Vol.2015-DBS-161 No.3
Vol.2015-IFAT-119 No.3
2015/8/5

NoSQL 上での ACID 属性を有するクエリ手法の提案

堀井洋[†1]　小野寺民也[†1]

NoSQL DB は，処理機能を制限することで，簡素なアーキテクチャ上で高い可用性やスケーラビリティを実現する．しかし，複雑なアプリケーションの要求に対応するため，クエリをサポートする NoSQL DB が普及され始め，トランザクション機能を NoSQL DB の基本操作のみで実現する研究も行われている．本論文では，その両者，すなわち，NoSQL DB がサポートするクエリと，トランザクション処理機能を，NoSQL DB が提供する基本操作のみで実現する手法を提案する．クエリ結果は，他トランザクションとの Read Committed の分離性を満たし，クエリ時点での最新とする一貫性を満たす．また，トランザクションの状態も NoSQL DB に保存することで原子性を満たし，NoSQL DB の既存の複製機構を利用することで堅牢性を維持する．我々は提案手法を MongoDB を用いて実装し，YCSB を用いて評価を行ったところ，提案手法は単純に従来手法を拡張した実装に比べ，高い性能向上を示すことがわかった．

Transactional Query Processing on NoSQL Database

HIROSHI HORII[†1]　TAMIYA ONODERA[†1]

NoSQL supports high scalability with limited capabilities for transactions or query processing. Though emerging NoSQL, such as MongoDB, supports certain queries, and researchers are developing the techniques to enable transaction processing in the existing non-transactional NoSQL, there is no solution for both limitations. We are developing a new library, MongoTx, which enables transaction processing in MongoDB without loss of any query capabilities. MongoTx stores dirty data, which a client has modified but not yet committed, in MongoDB along with the committed data. Because the dirty data must not appear in the query results, MongoTx hides the dirty data from the queries. To efficiently process queries and transactions, MongoTx uses a special data model to store the data in MongoDB. With this data model, MongoTx handles the same queries as MongoDB while guaranteeing the atomicity, isolation, and consistency of the transactions. With the evaluation of YCSB, MongoTx showed better throughput than a naïve extension of the existing work.

1. はじめに

NoSQL DB は，簡素な構成で機能性を最小限にすることで，データの格納，取得に対する高いスケーラビリティと可用性を実現する．様々なアプリケーションが NoSQL DB を利用する中，省かれた機能，特に，クエリ機能とトランザクション機能が求められている[20]．そのため，MongoDB[12]や Cassandra[14]といったクエリ機能を有する NoSQL DB が広く普及している[17]．また，トランザクション機能を有さない NoSQL DB 上で，トランザクション機能を提供する研究も広く行われている[3][6]．しかし，トランザクション機能を有さない NoSQL DB 上で，両機能を実現する手段は存在しない．

NoSQL DB にトランザクション機能を追加する場合，トランザクションの状態とコミット前のデータ（Dirty Data）を NoSQL DB に格納する[15]．例えば，レコード r_1，r_2 をテーブル D に挿入した後，レコード r_t をテーブル T に挿入する場合を考える．r_1，r_2 の値には r_t の主キーを含んでいるものとする．r_1，r_2 を取得したクライアントは，r_t が T に格納されている時にのみ，r_1，r_2 の値を照会できるという条件を導入することで，r_1，r_2 の 2 つの挿入処理を原子的に扱える（r_t はトランザクションのコミット状態を表し，r_t 挿入前の r_1，r_2 は Dirty Data となる）．複数レコードの原子的な挿入が可能であれば，多版型並行性制御（MVCC）を利用し，原子性，分離性，一貫性を有する更新，削除機能も実現可能となる．例えば，アプリケーションが利用するキーの値とバージョン番号の組み合わせを D の主キーとし，クライアントが常に最新のコミット済みのバージョンを利用することで，MVCC を実現可能である．上記仕組みを応用し，Percolator [6]，Omid [3]，Tell [4]は，BigTable [2]，HBase [13]，RamCloud [5]上での，トランザクション機能を実現している．

一方， NoSQL DB への Dirty Data や複数バージョンの格納は，既存 NoSQL DB のクエリ機能の利用に，オーバヘッドをもたらす．例えば，クライアントが D に対してクエリを行い r_1 を取得しても，r_1 がコミット済みの最新バージョンか確認する必要がある．コミット済みの確認には r_t を照会する必要があり，最新バージョンの確認には他バージョンの存在を照会する必要がある．そのため，既存手法[6]では，全レコードを走査する機能のみ提供され，クエリ機能は実現されていない．

本論文では，MongoDB 上でトランザクション機能を有するクエリ処理手法を提案する．MongoDB は，JSON 文書を格納し，JOIN 処理を含まないクエリ機能を提供する NoSQL DB で，トランザクション機能を持たない．提案手法を利用することで，アプリケーションは，トランザクションの区

†1 日本アイ・ビー・エム（株）東京基礎研究所
IBM Research – Tokyo

Vol.2015-DBS-161 No.3
Vol.2015-IFAT-119 No.3
2015/8/5

間を指定でき，区間内で呼び出されたクエリや更新を，原子性，分離性，一貫性を保ち，処理することが可能となる．

クエリ処理のオーバヘッドを軽減するために，提案手法では，JSON 文書の新しい格納方法，Single Document Two Versions（SD2V）を利用する．SD2V では，各 JSON 文書が最大 2 つのバージョンを保持する．トランザクションで JSON 文書を更新する際，新しいバージョンを追加し，コミット後に古いバージョンを新しいバージョンに置き換える．MongoDB に格納される 1 つの JSON 文書に全てのバージョンが含まれるため，最新バージョンの認識が容易となり，また，1 つのバージョンしか存在しない場合は，トランザクション状態を確認しないで照会することが可能となる．さらに，提案手法では，JSON 文書内のバージョンの配置方法を工夫することで，トランザクション処理を有さない他アプリケーションとの互換性を保ちながら，MongoDB が提供するクエリ機能の多くを実現可能とする．

本研究の貢献は，以下である．

- MongoDB が提供する機能のみを利用して，MongoDB の更新，クエリ機能に ACID 属性を付与する手法．
- 既存に MongoDB に格納された JSON 文書と互換性があり，効率的にクエリ可能な，新しい JSON 文書の格納方法 SD2V．
- 従来手法を単純に拡張した手法に比べ，YCSB を最大 2 倍の性能向上を実現する提案手法の実装．

2. 背景

最も広く利用される NoSQL DB の 1 つである，JSON 文書を保存する MongoDB と，その背景を示す．

2.1 JSON 文書

JSON は木構造の文書を表記するデータ記述言語である．JSON オブジェクトは，文字列と値をコロン（:）で示す複数のメンバを，カンマ（,）で区切り中括弧（{と}）で囲むことで構成される．値は文字列，数字，配列の他に，JSON オブジェクトを指定できる．配列は，要素をカンマ（,）で区切り各括弧（[と]）で囲むことで表される．図 1 に JSON オブジェクトの例を示す．なお，本稿では，MongoDB に倣い，メンバをフィールド，フィールドの文字列と値をフィールド名とフィールド値と呼ぶ．また，JSON オブジェクトを文書，フィールド値の文書を子文書と呼ぶ．

2.2 MongoDB

MongoDB は文書をコレクション（DBCollection）と呼ぶグループに分けて格納するデータベースである．一般的な NoSQL DB と同様，MongoDB は，シャーディング，複製，ロードバランシングの機能を持ち，データはシェアード・ナッシング方式[7]で管理される．

```
{  _id    : "Mongo001",
   name   : "Jason",
   amount : 100,
   friend : {  _id  : 10,
               name : "Freddy" },
   group  : [
                { name : "MongoDB"},
                { name : "MySQL"} ] }
```

図 1　JSON オブジェクトの例

Figure 1　An example of a JSON object.

アプリケーションは，MongoDB の API である insert，find，update，remove 関数を利用し，文書の挿入，クエリ，更新，削除を行う．また，findAndModify や findAndRemove 関数を利用して，文書の照会と更新を同時に行うことも可能である．格納される各文書は，_id フィールド値によりコレクション内で識別され，各関数は，文書ごとに原子的に処理される．

MongoDB は，関数で処理対象の文書を指定するために，独自のクエリを提供する．クエリは，比較演算（例：$gt，$lt）や，論理演算（例：$and，$or）等，様々な演算子を利用して，格納された文書の各フィールドに対する制約の組み合わせから構成され，JSON 文書として表される．

図 2 は，JSON をプリミティブ型として扱う Java に似せた言語で記述した，TPC-C [9]の payment 手続きを模したプログラムである[a]．まず，1 文書の照会に特化した関数 findOne 関数を利用して，顧客 ID（custId）に対する顧客情報（cust）情報を顧客情報コレクション（custs）から取得する（行 3）．次に，顧客情報の年間注文数（cust["YTD_PAYMENT"]）を update 関数を用いて更新し（行 4-5），注文数（amount），顧客 ID，倉庫 ID（whId）を履歴コレクションに insert 関数を用いて挿入する（行 6-8）．最後に，同じ倉庫の顧客による全注文数を，find 関数を用いて返す（行 10-13）．

MongoDB は，複数文書の更新を原子的に行えない．そのため，顧客情報の更新後（行 5）に履歴の挿入（行 8）が失敗する場合がある．アプリケーションが更新の補償処理を行えるが，補償処理中のクライアント障害には対応できない．また，同じ顧客情報を同時に更新時（行 5），履歴と顧客情報の注文数の総和の整合性が崩れる可能性がある．

2.3 NoSQL DB 上でのトランザクション処理

単一の更新処理のみ原子的な処理を保証する NoSQL DB 上で，複数の更新処理を原子的に処理可能とする手法が，Percolator [6], Omid [3], Tell [5]等で提案されている．NoSQL DB とは別サーバーを用いてコミットしたトランザクションを管理する Omid, TEL は，追加したサーバーに可用性が依存する．一方，Percolator では，BigTable のみを利用して，複数更新処理の原始的処理を実現する．

a) TPC-C の payment 手続きにおける DISTRICT を省いている．また，TPC-C では顧客の注文総数を返さない．

Vol.2015-DBS-161 No.3
Vol.2015-IFAT-119 No.3
2015/8/5

```
 1:DBCollection custs, hists;
 2:int payment(int whId, int custId, int amount) {
 3:  DBObject cust = custs.findOne({_id:custId});
 4:  cust["YTD_PAYMENT"] += amount;
 5:  custs.update({_id:custId}, cust);
 6:  DBObject hist = {
 7:   C_ID:custId, W_ID:whId, AMOUNT:amount };
 8:  hists.insert(hist);
 9:  int total = 0;
10:  DBCursor cursor =
11:    hists.find({W_ID: whId, C_ID:custId});
12:  while (cursor.hasNext())
13:    total += cursor.next()["AMOUNT"];
14:  return total;
15:}
```

図 2　MongoDB API を用いた payment 手続き

Figure 2　Payment procedure with MongoDB API.

Percolator のクライアントは, 複数のレコードを更新する際，各レコードを 2 回更新することで，原子的処理を実現する．1 巡目の更新でアプリケーションの更新データを全レコードに反映し，2 巡目の更新で更新データがコミットされたことを全レコードにマークする．1 巡目と 2 巡目は同じ順番でレコードを更新され，1 巡目の更新で最初に更新されるレコードへのリンクを全レコードに記録し，2 巡目の更新でそのリンクが削除する．2 巡目の最初のレコードが更新された時点で全レコードの更新がコミットされたとみなされる．他のクライアントが未コミットのレコードを取得した場合，リンクを辿ることでコミット済みか否かを判断でき, 長期間 2 巡目の更新がされてないレコードは，リンク先のレコードにアボートを記録することで，全更新のロールバックを行うことができる. 全ての更新は CAS を用いて行われ，BigTable の機能を利用してバージョン付けがされる．

Percolator は BigTable のバージョン機能を前提とするが，MongoDB にはバージョン機能がない．しかし，ObjectId を工夫することにより，同様なバージョン機能を実現することが可能である．例えば，アプリケーションが利用する ObjectId の値（`"Mongo01"`）にバージョン情報（`ver` フィールド）を追加することで, 図 1 の複数バージョン（`v1`, `v2`, `v3`）を MongoDB に以下のように格納できる．

```
{_id: {key: "Mongo01", ver: 1}, amount: 100,…}  :v1
{_id: {key: "Mongo01", ver: 2}, amount: 300,…}  :v2
{_id: {key: "Mongo01", ver: 3}, amount: 500,…}  :v3
```

上記のバージョン付けは，BigTable と同様なバージョン機能を実現し，かつ，MongoDB のクエリ機能の利用を可能とするが，MongoDB に高いオーバヘッドをもたらす．アプリケーションが従来どおり `find` 関数を利用した場合，その結果には古い文書，未コミット文書が含まれる可能性はあるが，再度 MongoDB に後続のバージョンの有無，コミット状態を問い合わせることで，容易にクエリ機能を実現することができる．しかし，これらの追加問い合わせは，MongoDB に高い負荷を与える．

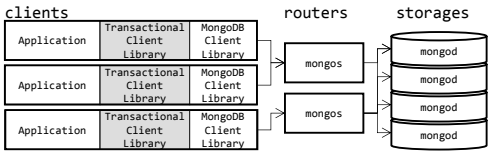

図 3　提案手法の構成（太線は物理筐体の境界を表す）

Figure 3　An architecture of our proposed method

3. SD2V を用いたトランザクション処理

本章では, MongoDB にトランザクション処理機能とクエリ機能を提供可能とする新しい文書格納方式を提案する．

3.1 設計

提案手法は，アプリケーションに対する，MongoDB への新しいライブラリとして稼動する．図 3 にその構成を示す．

クライアント上のアプリケーションは MongoDB 内の文書にアクセスする際，提案手法を実装するライブラリを利用する．そのライブラリ内部では，MongoDB 標準ライブラリが利用され，MongoDB 標準のルーター（`mongos`）を介してデータベース（`mongod`）内の文書がアクセスされる．

図 4 に，提案手法を利用した payment トランザクションの手続きを示す．提案手法では，MongoDB と同様，コレクション（`TxDBCollection`）に対して，格納する文書に対する処理を行う．異なる点は，セッション（`TxSession`）を用い，`begin`，`commit` 関数を呼ぶことで，トランザクションの範囲を指定するできる点である．図 4 では，行 4 から行 16 の間の処理が，1 つのトランザクションとして処理される．すなわち，update 関数による更新（行 7）と insert 関数による挿入（行 10）は，原子的に処理される．

提案手法では，コミット処理（行 16）が終了するまで更新中の文書が他のトランザクションで利用できない，Read Committed 分離性[1]が提供される．また，提案手法では，Read Committed 分離性を提供する商用データベース同様[10][11], クエリした時点で最新の文書を提供する一貫性を提供され，MongoDB が正常稼動する限り，堅牢性は保たれる．

3.2 トランザクション状態の格納

提案手法は，全トランザクションの状態を特別なコレクション（`txs`）に格納する．`begin` 関数では，トランザクション状態を表す文書（TxState）が挿入される．その文書の `state` フィールドは，トランザクション活動中を示す `"active"`の値が格納され，`commit` 関数では`"committed"` の値が，`rollback` 関数では`"roledback"`の値となる．値の遷移は，`findAndModify` 関数を利用し，負荷逆である．

TxState は，トランザクション ID（TxId）で識別される．TxId は，各クライアントが，クライアント識別子を用いて生成する．TxState は，トランザクションの開始時刻を示すフィールド（`ts` フィールド）を持ち，`begin` 関数で挿入時に，MongoDB サーバーの時刻が自動挿入される．

Vol.2015-DBS-161 No.3
Vol.2015-IFAT-119 No.3
2015/8/5

```
 1:TxSession session;
 2:TxDBCollection custs, hists;
 3:int payment(int whId, int custId, int amount) {
 4:  session.begin();
 5:  DBObject cust = custs.findOne({_id:custId});
 6:  cust["YTD_PAYMENT"] += amount;
 7:  custs.update({_id:custId}, cust);
 8:  DBObject hist = {
 9:   C_ID:custId, W_ID:whId, AMOUNT:amount };
10:  hists.insert(hist);
11:  int total = 0;
12:  TxDBCursor cursor =
13:    hists.find({W_ID:whId, C_ID:custId});
14:  while (cursor.hasNext())
15:    total += cursor.next()["AMOUNT"];
16:  session.commit();
17:  return total;
18:}
```

図 4　提案手法を用いた payment トランザクション

Figure 4　Payment transaction with our proposed method

state フィールドの値が"committed"に遷移した時点で，TxState に対応するトランザクションのコミットが確定する．また，"roledback"に遷移した時点で，トランザクションのロールバックが確定する．"roledback"への遷移は，トランザクションを起動したクライアント以外も実行可能で，一定時間以上"active"状態に対して行われる．

3.3 Single Document Two Versions (SD2V)

提案手法は，MongoDB のクエリ機能と，トランザクション処理機能を提供するため，NoSQL DB への新しいデータ格納方法，Single Document Two Versions（SD2V）を利用する．SD2V は JSON 文書の格納にに限定しないが，本稿では JSON 文書の格納を前提に記述する．

SD2V は，セーフフィールド群（Safe Fields），アンセーフフィールド群（Unsafe Fields），メタデータフィールド群（Metadata Fields）から構成される．Safe Fields は，コミット済みトランザクションにより更新された値が格納される．つまり，Safe Fields のみを照会することにより，アプリケーションは Read Committed 分離性を保証される．そのため，本稿では，Safe Fields を，セーフバージョン（Safe Version）と呼ぶ．Safe Version の各フィールド名は，アプリケーションが指定したフィールド名と同じであるため，アプリケーションのクエリをそのまま実行することが可能である．

一方，Unsafe Fields は，特別なフィールドである_unsafe フィールドに，子文書として格納される．トランザクションが更新中，全ての更新は_unsafe フィールドに格納され，TxState の state フィールドの値が，"committed"に遷移した後，全更新が Safe Fields にコピーされる．各フィールドは，更新中のフィールド以外，Safe Fields と同じである．コミットが確定していない可能性があるため，Unsafe Fields をアンセーフバージョン（Unsafe Version）と呼ぶ．Unsafe Fields は，トランザクションが終了後，削除可能となる．

```
{ _id    : "Mongo001",                              safe fields
  Name   : "Jason",
  amount : 100,
  friend : { _id: 10, name: "Freddy" },
  group  : [{ name: "MongoDB"}, { name: "MySQL"} ],
  _unsafe: { _id:  1,                              unsafe fields
             name: "Jason",
             amount: 2000,
             friend: { _id: 10, name: "Freddy" },
                    group: [ { name: "MongoDB"},
                             { name: "MySQL"} ] },
  _xTxId : "tx1",                                  metadata fields
  _sTxIds: [],
  _insert: false, _remove: false }
```

図 5　SD2V 文書例

Figure 5　An example of a SD2V document

Metadata Fields は，Unsafe Version を生成したトランザクションの TxId を保存する_xTxId フィールド，削除もしくは挿入中の際 true 値となる_insert，_remove フィールドから構成される．トランザクションが終了後，Metadata Fields は削除される．Metadata Fields が存在している場合，生成したトランザクション以外のトランザクションは，Metadata Fields を修正することができない．

SD2V 文書の更新は，全て findAndModify 関数を利用して行われる．そのため，1 つの文書を同時に複数のクライアントが更新することはない．

図 5 に，図 1 の文書の amount フィールドを 200 に修正した SD2V 文書を示す．全ての Unsafe Fields は，更新された amount フィールド以外，Safe Fields と同じである．また，tx1 で識別されるトランザクションとして，文書が更新されていることが分かる．

3.4 トランザクション状態の格納

提案手法を利用した図 4 のトランザクション処理例を，図 6 に示す．1 列目は txs，2 列目は custs，3 列目は hists が格納する文書（抜粋）を表し，A，B，C，D は図 4 の payment トランザクションの 4 つの局面を表している．

まず，図 4 行 4 でトランザクションを開始した際，図 6-A のように，クライアントは生成した TxId（"tx1"）を用いて TxState を txs に挿入する．次に，行 7 で custs の文書更新，行 10 で hists に文書挿入し，図 6-B のように，2 つの Unsafe Version を生成する．また，_xTxId フィールド値には，tx1 が指定される．行 16 でコミット処理された場合，まず，図 6-C のように，txs 内の TxState の state フィールドが"committed"に更新される．そして，図 6-D のように，2 つの Unsafe Version が，Safe Version にコピーされ，行 16 の commit 関数が終了する．

3.5 3 フェーズのクエリ処理

アプリケーションが要求するクエリを，MongoDB は Safe Fields を用いてクエリ処理するため，そのクエリ結果は Read Committed 分離性を満たす．しかし，図 6-C のように，すでにコミットされたトランザクションの更新が Unsafe Version として格納されている場合，最新の文書を照会する

ことができず，一貫性が崩れる．そのため，提案手法では，3 フェーズのクエリ処理を行う．

(1) **第 1 フェーズ：アンセーフ クエリ**

まず，格納されている Unsafe Version を対象に，クエリを行う．Unsafe Version は，Safe Version と同じフィールドを持つ_unsafe 子文書であるので，アプリケーションが指定するクエリ内のフィールド名に，_unsafe.をつけることで，Unsafe Version を対象にクエリをすることができる．例えば，図 4 行 13 のクエリは，Q1 のように修正される．

```
{ _unsafe.W_ID: whId, _unsafe.C_ID: custId }      :Q1
```

Unsafe Version を生成したトランザクションは，すでにコミットされている可能性がある．そのため，クライアントは，Q1 結果の TxId から txs を用いてトランザクションの状態を調べ，コミット済みの場合は，Safe Version へコピーする．

(2) **第 2 フェーズ：セーフ クエリ**

次に，アプリケーションが指定するクエリを利用して，格納されている Safe Version を対象に，クエリを行う．クエリ結果の SD2V 文書には，Unsafe Version が含まれている場合がある．Unsafe Version はコミットされている可能性があるため，クエリ結果の TxId から txs を用いてトランザクションの状態を調べ，コミット済みの場合は，Safe Version へコピーする．

(3) **第 3 フェーズ：キャッチアップ クエリ**

最後に，第 2 フェーズで Safe Version にコピーされた文書のみを対象に，クエリを行う．例えば，図 6-C に対する第 2 フェーズのクエリで，ObjectId 103 の Safe Version が Safe Version にコピーされた場合，Q2 のようなクエリが利用される．

```
{ _unsafe.W_ID: whId, _unsafe.C_ID: custId,      :Q2
  _id: {$in: 103 } }
```

(4) **クエリ結果の生成**

クエリ結果は，第 2 フェーズのクエリ結果中の Unsafe Version を持たない Safe Version と，第 3 フェーズのクエリ結果中の Safe Version から構成される．

4. 評価

我々は，提案手法の Java 実装 MongoTx と Yahoo Cloud Serving Benchmark（YCSB），US Zip コードを用いて，提案手法のトランザクション処理性能，クエリ処理性能を評価した．評価対象として，2.3 章で記載した既存手法を拡張した実装（MVCC）を利用した．なお，MVCC も MongoTx と同様，Read Committed を提供し，最適化を行っている．Intel(R) Xeon(R) E5-2680（2.70GHz，8 コア，ハイパースレッディング），32GB メモリ，SAS ハードディスクの IBM BladeCenter HS23 を利用し，2 台を mongos 用，5 台の mongod 用，2 台をクライアント用に構成して，評価を行った．OS は，Redhat Enterprise Linux 6.3 を用いた．

txs	custs	hists
A. トランザクションの開始		
{_id: "tx1", state: "active", ts: 080520140101}	{ _id: 1, name: "Jason", YTD_PAYMENT: 100}	{ _id: 101, C_ID:1, W_ID:2, AMOUNT: 100 } { _id: 102, C_ID:2, W_ID:2, AMOUNT: 200 }
B. custs and hists への更新，挿入後		
{ _id: "tx1", state: "active", ts: 080520140101}	{ _id: 1, name: "Jason", YTD_PAYMENT: 100, _unsafe: { _id: 1, name: "Jason", YTD_PAYMENT:200}, _xTxId: "tx1"}	{ _id: 101, C_ID:1, W_ID:2, AMOUNT: 100 } { _id: 102, C_ID:2, W_ID:2, AMOUNT: 200 } {_id: 103, _unsafe:{ C_ID:1, W_ID:2, AMOUNT: 200}, _xTxId: "tx1", _insert: true}
C. トランザクションのコミット		
{ _id: "tx1", state:"committed", ts: 080520140102}	{_id: 1, name: "Jason", YTD_PAYMENT: 100, _unsafe: { _id: 1, name: "Jason", YTD_PAYMENT:200}, _xTxId: "tx1"}	{ _id: 101, C_ID:1, W_ID:2, AMOUNT: 100 } { _id: 102, C_ID:2, W_ID:2, AMOUNT: 200 } {_id: 103, _unsafe:{ C_ID:1, W_ID:2, AMOUNT: 200}, _xTxId: "tx1", _insert: true}
D. トランザクションの終了		
{ _id: "tx1", state:"committed", ts: 080520140102}	{ _id: 1, name: "Jason", YTD_PAYMENT: 200}	{ _id: 101, C_ID:1, W_ID:2, AMOUNT: 100 } { _id: 102, C_ID:2, W_ID:2, AMOUNT: 200 } {_id: 103, C_ID:1, W_ID:2, AMOUNT: 200}

図 6　SD2V を利用したトランザクション処理

Figure 6　Transaction processing with SD2V

YCSB は，クラウド上のストレージシステムの評価を目的とした，単純な NoSQL DB の API を利用するベンチマークである．本評価では，6 つの定義済みワークロードの内，update-heavy，read-mostly，read-only，short-ranges のワークロードを利用する．それぞれ，50:50，95:5，100:0，95:5 の照会と更新の割合である．short-ranges 以外は主キーを指定した照会，更新処理から構成され，short-ranges は範囲指定のクエリが利用される．short-ranges のクエリは，必ず 1 文書を返すように設定し，また，zipfian を用いて，10 万件の文書から，クライアントがアクセスする文書を決定すりょうに設定した．YCSB は元々，複数の処理を行うトランザクション処理 には対応していないため，我々は 5 つの YCSB の処理を 1 つのトランザクションとして処理するように，ベンチマークを修正した．本稿では，終了した YCSB の処理数を基に，スループットを計測した．図 7 はその結果を示す．

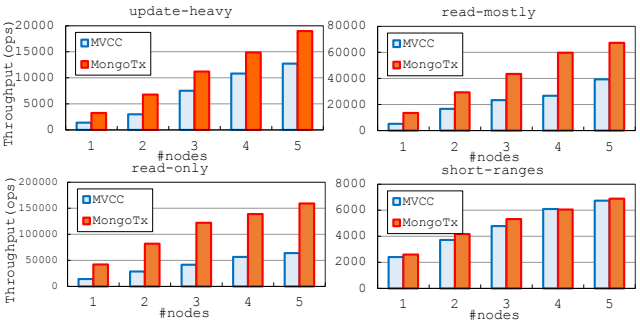

図 7　MongoTx と MVCC のスケーラビリティ
Figure 7　YCSB throughputs of MongoTx and MVCC

図 7 より，short-ranges 以外のワークロードでは，MongoTx が良いスループットを示しており，read-only では，2 倍以上のスループットの向上が見られる．MVCC では，文書の主キーは`_id` 子文書の一部として格納される．そのため，YCSB クライアントは，MongoDB のインデックス機能を利用して文書を獲得する．一方， MongoTx では`_id` フィールド値が主キーとなる．インデックス経由のデータアクセスと比較し，`_id` フィールド値経由のアクセスの方が MongoDB は効率的に照会処理が可能である．そのため，MongoTx の方が MVCC よりも性能が高くなる．

次に，US Zip コード[21]と地点の組み合わせを格納した MongoDB を利用し，ある地点から近い点を MongoDB の `$near` 操作を用いてクエリし，最も近い点を更新するトランザクションのレイテンシを調べた．`$near` 操作の引数(ラジアン距離)を変更することで，取得する文書を変更した．なお，1 台の `mongod` サーバ，1 台のクライアント スレッドを利用した．

図 8 は，1 つのトランザクション処理に必要な MongoDB API の実行数と，レイテンシの関係を示す．1 文書を返すクエリの場合，MongoTx と MVCC の性能は同じだが，20 文章を返すクエリの場合は，MongoTx の性能が優れている．これは，MVCC は，最新のコミット済みバージョンを API を通じて照会する必要があるのに比べ，MongoTx は最悪 3 回の API 呼び出しで，クエリが処理されるためである．

5. おわりに

本稿では，MongoDB の実装を修正せずに，MongoDB にトランザクション処理を実現する手法を提案した．提案手法では MongoDB のクエリを利用でき，また，従来のアプリケーションが格納した文書を，マイグレーションせずに利用することができるため，従来手法[18][19]とは異なり，容易に既存アプリケーションへの適用が可能である．提案手法は，MongoDB 以外にも適用可能であるため，今後，様々な NoSQL DB へのトランザクション機能の追加を検討していく．

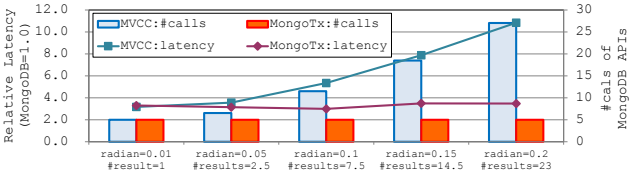

図 8　US Zip コードを利用したクエリの性能
Figure 8　Latencies of `$near` query in check-in transaction

参考文献

1) Berenson, H., et al.., P. A Critique of ANSI SQL Isolation Levels, In SIGMOD 1995.
2) Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., Bigtable: A Distributed Storage System for Structured Data, In TOCS 2008.
3) Ferro, D. G., Junqueira, F., Kelly, I., Reed, B., and Yabandeh, M., Omid: Lock-free Transactional Support for Distributed Data Stores, In ICDE 2014.
4) Loesing, S., Pilman, M., Etter, T., and Kossmann, D: On the Design and Scalability of Distributed Shared-Data Databases, In SIGMOD 2015.
5) Ongaro, D., Rumble, S., Stutsman, R., Ousterhout, J., and Rosenblum, M.: Fast Crash Recovery in RAMCloud, In SOSP 2011.
6) Peng, D. and Dabek, F., Large-Scale Incremental Processing Using Distributed Transactions and Notifications, In OSDI 2010.
7) Stonebraker, M. The Case for Shared Nothing. Database Engineering Bulletin Vol. 9, No. 1, 1986, 4-9.
8) Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S. , Liu, H., Wyckoff, P., and Murthy, R., Hive - A Warehousing Solution Over a Map-Reduce Framework, In VLDB 2009.
9) TPC benchmark C standard specification version 5.11. http://www.tpc.org/tpcc/spec/tpcc_current.pdf, 2010.
10) IBM DB2. http://www.ibm.com/software/data/db2/
11) IBM Infomix. http://www.ibm.com/software/data/informix/
12) MongoDB. http://www.mongodb.org/, 2009.
13) Apache HBase. http://hbase.apache.org/, 2008.
14) Apache Cassandra. http://cassandra.apache.org/, 2008.
15) Perform Two Phase Commits, http://bit.ly/1hxifWN.
16) Optimistic transactions in MongoDB, http://bit.ly/1E8kPPp.
17) DB-Engines Ranking (Feb. 9, 2015), http://bit.ly/JaDlgu
18) DB2 JSON, http://ibm.co/1uAvSOs
19) TokuMx, http://www.tokutek.com/tokumx-for-mongodb/
20) NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex, http://bit.ly/QTwHzl
21) Zip data of the United States: http://media.mongodb.org/zips.json