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の設計

矢口 堯[†] 首藤一幸[†]

1. はじめに

分散データストアでは，データの複製を地理的に分散配置することで，アクセスを高速化する手法が用いられている．クライアントが地理的に近いデータの複製にアクセスすることで，アクセス遅延が小さくなる．

しかし，この手法を用いるためには，データの整合性をある程度犠牲にせざるを得ない．そこで，データの分散配置手法を利用し性能を高めつつ，より強い整合性を実現するための研究が多く行われており，特に本研究の対象である因果整合性はその代表である．[1)2)3)4)]

本研究は，因果整合性のない分散データストアの上に被せることで因果整合性を実現するミドルウェアを設計する．ミドルウェア層で因果整合性を保証する事で，基本的な機能を備えた既存の分散データストアを修正することなく再利用できる．ミドルウェア層で因果整合性を実現する既存手法[2)] は，後述する因果関係が巨大化しない状況下でのみ適用可能である．本稿では，因果関係の情報を分割して管理することで，そのような制限なく適用可能な手法を提案する．

2. 因果整合性

因果整合性とは，データ間の依存関係（因果関係）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，データストアから読み出したデータ $x = 1$ と $y = 2$ の情報を踏まえて，データ $z = 3$ を書き込むとする．この時データストアには，$z = 3$ よりも先に $x = 1$ と $y = 2$ が書き込まれたと考えるのが自然である．逆に，$z = 3$ は読み出せるが先に書き込まれたはずの $x = 1$ または $y = 2$ は読み出せない（または古いデータを読み出す）という挙動はクライアントの期待に反しており，望ましくない．因果整合性はこのような挙動が生じないことを保証する．

上述のように，$z = 3$ が $x = 1$ の存在を前提とする時，$z = 3$ は $x = 1$ に依存するといい，$z = 3$ と $x = 1$ は依存関係にあるという．上記の例では，$z = 3$ は $y = 2$ にも依存する．また，依存関係には推移性がある．先の条件に加え，データ $w = 4$ が $z = 3$ に依存するならば，$w = 4$ は $x = 1$ と $y = 2$ にも依存する．このようにして，依存関係は以下のような階層構造をなす．

$$\begin{matrix} x=1 \searrow \\ y=2 \nearrow \end{matrix} z = 3 \rightarrow w = 4$$

因果整合性のない分散データストアは依存関係を解決せずに同期を行うため，依存関係が崩れた状態がクライアントから見える可能性がある．そこで，提案ミドルウェアをデータストアの上に被せ，この依存関係の崩れをクライアントから隠蔽する ．

3. 提案手法

本節では，分散データストアの上に被せることで因果整合性を実現するミドルウェアの設計を提案する．

アーキテクチャの全体像を示す．分散データストアは地理的に離れた複数のデータセンターに分散しており，それぞれデータの複製を保持する．そして，各クライアントは地理的に近いデータセンターにアクセスする．データセンター内では，ミドルウェアを経由してデータストアにアクセスし，読み書き処理を完了する．以下では，ミドルウェアの基礎的なプロトコルを示す．

3.1 因果整合性を保証する基礎的なプロトコル

提案ミドルウェアは，データストア内の依存関係が崩れた状態をクライアントから隠蔽する．そのためには，クライアントに要求されたデータを見せる前に，そのデータが依存するデータ群の存在を確認すれば十分である．以後，この確認処理を依存関係の解決と呼ぶ．

あるデータの依存関係の解決を行うためには，そのデータの依存関係を全て取得する必要がある．既存手法[2)] では，データの書き込み時にそのデータの全ての依存関係の一覧をデータの値と共に保存し，データ

† 東京工業大学大学院
Tokyo Institute of Technology

の読み出し時には，データの値と共にそのデータの全ての依存関係の一覧を取得する．しかしながら，依存関係は実行時間と共に巨大化するため，この方法ではデータの更新毎にその巨大な依存関係の更新も行わなければならず，効率が悪い．

そこで提案プロトコルでは，依存関係の一覧の取得を再帰的に行う．この方式では，各データが直接依存するデータ群の一覧，つまり一階層目の依存関係のみを保存する．そして読み出し時には，一階層目から順に，各データの依存関係を辿り，次の階層の依存関係を取得し，これを繰り返す．このようにして，全ての依存関係を取得し，依存する全てのデータ群の存在を確認する．

しかしながら，この方式では，並行な書き込みにより依存関係情報が上書きされる場合，全ての依存関係を辿れない可能性がある．以下では，この問題の解決方法を示す．

3.2 依存関係の効率的な更新方法

並行な書き込みを防ぐ一般的な方法は排他制御である．しかし，排他制御手法を用いるためには，読み書き処理毎にデータセンター間で通信する必要があり，性能が下がる．

我々は，データセンター内とデータセンター間での並行な書き込みに対する対処法を区別する手法を提案する．まず，データセンター内の並行な書き込みは排他制御を行うことで対処する．一方で，データセンター間の並行な書き込みは保存に指定したキーの変換により対処する．データセンター毎に異なるキーでデータを保存することで，上書きを回避する．そして，データの読み出し時は，データセンター毎の書き込み内容を適切にマージする．このようにすると，クライアントからは複数のキーを扱うことを隠蔽しつつ，上記の問題を解決できる．

提案手法を用いるとデータ量は増加するが，高々データセンターの数倍で抑えられ，データセンター間の排他制御を行わないため，性能の大きな低下を防ぐことができる．

4. 最適化手法

上述のプロトコルの最適化手法を 2 つ述べる．一つ目の手法は，依存関係を辿る回数を少なくすることで，読み出し遅延を小さくする．データの値と共に保存する依存関係を，1 階層目だけでなく，k 階層目までの全てとすることにより，依存関係の解決時に，一度のデータストアへのアクセスで k 階層分の依存関係を辿ることができる．当然，管理する依存関係が増えるため，使用する記憶領域は増加する．従って，この手法は性能とデータサイズのトレードオフを調整するものである．

次に，依存関係の解決時間をクライアントが許容できない場合に対処する手法を提案する．そのような場合は，依存関係を既に解決済みの古いデータを返すことで対処するのが一般的である．素朴な方法は，クライアントが指定する制限時間内ならば，通常通りデータを返し，制限時間まで待っても依存関係の解決が終わらない場合は，古いデータを返すという方式である．しかし，この方式では古いデータを返す場合は常に制限時間まで待つ必要がある．

我々は，依存関係の深さを見積もることにより，依存関係の解決に要する時間を予測する手法を提案する．その予測から，制限時間内でも依存関係の解決が終わらないと判断した場合は，制限時間まで待たずに古いデータを返す．これにより，待ち時間が減少し，性能が向上すると考える．

5. ま と め

本稿では分散データストアの上に被せることで，因果整合性を実現するミドルウェアの設計を提案した．さらに，最適化手法を 2 つ提案した．今後は，提案ミドルウェアを実装し，実験・評価を行う．

参 考 文 献

1) Sérgio Almeida, Joao Leitao, 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.
2) 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.
3) 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.
4) 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.