Vol.2017-OS-139 No.17
2017/3/2

トランザクション処理システムのリカバリ可能性の再考

神谷 孝明[1,a)]　星野 喬[2]　川島 英之[3]　建部 修見[3]

概要：データベースシステムなどのトランザクション処理システムにおける必須要件の一つとして障害からのリカバリが挙げられる．この要件を満たすためにこれまで様々な技術・手法が提案されてきた．しかし，それらは体系化されておらず，各既存研究が並行性制御や WAL（ログ先行書き込み）を独自の方法で組み合わせてリカバリ可能という目的を達成し，なおかつ高性能なトランザクション処理システムを目指している．本論文は，リカバリ可能なシステムの十分条件とは何かを再考し，各手法において，この十分条件が何によって満たされているかという観点で既存研究の整理を行う．また，リカバリ可能な十分条件を満たし，かつコストが小さいと考えられる新手法（Silo-TID 方式）を提案する．

1. はじめに

トランザクションとは，一連の操作を一つの実行単位としてまとめたものである．トランザクションは完全に成功（コミット）するか，全て無かったことになる（アボート）かのどちらかの状態しか許されない（原子性）．コミットしたトランザクションによる結果は失われてはならない（永続性）．また，実行途中のトランザクションによる影響を他のトランザクションは受けない [*1]（独立性）．また，トランザクションの前後でデータの整合性が失われてはならない（一貫性）．これらの特性はまとめて ACID と呼ばれる．一方で，システムには障害がつきものである．システムに障害が発生すると，原子性，一貫性，永続性のいずれかが失われる可能性がある．そこで，一般にトランザクション処理システムでは，リスタート時にリカバリを行うことで，トランザクションの ACID 特性を保証する．本稿ではシステムに障害が発生してもリカバリ操作によってトランザクションの ACID 特性が保証されるシステムのことを，リカバリ可能なシステムと呼ぶ．

トランザクション処理システムを分類する既存の枠組みとして steal/no-steal，force/no-force が存在する．この分類によって，リカバリ操作に redo 操作が必要か，undo 操作が必要かが分類される（**表 1**）．しかし，近年のトランザクション処理システムの設計は多岐に渡っており，それらがどのようにリカバリ可能性を満たしているかはこの分類からは一概に言うことができない．そこで，本稿はリカバリ可能なシステムの十分条件とは何かを再考し，各システムにおいてこの十分条件がどうやって満たされているかという観点で新たに分類を作成し，既存研究の整理を行う．

表 1　steal/no-steal, force/no-force とリカバリ時に必要な操作

	No-Steal	Steal
No-Force	Redo	Redo & Undo
Force		Undo

また，Silo は epoch と TID（Silo-TID）を組み合わせて使用しているが，リカバリ可能性を満たすためには epoch は必ずしも必要ではない．そこで epoch を除外した Silo-TID を使う方式を提案する．Silo-TID を使った方式は，epoch すら存在せず，グローバルな順序番号を一切使用しないため，スケーラビリティを阻害する要因がない．

本稿の構成は以下の通りである．2 章ではトランザクション処理システムを分類する既存の枠組みである steal/no-steal，force/no-force を紹介する．3 章ではリカバリ可能性を定義する．4 章ではリカバリ可能性を保証するための並行性制御と WAL について説明する．5 章では既存研究を紹介し，並行制御と WAL の観点から分類を行う．6 章では Silo-TID を使う新たなシステムの方式を議論する．7 章では結論と今後の課題を述べる．

2. steal/no-steal, force/no-force

トランザクション処理システムを分類する既存の枠組みとして，steal/no-steal，force/no-force がある．これらの設計の選択によって，リカバリ時に必要とされる操作

[1] 筑波大学大学院 システム情報工学研究科
[2] サイボウズ・ラボ株式会社
[3] 筑波大学 計算科学研究センター
[a)] kamiya@hpcs.cs.tsukuba.ac.jp

[*1] トランザクションの分離レベルによって，どの程度の影響を許容するかが異なる．一般に分離レベルが強いほど独立性が高いがトランザクション処理性能は低く，分離レベルが弱いほど独立性は低くなるがトランザクション処理性能は高い．

(redo/undo) が異なる (表 1).

2.1 steal/no-steal

steal は，未コミットなトランザクションの更新を永続データベース上へ書き込むこと (dirty-write) を許す．そのため，トランザクションがアボートした時に，永続データベース上のデータを undo する必要がある．undo を可能とするために，データを単に上書きする場合でも，データの前の値を読み込んでログに undo 用の情報 (undo ログ) を記録する必要がある．一方，no-steal は，コミット済みのトランザクションの更新しか永続データベース上へ書き込まない．そのため，undo の必要がなく，リカバリは redo 操作のみで良くなるためシンプルなリカバリとなる．

多数のデータを更新するロングトランザクションがあった場合，steal ポリシーではメインメモリ上のバッファを永続データベース上へ適宜書き込むことができるため，メインメモリの容量が小さくてもトランザクションを継続して実行することができる．一方，no-steal ポリシーでは，バッファが一杯になっても，トランザクションがコミットするまで永続データベース上へ書き込むことが許されない．また，バッファ上のどのデータを永続データベース上に書き出してよいかをバッファマネージャーが判断するためのコストが発生する．

過去，メインメモリ容量が小さく，ディスクベースなデータベースが主流であった当時のトランザクションの研究では steal ポリシーが主流であった．しかし，近年はメモリの大容量化が進み，高性能なインメモリデータベースへのシフトが進んでいる．インメモリデータベースでは全てのデータをインメモリ上に置くことで高性能化を実現している．また，undo ログの作成は，単純にデータを上書きする場合でも一度元のデータの値を読み込む必要があるため，不要な読み込みが発生する可能性がある．そのため，インメモリデータベースを代表とする近年の高性能トランザクション処理システムの主流は no-steal ポリシーとなっている．

2.2 force/no-force

force は，トランザクションが更新した全てのデータを永続データベース上へ書き込むまでコミットすることを許さない．そのため，システム障害発生時でもコミット済みのトランザクションの更新は全て永続化されているため，リカバリに redo 操作が必要ない．一方，no-force はトランザクションのコミット時に更新したデータを永続データベース上へ書き込むことを強制しない．そのため，リカバリ時に redo 操作が必要となるが，通常のコミット時にはログ (WAL) を永続化することでトランザクションの永続性を保証し，永続データベース上のデータの更新を遅延させることができる．トランザクションのコミット時に更新したデータの数だけデータの書き込みが発生しうる force と比べて，ログはまとめて一度で書き込まれるため，no-force ポリシーの方が性能が高くなる．そのため，近年の高性能トランザクション処理システムの主流も no-force となっている．一方で，実用化が近いと期待されている NVM を用いると書き込み遅延は小さく高速になるため，この NVM を前提として，リカバリの単純化の目的から force を採用するトランザクション処理システムの研究も行われている [4], [5].

3. リカバリ可能性

2 章で述べた通り，今後の高性能トランザクション処理システムの主流になると考えられる no-steal/no-force のシステムを対象として，次の二つの条件が満たされることをリカバリ可能と定義する．

(1) w-r (write-read) 関係にあるトランザクションのコミット順序が w-r の順序と一致する
(2) w-w (write-write) 関係にあるトランザクションのログの redo 順序が実スケジュールと一致する

steal のシステムでは undo 操作が必要となるため，この二つの条件に加えてアボート順序を考慮する必要が発生するが，本稿では対象としない．すなわち，no-steal のみを対象とする．上記の二つの条件の説明に先立って，説明に用いる用語を定義する．

3.1 用語の定義

操作

トランザクションの操作には write, read, commit, abort の 4 つがあるとする．トランザクション t_i がリソース x を read/write することをそれぞれ $r_i(x)/w_i(x)$ と書く．トランザクション t_i が commit/abort することをそれぞれ c_i/a_i と書く．

スケジュール

操作である $o_i, o_j (o_i \neq o_j)$ について，o_i の後に，o_j が実行された場合，$o_i < o_j$ と書く．または $<$ を省略して $o_i o_j$ と書く．

トランザクションの依存関係

トランザクション $t_i, t_j (t_i \neq t_j)$ に対して，それぞれがアクセスした read と write の data record 集合を read-set, write-set と呼び，rs_i, ws_i, rs_j, ws_j と書く．

w-w, w-r, r-w (read-write), r-r (read-read) 関係を次のように定義する．$x \in ws_i \cap ws_j$ and $w_i(x) < w_j(x)$ の時，t_i, t_j は w-w 関係 (output-dependency, write-after-write) にあるという．$x \in ws_i \cap rs_j$ and $w_i(x) < r_j(x)$ の時，t_i, t_j は w-r 関係 (flow-dependency, read-after-write) にあるという．$x \in rs_i \cap ws_j$ and $r_i(x) < w_j(x)$ の時，t_i, t_j は r-w 関係 (anti-dependency, write-after-read) にあるという．$x \in rs_i \cap rs_j$ and $r_i(x) < r_j(x)$ の時，t_i, t_j は r-r 関係にあ

るという．

t_i と t_j に x-x(r-w/w-r/w-w のどれか) の関係がある時，t_i --> t_j@(x-x) と書く．

対象とするシステムの分離レベルはシリアライザブル（直列化可能）とする．このとき，t_1, t_2 の間に w-w，r-w，w-r のいずれかの関係があるとき，全ての関係において逆は成り立たない [14]．すなわち t_i --> t_j@(x-x) $\Rightarrow \neg(t_j$ --> t_i@(r-w))$\wedge\neg(t_j$ --> t_i@(w-r))$\wedge\neg(t_j$ --> t_i@(w-w))．

3.2 コミット順序

リカバリ可能なトランザクションシステムでは，w-r 関係にあるトランザクションのコミット順序が保たれなければならない．次に例で示すスケジュールでは，w-r 関係にあるトランザクションのコミット順序が保たれている．

$w_1(x)r_2(x)w_2(y)c_1c_2$

すなわち，t_1 が先に書いたデータを，後から t_2 が読む場合は，t_1 が先にコミットされなければならない．もし，先に t_2 がコミットしてしまった場合，後から t_1 がコミットする分には問題はないが，システム障害やユーザ意思により t_1 がアボートしてしまうと以下のようなスケジュールとなり問題が発生する．

$w_1(x)r_2(x)w_2(y)c_2a_1$

この時，t_2 はコミットしない t_1 によって更新されたデータを読んでいる（dirty-read）．本来は，t_1 がアボートになった場合は，t_1 の更新結果に依存する t_2 もアボートさせなければいけないが，既に t_2 の結果をクライアントに返してしまっていることが考えられるため，そのトランザクションをなかったことにはできない．t_2 はコミットしないトランザクションが更新したデータを用いた書き込みを行っているため，一貫性がなくなってしまう．また，t_2 を無理やりなかったことにしてしまうと，コミットしたはずの t_2 の更新 $w_2(y)$ がなくなり永続性が破綻する．

一方で，w-r 関係がなく，w-w 関係のみがあるトランザクションのコミット順序は保たれる必要はない．次のようなスケジュールを考える．

$w_1(x)w_2(x)c2$

この例では，t_1 が先に書いたデータを，後から t_2 が書いている．この時，t_2 は，t_1 の更新を読んでいないため，t_1 のコミット/アボートに t_2 の結果は左右されず，t_1 よりも先に t_2 をコミットすることが可能である．

同様に，w-r 関係がなく，r-w 関係のみがあるトランザクションのコミット順序も保たれる必要はない．次のようなスケジュールを考える．

$r_1(x)w_2(x)c2$

この例では，t_1 が読んだデータを，後から t_2 が書いている．この時，t_2 は，t_1 の更新結果に依存しないため，t_1 のコミット/アボートに t_2 の結果は左右されないため，t_1 よりも先に t_2 をコミットすることが可能である．

3.3 ログの redo 順序

WAL によってログだけ先に書いておき，永続データベース上への更新を遅延させる場合，ログの redo 順とスケジュール上の書き込み操作の順序を一致させる必要がある．

$w_1(x)w_2(x)c_1c_2$

というようなスケジュールでトランザクション t_1, t_2 実行されたとする．このとき，永続データベース上の x を更新する前に障害が発生した状況を考える．トランザクションの永続性を保証するために，ログを用いてコミットしたトランザクションの更新を redo する．このとき，スケジュール上で，$w_1 < w_2$ となっている場合は，w_2，w_1 の順でログを redo してしまうと，最終的な結果が異なったものになってしまうため，必ず w_1，w_2 の順番でログの redo をする必要がある．

4. リカバリ可能性の実現方法

リカバリ可能性を満たすための二つの要件を実現するために使用される並行性制御と WAL について説明する．

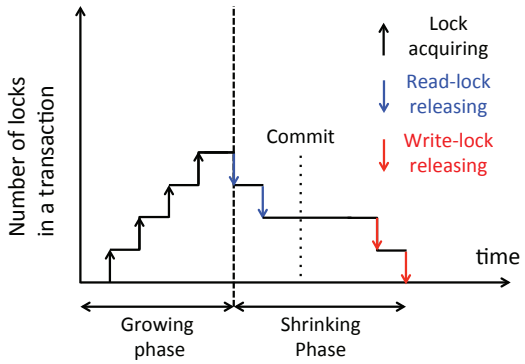
4.1 並行性制御

並行性制御はリソースへの排他を行うことで，同時に動くトランザクション間で ACID の独立性を保証する仕組みである．リカバリ可能性に必要な要件の一つである「w-r 関係にあるトランザクションのコミット順序」を保証するための並行性制御の方式として，悲観的並行性制御の S2PL（Strict 2-Phase Lock）および，SS2PL（Strong Strict 2-Phase Lock）[1], [14] がある．これらは 2PL（2-Phase Lock）に基づき，成長フェーズで全てのロックを獲得し，その後の縮退フェーズで全てのロックを解放する（**図 1**）．

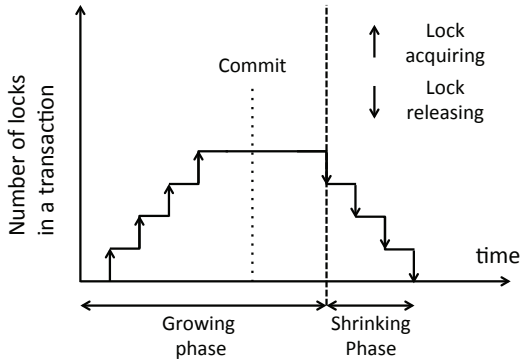
S2PL（**図 1(a)**）は write-lock（exclusive-lock）はコミット後に解放しなければならないが，read-lock（shared-lock）は 2PL に基づいた範囲でコミット前に解放することができる．SS2PL（**図 1(b)**）はコミット後に read-lock と write-lock を解放する．

S2PL ではコミット後に read-lock を解放するため，w-r，w-w 関係にあるトランザクションのコミット順序を保つが，r-w 関係についてはコミット順序を保たない．一方で，SS2PL では，w-r，w-w，r-w 関係にあるトランザクションのコミット順序を保つ．r-w 関係にあるトランザクションの順序は逆転しても不整合は発生しないことから，高性能化の目的のためには，より制約の緩い，read-lock を早く解放することが可能な S2PL を使用するべきだと考えられる．

楽観的並行性制御（OCC）では，read-lock を使用しない代わりにバージョン番号を用いてトランザクションのコミット時にバージョン番号によるバリデーションを行う．バリデーションによって，リソースへのアクセス競合を検知する．Silo で用いられている OCC は S2PL 相当なもの

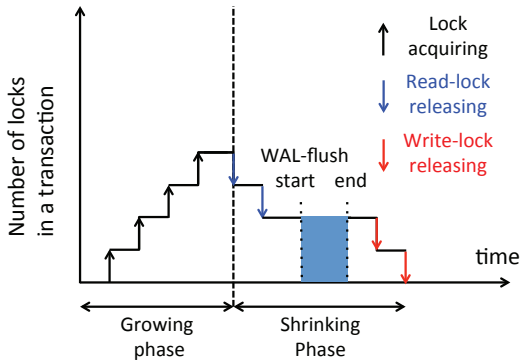


(a) S2PL

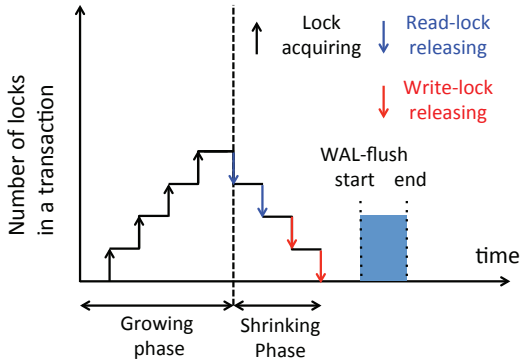


(b) SS2PL

図 1 (a) S2PL と (b) SS2PL のコミットタイミング．横軸が時間軸，縦軸が一つのトランザクションが獲得したロック数．S2PL はコミット前に read-lock を解放できるのに対し，SS2PL ではコミット後に read-lock を含む全てのロックを解放する．



(a) NLR



(b) ELR

図 2 S2PL における NLR と ELR のログ永続化タイミングの比較．NLR ではログの永続化（WAL-flush）が終わるまで，write-lock を解放できない，ELR ではログを永続化する前に write-lock を解放する．

であるため，本稿では断りのない限り並行性制御は S2PL を用いることを前提として議論を行う．

4.1.1 Normal Lock Release

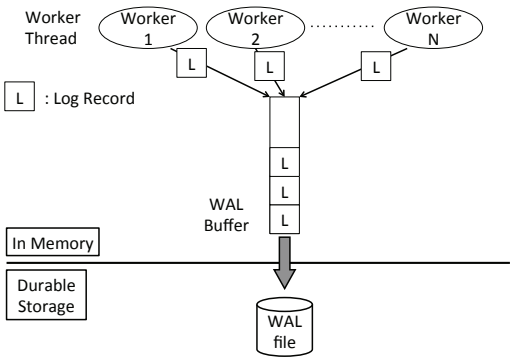
WAL を使ったシステムでは，全てのログの永続化後にトランザクションがコミットされるため，S2PL の write-lock の解放タイミングはトランザクションの全てのログの永続化後 (**図 2(a)**) となる．このロックの解放を NLR (Normal Lock Release) と呼ぶ．NLR を使う場合は dirty-read は発生しないため，「w-r 関係にあるトランザクションのコミット順序」は保証される．

4.1.2 Early Lock Release

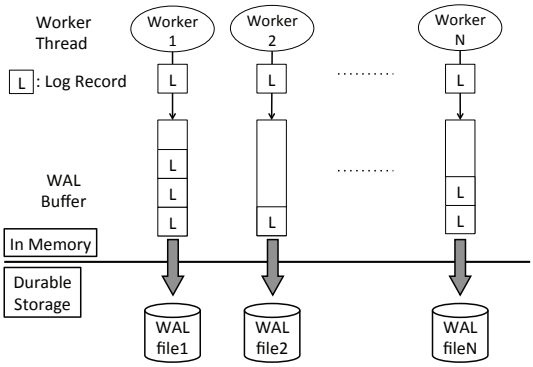
ELR（Early Lock Release）[3], [6], [10] はコミット前にロックを解放する手法である．ログの永続化はストレージ I/O を要するため，計算処理と比べて長い時間がかかる．NLR では，ログを永続化するまでロックを解放することができないが，ロックの保有期間が長いほどトランザクションの並列性を損なうので，ロックはできるだけ早くに解放するのが望ましい．そこで，ELR を採用した S2PL では，コミット前であるログの永続化前に write-lock を解放する (**図 2(b)**)．ELR を行うためには，ロックを早期解放したトランザクションに依存するトランザクションは，依存先である前のトランザクションがコミットした後に，コミットされなければならないという制約が発生する．これは，ELR を用いた場合は「w-r 関係にあるトランザクションのコミット順序」を何らかの方法で保証しなければならないことを意味する．この制約を満たす方法は既存研究毎に異なるため，具体例は 5 章で説明する．

4.2 WAL

WAL は永続データベース上のデータを上書きする前に，予めログを永続化することで，ACID の原子性と永続性を保証する仕組みである．ページの更新は後から In-Place で更新し，システム障害が発生した際はログを用いてリカバリを行う．リカバリ可能性に必要な要件の一つである「w-w 関係にあるトランザクションのログの redo 順序」を保証するためには，WAL によって書かれるログの適用順



(a) 直列 WAL．単一の WAL バッファ/WAL ファイルを持つ．



(b) 並列 WAL．複数の WAL バッファまたは複数の WAL ファイルを持つ．図に示す P-WAL のアーキテクチャはワーカースレッドの数だけ複数の WAL バッファと複数の WAL ファイルを持つ．

図 3 (a) 直列 WAL と (b) 並列 WAL のアーキテクチャの比較

序が問題となる．

4.2.1 直列 WAL

ARIES[9] や Aether[6] などで使われている WAL は，単一の WAL バッファ/WAL ファイルを利用している．このアーキテクチャを直列 WAL と呼ぶ．直列 WAL のアーキテクチャを**図 3(a)** に示す．直列 WAL では通常の実行時の順番にログを作成するだけで，メモリ上の WAL バッファ上でログが直列化される．WAL バッファのログは WAL ファイルに追記書き込みされる．そのため，リカバリ時は単一の WAL ファイルの先頭から順にログの redo を行うことで，実際のスケジュールと同じ順番で w-w 関係を保ったまま redo を行うことができる．すなわち，直列 WAL であれば「w-w 関係にあるトランザクションのログの redo 順序」がアーキテクチャによって保証される．

4.2.2 並列 WAL

直列 WAL に対して，複数の WAL バッファまたは，複数の WAL ファイルを持つものを並列 WAL と分類する．Silo[12]，P-WAL[16]，In-Page Logging[7] などは，フラッシュストレージや複数のストレージデバイスを使用することを前提に複数の WAL バッファ/複数の WAL ファイルを用いた並列 WAL を行う．並列 WAL である P-WAL のアーキテクチャを**図 3(b)** に示す．一般に並列 WAL では，複数の WAL バッファを用いることで直列 WAL で引き起こされる WAL バッファの競合を回避し，フラッシュストレージや複数のストレージへの並列書き込みを活用した高性能なログ書き込みを実現する．しかし，ログが単一の WAL バッファ/WAL ファイルにないためログが直列化されて記録されないため，ログに対応する操作の実際の順序が不明になってしまう可能性がある．すなわち，並列 WAL を用いた場合は「w-w 関係にあるトランザクションのログの redo 順序」がアーキテクチャによって保証されない．そのため，適切な方法で発行された順序番号をログに記録することで，実際の w-w 関係の順でトランザクションのログを redo を行う．この順序番号の発行方法は既存研究ごとに特色があるため，具体例は 5 章で説明する．

表 2 代表的な既存研究

	直列 WAL	並列 WAL
NLR	ARIES[9]	D-ARIES[11] IPL[7], [8] FlashLogging[2]
ELR	Aether[6]	Distributed Logging[13] P-WAL[16] Silo[12]

表 3 表 2 の各分類がシステムで別途満たさなければいけないリカバリ可能性の条件．w-w は「w-w 関係にあるトランザクションのログの redo 順序」，w-r は「w-r 関係にあるトランザクションのコミット順序」の略．

	直列 WAL	並列 WAL
NLR		w-w
ELR		w-r w-w

5. 既存研究の整理

代表的な既存研究を直列 WAL/並列 WAL，NLR/ELR の観点で分類した（**表 2**）．この分類により，システムがリカバリ可能性を保証するためにシステムが必要とする条件が異なる（**表 3**）．

5.1 直列 WAL

5.1.1 NLR

ARIES[9] は直列 WAL かつ NLR の代表的なシステムである．直列 WAL と NLR を使用しているため，w-w，w-r の条件が自動的に満たされる．各ログレコードにはログの論理的なアドレスを示す LSN（Log Sequnce Number）と呼ばれる一意な順序番号が割り当てられる．

5.1.2 ELR

Aether は ELR かつ直列 WAL である．直列 WAL を用いているため，w-w の条件は満たされる．ELR を用いて，メモリ上の WAL バッファにコミットログを挿入した後にロックを解放する．直列 WAL では，一つの WAL バッファを用いて先頭から順にログを永続化するため，後からロックを獲得したトランザクションのコミットログはその後に WAL バッファに挿入される．そのため，ELR であっても，特別な制御を要さずに，コミットログの永続化の順番がロックを獲得する順番と一致するため，w-r の順にトランザクションのログが永続化されるため，w-r の条件が満たされる．

5.2 並列 WAL

5.2.1 NLR

D-ARIES[11] は分散環境を対象とした研究である．分散環境ではノード間で通信してログをマージするコストが高いため，それぞれのノードがローカルにログを書く．全順序を割り当てる方式はコストが高いため，固定長のページ毎に最後にそのページを更新したログレコードを指すポインタとして，ノード番号とノード内におけるログレコードの論理アドレスを保持する．次に当該ページを更新するログは，一つ前の更新のログレコードの番号を記録する．各ログレコードは直前に同一のページを更新したログレコードを指すポインタを持つため，リカバリ時はこのポインタを元に，ページ毎にログレコードのリストを作成し，リストの先頭から redo を行うことで w-w の条件が満たされる．

In-Page Logging は，フラッシュストレージを対象とした研究である．フラッシュメモリの上書きに伴う高コストな消去操作の回数を減らすことを目的としている．ページの中に設けられたログ領域にログを追記することで，ログ領域が空いている間は上書きをせずに済む．同一ページへのログの書き込みが同一のログの領域に追記されるため，ログ領域が一杯になり，ログとデータのマージが必要になった場合は，各ページに含まれる領域の先頭からログを redo する．

FlashLogging は単一の WAL バッファを使用して単調増加する LSN を割り当てる．一方で，ログの書き込みリクエストは複数の WAL ファイルへラウンドロビン方式で分散させる．ストレージを複数使うことによりストレージの帯域幅の活用を行う．リカバリ開始地点は各ファイル内を二分探索し，以降はラウンドロビンでログを読み込むことで LSN 順にログがマージされる．LSN の大小関係が w-w 関係を反映するため，LSN 順にログを redo する．

5.2.2 ELR

P-WAL は，グローバルな共有カウンタを用いて，各ログレコードに単調増加する順序番号を割り当てている．DistributedLogging と同様の方法で，当該トランザクションのコミットログの順序番号以下の値を持つ全てのログレコードが永続化して初めて，トランザクションをコミットする．共有カウンタを用いた一意な順序番号割り当ては，並列度が高くなるとこの順序番号割当がスケーラビリティを妨げるボトルネックとなる可能性がある．

DistributedLogging は，シーケンス番号としてページとトランザクションの両方で用いる GSN なる順序番号を導入している．GSN は論理クロックにもとづいて計算される．GSN の大小関係が w-w 関係を反映するため，GSN の順にログを redo する．コミットログに GSN を割り当ててから，ELR によるロックの解放を行うことで，w-r の関係にあるトランザクション間で，コミットログの GSN の大小関係が w-r 関係を反映する．そのため，当該トランザクションのコミットログの GSN 以下の GSN を持つ全てのログが永続化してからトランザクションをコミットし，クライアントに結果を返すことで，w-r 関係にあるトランザクションのコミット順序を保っている．

Silo は，epoch（時間の区間）単位のグループコミットを採用している．同一 epoch の全トランザクションのログが永続化されて初めて，その epoch 内の全トランザクションがコミットとなる．Epoch 番号の大小関係が，w-w，w-r，r-w 関係を反映しており，epoch 番号の小さい順にトランザクションがコミットされるため，epoch をまたいだ w-r 関係にあるトランザクションのコミット順序は w-r の順を保つ．また同一 Epoch 内の w-r 関係にあるトランザクションは同時にコミットされるため，w に相当するトランザクションがアボートし，r に相当するトランザクションがコミットすることはない．よって，w-r 関係にあるトランザクションのコミット順序を保つ．また，リカバリ時はログは Epoch 単位で redo を行う．Epoch 内の全てのトランザクションがログが永続化された最新の epoch まで redo を行い，それ以上は redo を行わない．Epoch をまたいだ w-w 関係にあるトランザクションのログの redo 順序は，epoch の順で redo を行うことによって保証される．一方，同一 epoch 内の w-w 関係にあるトランザクションのログには TID（Transaction ID）がつけられている．P-WAL では共有カウンタを用いてログレコードにグローバルな順序番号を割り当てたが，Silo ではトランザクション単位で TID を分散方式で割り当てる．TID は 64-bit で上位ビットがコミット時の epoch 番号を含み，中位ビットが同一 epoch 内での順序番号を表し，下位の 3bit はステータスを管理するためのものとなっている．この TID は OCC で使用するバージョン番号として各データにも記録されており，次の三つの条件を満たす最も小さい番号が TID として決定される．

(1) 当該トランザクションによって読み書きしたどのデータの TID よりも大きい番号
(2) ワーカーが最近選んだ TID よりも大きい番号

(3) 現在のグローバル epoch 番号

この TID の順序は必ずしも r-w と r-r の実際の順序を反映しないが，(1) の条件により，TID は同一 epoch 内の w-r および w-w 関係にあるトランザクションの順序を反映する．これにより，同一 epoch 内のログは TID の順で redo を行うことで，w-w 関係にあるログの redo 順序は実際の順序と一致する．

6. 議論

Silo の TID の生成方法はアクセスしたデータをもとに分散方式で計算される．epoch 番号を TID に組み込むことで，epoch を跨いだトランザクション間での w-w, w-r, r-w 関係が，TID の大小関係に反映されるが，r-w はリカバリ可能なトランザクションシステムの要件ではない．すなわち，epoch のような集中型のタイムスタンプを一切使用せずに，リカバリ可能なシステムを作成できる可能性がある．

epoch 番号を上位ビットに組み込まない，アクセスしたデータのみに基づいて計算される Silo の TID を Silo-TID と呼ぶ．epoch は更新頻度が少ないとはいえ，グローバルな順序番号割当機構であるのに対し、TID 方式はスケーラビリティを阻害するグローバルな順序番号機構は一切存在しない．epoch を含めずとも，w-w, w-r の関係が Silo-TID の大小関係に反映される．

silo の論文で設定されている epoch は 40ms であるため，どんなに短いトランザクションであってもレイテンシは epoch に依存する．一方で，Silo の TID の順番でコミットを行うことで，レイテンシを大きく下げながら，Silo の高スループットなトランザクション処理性能を達成できる可能性がある．また，グローバルな共有カウンタ（LSN, epoch）を一切必要としないため，スケーラビリティを阻害する要因がない．

Silo では，epoch を荒い粒度（40msec 毎）で割り当てることにより，グローバルな順序番号割当ボトルネックを回避している．しかしながら，多数のコアを持つ環境の場合，同一 epoch による順序番号の配分のコストを償却するために，長い epoch を使用する必要がある [15]．epoch を使った方式ではどんなに短いトランザクションであっても，epoch 毎にトランザクションをコミットしクライアントに結果を通知するため，epoch が長いとその分遅延が大きくなる．

一方，Silo-TID のみを使う方式では，epoch を使わずにどの TID のトランザクションまでをコミットし，クライアントに結果を返してよいかを別の方法で計算する必要がある．現在考えている方式は，TID が欠損なく永続化された地点のトランザクションまでをコミットとする方法である（**図 4**）．欠損なく永続化されているか TID の地点を判断するために，各ワーカーが最後にログを永続化したトランザクションの TID（durableTID）を用いる．全ワーカーの durableTID の最小値が，欠損なくログが永続化された地点の TID となる．

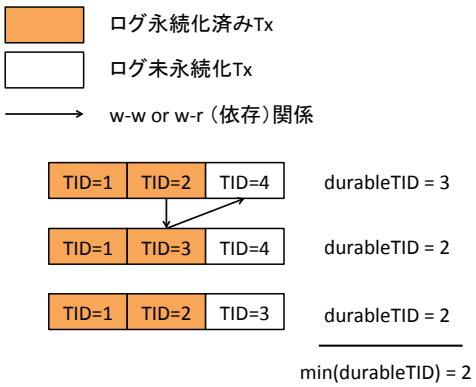


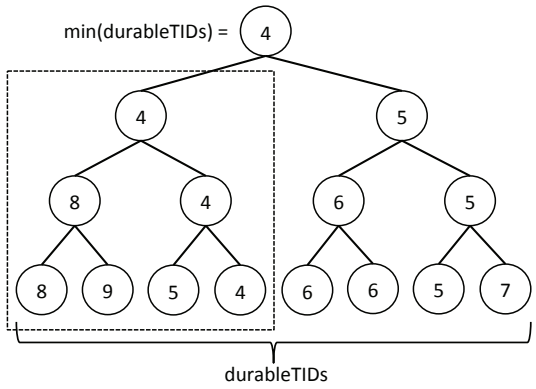
図 4 ワーカースレッドの数が 3 の時のトランザクション実行例．TID は各ワーカー内で単調増加する．また，同一レコードへアクセスする w-w，w-r 関係にあるトランザクションの TID も単調増加する．この例では，durableTID の最小値は 2 であるため，TID が 2 以下であるトランザクションをコミットし，クライアントに結果を返すことができる．

しかしながら，ナイーブに毎回最小値を計算する方法では，epoch と同様に大きな遅延が発生する可能性がある．そこで，考えているのが木構造を使った最小値計算のモデルである（**図 5**）．木構造を使う場合は値の読み込みだけでなく更新も必要となる．この更新は CAS（Compare and Swap）を使う．あるワーカーが更新する可能性のあるノードは，できるだけソケットのラストレベルキャッシュに載るようにスレッドを割り当てる．図 5 の例では，1socket につき 4 コアの CPU が載っているとした場合，点線に含まれるノードが同一ソケットのラストレベルキャッシュに載るようにスレッドを割り当てるといった工夫が考えられる．

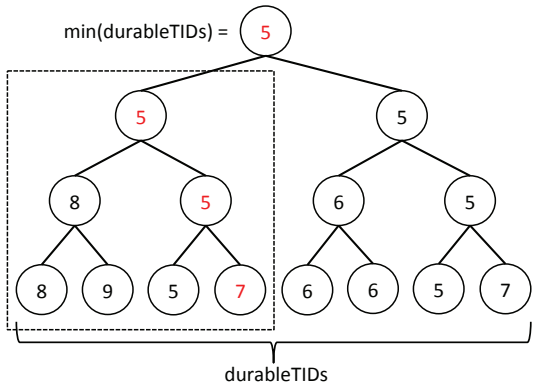
7. 結論と今後の課題

本稿ではトランザクションシステムのリカバリ可能性について再考を行い，no-steal/no-force システムにおけるリカバリ可能性を「w-r（write-read）関係にあるトランザクションのコミット順序が w-r の順序と一致する」，「w-w（write-write）関係にあるトランザクションのログの redo 順序が実スケジュールと一致する」の二つの条件を満たすことと定義した．既存研究を並行性制御と WAL の観点から分類し，各分類においてシステムが満たすべき条件を示した．また，LSN や epoch などのグローバルな順序番号を一切使用せず，リカバリ可能性を持つと考えられる Silo-TID 方式を提案した．今後の課題は，epoch を使った Silo 方式と Silo-TID 方式を実装し，スループットと遅延の面から両方式の性能を比較することである．

謝辞 本研究の一部は，JST CREST「ポストペタスケー



(a) 初期状態



(b) 初期状態から $\mathrm{durableTID}_4 = 7$ になった場合

図 5 ワーカー数が 8 の場合の木構造による min(durableTIDs) の計算例．葉ノードが各ワーカーの durableTID を表している．ワーカーはトランザクションログを永続化する際に，durableTID を更新する．ノードの更新の際は，一つ上の親ノードを見て，もし自分と同じ値であれば，親ノードの値を兄弟ノードの最小値で更新する．自分と違う値であればそれ以上何もしない．この操作を根まで再帰的に繰り返すことで，根ノードの値が durableTID の最小値となる．durableTID の更新時に木を更新することで，durableTID の最小値を求める際には親ノードの値を読むだけで済む．

ルデータインテンシブサイエンスのためのシステムソフトウェア」，JST CREST「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，JST CREST「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」，科研費「#16K00150」による．

参考文献

[1] Bernstein, P. A., Hadzilacos, V. and Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1987).
[2] Chen, S.: FlashLogging - exploiting flash devices for synchronous logging performance., *SIGMOD Conference* (2009).
[3] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., Wood, D. A., DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R. and Wood, D. A.: *Implementation techniques for main memory databasesystems*, Vol. 14, ACM (1984).
[4] Gao, S., Xu, J., Härder, T., He, B., Choi, B. and Hu, H.: PCMLogging - Optimizing Transaction Logging and Recovery Performance with PCM., *IEEE Trans. Knowl. Data Eng.*, Vol. 27, No. 12, pp. 3332–3346 (2015).
[5] Izraelevitz, J., Kelly, T. and Kolli, A.: Failure-Atomic Persistent Memory Updates via JUSTDO Logging., *ASPLOS*, pp. 427–442 (2016).
[6] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M. and Ailamaki, A.: Aether - A Scalable Approach to Logging., *PVLDB* (2010).
[7] Lee, S.-W. and Moon, B.: Design of flash-based DBMS - an in-page logging approach., *SIGMOD Conference*, p. 55 (2007).
[8] Lee, S.-W. and Moon, B.: Transactional In-Page Logging for multiversion read consistency and recovery, *2011 IEEE International Conference on Data Engineering (ICDE 2011)*, IEEE, pp. 876–887 (2011).
[9] Mohan, C., Haderle, D. J., 0001, B. G. L., Pirahesh, H. and Schwarz, P. M.: ARIES - A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging., *ACM Trans. Database Syst.* (1992).
[10] Soisalon-Soininen, E. and Ylönen, T.: Partial Strictness in Two-Phase Locking., *ICDT*, Vol. 893, No. Chapter 12, pp. 139–147 (1995).
[11] Speer, J. and Kirchberg, M.: D-ARIES - A Distributed Version of the ARIES Recovery Algorithm., *ADBIS Research Communications* (2005).
[12] Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S.: Speedy transactions in multicore in-memory databases, *SOSP*, pp. 18–32 (2013).
[13] Wang, T. and Johnson, R.: Scalable Logging through Emerging Non-Volatile Memory., *PVLDB* (2014).
[14] Weikum, G. and Vossen, G.: *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*, Morgan Kaufmann Publishers Inc. (2001).
[15] Yu, X., Bezerra, G., Pavlo, A., Devadas, S. and Stonebraker, M.: Staring into the abyss, *Proceedings of the VLDB Endowment*, Vol. 8, No. 3, pp. 209–220 (2014).
[16] 神谷孝明, 川島英之, 星野喬, 建部修見.: 並列ログ先行書き込み手法 P-WAL 情報処理学会論文誌データベース（TOD）, Vol. 10, No. 1 (2017) (to appear).