Vol.2016-DBS-163 No.3
Vol.2016-IFAT-123 No.3
2016/9/13

クライアントキャッシュ DB の提案

望月翔平[†1]　片山大河[†1]　金松基孝[†1]

概要：　社会インフラシステムなどにおいて，制御データを RDBMS で管理し，低レイテンシでアクセスできることが求められている．しかし，通常の RDBMS では要求を満たすことができない場合がある．そこで，RDBMS のクライアントモジュールに, 組み込み型のメモリ DB を配置することで RDBMS の一部のテーブルをキャッシュして参照を高速化する手法を提案する．クライアントモジュールに配置するメモリ DB として，SQLite のメモリ DB 機能を用いて実装・評価を行い，その有効性を確認した．

キーワード：DBMS，キャッシュ

Client Cache Database

SHOHEI MOCHIZUKI[†1]　TAIGA KATAYAMA[†1]
MOTOTAKA KANEMATSU[†1]

1. はじめに

社会インフラの制御システムで，制御データをリレーショナルデータベース管理システム（RDBMS）で管理することが求められている．これは，SQL を使うことで開発工数を抑えるためである．想定する制御システムでは，単一のクライアントが RDBMS で管理された制御情報を更新し，それとは別の複数クライアントが参照して機器を動作させる構成となっている．このシステムでは，短い周期で動作を行うため，毎周期で DB の参照のために使える時間が少ない．その周期のデッドラインを守るためには，レコード 1 件の取得に平均で 10 マイクロ秒以下の応答時間を要求する．通常のクライアント・サーバ型の RDBMS ではそのような応答時間の要求を満たすことができない．これは，ネットワーク経由でアクセスすることによるレイテンシが大きいためである．

そこで，クライアント・サーバ型の RDBMS のクライアントモジュールにインメモリ型の RDBMS を組込み，クライアント・サーバ型 RDBMS の DB（以下，サーバ DB）の一部のテーブルをキャッシュして参照を高速化するクライアントキャッシュ DB を提案する．

以降，2 章で提案システムの特徴と設計方針について述べ, 3 章で提案システムの詳細について述べる．4 章で実験結果を示し, 5 章で関連研究について述べ, 6 章で本論文をまとめる．

2. 提案システムの特徴と設計方針

クライアントキャッシュ DB の特徴は以下の通りである．

- クライアントモジュールに組込み型のメモリ DB を配置することによるデータ参照の高速化
- サーバ DB との自動同期

この章では，以上の特徴について説明する．

2.1 クライアント DB による参照の高速化

データ参照を高速化するため，クライアントモジュール内の DB（以下，クライアント DB）にサーバ DB の一部のテーブルをキャッシュする．クライアント DB は組込み型であり，アプリケーションからキャッシュしたテーブルへの参照は，プロセス間通信のレイテンシがなく，関数呼び出しでアクセス可能なため高速である．

また，クライアント API の内部で自動的にクライアント DB とサーバ DB どちらにクエリを実行するか判断する．そのため，アプリケーションはクライアントライブラリとリンクするだけで，今までと同じ使い方でクライアント DB を利用できる．

2.2 自動同期

クライアント DB にキャッシュしたデータをできるだけ新しく保つために，サーバ DB の更新をクライアント DB へ自動的に反映するようにした．更新クエリはサーバ DB

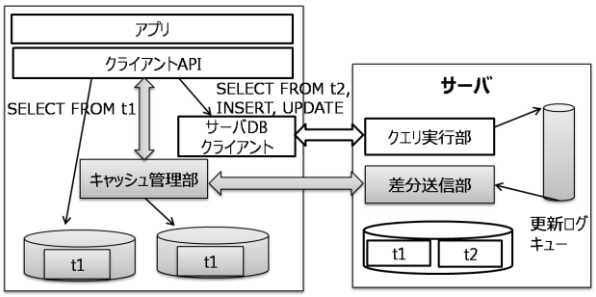

図 1 全体構成

†1 （株）東芝 インダストリアル ICT ソリューション社
IoT テクノロジーセンター

で実行し，一定の遅延後にクライアント DB に反映する．しかし，この自動同期の仕組みにより，アプリケーションがクライアント DB にデータ参照時に，サーバ DB の更新内容のクライアント DB への反映が行われた場合，アプリケーションの参照が妨げられ，レイテンシが大きくなる可能性がある．これは，RDBMS で同じレコードへの読込みと書込みが同時に起こる場合，ロックまたは MVCC（Multiversion Concurrency Control）等の並行性制御の仕組みが必要なためである．ロックの場合，一方のトランザクションはもう一方のトランザクションが終了するまで待たされる．MVCC は，複数バージョンのレコードを管理することで，書込み中も読込みが可能となる並行制御方式である．しかし，レコードの読込み・書き込みごとに短時間のロック（ラッチ）が必要でオーバーヘッドがかかることになる．

この問題を回避するため，クライアントはメモリ上に 2 つのクライアント DB を持たせる構成とする．一方をアプリケーションが専有できる参照用クライアント DB とし，もう一方をサーバ DB の更新を反映するための更新用クライアント DB とする．アプリケーションは参照用クライアント DB を専有できるため，常に高速に参照できる．更新用クライアント DB にサーバ DB の更新を反映後に，この 2 つの役割を切替えることで，アプリケーションはサーバ DB に書込まれた内容が反映されたクライアント DB にアクセスできるようになる．

切替は，アプリケーションのクライアント DB への参照を妨げないようにする．アプリケーションが参照中ならば，アプリケーションの参照が終わるまで待ってから切替を行った後，もう一方のクライアント DB にサーバ DB の更新を書込む．

このように 2 つのクライアント DB を用いるとクライアント DB の外部で排他制御が可能であり，クライアント DB 内部の排他制御を不要にできる．参照中かどうかのフラグは，参照開始時と終了時に 1 回ずつロックを取って書き換えるだけであり，レコードごとのロックを取る MVCC と比べてロックの回数が少なくて済む．

3. 提案システムの詳細

提案システムの構成のうち，新規追加部分を灰色で示す（図 1）．通常のクライアント・サーバ型の構成に加え，クライアントモジュールに 2 つのインメモリ型 DB をクライアント DB として配置する．また，キャッシュ管理部はクライアント DB を管理するためのモジュールで，クライアント DB の更新をバックグラウンドで実行できるよう，アプリケーションのクエリ処理とは独立したスレッドとして動作する．

サーバ側には，キャッシュ管理部と接続する差分送信部とクエリ実行部からサーバ DB の更新内容を取得するため

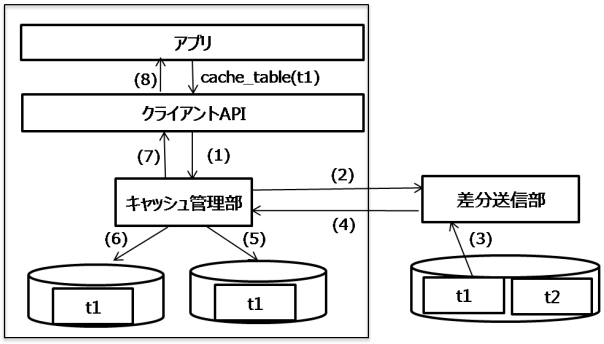

図 2 キャッシュの構築

のキューを追加する．差分送信部はキャッシュ管理部からの要求で，サーバ DB にアクセスしキャッシュ管理部にデータを返す処理を行う．

3.1 テーブルのキャッシュ構築方法

サーバ DB のテーブルをクライアント DB へキャッシュするには，始めにアプリケーションが明示的にキャッシュ構築用 API を呼び出す必要がある．具体的には，アプリケーションはテーブル名を引数に，cache_table というクライアント API を呼び出す．この API は，指定したサーバ DB 上のテーブルをクライアント DB にも作成し，サーバ DB からレコードを全て読み込む．また、この API が完了後は，キャッシュしたテーブルがサーバ DB で更新されると，自動的にクライアント DB 上にも反映する．API 呼び出し時の詳細な流れを図 2 に示す．まず，(1) キャッシュ管理部にキャッシュ要求を出し，(2) キャッシュ管理部がサーバ側の差分送信部に要求を送信する．(3) 差分送信部はサーバ DB にアクセスしてテーブルのスキーマと，そのテーブルの持つインデックスのスキーマ，テーブルの全レコードを取得し，(4) キャッシュ管理部に送信する．(5) (6) キャッシュ管理部は，受け取ったスキーマとレコードからテーブルの作成，インデックスの作成，レコードの追加を行う SQL 文を作成し，両方のクライアント DB に書込む．(7) クライアント API に完了を通知し，それを受けて，(8) アプリケーションに制御を戻す．

3.2 クライアント DB の自動更新

サーバ DB の更新をクライアント DB に自動的に反映する処理を図 3 に示す。

3.2.1 サーバからクライアントへのデータ送信

クライアントからサーバ DB に対して更新があった場合，クエリ実行部はコミット時に，サーバ DB に書込みを行うだけでなく，クライアントに送信するための更新ログを，テーブルごとに用意したメモリ上のキューにも追加する．キューの 1 つの要素はトランザクション単位のログとなっている．クライアント側のキャッシュ管理部は定期的に，差分送信部に新たな更新がないか問い合わせ，差分送信部はキューにデータが存在したらクライアントに送信する．

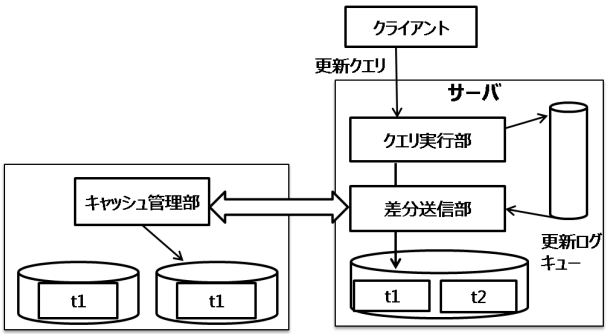

図 3　サーバ DB の更新のクライアント DB への反映

このログのフォーマットは，テーブル毎の変更レコードのリストとする．レコード単位とする場合，クライアント側でタプルを一意に定める値を指定した INSERT OR REPLACE 文を用いて更新することで，INSERT 文と UPDATE 文両方に対応できる．DELETE 文については削除したレコードの行番号のみの情報があればよい．

3.2.2 クライアント DB 更新と切替

キャッシュ管理部は，サーバから送信された更新ログを受信し，アプリケーションが参照していない更新用のクライアント DB を更新する．クライアントが受信した更新ログがすべて更新用クライアント DB に書込まれた時点で切替え，もう一方のクライアント DB を更新する（図 4）．

アプリケーションの参照を妨げずに切替えるために，アプリケーションの参照とキャッシュ管理部の更新が同一のクライアント DB に対して起こらないようにし，クライアント DB の外部で切替時にロックをとる．

切替の具体的な処理は次のように行う．アプリケーションがクライアント DB に参照中でない場合は，キャッシュ管理部は即座に切替える．アプリケーションが参照中の場合，キャッシュ管理部は，アプリケーションの次回の参照ではもう一方のクライアント DB を参照するように指示した上で，アプリケーションの参照が終了するのを待つ．参照が終了した時点で，キャッシュ管理部はもう一方のクライアント DB に書込みを開始する．

3.2.3 クライアント DB への反映の遅延に関する議論

サーバ DB で起こった更新がクライアント DB に反映され，アプリケーションがアクセス可能となるまでには遅延

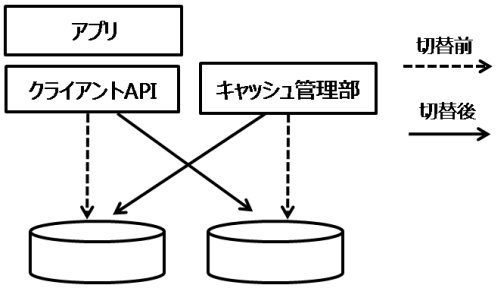

図 4　クライアント DB の切替

がある．遅延の要因としては次の 3 つがある．

1. 更新の差分をサーバから取得するのにかかる時間．ネットワークの遅延に依存する．
2. 差分をクライアント DB に書込む時間．
3. 書込みが完了してから切替完了までにかかる時間．

1 と 2 は一般的な非同期レプリケーションで問題となる遅延と同じである．2 では，2 つのクライアント DB へシーケンシャルに書込む必要があるが，メモリ DB への書込みであり，サーバ DB への更新よりも相対的に高速であるため，遅延が大きくなる可能性は低い．1 と 2 は並列化することで改善可能な処理である．3 は，アプリケーションの参照のトランザクションの実行時間に依存する．アプリケーション開発者が，長時間 1 つのトランザクションで参照を続けた場合には他のクライアントの更新が反映されないということに注意する必要がある．しかし，トランザクション開始時点のスナップショットが他のトランザクションで変化しないという挙動は，SERIALIZABLE の分離レベルと一致しているため，直感的な挙動である．

3.3 クエリの実行

3.3.1 参照・更新クエリの実行

アプリケーションが実行した参照クエリは，アクセスするテーブルが全てクライアント DB に存在する場合，クライアント DB で実行し，それ以外の参照クエリと全ての更新クエリはサーバ DB で実行する．そのため，2 つ以上のテーブルを参照する JOIN などの SQL は，参照する全てのテーブルがクライアント DB に存在する場合のみ，クライアント DB で実行するようにする．テーブル t1 がクライアント DB 上にあり，t2 はサーバ DB にのみ存在する場合の状況を図 1 に示す. なお, これらは自動的に判定するため，アプリケーションの開発者はどちらの DB にアクセスしているか意識する必要がない．

具体的な判定処理は次のように行う．アプリケーションから SQL が与えられたら，どちらか一方のクライアント DB で，SQL 文の実行計画を作成するプリペア処理を行う．プリペア処理に失敗した場合，クライアント DB 上に存在しないテーブルにアクセスしている可能性があり，サーバ DB でクエリを実行すると判定する．クライアント DB でのプリペア処理に成功した場合，そのクエリが更新クエリか参照クエリか判断できる．更新クエリの場合はサーバ DB で実行すると判定し，参照クエリの場合はクライアント DB で実行する．プリペアドステートメントを用いた実行する．

3.3.2 プリペアドステートメントを用いた実行

同じ SQL を複数回実行する場合や, パラメータの値のみを書き換えて複数回実行する場合には，アプリケーションは，SQL の実行計画を事前に作成しておきそれを使い回す機能であるプリペアドステートメントを利用することで高速化が期待できる．特に，クライアント DB にアクセスす

Vol.2016-DBS-163 No.3
Vol.2016-IFAT-123 No.3
2016/9/13

る場合には通信のレイテンシがないことから，単純なクエリの場合，クエリ実行時間に占める実行計画の作成時間が大きいため，効果が大きい．

プリペア処理についても，クエリ実行時と同様の判定を行い，クライアント DB かサーバ DB のどちらかで実行計画を作成する．クライアント DB で実行計画を作成する場合は，2 つあるどちらの DB にアクセスすることになるかこの時点では決定できないため，内部的に両方の DB でプリペアードステートメントを作成しておく．プリペア処理の API はステートメントのハンドルをアプリケーションに返す．API の内部では，プリペア処理がクライアント DB とサーバ DB のどちらで実行されたかを保存しておき，アプリケーションに返したハンドルと実際のクライアント DB またはサーバ DB のハンドルと関連付けて保存しておく．実際のクエリ実行のときには，その関連付けから実行するクライアント DB とサーバ DB のどちらで実行するか決める．このようにして，アプリケーションはサーバ DB とクライアント DB で同一の API を利用してプリペア処理が実行できる．

3.4 更新と参照

クライアント DB にキャッシュしたテーブルに対する更新クエリもサーバ DB で実行し，一定の遅延後にクライアント DB に反映する．そのため，アプリケーションがテーブル t1 をキャッシュしたクライアントから，以下の SQL を連続して実行することを考えると，次のような問題が起こる．

- INSERT INTO t1 VALUES（100）
- SELECT * FROM t1

INSERT 文はサーバ DB で実行し，その直後に SELECT 文はクライアント DB で実行するため，SELECT 文実行結果に，直前の INSERT 文の結果が反映されないことがある．このように，アプリケーションから見た場合，キャッシュしているテーブルに対する書込内容が，書込直後には見えない場合があり，直感に反することに注意する必要がある．

3.5 コード例

クライアントキャッシュ DB を使用したコード例を表 1 に示す．予めサーバ DB に以下のスキーマを持つテーブル存在するものとする．

- CREATE TABLE t1（col1 INTEGER PRIMARY KEY, col2 INTEGER）
- CREATE INDEX idx on t1（col2）

コード例では，db1 に接続後，テーブル t1 を cache_table 関数（2 行目）でキャッシュしている．その後、プリペア処理を実行し，ループ内で毎回変数の値のみを変えてクエリの実行を行っている．ユーザは，実際の処理がクライアント DB またはサーバ DB のどちらで行われるかを意識せず，プリペアドステートメントの作成，実行，結果取得の API を呼び出せる．そのため，キャッシュ DB の導入により必要となるコードの追加は，テーブルをキャッシュする cache_table の呼び出しのみであり，導入が容易である．

```
1  select_db(con,"db1");          //db1 に接続
2  cache_table(con,"t1");         //t1 をキャッシュ
3  prepare_stmt(con,"SELECT * FROM t1
     WHERE col2 = ?;", &stmt);    //プリペア処理
4  for (int i = 0; i < num; i++) {
5    ResultSet result; //クエリ結果を格納する変数の定義
6    int id = rand(size);         //乱数の取得
7    //'?'を変数 id の値で置き換える
     bind_stmt(con,stmt,1,INT,id);
8    execute(con,stmt);           //クエリの実行
9    store_result(user,stmt,&result); //結果の取得
10   free_result(user,stmt,result);   //結果の解放
11 }
```

表 1 コード例

3.6 実装

サーバ DB として，オープンソースの組込型データベース SQLite[1]をベースに，当社が開発したクライアント・サーバ型データベース TinyBrace®[3]を利用した．また，クライアント DB としても SQLite を利用した．これは，SQLite は DB ファイルを作成せずにメモリ上にデータを保持するメモリ DB 機能を持つことと，サーバ側とベースが同じであることからデータ型の互換性の問題が起こりにくいためである．クライアント DB として利用する SQLite については，ソースコードを変更せず，標準の API を利用する方針で開発を行った．

また，SQLite では，メモリ DB を複数コネクションから共有するには，Shared-Cache モードを利用する必要がある．このモードでは複数コネクションから同時に同一 DB に対しクエリを実行した時，その処理のほとんどが排他されるため，大きな速度低下が起こるが，提案する構成においては，同時に 1 つのコネクションからしかアクセスが起こらないためこのような問題は起こらない．また，SQLite では，スレッドセーフを保証するかどうかをコンパイル時に指定するコンパイルオプション DSQLITE_THREADSAFE がある．これをオフにしてコンパイルすることで，スレッド間の排他処理がコンパイル時に取り除かれるため，高速化できる．

4. 実験

提案システムの効果を測定するため，実験を行った．

4.1 実験内容

当社が開発した元々の TinyBrace®と，それにクライアントキャッシュ DB を追加した場合，そして SQLite のメモリ DB を直接使用した場合の 3 つの比較を行った．3.5 節のコ

ード例で説明したものと同じ処理を 1 クライアントから実行し，rand 関数の実行を除くループ 1 回あたりの処理の時間を測定した．つまり SELECT 文を実行する execute だけでなく，bind，store_result，free_result を含めた測定結果となっている．SQLite については，それぞれ対応する関数を直接呼び出している．テーブルサイズは 10000 件とした．また，1 回の SELECT 文で得られる行は 1 行だけとなっている．また，サーバとクライアントは同一マシン上で動作させているため，サーバとクライアント間の通信は同一マシン上の TCP 通信となっている．

実験環境は以下である．

- CPU：Intel core i7 4770K　3.5 GHz
- メモリ：16GB
- HDD：512GB
- OS：Windows8.1 Pro

4.2 測定結果

測定結果を表 2　測定結果 に示す．キャッシュありの場合，キャッシュなしの場合と比べて，大幅な高速化が達成できている．これは，クライアント DB にアクセスすることにより，サーバクライアント間の通信を省くことによるものだと考えられる．キャッシュなしの場合も，テーブルのサイズが大きくなく DB のキャッシュにのるサイズであることから，実際の検索処理にかかる時間は大きくないと推察される．そのため，実験で利用したクエリのような，インデックスの効く高速な検索においては，クエリの応答時間の多くが，通信のレイテンシであるといえる．

クライアント DB の実装に SQLite のメモリ DB を使用しているため，SQLite の API を直接使用してメモリ DB にアクセスした場合が，考えうる最も高速な値となるが，実験結果ではそれと比べキャッシュがありの場合，倍以上差がある．クライアント API を介して SQLite のメモリ DB にアクセスするオーバーヘッドが大きいためだと考えられる．

5. 関連研究

同様の構成を持つものとして Oracle® TimesTen Application-Tier Database Cache[2]がある．Oracle® Database の一部のテーブルを，アプリケーションと同一マシン上に置かれたインメモリ型の RDBMS Oracle® TimesTen にキャッシュすることで高速化する．提案手法と異なり，クライアント DB として 1 つのメモリ DB を使用するため，アプリケーションからの参照と Oracle® Database の更新の反映が Oracle® TimesTen に対して同時に行われる可能性があり，メモリ使用量の節約や，より新しいデータへのアクセスが可能な一方で，アプリケーションからの参照が一時的に妨げられる可能性がある．

	SQLite メモリ DB	キャッシュあり	キャッシュなし
応答時間（μ秒）	0.98	2.49	189.39

表 2　測定結果

6. あとがき

本論文では，クライアントモジュールに組込み型の DB を配置することで高速化するクライアントキャッシュ DB の提案した．本提案では，2 つのクライアント DB を置くことで，サーバ DB の更新の反映によりアプリケーションの参照が妨げられないようにした．また実験によりクライアントに DB を置かない場合と置く場合で比較し，置いた場合にレイテンシが削減されることを確認した．

今回の実験では，サーバ DB への更新を行っていない状況で応答速度を測定した．今後更新中の応答速度の測定や，クライアント DB に反映されるまでの遅延などの測定を行うことを予定している．

提案手法では，テーブル単位でキャッシュを行うようにした．しかし，サーバ DB のテーブルの一部のみをキャッシュしたいケースも考えられる．その場合，テーブル名に加え，WHERE 句を満たすレコードや指定したカラムのみのテーブルをクライアント DB に作成する機能を用意することが考えられる．

参考文献

[1] SQLite
https://www.sqlite.org/

[2] Oracle TimesTen Application-Tier Database Cache
https://www.oracle.com/database/timesten-cache/index.html

[3] 金松基孝，山地圭：組込み機器のデータ管理に適した軽量データベース TinyBrace，東芝レビュー，Vol.67，No.8，pp.11－14（2012）