分散データストアの因果整合性違反度測定手法

高塚　康成†　　首藤　一幸†

† 東京工業大学情報理工学研究科　〒 152–8552 東京都目黒区大岡山 2–12–1
E-mail: †takatsuka.y.aa@m.titech.ac.jp

あらまし　既存の分散データストアの多くは結果整合性を保証するのみであり，それら分散データストアに対し因果整合性を保証する手法が提案されている．しかし，それら手法を用いると整合性保証のためのオーバヘッドが生じ，アクセス性能が低下する．そこで，クライアントがどれだけ因果整合性違反を観測するかを測定できれば，性能を低下させてまで因果整合性を保証する必要があるかを判断できる．本研究は任意のワークロードにおける因果整合性違反度の測定手法を提案する．違反度の測定にはクライアントログを用いる．また，提案手法を既存の分散データストアに適用し，因果整合性違反がどの程度発生しているかを調べる．

キーワード　因果整合性/Causal Consistency，分散データストア，ログ解析

1. はじめに

分散データストアでは，アクセス性能の向上を目的に，データの複製を分散配置する．そのような手法を用いた場合，全ての複製が常に同一の値を示すように同期を維持するには大きなコストがかかり，データストアのアクセス性能を低下させてしまうため，複製間の整合性はある程度犠牲にせざるを得ない．したがって，多くの分散データストアは，複製間の整合性について結果整合性（eventual consistency）という弱い整合性モデルを採用している．

しかし，結果整合性を採用したデータストアでは，データの収束性が保証されるのみであり，データの更新順序は考慮せずに同期が行われるため，データ間の依存関係が崩れる可能性がある．すなわち，そのような分散データストアにおいて因果整合性（causal consistency）は保証されない．

分散データストアで因果整合性を保証する手法は研究されている．しかし，それら手法を用いると，データ間の依存関係を崩さない処理のために性能面でオーバヘッドが生じ，データストアの性能が低下してしまう．したがって，それら手法の利用を検討する場合，性能を低下させてまで因果整合性を保証する必要があるか判断する必要がある．その判断を行う上で，クライアントがどれほどの因果整合性違反を観測するか測定する手法は有効である．

本研究は，因果整合性を違反した読み出しがどれほど発生したか測定する手法を提案する．本手法は違反の数を調べるのみでなく，違反原因の分析も行う．

また，ケーススタディとして，提案手法を用いた実験により，既存の分散データストアにおいてクライアントがどれほどの因果整合性違反を観測するか調べた．既存の分散データストアとしては Apache Cassandra と Amazon DynamoDB を用い，アクセス対象のレコード数を変更した場合の違反度合いの変化を調べた．両データストアとも，アクセス対象のレコード数を変化させた場合に，因果整合性を違反した読み出しの数が大きく変化することを確認した．DynamoDB を用いた実験では，アクセス対象のレコード数が減少するほど違反読み出しの数が増加した．一方で，Cassandra を用いた実験では，DynamoDB とは異なり，ごく少数のレコードにアクセスが集中する場合に違反読み出しの数が減少した．

本論文の構成は以下の通りである．2 章で因果整合性について述べ，3 章で関連研究を述べる．4 章で提案手法を述べる．5 章で既存の分散データストアを対象とした，提案手法を用いた違反度測定実験の結果を示す．6 章で本研究の貢献と今後の課題を述べる．

2. 背　　景

本省では，整合性モデル，特に本研究が扱う因果整合性について述べる．

2.1 整合性モデル

複製間の整合性については，直線化可能性（linearizability）[17] や，順序整合性（sequential consistency）[19]，因果整合性（causal consistency）[10] [21]，結果整合性（eventual consistency）[14] など，様々な整合性モデルが提案されている．

弱い整合性モデルは，データストア利用者に古いデータを返す等の好ましくない状況を招く．したがって，直線化可能性や順序整合性のような強い整合性モデルを保証することが望ましい．しかし，それら強い整合性モデルは，データの複製を地理的に分散配置した状況では保証できないことが知られている [8] [22]．

既存の多くの分散データストアは，可用性を重視し，データの複製を地理的に分散配置するため，複製間の整合性については弱い整合性モデルである結果整合性を採用している．結果整合性は，新たに更新が行われない場合に，いつか全ての複製データが同一の値に収束することのみを保証した整合性モデルである．

本研究が扱う因果整合性も，データの複製を地理的に分散配置した状況で保証できる整合性モデルの一つである．結果整合性がデータの収束性を保証するのに対し，因果整合性はデータ間の依存関係が崩れないことを保証する．以下で，本研究が扱

う因果整合性について説明する．

2.2 因果整合性

因果整合性とは，データ間の依存関係が崩れないことを保証するデータストアの性質である．例えば，チャットアプリケーションにおいて，クライアント 1 により投稿されたメッセージ A を見たクライアント 2 が，A への反応としてメッセージ B を投稿した場合，第三者のクライアント 3 から見て，メッセージ B は見えるもののメッセージ A は見えないという状況は，文脈が読めないため望ましくない．因果整合性を保証した分散データストアでは，そのような状況は生じない．

2.3 データ間の依存関係

上述した例の場合，データ（メッセージ）B の書き込み（投稿）はデータ A の存在を踏まえて行われた．このように，データ B がデータ A の存在を前提とする時，B は A に依存するという．以下では，そのような依存関係を $A \leftarrow B$ で表現する．また，データ A をデータ B の依存先データ，データ B をデータ A の依存元データという．

因果整合性の定義に基づくと，データ A とデータ B の間に $A \leftarrow B$ の関係が生じるのは，以下のいずれかの場合である．ここで，$W(A)$ は A に関する書き込み操作を，$R(A)$ は A に関する読み出し操作を表すこととする．

- 同一クライアントによる書き込み後書き込み（W–W）
  あるクライアントが $W(A)$ の後に $W(B)$ を行った場合
- 任意クライアント間の読み出し後書き込み（W–R–W）
  あるクライアントが $W(A)$ を行い，その A を読み出した（$R(A)$ を行った）クライアントが，後に $W(B)$ を行った場合
- W–W と W–R–W に関する推移性
  例えば，あるクライアントが $W(A)$ を行った後に $W(A')$ を行い，さらに，任意のクライアントが $R(A')$ を行った後に $W(B)$ を行った場合など

3. 関 連 研 究

この章では，本研究の関連研究として，分散データストアで因果整合性を保証する研究と，整合性の違反度合いを測定する研究について述べる．

3.1 分散データストアで因果整合性を保証する手法

分散データストアで因果整合性を保証する手法は研究されている．その手法は大きく二つに分類できる．一つは，因果整合性を保証する機能を分散データストアに直接実装する手法である．もう一つは，クライアントと分散データストアの間に設けたミドルウェアで因果整合性を保証する手法である．後者の手法は，データベース管理システムの実装に手を加えることなく，既存の分散データストアに対し因果整合性の性質を付加することができるという利点がある．一方で，前者の手法は，後者の手法に比べ，ミドルウェアを用いることにより生じる通信遅延のオーバヘッドがないため性能面で勝る．

前者の手法としては，Chain reaction [11] や COPS [13], Eiger [20], Orbe [18] が，後者の手法としては，Bolt–on Causal Consistency [7] や矢口らの手法 [23] が挙げられる．

どちらの手法を用いたとしても，分散データストアで因果整合性を保証しようとすると，データ間の依存関係を崩さない処理のためのオーバヘッドが生じるため，結果整合性のみを保証した分散データストアに比べ性能が低下してしまう．したがって，これら手法の利用を検討する場合，性能を低下させてまで因果整合性を保証する必要があるか判断する必要がある．

3.2 整合性の違反度合いを測定する手法

分散データストアにおける整合性の違反度合いを測定する手法は研究されている．

Wada らの研究 [12] は，結果整合性のみを保証した分散データストアにおいて，同一のデータアイテムに対する連続した複数の書き込みが発生した時，直後の読み出し操作が古い方の値を読み出してしまう可能性がどれほどあるか，および，どれ程古い値が返され得るか（*staleness* と呼ばれる指標）について調査した研究である．

また，Bermbach らの研究 [15] では，*staleness* の測定に用いるクライアント数を増やし，測定した複数の *staleness* 情報を付き合わせることで，分散データストアの全複製が同一の最新値に収束するまでの時間を推定する手法を示している．

Bermbach らの後続研究 [6] では，上記手法を用いて，時期による Amazon S3 [3] の整合性保証度合いの変化を調査している．

上述した研究は全て，分散データストアが保つことのできる整合性の限界を知ることを目的とした研究である．すなわち，整合性違反が発生しやすいように設計した，固定のワークロードを用いることを前提としている．したがって，これら研究では，任意のワークロードにおいて，実際にクライアントがどれ程の違反を観測するか知ることはできない．本研究は，ワークロードを限定せず，クライアントがどれだけの因果整合性違反を観測するか測定する手法を提案する．

人為的に整合性違反が発生しやすい状況を作り出し，その際の整合性違反度合いを調査するツールがある．Jepsen [16] は，ネットワーク分断の存在下で，分散データストアが普段保証している整合性がどれだけ崩れるか検証することを目的とした，分断耐性テストツールである．iptables を用いて人為的にネットワーク分断を引き起こし，書き込みデータの損失などがどの程度発生するかを調査する．

4. 因果整合性違反度測定手法

本節では，クライアントがどれだけの因果整合性違反を観測するか測定する手法を示す．違反度合いの測定にはクライアントログを用いる．

以下では，まず，本手法の測定対象である因果整合性違反を定義し（4.1 節），違反度測定のおおまかな流れを示す（4.2 節）．次に，測定に用いる依存関係グラフについて述べ（4.3 節），依存関係グラフを用いた違反検出手法（4.4 節）を述べる．最後に，因果整合性違反原因の分析手法を述べる（4.5 節）．

4.1 因果整合性違反の定義

因果整合性違反は，データストア上の因果整合性違反と，クライアントの観測する因果整合性違反の二種類を定義できる．本研究が測定対象とするのは後者の違反である．

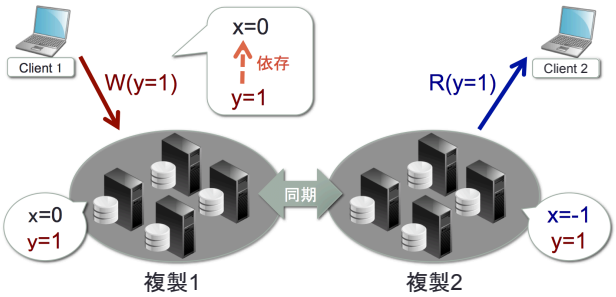

図 1　データストア上の因果整合性違反例

4.1.1　データストア上の因果整合性違反

データストア上の因果整合性違反とは，ある複製上に存在するデータについて，同一複製上にその依存先データが存在しない，もしくは依存先データの古いバージョンしか存在しない状態を指す．ここで，データ A がデータ B の古いバージョンであるとは，A と B の変数が同じで，B が A に依存する場合を指す．

この違反は，複製間における同期処理の遅れに起因する．例えば，図 1 において，クライアント 1 がデータ $x=0$ に依存するデータ $y=1$ を複製 1 に書き込み，そのデータが複製 2 に伝播されている．しかし，複製 2 には未だ $x=0$ が伝播されておらず，$x=0$ の古いバージョンである $x=-1$ が存在している．このようなデータストアの状態は，$y=1$ の依存先データが全て複製 2 へ伝播されるまで $y=1$ を読み出せないようにするなどの工夫をしない限り，複製 2 にアクセスしたクライアントが，$y=1$ は見えるものの，その依存先データである $x=0$ は見えないという結果を招き得る．つまり，データストア上の因果整合性が保証されない場合，クライアントが依存関係の崩れを観測する可能性が生じる．

4.1.2　クライアントの観測する因果整合性違反

データストア上に，上述した「データストア上の因果整合性」を違反した（つまり，依存先データが同一複製上に存在しない）データが存在したとしても，その依存関係の崩れをクライアントが観測するとは限らない．例えば，図 1 において，$y=1$ はデータストア上の因果整合性を違反したデータである．もし，クライアント 2 が $y=1$ を読み出した後に，$x=-1$ を読み出した場合，$x=-1$ は $y=1$ が依存する $x=0$ の古いバージョンであるため，クライアントは依存関係の崩れを観測したこととなる．しかし，もし，クライアント 2 が $y=1$ を読み出した後に，$x=0$ や $x=0$ より新しいバージョンを読み出した場合や，x に関して読み出しを行わなかった場合，クライアント 2 は依存関係の崩れを観測しない．また，そもそも，$y=1$ を読み出さなければ依存関係の崩れは観測されない．クライアントが依存関係の崩れを観測しなければ，そのクライアントからは，データストアが因果整合性を保っているように見える．

クライアントが依存関係の崩れを観測するのは，そのクライアントが過去に読み出したデータが依存するデータの古いバージョンを読み出してしまった場合である．

あるクライアントの読み出し操作 $R(x_i)$ は，以下の条件を満たす時，依存関係の崩れを観測したこととなる．ここで，データ x_i について，x は変数，i は値を表す．また，$R(x_i)$ を行ったクライアントが $R(x_i)$ の実行以前に読み出したデータの集合を $PastReadData$ とする．

$$R(x_i)\ is\ a\ violation \\ iff\ \exists data \in PastReadData[\exists x_j[x_i \leftarrow x_j \wedge x_j \leftarrow data]] \quad (1)$$

上記定義中の x_j は，過去に読み出したデータの一つが依存し，かつ，x_i より新しいバージョンのデータである．自分が過去に読み出したデータが依存する，より新しいバージョンのデータが存在しているにも関わらず，それを読み出せなかったため，$R(x_i)$ は因果整合性を違反した読み出しといえる．以下では，検証中の読み出し $R(x_i)$ を違反たらしめる x_j を $ViolationEvidence$ と呼び，VE と略す．VE は一つであるとは限らない．

本研究は，上記の定義に基づき，因果整合性を違反した読み出しがどれだけ行われたかを調べる．

4.2　違反度測定の流れ

提案手法は，ワークロード実行フェイズとログ解析フェイズに分けられる．

（1）　ワークロード実行フェイズ

任意の数だけクライアントを用意する．それらクライアントは，アクセスログを取りながら，分散データストアに対して読み書き処理を行う．ログの各エントリは，単一の読み出し処理や書き込み処理ごとに作成される．アクセスログの各エントリには以下の情報を記録する．

- オペレーションの種類（Read or Write）
- オペレーションが対象とする変数
- 読み書きした値
- オペレーション実行時のタイムスタンプ

（2）　ログ解析フェイズ

全クライアントのアクセスログから依存関係グラフ（4.3 節参照）を生成する．また，因果整合性の違反度合いを測定する対象のクライアントを一つ選び（以下，テストクライアントと呼ぶ），そのアクセスログから読み出しログのみを抽出する．抽出した読み出しログと依存関係グラフを比較することで，一つ一つの読み出しが因果整合性を違反した読み出しでないかを静的に検証する．以下では，検証中の読み出しを $VerificationTargetRead$ と呼び，R_v と略す．

4.3　依存関係グラフ

依存関係グラフとは，書き込み操作間の依存関係を示したグラフである．任意のデータについて，そのデータを書き込んだ書き込み操作は一意に定まるため，データ間の依存関係は書き込み操作間の依存関係に等しい．2.3 節で述べたように，書き込み操作間の依存関係は，W–W と W–R–W，および，それらの推移性による依存関係の三つである．したがって，あるデータがどのデータに依存するかは，そのデータに対応する書き込み操作から，W–W と W–R–W の依存関係を辿ることで知ることができる．

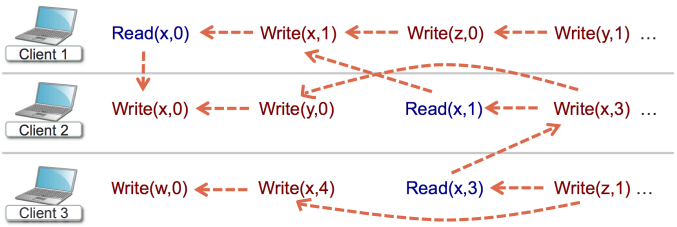

図 2　依存関係グラフの例

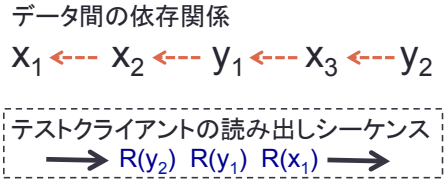

図 3　*ViolationEvidence* の検出が重複する例

依存関係グラフの実体は，各オペレーションを頂点とし，依存関係を辺とした有効グラフである．二種の依存関係は，オペレーション間に以下の三つの辺を張ることで表現する．

- W–W を表現
  - 1．ある書き込みから，それ以前の書き込みへ向う辺
- W–R–W を表現
  - 2．ある書き込みから，それ以前の読み出しへ向う辺
  - 3．ある読み出しから，その元書き込み（読み出したデータを書き込んだ書き込み操作）へ向う辺

図 2 に依存関係グラフの例を示す．簡単のため，図 2 に示したグラフは，1 と 2 の辺について、直前のオペレーションに対する辺以外は省略している．

4.3.1　依存関係部分グラフ

4.1.2 項で述べた因果整合性違反の定義に従い，ある読み出し操作 R_v が依存関係の崩れを観測したか否かを素朴に調べると，R_v 以前にテストクライアントが読み出した全てのデータについて，R_v が読み出したデータとの間に依存関係があるか，および，その依存関係上に R_v が読み出したデータより新しいバージョンのデータ（VE）が存在するかを調べなければならない．一つの読み出し操作を検証するたびに，それ以前に読み出した全てのデータについて上記のような検証を行うと，テストクライアントの読み出しログ全てを検証するのに膨大な計算量を要する．また，そのような素朴な手法を用いると，VE の検出に重複が出る．例えば，図 3 において，テストクライアントの読み出し操作 $R(x_1)$ が因果整合性を違反した読み出しであるか検証する場合，過去に y_1 と y_2 を読み出しているため，素朴な手法では，y_1 と y_2 のそれぞれと x_1 の依存関係上に x_1 より新しいバージョンが存在するか調べることとなる．y_1 と x_1 の依存関係を調べると，x_2 が VE として検出される．さらに，y_2 と x_1 の依存関係を調べると，x_2 と x_3 が VE として検出され，x_2 の検出が重複する．

ここで，過去に読み出されたデータ群のうち，依存関係上最も新しいデータ群（以下，*LatestData*）を導入する．*LatestData* とは，過去に読み出された他のデータからは依存されないデータ群のことである．例えば，図 2 において，テストクライアントが $x=1$ と $x=3$，$y=1$，$z=1$ を過去に読み出している状況では，$y=1$ と $z=1$ が *LatestData* に当たる．

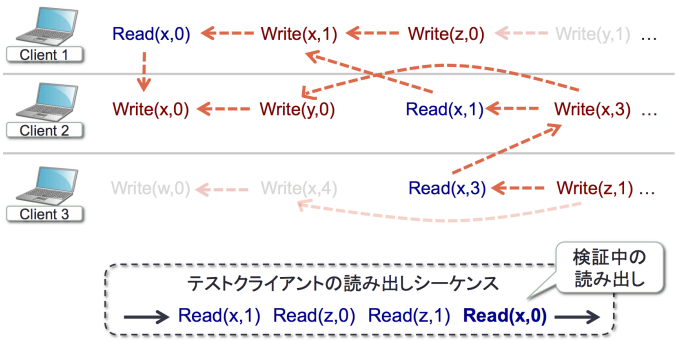

図 4　依存関係部分グラフの例

上述した重複検出を避けるには，*LatestData* についてのみ，R_v が読み出したデータとの比較検証を行えば良い．なぜなら，*LatestData* 以外の過去に読み出されたデータは，*LatestData* に含まれるデータのうち少なくとも一つから依存されるため，*LatestData* 以外の過去に読み出されたデータと R_v が読み出したデータの依存関係上に VE が存在した時，その VE は，*LatestData* に含まれるデータと R_v が読み出したデータの依存関係上にも存在するからである．

したがって，依存関係グラフにおいて VE が存在し得るのは，*LatestData* に対応する書き込み群（以下，*LatestWrites* とする）が依存する部分グラフ上である．図 2 の例において，テストクライアントが $x=1$ と $z=0$，$z=1$ を過去に読み出している場合，*LatestWrites* に当たる $Write(z,0)$ と $Write(z,1)$ が依存する部分グラフに関して調査を行えば十分である．

また，4.1.2 項の定義より，VE は R_v が読み出したデータに依存するデータである．したがって，依存関係グラフにおいて VE が存在するのは，R_v が読み出したデータに対応する書き込み（以下，*VerificationTargetWrite* と呼び，W_v と略す）に依存する部分グラフ上である．図 2 の例では，$Write(x,0)$ が W_v である時，W_v に依存する，各クライアントにおける依存関係上最も古い書き込みである $Write(x,1)$ と $Write(y,0)$，$Write(z,1)$ 以降について調査を行えば十分である．

したがって，依存関係グラフにおいて VE が存在し得るのは，*LatestWrites* が依存し，かつ，W_v に依存するオペレーションで構成される部分グラフ上である．この限定したグラフを依存関係部分グラフと呼ぶ．図 4 は，上述した例の場合に生成される依存関係部分グラフを示したものである．依存関係部分グラフの中に W_v と同じ変数に関する書き込みがあれば，すなわち，R_v は因果整合性を違反した読み出しである．

4.4　因果整合性違反の検出手法

まず，違反検出の準備として，4.3 節の定義に従い，全クライアントのアクセスログから依存関係グラフを作成し，また，テストクライアントのアクセスログから読み出しログのみを抽出する．抽出した読み出しログについて，最初の読み出しから順に違反を観測していないか検証していく．

以下では，$LatestWrites$ に含まれる書き込みのうち，クライアント i による書き込みを LW_i と略す．

まず，R_v が読み出したデータの元書き込み W_v がクライアント i による書き込みであった場合，依存関係グラフを参照し，W_v と LW_i のどちらがより新しい書き込みかを調べる．

W_v が LW_i より古い書き込みであった場合，W_v は $LatestWrites$ から依存される書き込みであるため，VE があるか調べる．4.3.1 項で述べたように，依存関係グラフから，現 $LatestWrites$ が依存し，W_v に依存する依存関係部分グラフを生成し，その中に W_v と同じ変数への書き込みがあるかどうかを調べる．W_v と同じ変数への書き込みがあれば，R_v は新しいデータを読み出せていなかったことがわかり，因果整合性を違反した読み出しとなる．

W_v が LW_i より新しい書き込みだった場合，過去に読み出したデータより依存関係上新しいデータを読み出したこととなるので，R_v は因果整合性違反を観測しない．また，そもそも，未だクライアント i の LW が存在しない（$LatestWrites$ 中に LW_i が存在しない）場合も，過去に読み出したデータからは依存されないデータを読み出したこととなるため，やはり，R_v は因果整合性違反を観測しない．これらの場合，W_v や W_v が依存する書き込みにより $LatestWrites$ が更新される可能性があるため，W_v を考慮して $LatestWrites$ の更新を行う．

また，$W_v = LW_i$ の場合も同様に，R_v は過去に読み出した最新のデータを読み出したこととなるため，因果整合性違反は観測されない．この場合は $LatestWrites$ を更新する必要もない．

以上をまとめると，あるクライアントが因果整合性違反を観測したかどうかを検証するアルゴリズムは以下の通りである．

（1） 全クライアントのアクセスログから依存関係グラフを作成する．

（2） テストクライアントを定め，テストクライアントのアクセスログから読み出しログのみを抽出する．

（3） 抽出した読み出しログの最初の読み出し操作を R_v とし，R_v により読み出されたデータの元書き込み W_v を特定する．

（4） W_v がクライアント i による書き込みの場合，W_v と LW_i のどちらが新しい書き込みか調べる．

- W_v が LW_i より古い書き込みの場合

  W_v と $LatestWrites$ から依存関係部分グラフを生成し，そのグラフ中に W_v と同じ変数への書き込みが存在するか調べる．存在すれば，R_v は因果整合性を違反した読み出しである．
- W_v が LW_i より新しい書き込みの場合，および，LW_i が存在しない場合

  検証中の読み出しは因果整合性を違反した読み出しではない．W_v を考慮して $LatestWrites$ を更新する．
- $W_v = LW_i$ の場合

  検証中の読み出しは因果整合性を違反した読み出しではない．

（5） 抽出した読み出しログにおける R_v の次の読み出しを次の R_v に設定し，その元書き込み W_v を特定して（4）の操作に戻る．抽出した読み出しログにおける最後の読み出しを検証し終えるまで同様の操作を繰り返す．

4.5 因果整合性違反の内訳分析

本研究は因果整合性を違反した読み出しの数を測定するのみでなく，それぞれの違反読み出しが，どのような種類の依存関係違反を観測したかまで分析する．

4.5.1 依存関係の分類

本研究では，データ間の依存関係を次の五通りに分類する．

- $WWuni$

  同一変数間の W–W による依存関係
- $WWdiff$

  $WWuni$ を除く，任意変数間の W–W による依存関係
- $WRWuni$

  同一変数間の W–R–W による依存関係
- $WRWdiff$

  $WRWuni$ を除く，任意変数間の W–R–W による依存関係
- $Others$

  上記以外の依存関係（W–W と W–R–W を組み合わせた依存関係）

4.5.2 内訳分析手法

4.1.2 項で定義した通り，ある R_v が依存関係の崩れを観測したと判定されるのは，テストクライアントが過去に読み出したデータの元書き込み（以下，$W_{PastRead}$ とする）が依存し，かつ，R_v の元書き込み W_v に依存する，W_v と同じ変数への書き込み（すなわち，VE）が存在する場合である．4.4 節で述べた通り，依存関係の種類を考慮せず，ただ VE を見つけるのみであれば，VE が $W_{PastRead}$ からどのような依存関係で依存され，また，VE が W_v にどのような依存関係で依存するかは限定しない．

ここで，検出された VE について，$W_{PastRead}$ から VE への依存関係（$VE \leftarrow W_{PastRead}$）と，$VE$ から W_v への依存関係（$W_v \leftarrow VE$）について，その種類を 4.5.1 項で述べた各種依存関係のどれか一つに限定してもなお両依存関係が成立しているのであれば，4.1.2 項の定義より，R_v はその限定した種類の依存関係について違反を観測したこととなる．

本研究は，検出したそれぞれの VE について，W_v への依存関係と $W_{PastRead}$ からの依存関係の種類を限定し，なお両依存関係が成立するかどうか調べることで，R_v がどのような種類の依存関係違反を観測したか分析する．

ある VE が特定種類の依存関係違反に該当するかどうかの判定は，次のようにして行う．まず，依存関係部分グラフを用いて，W_v から特定種類の依存関係のみを逆に辿ることで，検証中の VE にたどり着くことができるかを調べる．辿り着くことができた場合，今度は VE から，特定種類の依存関係のみを逆に辿り，テストクライアントが過去に読み出したデータの元書き込みのうち，いずれか一つに辿り着くことができるか調べる．辿り着くことができれば，その VE を持つ R_v は，特定種類の依存関係違反を観測したこととなる．

本研究では上記手法を用い，検出したそれぞれの VE

が，4.5.1 項で述べた $WWuni$ と $WWdiff$，$WRWuni$，$WRWdiff$ のいずれに該当するかを検証し，どれにも該当しない VE を $Others$ として分類した．

本研究では，W–W や W–R–W の依存関係を，同一変数間のみに認める場合と任意変数間に認める場合の二通りで分類した．しかし，アプリケーションのセマンティクスを考慮すると，一部の異なる変数間において依存関係が認められない場合がある．例えば，あるクライアントが記事 A に投稿を行った後に，全く関係ない記事 B に投稿を行った場合，その二つの投稿（書き込み）には依存関係が存在しない．アプリケーションのセマンティクスを取り入れ，特定の変数間にのみ依存関係を認めた内訳分析手法の考案は今後の課題である．

5. 実　　験

ケーススタディとして，提案手法を用いた実験により，既存の分散データストアでどれほど因果整合性違反が発生するか調べた．違反度測定対象の分散データストアとしては，Apache Cassandra [4] [9] と Amazon DynamoDB [1] を用いた．

5.1 両データストアを用いた実験の共通設定

本論文では，両データベースについて，アクセス対象のレコード数を変えた場合の違反度合いの変化を調べた．

データストアにアクセスするクライアントとして Yahoo! Cloud Serving Benchmark（YCSB）[5] を使用した．YCSB は NoSQL 向けのベンチマークツールであり，読み出しおよび書き込みの比率や，データベースへのアクセス分布，また 1 秒あたりに発行する処理要求数の目標値（スループット）を指定できる．ただし，YCSB にはアクセスログを取る機能は無いため，読み書き処理実行時にログを取るように追加実装を施した．

依存関係グラフ作成工程の中に，全ての読み出し操作について，その元書き込みを特定する工程がある．同じ変数に関して全く同じ値の書き込みが存在しないことを仮定した場合，アクセスログ全体を走査することで，読み出したデータの変数と値から元書き込みを特定することは可能である．しかし，一つの読み出しの元書き込みを特定するたびにアクセスログ全体を走査すると依存関係グラフの作成に膨大な時間がかかってしまう．したがって，本実験では，元書き込みの特定を効率化するために，書き込む値にクライアント ID とタイムスタンプ情報を埋め込むように YCSB の実装を変更した．読み出した値に埋め込まれたそれら情報を用いることで，元書き込み特定のために走査するアクセスログの範囲を大幅に削減することができる．

両実験では，YCSB で指定できるワークロードの各種パラメータについて，スループット以外は同じパラメータを用いた．ワークロードの読み書き比率は 1:1 とし，アクセス分布は Zipf 分布を用いた．1 レコードあたりの容量は 100 byte とし，1 クライアントあたり 5 万オペレーションを実行した．また，データストア上のレコード数は 1000 万レコードとした．

5.2 Apache Cassandra を対象とした実験

OSS である Apache Cassandra について，アクセス対象のレコード数を変えた場合の違反度合いの変化を調べた．

表 1　利用したマシンの構成

OS	Linux 3.16.0–45–generic, Ubuntu 14.04.2 LTS
CPU	2.40 GHz Xeon(R) E5620 × 2
メモリ	8 GB RAM
Java	Java SE 8 Update 45

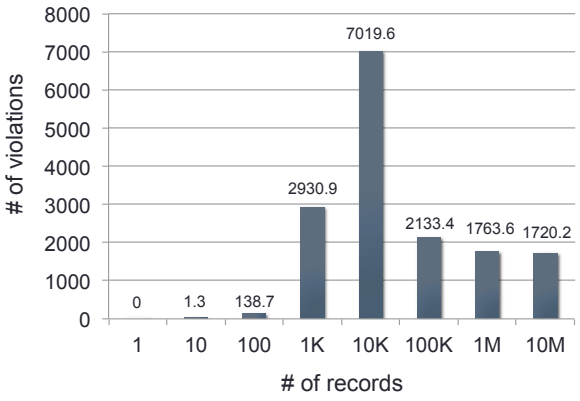

図 5　因果整合性を違反した読み出しの数（Cassandra）

5.2.1 実 験 環 境

実験に用いた Cassandra の構成はは以下の通りである．

- 3 データセンタ
- 1 データセンタあたり 2 ノード
- 1 データセンタにつき 1 つの複製

表 1 に利用したマシンの構成を示す．ストレージデバイスには SSD を使用した．

YCSB の実行用に，データセンタごとに 1 つ，合計で 3 台のサーバを用意し，異なるサーバ上の YCSB が互いに異なるデータセンタにアクセスするように設定した．また，YCSB による読み書き処理のスループットは 1000 ops/sec に設定した．

5.2.2 実 験 結 果

アクセス対象のレコード数を変更することで，因果整合性を違反した読み出しの数がどのように変化するかを調べた．図 5 は，1 クライアントあたりの違反読み出し数を示したものである．

アクセス対象のレコード数が過度に少ない場合はほとんど違反が発生しなかった．これは，アクセス対象レコードが極端に少ない場合，データのほとんどが CPU のキャッシュメモリに乗り，同期処理が高速に行われるためであると考察される．また，アクセス対象のレコード数を増やすと違反数が減少する傾向があった．これは，アクセス対象のレコードが増えるほど，同一データに対するアクセス間隔が，同期処理にかかる時間より長くなり，古いバージョンを読み出す可能性が低下するためであると考えられる．アクセス対象のレコード数をさらに増やすと，違反数が増減しなくなった．これは，アクセス対象レコードの増加に伴いほとんどのデータについて古いバージョンを読み出すことがなくなる一方で，Zipf 分布に従いアクセス頻度が高いままのデータについては，アクセス対象のレコードが増えようと同様のアクセス頻度が維持されるため，それらデータに関する違反読み出しの数は変わらないためであると考えられる．

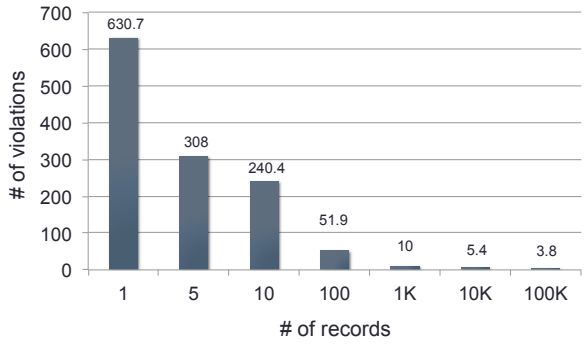

図 6　因果整合性を違反した読み出しの数（DynamoDB）

また，違反の内訳分析をしたところ，4.5.1 項で述べたところの *Others* しか検出されなかった．つまり，クライアントから *W–W*，または *W–R–W* のみによる依存関係の崩れは観測されなかった．

5.3　DynamoDB を対象とした実験

PaaS である Amazon DynamoDB について，データストア上のレコード数を変更した場合の違反度合いの変化を調べた．

5.3.1　実 験 環 境

DynamoDB を起動するリージョンとしては，バージニア北部を指定した．DynamoDB を利用する場合，DynamoDB に対して 1 秒間あたりに発行されるであろう読み書きリクエストの量（provisioned throughput）をあらかじめ指定する必要がある．その指定値を超えた分の読み書きリクエストは処理されない．本実験では，1 秒間あたりに 1000 オペレーションの読み出し，および書き込みを行えるように provisioned throughput を設定した．

YCSB の実行には Amazon EC2 [2] を用いた．YCSB の実行用に，DynamoDB と同じくバージニア北部リージョンで 3 台の EC2 インスタンスを立ち上げた．また，YCSB による読み書き処理のスループットは 500 ops/sec に設定した．

5.3.2　実 験 結 果

アクセス対象のレコード数を変更することで，因果整合性を違反した読み出しの数がどのように変化するかを調べた．図 6 は，1 クライアントあたりの違反読み出し数を示したものである．アクセス対象のレコード数が過度に少ない場合にのみ，違反が発生しているのがわかる．アクセス対象のレコード数が 100 以上になると，違反の数が急激に減少している．このことから，DynamoDB を用いた場合，極少数のデータに多量のアクセスが集中しない限り，クライアントは因果整合性違反を観測しないことがわかる．

また，違反の内訳分析をしたところ，Cassandra の場合とは異なり，*Others* 以外の各種依存関係について違反が観測されていることがわかった．図 7 は，アクセス対象のレコード数を 10 レコードとした場合に観測した各種違反の数を示したものである．違反の多さは $Others > WWdiff > WWuni > WRWuni > WRWdiff$ の順であった．アクセス対象のレコード数を変えた場合も，この順序は変わらなかった．

また，図 8 は $WWuni$ と $WWdiff$ の違反数を，図 9 は

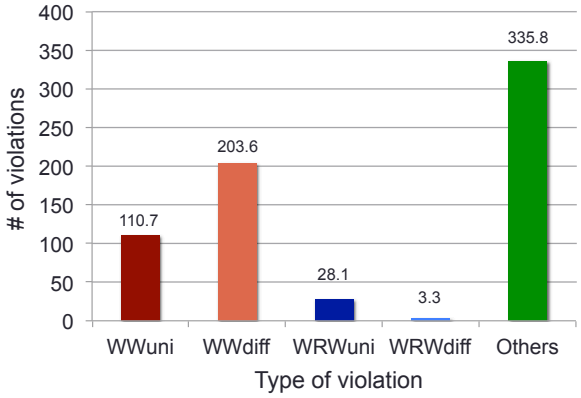

図 7　観測した違反の内訳分析（10 レコード）

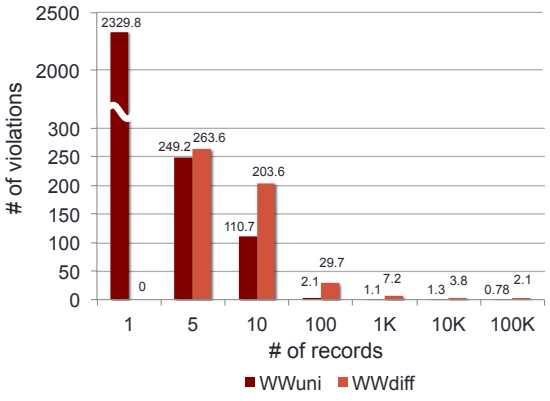

図 8　WW の違反を観測した数

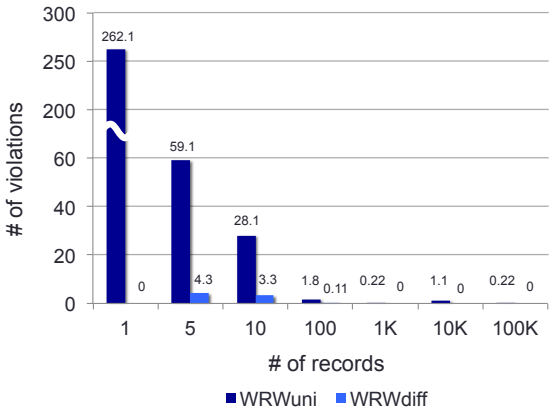

図 9　WRW の違反を観測した数

$WRWuni$ と $WRWdiff$ の違反数を，図 10 は *Others* の違反数を示したものである．図 6 の場合と同様に，アクセス対象のレコード数が 100 以上になると，各種違反の数が急激に減少した．

アクセス対象のレコード数が 1 の場合，当然，同一変数しか読み書きせず，異なる変数間における依存関係違反を観測し得ないため，$WWdiff$ と $WRWdiff$ の違反数は 0 となっている．

これらの実験結果から，5.2.1 項，および，5.3.1 項で述べ

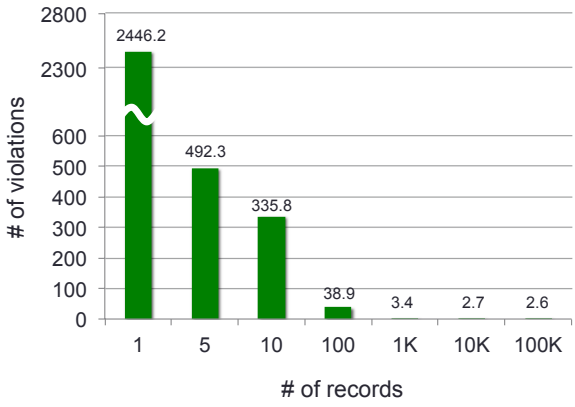

図 10　*Others* の違反を観測した数

た設定においては，極少数のレコードに多量のアクセスが集中する場合，Cassandra を用いた方が，クライアントが違反を観測する可能性が低く，そうでない場合，DynamoDB を用いた方が，クライアントが違反を観測する可能性が低いことがわかった．

6. まとめと今後の課題

本論文では，クライアントのアクセスログを用いることで，結果整合性のみを保証した分散データストアにおける因果整合性の違反度合いを測定する手法を提案した．既存研究との差分は二つある．一つは，測定対象のワークロードを限定せず，読み出し操作と書き込み操作で構成された任意のワークロードで違反度を測定する手法を示したこと，もう一つは，測定対象である因果整合性の詳細化を行ったことである．

また，既存のデータストアを用いた実験により，提案手法により違反を検出できることを確認した．本論文では，ケーススタディとして，アクセス対象のレコード数を変更した場合の違反数の変化を調べた．アクセス対象のレコード数を変更することで，違反度合いが大きく変化することを確認した．

アプリケーションのセマンティクスを取り入れた内訳分析手法の考案は今後の課題である．また，提案手法において，単一の読み出し操作および書き込み操作以外の操作（スキャン操作など）に対応することは今後の課題である．

謝辞本研究は JSPS 科研費 25700008，26540161 の助成を受けたものです．

文　献

[1] Amazon Web Services. Amazon DynamoDB. http://aws.amazon.com/dynamodb/.
[2] Amazon Web Services. Amazon EC2. http://aws.amazon.com/ec2/.
[3] Amazon Web Services. Amazon S3. http://aws.amazon.com/s3/.
[4] The Apache Software Foundation. Apache Cassandra. http://cassandra.apache.org/.
[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB, In Proc. SOCC ' 10, pp. 143–154, 2010.
[6] David Bermbach, and Stefan Tai. Benchmarking Eventual Consistency: Lessons Learned from Long–Term Experimental Studies. In Proceedings of the 2nd IEEE International Conference on Cloud Engineering, Best Paper Runner Up Award, 2014
[7] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on causal consistency. In Proceedings of the 2013 international conference on Management of data, pp. 761–772. ACM, 2013.
[8] Seth Gilbert, and Nancy Lynch. Brewer' s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. SIGACT News, Vol. 33, No. 2, pp. 51—59, 2002.
[9] Avinash Lakshman, and Prashant Malik. Cassandra – A Decentralized Structured Storage System, Proc. LADIS ' 09, 2009.
[10] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, P.W. Hutto. Causal Memory: Definitions, Implementation and Programming. Distributed Computing, 9(1):37–49, 1994.
[11] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In Proceedings of the 8th ACM European Conference on Computer Systems, pp. 85–98. ACM, 2013.
[12] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade–offs in Comercial Cloud Storages: the Consumers' Perspective. In Proceedings of the 5th Biennial Conference on Innovative Data Systems Research, pp. 134–143, 2011.
[13] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 401–416. ACM, 2011.
[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon' s Highly Available Key–value Store. Proc. SOSP 2007, 2007.
[15] David Bermbach, and Stefan Tai. Eventual Consistency: How soon is Eventual? An evaluation of Amazon S'3s consistency behavior. In Proceedings of the 6th Workshop on Middleware forService Oriented Computing, pp. 1:1–1:6, 2011.
[16] Kyle Kingsbury. Jepsen. http://jepsen.io/.
[17] Maurice P. Herlihy, and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects, ACM Trans. Program. Lang. Syst., Vol. 12, No. 3, pp. 463–492, 1990.
[18] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: scalable causal consistency using dependency matrices and physical clocks. In Proceedings of the 4th annual Symposium on Cloud Computing, p. 11. ACM, 2013.
[19] Hagit Attiya, Jennifer L. Welch. Sequential consistency versus linearizability. ACM Trans. Comput. Syst., Vol. 12, No. 2, pp. 91–122, 1994.
[20] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In NSDI, pp. 313–328, 2013.
[21] Leslie Lamport. Time, clocks, and the ordering of events in a distributed. Commun. ACM, 21(7):558–565, 1978.
[22] Eric A. Brewer. Towards robust distributed systems. PODC Keynote, 2000.
[23] 矢口 尭, 首藤 一幸. 広域分散データストアに因果整合性を付加するミドルウェア. 第 7 回データ工学と情報マネジメントに関するフォーラム, 2015.