ScalarDL Validator: データベースにおける非同期的ビザンチン故障検知

根本　潤† 山田　浩之†

† Scalar, Inc. 〒 162–0828 東京都新宿区袋町 5-1 FARO 神楽坂 209
E-mail: †{jun.nemoto,hiroyuki.yamada}@scalar-labs.com

あらまし 本論文では，非同期的なビザンチン故障検知が可能なデータベースミドルウェア ScalarDL Validator を提案する．これまで，データベースにおける改ざんの問題に対応するため，2 つのデータベースレプリカを用いて同期的にビザンチン故障を検知するプロトコルが提案されている．従来の検知プロトコルは，Strict Serializablity を保証しつつ，トランザクションを並列に実行可能だが，トランザクションの実行時にレプリカ間で同期的なコンセンサス処理を必要とするため，レイテンシが大きいという課題がある．これに対して，ScalarDL Validator では，実行時にコンセンサス処理をすることなく，故障の検知をトランザクション実行後に非同期的に行うことで実行時のレイテンシを短縮する．プロトタイプを用いた初期評価により，従来方式と比較してトランザクションスループットおよびレイテンシが改善することを示す．

キーワード ビザンチン故障検知，トランザクション処理，分散型台帳

1 はじめに

近年，国内外を問わず製造業を中心に検査不正や検査データの改ざんが横行し，社会問題となっている．また，企業や官公庁に対するサイバー攻撃も増加している．このような悪意ある攻撃に対応するため，ビザンチン故障遮蔽 (Byzantine Fault Tolerance，BFT) 技術 [2,3] およびビザンチン故障検知 (Byzantine Fault Detection，BFD) 技術 [7] が広く研究されている．

ビザンチン故障遮蔽技術はデータベースシステムにも応用されており，不可分性，一貫性，独立性を保ちながらトランザクション処理を行うことができる [6,10]．これらの技術でビザンチン故障を遮蔽するには，複数のデータベースレプリカを必要とする．許容する故障レプリカ数を f とすると，少なくとも $3f+1$ のレプリカが必要である．悪意ある攻撃は，一般に同一管理ドメイン [1]内のレプリカに波及しうるため，これらの複数のレプリカはそれぞれ別の管理ドメインで運用しなければならない．したがって，ビザンチン故障を遮蔽するためには，単一組織で運用されることが多いデータベースシステムを，管理権限の異なる少なくとも 4 つの組織で運用しなければならず，管理上大きな負担を強いる．

ビザンチン故障検知技術は，そうした負担を軽減しつつ悪意ある攻撃に対応するアプローチの 1 つである．これらの検知技術は，ビザンチン故障の検知のみを行うため，遮蔽技術のように運用を継続することはできないが，故障レプリカ数 f に対して $f+1$ のレプリカさえあれば利用可能である．すなわち，最低 2 つの管理ドメインで，ビザンチン故障を検知することができる．ビザンチン故障検知技術として，PeerReview [7] や ScalarDL [12] がある．

ScalarDL は，ビザンチン故障検知のためのデータベースミドルウェアであり，Strict Serializability を保証しつつ，互いにコンフリクトのないトランザクションを並列に実行可能である．ScalarDL は，Ledger と Auditor と呼ばれる 2 つのコンポーネントが有する 2 つのデータベースレプリカを用いて同期的にビザンチン故障を検知する．すなわち，トランザクション実行時に，2 つのレプリカ間で矛盾があればその場で検知する．具体的には，次の 3 フェーズで検知を行う．1) Auditor がクライアントからのトランザクション要求をコンフリクトの有無に基づいて順序付けする (Ordering フェーズ)，2) Ledger が決められた順序にしたがってトランザクションを実行しコミットする (Commit フェーズ)，3) Auditor が Ledger のトランザクション実行結果と実行順序を検証する (Validation フェーズ)．この 3 フェーズプロトコルは，いずれかのレプリカが，Strict Serializablity に違反したトランザクション実行や，データ改ざんなどのビザンチン故障を引き起こした場合，その場で検知が可能である．しかし，トランザクション実行時，Ledger と Auditor は，Ordering フェーズと Validation フェーズで同期的にコンセンサス処理 (合意形成) を行わなければならないため，レイテンシが大きいという課題がある．

そこで，本論文では，トランザクション実行時にコンセンサス処理を行わず，実行後に非同期的にビザンチン故障を検知する ScalarDL Validator を提案する．ScalarDL Validator の非同期的なビザンチン故障検知により，実行時の即時検知は不可能となるが，トレードオフとしてレイテンシを短縮する．具体的には，まず，クライアントが Ledger にトランザクション要求を送信する際と，Ledger からその実行結果を受領する際に，トランザクションの入出力内容と順序情報を記録しておく．そして，ユーザからの要求に応じて，記録したそれらの情報に基づいて，クライアント側でトランザクションを再実行するとと

1：ここで管理ドメインとは，単一の組織や管理権限下で運用されるノードやネットワークの集合のことを言う．

もに，コンフリクト関係を再現することで，実行時の挙動が正しかったかどうかを検証する．

本論文の構成は以下の通りである．まず，2 章で従来技術とその課題について述べる．次に，3 章と 4 章で，提案する ScalarDL Validator の設計と実装についてそれぞれ述べる．そして，5 章で予備的な評価実験を行い，6 章でまとめを述べる．

2 従来技術と課題

本章では，従来のビザンチン故障遮蔽技術，および検知技術について，データベースシステムへの応用とその課題を中心に述べる．

2.1 ビザンチン故障遮蔽技術

ビザンチン故障遮蔽技術は，データの改ざんのような悪意ある攻撃などの任意の故障を遮蔽するため，State Machine Replication (SMR) の文脈で広く研究されてきた [2,3]．本論文では，SMR ベースのビザンチン故障遮蔽技術を BFT SMR と分類する．BFT SMR は，故障レプリカを許容するため，クライアントからの要求を複数のレプリカ間に複製する．許容する故障レプリカ数を f とすると，少なくとも $3f+1$ のレプリカが必要である．BFT SMR では，レプリカ間で同じ状態を得るため，通常，クライアントからの要求はシーケンシャルに実行される．

データベースシステム向けに BFT SMR を拡張する研究も行われている [6,10]．本論文では，これらを BFT DB と分類する．Byzantium [6] は，データベースにビザンチン故障遮蔽技術を適用し，複数のトランザクションを ACID かつ並列に実行可能にした BFT DB の 1 つである．Byzantium は，レプリケーションに PBFT [3] を用いているため，$3f+1$ のレプリカが必要である．Basil [10] は，各トランザクションを複数のレプリカにブロードキャストして，コミットまたはアボートを独立に決定および投票させる．そして，クライアントがその投票を集め，クォーラムベースで最終決定を行う．Basil は，レプリカとのやりとりを，トランザクションの送信，実行，結果受信からなる一度のラウンドトリップ通信に抑えるため，$5f+1$ のレプリカが必要である．

2.2 ビザンチン故障検知技術

ビザンチン故障検知技術は，BFT SMR や BFT DB とは異なるアプローチで，ビザンチン故障を扱う技術である．ビザンチン故障検知技術 (BFD) は，f の故障レプリカを検知するにあたって，$f+1$ のレプリカさえあればよいが，遮蔽技術のように運用を継続することはできない．BFD 研究の先駆けである PeerReview [7] は，プライマリノードでハッシュチェーンで繋がれた全順序の実行ログを作成し，それを witness と呼ばれるセカンダリノードでシーケンシャルに再実行することで，レプリカ間の状態が同じであることを検証する．SMR に似た特徴から，PeerReview は BFD SMR に分類する．PeerReview は，データベースシステムに応用することも可能だが，セカンダリノードにおけるログのシーケンシャルな再実行が，トランザクションの並列実行を阻害するという課題がある．

ScalarDL [12] は，BFT DB の必要レプリカ数が多いという課題や，BFD SMR およびその拡張におけるトランザクション並列実行に関する課題を解決した BFD DB である．ScalarDL は，Ledger と Auditor と呼ばれる 2 つのコンポーネントが有する 2 つのデータベースレプリカを用いて同期的にビザンチン故障を検知する．ここで，同期的とは，トランザクション実行時に，2 つのレプリカ間で矛盾があればその場で検知することを意味する．対して，ScalarDL Validator は，Ledger と Auditor のインタラクションによる同期的なビザンチン故障検知を放棄するトレードオフとして，非同期的なビザンチン故障検知の採用により，トランザクションスループットとレイテンシを性能向上させる BFD DB の新しいアプローチである．以下，2.3 節で同期的なアプローチにおける性能上の課題について詳細に述べる．

2.3 課　　題

ScalarDL における従来の同期的なビザンチン故障検知の流れは次の通りである．1) Auditor がクライアントからのトランザクション要求をコンフリクトの有無に基づいて順序付けする (Ordering フェーズ)，2) Ledger が決められた順序にしたがってトランザクションを実行しコミットする (Commit フェーズ)，3) Auditor が Ledger のトランザクション実行結果と実行順序を検証する (Validation フェーズ)．この 3 フェーズプロトコルは，いずれかのレプリカが，Strict Serializablity に違反したトランザクション実行や，データ改ざんなどのビザンチン故障を引き起こした場合，その場で検知が可能である．しかし，トランザクション実行時，Ledger と Auditor は，Ordering フェーズと Validation フェーズで同期的にコンセンサス処理 (合意形成) を行わなければならないため，レイテンシが大きいという課題がある．

3 設　　計

本章では先述の課題を解決する ScalarDL Validator の設計について述べる．

3.1 設 計 方 針

ScalarDL Validator は，従来の ScalarDL と同様のシステムモデルを前提とする．すなわち，ビザンチン故障したノードは任意の挙動を取りうる想定であり，同一管理ドメインのノード間でビザンチン故障は伝搬しうる (管理ドメインが異なればビザンチン故障は伝搬しないし，結託もない) 想定である．また，必要とするレプリカ数 (管理ドメイン数) についても，ScalarDL 同様に実用性を鑑みて 2 つを維持する．

ScalarDL Validator が保証する安全性 (safty) とライブネス (liveness) の基準についても，ScalarDL 同等を目指す．すなわち，安全性とはデータベースシステムが Strict Serializability を保証することであり，故障レプリカの数に応じて以下のシステム特性を提供する．

故障レプリカがない場合: 両レプリカとも故障していない場

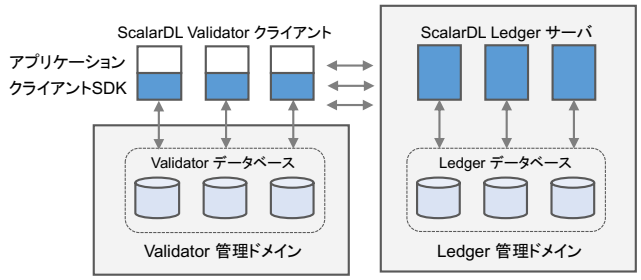


図 1: ScalarDL Validator システム構成

Algorithm 1: Logging フェーズ

```
// client: An abstract client for Ledger
1 Function logging(request)
2     beginClock = getClock()
3     registerRequest(request, beginClock)
4     result = client.execute(request)
5     endClock = getClock()
6     registerResponse(request, result, endClock)
```

合，ScalarDL Validator は，安全性とライブネスを保証する．

故障レプリカが 1 つの場合: いずれかのレプリカが故障している場合，ScalarDL Validator は，安全性を保証する．すなわち，正常なクライアントがデータベースシステムで発生したビザンチン故障を非同期的に検知する．例えば，データベースシステムが，Strict Serializablity に違反するようなスケジュールでトランザクションを実行した場合，正常なクライアントは後から遡ってそれを検知可能である．

故障レプリカが 2 つの場合: 両レプリカが故障している場合，ScalarDL Validator は，安全性もライブネスも保証しない．ただし，管理ドメイン同士が結託しないかぎり，2 つのレプリカが不正に同一になる可能性は極めて低いことから，多くの場合，正常なクライアントにより故障 (例えば，改ざんのような状態の不一致) を検知可能である．

3.2 システムの概要

図 1 に ScalarDL Validator のシステム構成図を示す．ScalarDL Validator システムは，Validator と Ledger の 2 つのコンポーネントで構成される．前者の Validator は，Validator クライアントと Validator データベースで構成される．Validator クライアントは，Ledger サーバへアクセスするためのクライアントとしての役割に加えて，Validator データベースにトランザクション要求の内容や実行結果を保存するなど，非同期的にビザンチン故障検知を行うための機能を提供する．後者の Ledger は，従来と同様に，クライアントからの要求に応じてトランザクションの実行とコミットを行う．

Validator と Ledger は，それぞれ異なる管理ドメインで各々データベースレプリカを管理する．各レプリカでは，トランザクションが読み書きするデータに加えて，その読み書きのロジックを定義した決定的なプログラムを保持する．クライアントはこのプログラムの識別子と必要なパラメータを指定して Ledger にトランザクションの実行を要求する．また，各レプリカは，シングルノードで構成してもよいし，性能やクラッシュ故障耐性を高めるために複数ノードで構成してもよい．

3.3 検知プロトコル

ScalarDL Validator における非同期的ビザンチン故障検知プロトコルの基本的な考え方は，従来の同期的なプロトコルのように Strict Serializability が保証されたトランザクションの順序を事前に決定するかわりに，Ledger がコミットしたトランザクションの正しさを，後から Validator が検証するというものである．この事後検証のため，提案プロトコルは，Logging フェーズと Validation フェーズで構成される．Logging フェーズでは，Validator クライアントが，Ledger にトランザクション要求を送信するのに先だって，その要求内容 (プログラム識別子や引数など) と時刻を記録する．また，Ledger からトランザクションの実行結果を受領したら，その内容と時刻を記録する．Validation フェーズでは，Validator クライアントが，Logging フェーズで記録されたトランザクションの要求内容に基づいてトランザクションを再実行し，Ledger がコミットした際の実行結果と同一かどうかを検証する．さらに，実行時刻と読み書き対象に基づいて，コンカレントかつコンフリクトのあるトランザクションを特定し，MVSG (Multi-Version Serialization Graph) を用いて Strict Serializable な実行になっていたかを検証する．MVSG は，トランザクション間の Reads-From 関係に加え，バージョン順序に関するエッジを張った有向グラフであり，グラフに循環がないことを確認することで与えられたスケジュールがシリアライザブルかどうかを判定する [1]．文献 [11] におけるブラックボックス環境における Serializablity 検証とは異なり，バージョン順序が得られる前提のため，多項式時間で判定可能である．具体的には，V をノード数 (ここではトランザクション数)，E をコンフリクト関係に基づいて張られたエッジの数とすると，$O(V+E)$ で判定可能である．以下，それぞれのフェーズについて詳細に説明する．

3.3.1 Logging フェーズ

Logging フェーズの処理フローを Algorithm 1 に示す．Logging フェーズでは，まず，クライアントがアプリケーションプログラムから受領したトランザクション要求を，取得した時刻 (開始時刻) とともに Validator データベースに記録する (Line 1–2)．トランザクション要求は，$\langle n, f, a, s\rangle$ で構成され，n は要求を識別するための一意なトランザクション ID (例えば，UUID)，f はプログラム (トランザクションの実行ロジック) の識別子，a はプログラムの引数，s は，n，f，a から生成されたメッセージ認証コードである．メッセージ認証コードとしては，例えば，クライアント秘密鍵による署名や，クライアントと Ledger で共有する鍵から生成したハッシュ値 (i.e., Hash-based Message Authentication Code, HMAC) などが利用できる．

次に，クライアントは Ledger に要求を送信してトランザクションを実行する (Line 4)．Ledger におけるトランザクション処理は，従来プロトコルの Commit フェーズと同様であるため

詳細は割愛するが，Ledger はプログラムに記述された内容にしたがってアトミックにデータベースのレコードを読み書きし，トランザクション ID 毎に状態 (コミットまたはアボート) を記録する．また，Ledger はクライアントに対して，プログラムの実行結果とともに，読み書きした各レコードについてプルーフと呼ぶトランザクション実行時の入出力情報を作成し，応答する．プルーフは $\langle k, v, n, d, s \rangle$ で構成され，k はレコードの主キー，v はレコードのバージョン番号，n はトランザクション ID，d は当該トランザクションが入力としたレコードの主キーとバージョン番号の集合，s は k，v，n，d から生成されたメッセージ認証コードである．

Ledger から実行結果とプルーフを受領後，クライアントは，取得した時刻 (終了時刻) とともにそれらを記録する (Line 5–6)．より具体的には，プルーフから当該トランザクションの Read-Write セットを作成した上で，開始時に記録したトランザクション要求と紐付けて保存する．ここで，Read-Write セットとは，当該トランザクションが読み書きしたレコードの主キーとバージョン番号の集合であり，レコードの値そのものは含まない．Read-Write セットはプルーフの k，v，d から作成可能である．

3.3.2 Validation フェーズ

Validation フェーズの処理フローを Algorithm 2 に示す．Validation はレコードのバージョン単位，または，トランザクション単位で実行する．レコードのバージョン単位で行う場合は，主キー *key* とバージョン番号 *version* を指定してプルーフを取得し，当該バージョンを書いたトランザクションを特定してから Validation を行う (Line 2–3)．トランザクション単位で行う場合は，事前にトランザクション ID が判っているため，そのまま Line 3 以降を実行する．

Validation フェーズの検証処理では，当該トランザクションの入出力が正しかったのかどうかを検証する入出力チェック (Line 3–6) と，当該トランザクションとコンカレントで，かつコンフリクトのあるトランザクションの集合が正しいスケジュールで実行されたのかを検証する Serializability チェック (Line 7–9) の 2 つを行う．

入出力チェックでは，当該トランザクションが書いたレコードのバージョン全てについて，Validator データベース内に保存したレコード毎のプルーフを取得し，その入出力を検証する (Line 5–6)．検証の主たる処理は，*deriveStates*() 関数で行われる．はじめに，当該レコードを出力するにあたって使用した入力レコードの集合を得る (Line 11–14)．入力レコードの主キーとバージョン番号は，各プルーフから得られる．また，同じくプルーフから得られるトランザクション ID を用いて，当該トランザクション要求に応じて実行したプログラムとその引数を得た上で，プログラムを再実行する (Line 15–17)．再実行した結果得られる状態，すなわち出力レコードが Ledger 側と一致しない場合は検証エラーを送出し，一致する場合はそれらを Validator のデータベースに書き出す (Line 18–21)．

Serializability チェックの主たる処理は，*createMVSG*() 関数で行われる MVSG 作成処理 (Line 22–36) と，作成した MVSG に循環がないかを確認する処理である (Line 8–9)．MVSG は，対象トランザクションとコンカレントな候補トランザクションを検索し (Line 31)，それらの候補トランザクションがコンフリクト関係にある場合に，対象トランザクションから候補トランザクションにエッジを追加する (Line 32–35) ことで作成していく．コンカレントな候補トランザクションは，Logging フェーズで各トランザクション要求に紐付けて記録した開始終了時刻に基づいて検索する．また，コンフリクト関係にあるかどうかは，同様に記録した Read-Write セットを用いて，トランザクションが読み書きしたレコードとそのバージョンに基づいて判定する (Line 34)．起点とする検証対象のトランザクションと直接コンカレントかつコンフリクトな関係にない場合でも，推

Algorithm 2: Validation フェーズ

```
1  Function validate(key, version)
2      proof = getProof(key, version)
3      request = getRequest(proof.txId)
4      foreach (k, v) ∈ request.writeSet do
5          p = getProof(k, v)
6          deriveStates(p)
7      graph = createMVSG(proof.txId)
8      if graph has a cycle then
9          throw a validation error
10 Function deriveStates(proof)
11     states = {} // An array of record
12     foreach input ∈ proof.inputs do
13         record = getRecord(input.key, input.version)
14         states.push(record)
15     request = getRequest(proof.txId)
16     function = getFunction(request.functionId)
17     recomputedStates = function(states, request.args)
18     if recomputedStates is diverged then
19         throw a validation error
20     else
21         putRecords(recomputedStates)
22 Function createMVSG(originTxId)
23     graph = {} // A map from transaction ID to node
24     todo = {} // A stack for outstanding transactions
25     todo.push(originTxId)
26     while todo is not empty do
27         txId = todo.pop()
28         if graph does not have txId then
29             request = getRequest(txId)
30             graph.put(txId, Node(txId))
31             candidates = getConcurrentTransaction(txId)
32             foreach candTxId ∈ candidates do
33                 r = getRequest(candTxId)
34                 if hasConflict(request, r) then
35                     Add an edge from txId to candTxId
36                     todo.push(candTxId)
```

移的に関係するトランザクションが Serializability 違反を犯す可能性があるため，コンカレントかつコンフリクトのあるトランザクションは再帰的に確認し，グラフに追加していく (Line 36)．

上記のように MVSG を作成し，循環がないかを確認することで Ledger 側のトランザクション実行がシリアライザブルであることは検証可能だが，タイムトラベル異常のような Linearizability 違反は検出できない．そのため，例えば，文献 [12] で例示されているように，不当に古いバージョンを参照しつつ，Serializability 違反にならないスケジュールでトランザクションをコミットさせるといった不正ができてしまう．クライアント側で記録したトランザクションの開始終了時刻に基づいて各バージョンを読んでもよい時間 (可視時間と呼ぶ) を求めておき，あるバージョンを読んだトランザクションが，当該バージョンの可視時間と整合するかどうかを確認することで Strict Serializability を保証することができる．可視時間算出方法の詳細検討は今後の課題である．

3.4 障害復旧処理

クライアントや Ledger の障害，ネットワーク障害など様々な理由により，トランザクション要求を送信したものの，応答を得られないまま，Logging フェーズが正常終了しないことがある．ここでは，そうしたトランザクションの障害復旧処理について簡潔に述べる．

障害復旧処理では，まず，終了時刻が打刻されていない不定トランザクションを特定する．各不定トランザクションについて，Ledger に当該トランザクションの状態を問い合わせる．問合せの結果，コミットまたはアボートが確定している場合，現在の時刻をもって終了時刻を打刻する．コミット済みの場合には，Read-Write セットも併せて取得し，当該トランザクション要求のレコードに紐付けて保存する．問合せの結果，状態が不明の場合，当該トランザクションは処理中の可能性があるため，判定を見送る．一定時間後を経てリトライしても不明のままならば，アボート済みの場合と同様に終了時刻のみ打刻する．Ledger におけるトランザクション処理の (コミットかアボートのいずれかに至るまでの) タイムアウトを考慮してリトライを行えば，Validator 側で判断を誤ることはない．Ledger がタイムアウト以上に不当にコミットを先送りしたとしても，Validator 側に残された記録から矛盾は検出可能である．

4 実　　装

本章では 3 章で述べた非同期的ビザンチン故障検知プロトコルを実装する Ledger と Validator の詳細について述べる．

4.1 Ledger

非同期的ビザンチン故障検知プロトコルにおける Ledger のトランザクション処理は，従来プロトコルの Commit フェーズと同様であり，従来の ScalarDL Ledger の実装をそのまま流用する．したがって，ScalarDL Validator システムにおけるトランザクションは，コントラクト [12] と呼ぶ決定的なプログラムを用いた one-shot モデルで実行される．Ledger データベースは多次元マップの Key-Value 形式で抽象化されており，コントラクトを介して ACID に読み書きできる．

Ledger は，各機能を下位のデータベースシステムに非依存で実現するため，データアクセス層を抽象化している．抽象化されたデータアクセス層を実装するトランザクションマネージャとして，ScalarDB [9] を使用している．ScalarDB は，高スケーラブルな分散トランザクションマネージャであり，ACID なトランザクション機能をもたない NoSQL などを含めた多様なデータベース上で，ACID トランザクションを実現する．ScalarDB の利用により，Ledger データベースとして，PostgreSQL や MySQL などの ACID なデータベースに加えて，Apache Cassandra や Amazon DynamoDB など，性能やコスト，スケーラビリティの要件に応じて ACID でないデータベースも使用することができる．

4.2 Validator

4.2.1 概　　要

Validator は，3.3.1 節と 3.3.2 節で述べた Logging フェーズと Validation フェーズを実装する．現在のプロトタイプでは，Validator は，ScalarDL のクライアント SDK と同様に，Ledger へアクセスするライブラリとして実装されている．アプリケーションは Validator 用のクライアント SDK を用いて，認証用の鍵登録，コントラクトの登録，実行を行う．また，Validator データベースへのアクセスは，Ledger 同様に抽象化されている．そのため，ScalarDB を用いることで，下位データベースとして多様なデータベースを選択できる．

4.2.2 時 刻 管 理

提案プロトコルが非同期的に Serializability を検証する上で重要な情報の 1 つは，Logging フェーズで取得するトランザクションの開始終了時刻である．Validator では，この開始終了時刻として Lamport の論理クロック [8] を応用する．

開始終了時刻として，TrueTime [5] や Timestamp Oracle を採用しない理由は次の通りである．まず，TrueTime のような物理的な時刻を使用しないのは，全てのシステムでそうした正確な時刻管理サービスが利用できるとは限らないためである．汎用的な NTP サーバで TrueTime を提供することも考えられるが，その運用は必ずしも容易ではない．また，中央集権的な Timestamp Oracle サーバは，単一障害点となりやすい課題がある．仮に複数サーバで運用するとしても，性能スケーラビリティを維持しつつ，時刻の単調増加を保証するのは容易ではない．そこで，クライアントが分散かつスケーラブルな方法で時刻を打刻するため，Lamport の論理クロックを応用する．

ScalarDL Validator では，複数のクライアント (スレッド) がトランザクション要求を発行する．また，そのクライアントの数は求められる性能などの要件に応じて増減する可能性がある．そのため，Lamprot の論理クロックのアルゴリズムのように，クライアント間で互いにクロックを送受信するアプローチを取るのは現実的ではない．そこで，ScalarDL Validator では，この送受信処理をクライアント間で共有している Validator デー

タベース上のクロックテーブルの読み書き操作で代用する．

具体的には，クロックテーブルに各クライアントが使用するローカルカウンタに相当する複数のカウンタレコードを用意する．用意するカウンタの数については，4.2.3 節で議論する．クライアントは，トランザクションの開始終了時にクロックテーブルをスキャンし，得られたカウンタの最大値に 1 加えた値をその時点の論理クロックとして使用する．また，クロックテーブルのレコードも，新しい論理クロックの値に更新する．

なお，コンカレントなトランザクションを漏れなく特定する目的においては，各クライアントによるクロックテーブルのスキャンと更新はトランザクションとして実行する必要はなく，古いクロックをスキャンしてしまっても問題ない．これは，もしある 2 つのトランザクションが実際にコンカレントであるならば，一方の開始 (または終了) にともなうクロック更新が，必ずもう一方の終了 (または開始) にともなうスキャンによって観測され，開始終了時刻がオーバーラップするからである．あるトランザクション T_i の実際の開始時刻は T_i のクロック更新より後であり，それとは別のトランザクション T_j の終了に伴うスキャンは T_j のコミットより後に発生するため，もし T_i と T_j がコンカレントであれば T_j のスキャンは必ず T_i のクロック更新を観測する．古いクロックを取得することで，実際にはコンカレントでないトランザクションに対して，あたかもコンカレントであるかのように時刻を付与してしまうことはあるが，Serializability チェックの結果には影響しない．

4.2.3 時刻管理の最適化

論理クロックのスキャンと更新はデータベースへのアクセスをともなう比較的重い処理であるため，クロックテーブルのカウンタレコード数やその更新頻度は重要なチューニングパラメータである．

クロックテーブルにおけるカウンタレコードは，いくつかのクライアントスレッドで共有して利用することでその数を削減することができる．これにより，スキャンするレコード数が減少する一方，論理クロックの更新が競合して性能が低下する可能性がある．

また，時刻取得時，対応するカウンタレコードの値を参照するのみとし，定期的にスキャンと更新を省略することもできる．この場合，あたかもコンカレントであるかのように見えるトランザクションが増大することとなる．これによる弊害は 2 つある．1 つは，Serializability チェックでコンカレントなトランザクションを抽出する際に候補トランザクションが増えてしまう点である．ただし，コンフリクトがなければグラフそのものが大きくなるわけではない．もう 1 つは，Validator として後から確認できるバージョンの可視時間が伸びることになるため，Ledger 側で不正に古いバージョンを読んでも露見しない可能性が高まる．ただし，Validator による時刻管理と，Ledger のトランザクション管理は独立，すなわち異なる管理ドメインで行われる処理であり，Ledger が論理クロックの進行を悪用し，都合のよい契機で不正を行うことは困難であると考える．

5 評　　価

本章では，ScalarDL Validator が従来方式と比較して，どの程度トランザクション性能が向上するかを評価する．比較対象は，Auditor 構成の ScalarDL [12] である．

5.1 ワークロード

ワークロードとしては，クラウドサービス向けのベンチマークで，OLTP システムの評価にも使用されている YCSB [4] を使用した．複数種ある YCSB のワークロードのうち，読込み専用の Workload C と読み書き両方を行う Workload F を使用した．Workload C では，各トランザクションが 2 つのレコードを読み込み，Workload F では，各トランザクションが 1 つのレコードを読み書きする．データベースのレコード数は 100 万行，ペイロードサイズは 100 バイトとした．なお，本実験では，ScalarDL Validator の基礎的な性能特性を明らかにするため，アクセス対象のレコードは一様ランダムに決定した．

5.2 実 験 条 件

実験は，Amazon Linux 2 が動作する AWS EC2 のインスタンスを使用して行った．Ledger，Auditor，Validator の各サーバ用のインスタンスとして，c5d.4xlarge (8 CPU コア，32GB メモリ, NVMe SSD) を使用した．Auditor 構成の ScalarDL では，Ledger と Auditor で 2 つのインスタンスを使用し，それぞれのインスタンスでデータベースサーバも動作させる．Validator 構成においても，Ledger と Validator で 2 つのインスタンスを使用するが，Validator 本体はクライアント上で動作するため，1 つは，Validator データベース用のインスタンスとして使用する．また，クライアントには，c5.2xlarge (4 CPU コア，16GB メモリ) を使用した．

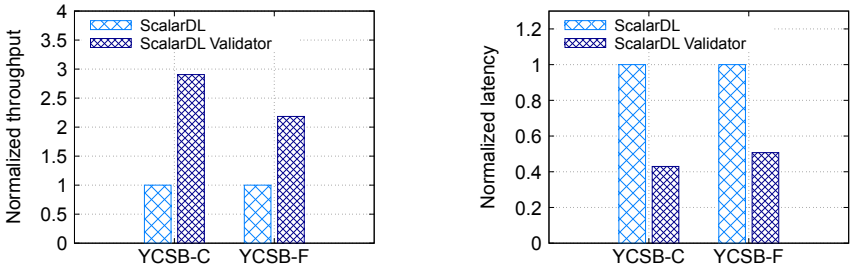
データベースとしては，PostgreSQL v14.5 を使用した．PostgreSQL は，`max_connections` を 500，`shared_buffers` と `max_wal_size` を 16GB，`max_locks_per_transaction` を 512 に設定した．また，`isolation_level` に関しては，各々Serializablity 保証に必要な要件に合わせて，Auditor 構成では，`READ_COMITTED`，Validator 構成では `SERIALIZABLE` とした．

Auditor，Validator いずれの構成についても，2 つの管理ドメインを作成した．それぞれの管理ドメインは，別々の組織，管理者が運用することを想定し，別々のネットワークで構成した．また，Auditor，Validator で使用するメッセージ認証コードは，どちらも電子署名を用いて生成した．使用した電子署名のアルゴリズムは，SHA-256 ハッシュによる ECDSA である．

実験における測定時間は，ウォームアップ 10 秒，実測 60 秒で，各々3 回測定した値の平均値を使用する．

5.3 YCSB 実験結果

YCSB の各ワークロードを実行した際のトランザクション性能の比較を図 2 に示す．トランザクションスループットは，クライアントスレッド数を 1〜96 まで増やしたときのピーク値で比較し，レイテンシは，クライアントスレッド数が 1 の場合の



(a) スループット改善効果　　(b) レイテンシ改善効果

図 2: YCSB 実験結果

値で比較した．いずれも ScalarDL を 1 とした場合の相対値である．YCSB-C，YCSB-F どちらのワークロードも全体として傾向は同じであり，Validator 構成は，Auditor 構成と比較して約 2.2 倍〜2.9 倍のトランザクションスループットが得られた．Auditor 構成は，同期的なビザンチン故障検知を行うために，2 相ロックの亜種によるトランザクションの順序付けや，コントラクトの再実行が必要である．そのため，それらが Ledger 単体のコントラクト実行やコミット処理と同等以上のレイテンシを要した結果，このような性能差になったと考える．

6 おわりに

本論文では，非同期的なビザンチン故障検知が可能なデータベースミドルウェア ScalarDL Validator を提案した．ScalarDL Validator は，トランザクション実行時に同期的にコンセンサス処理を行って故障を検知するのではなく，トランザクション実行後，事後的に検証処理を行うことで非同期的にビザンチン故障を検知する．プロトタイプを用いた初期評価では，従来方式と比較してトランザクションスループットおよびレイテンシが改善することを示した．

今後の課題は，Strict Serializability 保証のためのバージョン可視時間を用いた検証方式や障害復旧処理方式の詳細検討，および Validation フェーズや論理クロック最適化に関する評価実験の実施などである．

文　献

[1] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[2] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[3] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186. USENIX Association, 1999.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154. Association for Computing Machinery, 2010.

[5] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 251–264. USENIX Association, 2012.

[6] R. Garcia, R. Rodrigues, and N. Preguiça. Efficient middleware for byzantine fault tolerant database replication. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 107–122. Association for Computing Machinery, 2011.

[7] A. Haeberlen, P. Kouznetsov, and P. Druschel. Peerreview: Practical accountability for distributed systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 175–188. Association for Computing Machinery, 2007.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[9] Scalar, Inc. Scalardb. `https://github.com/scalar-labs/scalardb`, 2023.

[10] F. Suri-Payer, M. Burke, Z. Wang, Y. Zhang, L. Alvisi, and N. Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 1–17. Association for Computing Machinery, 2021.

[11] C. Tan, C. Zhao, S. Mu, and M. Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[12] H. Yamada and J. Nemoto. Scalar dl: Scalable and practical byzantine fault detection for transactional database systems. *Proc. VLDB Endow.*, 15(7):1324–1336, 2022.