預金口座を対象としたプロトタイプを実装し実験を行った．

4.1 実装

この実装では，RDBMS は MySQL 5.7，トランザクション機能は MySQL のデータベースエンジン InnoDB，プログラムは Java1.8.0_92，DBMS と Java のインタフェースは JDBC ドライバを使用した．

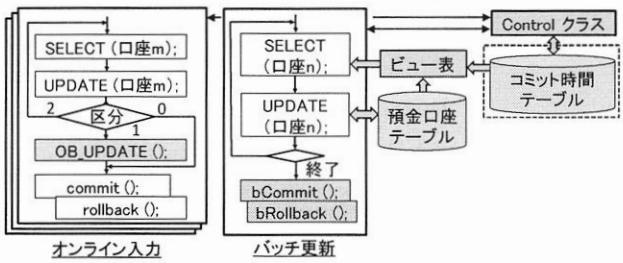


図 6　プロトタイプのソフトウェア構成

プロトタイプのソフトウェア構成を図 6 に示す．預金口座テーブルは業務データである口座番号，残高を保存し，コミット時間テーブはバッチ更新のコミットの時間を管理する．バッチ更新後はビュー表を経由して検索することで図 1 に示す有効データのみを検索する構成にする．なお，預金テーブルの更新は直接テーブルを操作する．アプリケーションプログラムの同時実行制御は図 6 の Control クラスを実装し，アプリケーションプログラムであるオンライン入力，バッチ更新のプログラムから，このクラスの sync メソッドを呼び出すことで実行する．なお，各プログラムの同時実行は Java のスレッドで実装し，sync メソッドは直列化可能スケジュールを構成するため Synchronized キーワードを付加して実装，同期化した．

図 6 で，オンライン入力は 1 件の口座に対して読込みと更新結果の書込みを行い，コミットの直前にバッチ更新の実行状況を確認して表 1 の区分に従って処理を実行する．ここで，バッチ更新の状況はトランザクション時間に基づいて確認する．すなわち，オンライン入力のコミットのトランザクション時間は実時間を使用しているが，バッチ更新完了時刻は未来のトランザクション時刻であるため，この時刻までは区分 1 として処理する．従って，バッチ更新の開始，終了では実時間より遅い時間を設定し，処理順序の逆転を防いでいる．なお，区分 2 の場合でテーブルに書込み済みの場合には，一旦，書込みを取消し，バッチ更新の結果を読み出して処理を再実行する．

図 7　バッチ終了時刻の変換方式

```
-- 更新区分Pを取得するビュー表の作成
create view P_v as select K, if (Tu, max(P), 0) as P
from 預金口座 as G2 left outer join コミット時間 as C2
using (Ta) where G2.Td ='99999999999999999' group by K;

-- 預金口座のビュー表の作成
create view 預金口座ビュー as select K, Td, P,
coalesce (C1.Tu, G1.Ta) as Ta, D, A from 預金口座 as G1
left outer join コミット時間 as C1 using (Ta)
where Td='99999999999999999' and D=0
and P = (select P from P_v G2 where G1.K=G2.K) order by K;
```

図 8　ビュー表を作成する SQL 文

バッチ更新は多数の口座に対して金額の読込みと更新結果の書込みを順次行い，最後にバッチ更新のコミット（bCommit）あるいはロールバック（bRollback）を実行す

る．更新データはバッチ更新終了のトランザクション時刻t_uで保存する．ここで，バッチ更新は長時間に及ぶため，終了時刻は予測できない．そこで，図7に示すように預金口座テーブルでは先頭を「@」とした仮の時刻を保存し，コミット時間テーブルと結合することで時間の変換を行う．このビュー表は図8に示すSQL文で作成した．すなわち，本実装ではバッチ更新のコミットはコミット時間テーブルにバッチ更新終了時間であるT_uを設定する処理である．この時間の設定によりバッチ更新およびOB更新結果が有効になり，図1に示す優先順位で検索される．

図7でコミット時間テーブルの$T_{_id}$はバッチの番号であり，$T_{_id}=1$はコミット済のため，預金口座テーブルのT_aの値がT_uの値に変換され，検索可能になる．一方，$T_{_id}=2$はコミット前であり，預金口座テーブルの$K=2$のデータはバッチ更新により追加されているため，検索対象にはならない．また，図8で現在時刻nowは時間の最大値として実装している．

なお，オンライン入力，バッチ更新共に，コミット，ロールバックは直列化可能スケジュールを構成する必要があるため，Syncメソッドで直列に実行した．なお，バッチ更新では完了後に不要なデータを削除する．例えば，コミットでは，図1のD2に示すようにバッチ更新とOB更新の結果が保存されるため不要なバッチ更新結果は削除し，オンライン入力には削除時刻にt_uを設定する．また，ロールバックではバッチ更新自体が取り消されるため，バッチ更新，OB更新の両方の結果を取り消し，オンライン入力結果のみを残す．

4.2 実験1：ACID特性維持の検証

プロトタイプを使用して実験と評価を行った．実験はスタンドアロンのワークステーションで行い，CPUはCire i7-6700 (3.6GHz)，メモリ16GB，OSはWindows 10 Pro (64bit)，ディスクは512GB SSDである．

まず，実験1として，ACID特性が維持されていることを検証するため，口座番号1から500の口座を対象に，以下の実験を行った．最初に，各口座の残高には初期値として$10{,}000+口座番号\times 100$を設定しておく．次に，idが1から5の5本のオンライン入力プログラムと，1本のバッチ更新プログラムを並行して実行し，オンライン入力では各々1から100，101から200というように連続する100件の口座に4000を加算した．バッチ更新では口座番号の昇順に全ての口座の残高を1/2にした．

また，オンライン入力実行中の状況を作るため，図6のオンライン入力プログラムのselect命令とupdate命令の処理の間に50ミリ秒の遅延を入れて競合を発生しやすくし，さらにコミットあるいはロールバック後には直ちに次の処理を開始する構成にした．バッチ更新はオンライン入力との間で直列化可能スケジュールを構成できることを検証するため，オンライン入力開始から1秒の遅延を経て開始し，オンライン入力プログラムの実行中に完了する構成にした．なお，バッチ更新の開始と終了のトランザクション時刻は実時刻の1秒後とした．実験としては，コミットとアボートによるロールバックの2つのケースを実行した．なお，参考としてバッチ更新のみを行うケースも実行した．

図9に口座番号1から70までの実験結果を示す．コミットのケースでは口座番号17から46でOB更新が実行された．図9に示すように口座番号46までは4000を加算してから1/2にした残高，すなわち，まず，オンライン入力が実行され，その後でバッチ更新を実行したのと同様の結果が得られた．一方，口座番号47以降は，まず1/2にしてから4000を加算した残高，すなわち，まず，バッチ更新が実行され，その後でオンライン入力が実行されたのと同様の結果が得られた．すなわち，分離性が維持され，直列化可能スケジュールを構成することができた．従って，整合性も維持されていた．

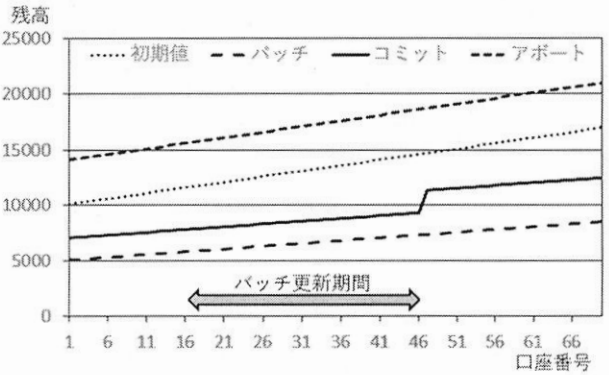


図9 バッチ更新をコミットした場合の結果

また，アボートのケースでは初期値に対し4000の加算のみが実行された．これは，オンライン入力のみが行われたのと同等の結果であり，分離性，整合性に加えて原子性が維持された．なお，3.3節に説明したように，持続性はMySQLの機能により提供されている．従って，本方式によりACID特性を維持したバッチ更新の実装ができた．

4.3 実験2：効率性の評価

次に，本提案方式によって，図3に示すように従来の時制更新方式の課題，すなわちオンライン入力におけるバッチ更新のコミット待ちが解消されたことを確認するため，本方式と従来の方式の効率の比較評価を行った．効率は，オンライン入力の処理が起動されてから，データベースアクセスのトランザクション開始までの時間によって計測した．なお，オンライン入力の競合を除いては，実験の環境は実験1と同様である．

ここで，従来方式ではバッチ更新コミット直後のトランザクションに待ちが発生することが分かっている．このため，図10に示すように各々のケースの待ち時間を長さの

降順の順位で並べたグラフを作成し比較した．図10でC_8は本方式において図3の競合するオンライン入力の数を8にしたケースであり，S_2，S_4，S_8は従来の方式で競合の数を各々2，4，8にしたケースである．

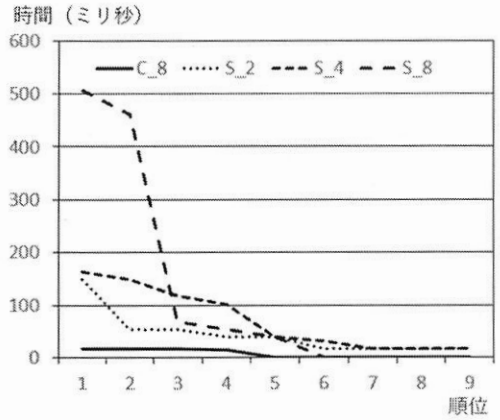


図10　効率性の比較評価

図10に示されるように従来の方式では競合の数が増えるに従って待ち時間が増加し，8の場合には500ミリ秒に達した．一方，本方式では，8の場合であっても待ち時間はほとんど発生しなかった．

5．考察

従来の時制更新方式で直列化可能スケジュールを構成するためには，図3に示すようにバッチ更新のコミットがボトルネックとなる．このため，図10に示すように競合によりオンライン入力の実行時間が長くなるに従い，待ち時間が増加した．本提案方式は表1の区分2に示す手順を追加することで，この待ち時間の解消を図ったものである．本方式により，実験1，実験2に示すようにACID特性を維持したバッチ更新という特徴を維持したままで，待ち時間の増大を抑止できることが分かった．

さらに，従来の方式では競合するトランザクションの完了を管理する必要があった．すなわち，実行中の全てのトランザクションに対して，バッチ更新との競合の有無を確認し，それらが全て完了したことを判定した上でバッチ更新のコミットを行う必要があった．しかし，本方式では図6のオンライン入力プログラムに示すように，個々のオンライン入力トランザクションが自律的に区分を判定して必要なデータ操作を行うことが可能になった．従って，システム全体としても効率が向上できると考えられる．

ただし，実際の業務の視点では，区分2はトランザクションのロールバックに相当する．例えば，ユーザがデータを入力していた場合には，一旦，入力が破棄されることになる．従って，実際の適用にあたっては，ユーザの操作を考慮する必要があると考えられる．例えば，複雑な書類の情報を入力する操作では，入力作業自体に時間を要するため，少々の待ち時間は許容される可能性が高い．逆に，時間をかけて入力した結果の破棄はユーザに大きな負担となると考えられる．従って，この場合には従来の方式が有効と考えられる．一方で，預金の引出しなどでは，引出し中に残高が変わっても影響は少なく，むしろ，オンライン入力待ちの時間を抑止することが重要になる．このような分野では，本方式が有効と考えられる．

6．まとめ

時制更新方式は，長時間のバッチ更新を，ACID特性を維持しながらオンライン入力と同時実行できるという特徴を持っており，先行研究により分散環境や相互に関連するデータの一括更新で有効であることを示してきた．しかし，従来の方式で直列化可能スケジュールを構成するためには，バッチ更新結果をコミットして有効にする際に，一旦，競合するオンライン入力を停止する必要があった．

そこで，本研究ではバッチ更新完了時点で実行中のオンライン入力に対し，一旦，バッチ更新の結果を有効にし，オンライン入力完了時点でバッチ更新の結果に対してオンライン入力を再実行した結果を有効にするという方式を提案した．さらに，実験によりバッチ更新を，ACID特性を維持したトランザクションとして実行しながら，コミットの待ち時間を抑止できることを確認した．

現在，時制更新方式をベースにしたNoSQL，特にMongoDBのトランザクション処理の研究を進めている．次の段階として，本研究成果の応用によりMongoDBのトランザクション処理の効率化が可能になると考えている．

謝辞

本研究はJSPS科研費15K00161の助成を受けたものです．また，本研究の動機はノンストップシステムのバッチ更新を目的とした特許[2)]の効率的な実装方法の開発にあります．特許成立を支援頂いた三菱電機インフォメーションシステムズ（株）の各位に感謝いたします．

参考文献

1）　喜連川優（監訳），J. Gray, and A. Reuter（著），“トランザクション処理－概念と技法”，日経BP社（2001）．

2）　三菱電機インフォメーションシステムズ（株），工藤司，“データベースシステム”，特許第4396988号（2009）．

3）　T. Kudo, Y. Takeda, M. Ishino, K. Saotome and N. Kataoka, “Application of a Lump-sum Update Method to Distributed Database”, Int. Journal of Informatics Society, Vol. 6, No. 1, pp. 11-18 (2014).

4）　T. Kudo, Y. Takeda, M. Ishino, K. Saotome and N. Kataoka, “Evaluation of Lump-sum Update Methods for Nonstop Service System”, Int. Journal of Informatics Society, Vol. 5, No. 1, pp. 21-28 (2013).

5） 白鳥則郎（監修），“データベース—ビッグデータ時代の基礎—”，共立出版（2014）.

6） P. A. Bernstein, V. Hadzilacos, and N. Goodman, “Concurrency Control and Recovery in Database Systems”, Addison-Wesley (1987).

7） T. Wang, J. Vonk, B. Kratz, and P. Grefen, “A survey on the history of transaction management: from flat to grid transactions”, Distributed and Parallel Databases, Vol. 23, Issue 3, pp. 235-270, 2008.

8） H. Garcia-Molina, and K. Salem, “SAGAS”, Proc. of the 1987 ACM SIGMOD Int. Conf. on Management of data, pp. 249-259 (1987).

9） A. Silberschatz, H.F. Korth, and S. Sudarshan, “Database System Concepts”, McGraw-Hill Education (2010).

10） T. Kudo, Y. Takeda, M. Ishino, K. Saotome and N. Kataoka, “A Mass Data Update Method in Distributed Systems”, Procedia Computer Science Vol. 22, pp. 502-511 (2013).

11） R. Snodgrass and I. Ahn, “Temporal Databases”, IEEE COMPUTER, Vol. 19, No. 9, pp. 35-42 (1986).

12） B. Stantic, J. Thornton and A. Sattar, “A Novel Approach to Model NOW in Temporal Databases”, Procs. 10th Int. Symp. on Temporal Representation and Reasoning and Fourth Int. Conf. on Temporal Logic, pp. 174-180 (2003).

13） H. Berenson et al., “A critique of ANSI SQL isolation levels”, ACM SIGMOD Record, Vol. 24, No. 2, pp.1-10 (1995).

14） R. T. Snodgrass, “Developing time-oriented database applications in SQL”. Morgan Kaufmann Publishers (2000).

15） T. Kudo, Y. Takeda, M. Ishino, K. Saotome and N. Kataoka, “A batch Update Method of Database for Mass Data during Online Entry”, ADVANCES IN KNOWLEDGE-BASED AND INTELLIGENT INFORMATION AND ENGINEERING SYSTEMS, IOS Press, pp. 1807-1816 (2012).