Vol.2020-OS-150 No.11
2020/7/31

分散KVSにおけるプロキシを用いたトランザクション実装

宮本 基志[1]　三輪 竜也[1]　川島 龍太[1]　松尾 啓志[1]

概要：ビッグデータの登場により, 大規模なデータの保存や高速な検索処理が可能な分散キーバリューストアが一般的となった. Apache Cassandra は全ての KVS ノードがクライアントからの要求を受け付けることで, 高可用性やスケーラビリティ, 高いスループットを実現したが, P2P 型の分散 KVS は, その構成により高速なトランザクション処理を行うことが困難である. 本研究では総外部接続速度が 1Gbps から 10Gbps の比較的中規模な分散 KVS の利用を想定し, 分散 KVS に対して集中型のプロキシを用いトランザクション処理を提供する手法について提案する. トランザクション機能を実装したプロキシと Cassandra との軽量トランザクションを行う実行時間の比較において，KVS ノード数を 1 台から 6 台に増加させた場合，Cassandra では約 45%，プロキシでは約 13%の実行時間の増加となり, 増加割合は約 3 分の 1 となった. また，KVS ノードが 6 台のときには Cassandra と比較し提案手法は 50%-60%の実行速度となり，提案手法は Cassandra より高速に軽量トランザクションを行うことができることを確認した. さらに，単純なトランザクション処理性能をリレーショナル DB である MariaDB と比較したところ，実行速度は約 3 分の 1 となり，従来のリレーショナル DB と比べても大差ない速度で動作することを確認した.

キーワード：分散キーバリューストア, トランザクション, Paxos

1. はじめに

Apache Cassandra [1] に代表される P2P 型の分散キーバリューストア (Distributed Key-Value Store: D-KVS) は, 全ての KVS ノードがクライアントからの要求を受け付けることで, 高可用性やスケーラビリティを確保しつつ，高スループットを実現できる. しかし，P2P 型の D-KVS では，全ての KVS ノードがクライアントからの要求を受け付けるため，KVS ノード間でのデータの一貫性保証が必要である. これは，Paxos [2] などの分散合意プロトコルを用いることで実現可能であるが, データの一貫性保証のためにノード間での通信が多発し, 排他制御やトランザクション処理を効率的に行うことが困難である.

我々の行った先行研究 [3] では，DB ノード群へのクライアント群からの総接続速度が 1Gbps から 10Gbps を想定した比較的中規模の D-KVS を想定し, メニーコアプロセッサを活用したプロキシサーバーによるロックの管理とプロキシ内の転送速度の高速化，排他制御の実装を行った. プロキシ一台でクライアントからの要求を受け付け、ロック管理を行うことで，KVS ノード間で行われる排他制御のための通信を削減しつつ, 分散合意プロトコルを使用した場合と同様に強い一貫性を保証する. さらに，18 コア/36 スレッドの Intel Core i9 -7980XE 上に DPDK によるプロトコルスタックを実装することにより，プロキシの中継速度として約 8M クエリ/秒 (10Gbps) を実現した. しかし，従来のプロキシでは，排他制御のみを提供しているため，トランザクション機能は未実装であった.

そこで本研究では，ACID に準拠したトランザクション機能を持たない D-KVS に対し，プロキシを用いた ACID に準拠したトランザクション機能の実装を行う. プロキシ一台でロックの管理とトランザクション処理を行うことで，KVS ノード間における一貫性保証のための通信を削減し, トランザクション処理を効率的に行う. また，トランザクション機能を持たない KVS においてもプロキシを用いることで，トランザクション機能を利用することが可能となる.

本稿の構成は以下の通りである. まず，第 2 章では D-KVS のトランザクションを実装する上での問題点について説明する. 次に，第 3 章では関連研究について述べ, 第 4 章で先行研究とその問題点について説明する. 第 5 章で提案手法とその実装について述べ, 第 6 章で提案手法の性能比較と考察を行い, 第 7 章でまとめと今後の課題を述べる.

[1] 名古屋工業大学大学院
Nagoya Institute of Technology

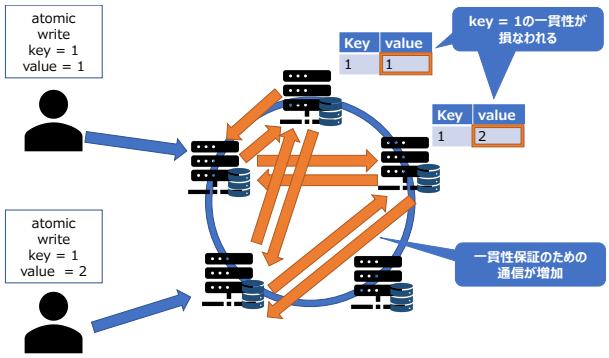

図 1 Paxos による一貫性制御

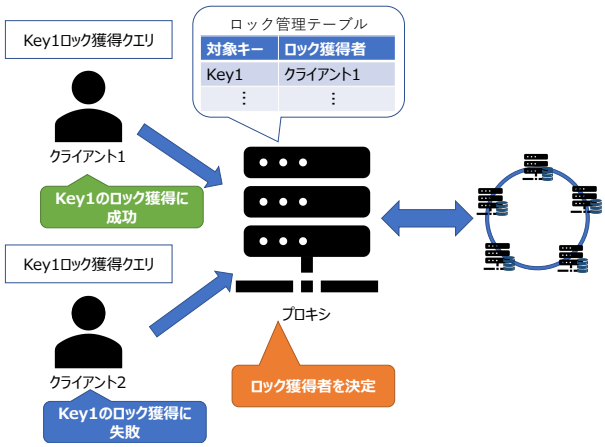

図 2 プロキシによる排他制御

2. D-KVS の問題点

Amazon DynamoDB [4], Apache Cassandra, Riak [5] などの D-KVS は，P2P 型のモデルを採用している．P2P 型のモデルでは KVS ノードに特権を持つホストが存在せず，合意形成には各 KVS ノード同士での分散合意プロトコルによる通信を必要とする．P2P 型は特定のホストに権限が集まらないため耐障害性に優れる．また，P2P 型の D-KVS は，全ての KVS ノードがクライアントからの要求を受け付けるため，BASE 特性 [6] の一つである結果整合性 (Eventual Consistency) を持つ．しかし，想定するアプリケーションが厳密な整合性や一貫性を必要とする場合は，データベース側においてデータの強い一貫性保証が必要となる．Cassandra や Redis，DynamoDB といった D-KVS では複数のデータをアトミックに更新するため，分散合意プロトコルを用いて排他制御やトランザクション処理を実装している [7] [8] [9]．しかし，代表的な分散合意プロトコルである Paxos では，KVS ノード間で最低 2 往復の通信が必要であるため，D-KVS において効率的に排他制御やトランザクション処理を行うことは困難である (図 1)．

3. 関連研究

トランザクション機能を持たない D-KVS 上でトランザクション処理を行う方法が提案されている．ScalarDB [10] はクライアント側のライブラリとして実現され，ストレージシステムとして使用される Cassandra などの KVS が提供しているアトミックなデータ更新機能を使用し，トランザクションやロック状態の調整を行いトランザクション機能を提供する．しかし，この手法ではトランザクションの状態管理に分散合意プロトコルを使用する KVS の機能を必要とするため，トランザクション処理を行うための通信が多く，高速にトランザクション処理を行うことが困難である．

また，近年 NewSQL と呼ばれる分散環境でトランザクション処理を行う方法が提案されている．Google Spanner [11] では，全てのトランザクション処理のコミットに絶対時刻を付加することで順序を管理し，データの一貫性を担保している．しかしこの手法では，許容する時刻の差異を大きくするほど適用にかかる時間が大きくなってしまうため，精度の高い時計が不可欠である．よって，この手法では特別なハードウェアである原子時計や GPS を利用した高精度な時計がノード数に比例した数必が要である．また，Amazon Aurora [12] では，分散ストレージ上に構築されたデータベースでトランザクション処理を行う．Aurora では 6 台のレプリカが存在しているが，コミット時には 4 台のレプリカによるジャーナルの永続化が完了した時点で成功とみなすことで高速にトランザクションを適用している．また，分散ストレージが必要に応じて保持するデータをジャーナルから作成することができるため，トランザクションの適用処理を軽量化することが可能となる．しかし，この手法は DBMS だけでなく専用の分散ストレージが必要となるため容易に実装，配置することは困難である．

4. 先行研究

我々の行った先行研究では，中規模の D-KVS を想定し，メニーコアを活用した集中型のプロキシによるクエリ集約手法の高速化と，排他制御手法を提案した [3] (図 2)．プロキシ内で排他制御にかかるロックの管理を行うことで，一貫性保証のための通信を削減し，Paxos と同様に強い一貫性を保証した．Paxos を用いた Cassandra のロック獲得と，先行研究によるロック獲得を比較した性能評価では，先行研究のスループットが約 30%向上し，ロック獲得によるレイテンシは約 35 倍高速化した．しかし，プロキシを用いたトランザクション処理は未実装であった．

5. 提案手法

本章では，プロキシを使用することにより，ACID に準拠したトランザクション機能を提供する手法を提案し，その実装について述べる．この手法により，KVS ノード間の一貫性保証のための通信を削減し，トランザクション処理の効率化を行う．また，ACID 特性に準拠したトランザクション機能を持たない D-KVS に対してプロキシを用い，

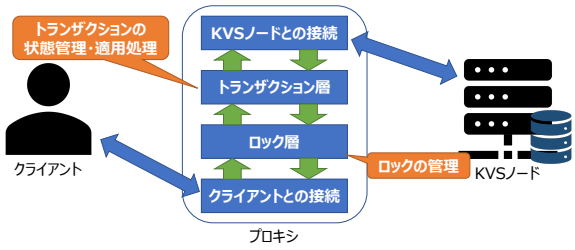

図 3 提案手法の構造

ACID に準拠したトランザクション機能を提供することも可能となる．

5.1 トランザクションの実装

本研究では，トランザクション機能を提供するため，ロック層とトランザクション層に分けて実装を行った (図 3)．ロック層では，トランザクションの対象となっているキーのロックの管理を行う．トランザクション層では，トランザクションの状態管理や適用処理などを行う．

トランザクション層では SQL を処理可能なデータベース (以下，プロキシ内 DB) と，クエリ保存領域を使用しトランザクションの状態管理を行う．実装ではプロキシ内 DB として SQLite3 [13] を，インメモリ DB として使用した．プロキシ内 DB のインスタンスとクエリ保存領域はトランザクション毎に作成される．

トランザクション中の書き込み操作によって変更された値は KVS に送信されないため，クライアントが KVS に対してトランザクション中に書き込まれたキーに対する取得操作をしても，KVS 側からは値を取ることができない．提案手法ではプロキシ内 DB に一旦すべてのクエリを処理させることで，トランザクションの適用によって書き込まれる値と同様の値をプロキシ内 DB に保持する．そして，クライアントからの取得要求に対してプロキシ内 DB が応答することにより，トランザクション中に変更された値を正常に読み出すことができる．

クエリ保存領域はトランザクション中の書き込みクエリを保存する領域である．この領域に書き込まれたクエリはコミット時に KVS に対して送信される．クエリ保存領域に保持されているトランザクションに含まれるクエリを，KVS に送信する直前に永続記憶装置に書き込むことにより，ジャーナル処理を実現することが可能であり，プロキシサーバーのダウンなどの異常終了時の際の復旧と処理継続が可能となる．

5.2 トランザクション処理における動作

プロキシが処理するクエリはトランザクションの制御を行うクエリと，書き込み読み込みを行う通常のクエリに分けられる．トランザクションの制御を行うクエリは BEGIN クエリ，COMMIT クエリ，ROLLBACK クエリに分けることができる．また，通常のクエリはトランザクション実行中か否かによって処理が変化する．本節では，次の 6 動作について説明する．

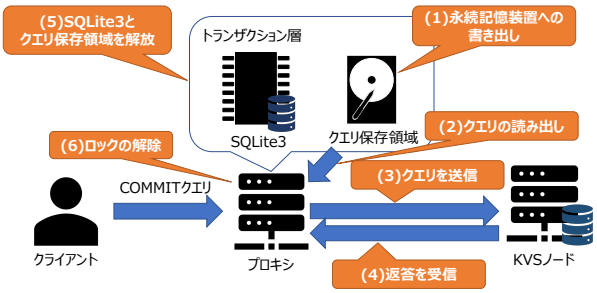

図 4 コミット処理の動作

- BEGIN クエリ
  トランザクションの開始を行うクエリ
- COMMIT クエリ
  トランザクションの適用処理を行うクエリ
- ROLLBACK クエリ
  トランザクションの破棄処理を行うクエリ
- 非トランザクションクエリ
  トランザクションが開始していない場合の読み込み書き込みなどのクエリ
- トランザクション中の書き込みクエリ
  トランザクション中のデータの追加・更新を行うクエリ
- トランザクション中の読み込みクエリ
  トランザクション中に送られる読み込みクエリ．トランザクション中に更新されたキーか否かに関わらず同一の処理を行う．

5.2.1 BEGIN クエリ

BEGIN クエリを受け取るとロック層ではデータベースのロックを行い，トランザクション層ではデータベースインスタンスの作成とクエリ保存領域の確保を行い，BEGIN クエリを KVS 送ることなく返答をクライアントへ返す．

5.2.2 COMMIT クエリ

COMMIT クエリではトランザクションの適用処理を行う．トランザクションの適用では次の操作を行う (図 4)．

(1) トランザクション処理実行中にプロキシに異常が発生した場合にデータの修復を可能とするため，クエリ保存領域を永続記憶装置へ書き出す
(2) クエリ保存領域に書き込まれているクエリを読み出す
(3) 読み出したクエリを KVS へ送信する
(4) KVS から返答を受け取る
(5) クエリ保存領域とデータベースインスタンスを削除する (トランザクションにおけるデータの適用は完了したため)
(6) ロックを解除する

5.2.3 ROLLBACK クエリ

ROLLBACK クエリではトランザクションの破棄を行う．破棄処理ではクエリ保存領域とデータベースインスタ

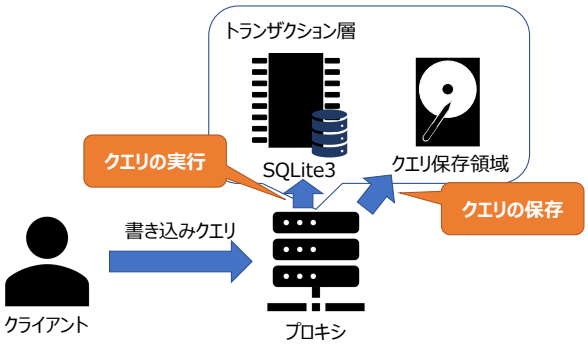

図 5 トランザクション中の書き込みクエリの動作

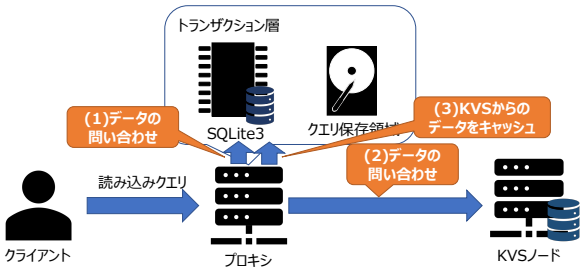

図 6 トランザクション中の読み込みクエリの動作

ンスの解放とロックの解除を行う．KVS との送受信は行わない．

5.2.4 非トランザクションクエリ

トランザクションが開始していない場合のクエリは，すでにトランザクションが開始されているキーに対する操作以外は，プロキシ内では特別な操作を行わず，KVS にそのまま送信される．

5.2.5 トランザクション中の書き込みクエリ

トランザクション実行中の書き込みクエリの動作を図 5 に示す．この動作では，データベースでクエリの実行し，クエリ保存領域にクエリを保存する．その後，KVS へのクエリの送信はせず返答をクライアントへ返す．書き込みクエリの処理ではクエリ保存領域の永続記憶装置への書き込みは行わない．データベースでクエリを実行することで，KVS への書き込みを行うことなくトランザクション中に正しい値の読み込みを行うことが可能となる．

5.2.6 トランザクション中の読み込みクエリ

トランザクション中に読み込みクエリが送られた場合は次の動作を行う (図 6)．

(1) プロキシ内 DB に要求されたデータが存在するか問い合わせ存在する場合はその値を返す
(2) KVS に問い合わせる
(3) KVS からの返答でデータが存在していればキャッシュを行うためプロキシ内 DB に書き込む

この (1) の処理を行うことで，トランザクション中に書き込まれたデータを参照可能としている．

6. 評価

プロキシを使用したことによる性能低下を確認するため，YCSB を用いた性能比較を行った．Cassandra は ACID 準拠のトランザクション機能を持たず，軽量トランザクションと呼ばれる条件を満たすレコードのアトミック更新機能を持つ．そこで，軽量トランザクションにおけるプロキシの有効性を確認するため，Cassandra の軽量トランザクションとプロキシに実装した軽量トランザクションで性能比較を行った．さらに，提案手法と MariaDB のトランザクション処理における性能評価を行った．

表 1 評価に使用する計算機の仕様

	クライアント	KVS ノード	プロキシ
OS	Ubuntu 18.04.4 LTS	Ubuntu 16.04.1 LTS	
CPU	Intel Core i5-4460 (3.20GHz)		
メモリ	16GB		
NIC	Realtek RTL8111G (1Gbps)		

表 2 YCSB のワークロード

ワークロード	動作
WorkloadA	50%読み込みと 50%書き込み
WorkloadB	95%読み込みと 5%書き込み
WorkloadC	100%読み込み
WorkloadD	新規データの追記．新しいデータほど読み込まれやすい
WorkloadE	データの範囲読み込み
WorkloadF	データの読み込み・変更・書き込み処理

6.1 評価環境

プロキシは 5 章で述べた実装方式により実現した．クライアントとの通信には gRPC [14] を用い，Cassandra との通信には DataStax C/C++ Driver for Apache Cassandra [15] を用いた．KVS ノードには Cassandra 3.11.6 を使用した．計算機はプロキシ，クライアント，KVS ノードでそれぞれ別の計算機を使用した．それぞれの仕様を表 1 に示す．

6.2 YCSB による評価

実装したプロキシによる性能低下を評価するため，YCSB を使用したクライアントとプロキシ，KVS ノードを 1Gbps のネットワークに接続し，スループットを測定した．測定ではプロキシを使用せず YCSB と KVS ノードを直接接続した場合と，YCSB とプロキシ，KVS ノードを接続した場合の 2 つを評価し比較を行った．YCSB では標準で 6 つのワークロードを提供している (表 2)．評価では実装したプロキシが現時点では対応していない WorkloadE を除いた 5 つのワークロードを使用し行った．ベンチマークツールは YCSB-C [16] を用いた．

評価結果を図 7 に示す．グラフの縦軸がスループット，横軸が各スループットを表しており，青色の棒グラフはプロキシを使用せず直接 Cassandra にアクセスした場合，オレンジ色のグラフはプロキシを使用した場合をそれぞれ表している．グラフより，提案手法は全てのワークロードに

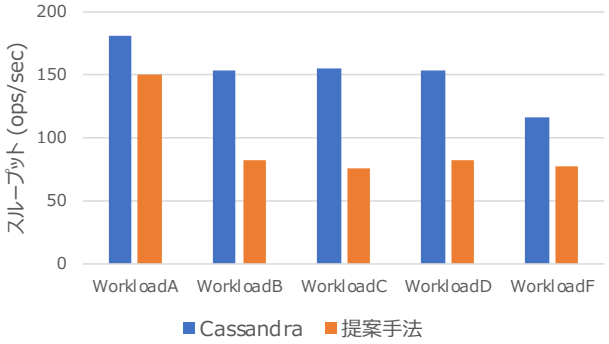

図 7 YCSB による評価結果

```
INSERT INTO usertable(
    y_id,
    field0, field1, field2, field3, field4,
    field5, field6, field7, field8, field9,
) VALUES (
    '1',
    'PK8Mi...', 'nQRS7...', 'hPZhB...', 'ICuQS...', 'zZzN9...',
    '0wvP4...', 'tWXVQ...', 'kVBFt...', 'BKSvm...', 'uPfyX...'
) IF NOT EXISTS
```

図 8 軽量トランザクションクエリの例

おいて Cassandra と比較しスループットが約 17%から約 51%低下した．これは，トランザクションを行わない場合でも，プロキシを通ることで遅延が発生することや，評価に使用したプロキシの実装が OS の提供するソケットを使用したため，システムコールによるコンテキストスイッチのオーバヘッドが発生したためである．しかし，既に我々の研究室では文献 [3] で示したように，DPDK とメニーコアプロセッサを用いたプロキシの実装により 10Gbps のスループットを実現している．今後は今回提案したプロキシにも同様の手法を導入する予定である．

6.3 軽量トランザクションの評価

実装したプロキシの有効性を評価するため，Cassandra の軽量トランザクションと，プロキシで実装した軽量トランザクションの比較を行った．評価のため軽量トランザクションの実行時間を計測するベンチマークツールを作成した．ベンチマークツールでは，Cassandra のデータベースドライバとして DataStax Java driver for Apache Cassandra のバージョン 3.0.0 を使用した．評価では，ベンチマークツールと KVS を直接接続した場合と，ベンチマークツールとプロキシ，KVS ノードを接続しプロキシを使用した場合の 2 つを評価し比較を行った．また，KVS ノード数を 1 台と 6 台の場合でそれぞれ評価を行った．評価では，図 8 で示す形式の軽量トランザクションクエリの実行時間を 1000 回計測し，1000 回の平均値，中央値を算出した．

評価結果を図 9 に示す．縦軸が実行時間，横軸が評価対象を表している．評価対象は左から，Cassandra を直接使用し，KVS ノードが 1 台と 6 台，プロキシを使用し，KVS ノードが 1 台と 6 台の場合をそれぞれ示す．青色の棒グラ

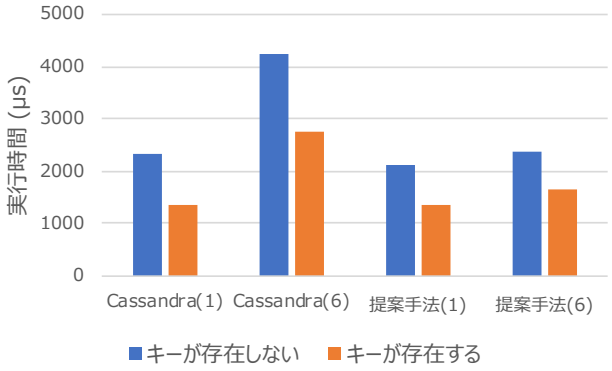

図 9 軽量トランザクションの評価結果

```
BEGIN;
INSERT INTO bench(pk,     field1, field2, field3)
          VALUES ('1...', 10790,  48968,  65922);
INSERT INTO bench(pk,     field1, field2, field3)
          VALUES ('q...', ...);
        .
        .
        .
INSERT INTO bench(pk,     field1, field2, field3)
          VALUES ('7...', ...);
INSERT INTO bench(pk,     field1, field2, field3)
          VALUES ('G...', 17424,  36284,  77950);

COMMIT;
```

図 10 トランザクションクエリの例

フは KVS ノード内にキーが存在していない場合，オレンジ色の棒グラフはキーが存在している場合をそれぞれ示す．KVS ノード数が 1 台のとき，提案手法における Cassandra と比べた実行時間は，キーが存在しない場合では約 90%，キーが存在する場合では約 99%となり，オーバヘッドが小さいことを確認した．また，KVS ノード数が 6 台のとき，提案手法の Cassandra と比較した実行時間は，キーが存在していない場合は約 56%，キーが存在している場合では約 60%となり，プロキシを用いない場合より高速に動作することを確認した．これは，KVS ノード間で行われる一貫性保証のための通信をプロキシを使用したことにより削減したためである．次に，KVS ノード数を 1 台から 6 台に増加させた場合，実行時間は Cassandra ではキーが存在していない場合は約 45%増加したのに対し，提案手法では約 13%の増加にとどまった．これは，KVS ノード数が増加したことにより，プロキシを用いなかった場合は一貫性保証のための通信が増加したのに対し，提案手法では一貫性保証をプロキシ内で行い，KVS ノード間の通信が削減されたためである．以上より，提案手法であるプロキシを用いたトランザクション処理の軽量トランザクションにおける有効性を確認できた．

6.4 トランザクションの評価

実装したプロキシのトランザクションにおける有効性を評価するため，MariaDB [17] のトランザクションと，プロ

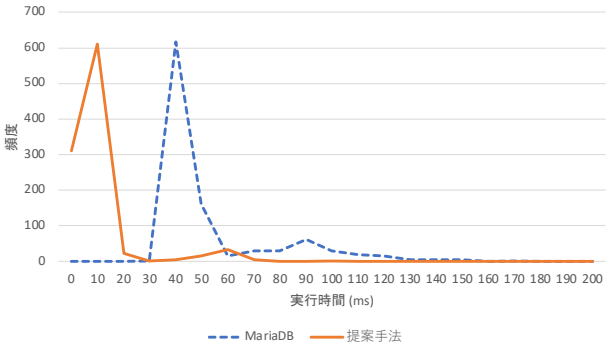

図 11 トランザクションの評価結果のヒストグラム

表 3 トランザクションの評価結果

実行時間 (ms)	平均値	中央値	最大値	最小値
MariaDB	55.804	41.798	408.511	33.328
提案手法	16.809	16.464	108.419	7.119

キシ内に実装したトランザクションの比較を行った．測定では，ベンチマークツールを実装したクライアントノードと MariaDB を使用した場合と，ベンチマークツールとプロキシ，KVS ノードを接続し使用した場合の 2 つを評価と比較を行った．評価では，任意のクエリを含むトランザクションの実行時間を計測するベンチマークツール [18] を作成し使用した．評価では図 10 で示した 10 個の INSERT クエリを 1 つのトランザクションとし，そのトランザクションを 1000 回実行し 1 回あたりの実行時間の平均値と中央値を算出した．

評価結果のヒストグラムを図 11 に，評価の集計結果を表 3 にそれぞれ示す．ヒストグラムの縦軸が頻度，横軸が実行時間をそれぞれ表している．青色のグラフは MariaDB の，オレンジ色のグラフは提案手法の結果をそれぞれ示す．グラフから MariaDB は 40 ミリ秒から 50 ミリ秒の間にピークが現れているのに対し，提案手法では 10 ミリ秒から 20 ミリ秒の間に現れており，提案手法がリレーショナル DB と同等かそれ以上の性能を提供できることを確認した．また，集計結果から提案手法は MariaDB に比べ実行時間が約 1/3 であることが確認できる[*1]．そして，最小値では提案手法は MariaDB の約 4.68 倍の速度でトランザクションを処理を行っていることが確認できた．

7. まとめと今後の課題

本稿では，D-KVS にトランザクション機能を実装したプロキシを導入し，ACID に準拠したトランザクション機能をそれを持たない KVS に対して提供する仕組みを提案した．トランザクション中のデータの参照にインメモリデータベースを使用し，トランザクションの状態保持にクエリ保存領域を使用する形で実装を行った．軽量トランザクションを用いた評価では，KVS ノード数が 6 台のときでは提案手法は Cassandra と比べ約 56%-60%程度の実行速度となり，KVS ノード数を 1 台から 6 台に増加させた際には，Cassandra は実行時間が約 45%増加したのに対し提案手法は約 13%にとどまった．以上から軽量トランザクションにおけるプロキシの有効性を確認した．また，リレーショナル DB との比較では実行時間が約 1/3 となることを確認した．

今後の課題として，DPDK を使用したプロキシにトランザクション機能を実装しスループットの改善を行うことや，プロキシが単一障害点となる問題の解消などがある．

[*1] MariaDB に比べて，現在プロキシに実装しているトランザクション処理は機能が少ない．従ってこの実験は高速化が目的ではなく，プロキシによるトランザクション処理の実現が軽量に行えることを評価することが目的である．

参考文献

[1] Lakshman, A. and Malik, P.: Cassandra: A Decentralized Structured Storage System, *SIGOPS Oper. Syst. Rev.*, Vol. 44, No. 2, pp. 35–40 (online), DOI: 10.1145/1773912.1773922 (2010).
[2] Lamport, L.: The Part-Time Parliament, *ACM Trans. Comput. Syst.*, Vol. 16, No. 2, pp. 133–169 (online), DOI: 10.1145/279227.279229 (1998).
[3] 三輪竜也，川浪大知，川島龍太，松尾啓志：分散 KVS におけるメニーコアを活用したプロキシによるクエリ集約及び排他制御，技術報告 3 (2019). 情報処理学会 研究報告 オペレーティングシステム研究会 Vol.2019-OS-147.
[4] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: Amazon ' s Highly Available Key-Value Store, *SIGOPS Oper. Syst. Rev.*, Vol. 41, No. 6, pp. 205–220（オンライン），DOI: 10.1145/1323293.1294281 (2007).
[5] Riak: Key Value Database — NoSQL Key Value Database — Riak KV — Riak. `https://riak.com/products/riak-kv/index.html`,(参照: 2020-06-15).
[6] Pritchett, D.: BASE: An Acid Alternative, *Queue*, Vol. 6, No. 3, pp. 48–55 (online), DOI: 10.1145/1394127.1394128 (2008).
[7] DataStax: Using lightweight transactions. `https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useInsertLWT.html`,(参照: 2020-06-15).
[8] Redis: Transactions Redis. `https://redis.io/topics/transactions`,(参照: 2020-06-15).
[9] Amazon: DynamoDB トランザクションで複雑なワークフローを管理する - Amazon DynamoDB. `https://docs.aws.amazon.com/ja_jp/amazondynamodb/latest/developerguide/transactions.html`,(参照: 2020-06-15).
[10] Scalar: GitHub - scalar-labs/scalardb: A library that makes non-ACID distributed databases/storages ACID-compliant. `https://github.com/scalar-labs/scalardb/`,(参照: 2020-06-15).
[11] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Woodford, D., Saito, Y., Taylor, C., Szymaniak, M. and Wang, R.: Spanner: Google's Globally-Distributed Database, *OSDI* (2012).
[12] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T. and Bao, X.: Amazon Aurora: De-

sign Considerations for High Throughput Cloud-Native Relational Databases, *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17 , New York, NY, USA, Association for Computing Machinery, pp. 1041–1052 (online), DOI: 10.1145/3035918.3056101 (2017).

[13] SQLite: SQLite Home Page. `https://www.sqlite.org/index.html`,(参照: 2020-06-15).

[14] Google: gRPC - A high-performance, open source universal RPC framework. `https://grpc.io/`,(参照: 2020-06-15).

[15] DataStax: GitHub - datastax/cpp-driver: DataStax C/C++ Driver for Apache Cassandra. `https://github.com/datastax/cpp-driver`,(参照: 2020-06-15).

[16] basicthinker: GitHub - basicthinker/YCSB-C: Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB (https://github.com/brianfrankcooper/YCSB/wiki). `https://github.com/basicthinker/YCSB-C`,(参照: 2020-06-15).

[17] MariaDB: MariaDB Foundation - MariaDB.org. `https://mariadb.org/`,(参照: 2020-06-15).

[18] SiLeader: GitHub - SiLeader/cxx-transaction-benchmark-tool: Transaction Benchmark tool for C++. `https://github.com/SiLeader/cxx-transaction-benchmark-tool`,(参照: 2020-06-15).