D-016

Key-Value Store 型 DBMS における SQL アクセス機能の実装と評価
Design and implementation of SQL access feature on a key-value store DBMS

嶋村 誠 * 片山大河 * 山地 圭 * 金松基孝 *
Makoto Shimamura Taiga Katayama Kei Yamaji Mototaka Kanematsu

1. はじめに

近年、簡易なデータベース管理システム (DBMS) として Key-Value Store (KVS) [1, 2, 3, 4, 5] と呼ばれるシステムが注目を集めている。KVS では任意の保存したいデータ (Value) に対し、対応する一意のキー (Key) を設定し、これらをペアで保存する。KVS は従来のリレーショナル型 DBMS (RDBMS) とは異なり、多数のマシンで構築したクラスタ上にデータを分散させたデータ管理や検索を行うのが容易であるものが多い。しかし、これらの KVS では RDBMS における SQL のような共通のインターフェースは持たず、各々の KVS がデータアクセスのための独自の API を用意している。従って、RDBMS に比べて、データの検索・集計処理のために開発者が書くプログラムの量が多くなってしまう。

様々な KVS や RDBMS に既に保存されているデータを横断的に活用するにあたっては、同一のインターフェースを利用できることが望ましい。そこで、KVS 上で SQL を用いることを考える。KVS へのアクセスインターフェースとして SQL を用いることができれば、以下の利点がある。

- **データの検索・集計が容易**　データの集計・検索処理のためのプログラミングの手間を削減できる。例えば、ある条件に当てはまるデータを検索し、検索結果のデータ中の最大値を求める処理が SQL 文 1 つで実行できる。C 言語でこれを行うには 10 行以上のプログラムが必要になる。
- **SQL を知っている開発者が容易に移行できる**　SQL を学んでいる開発者は多い。このため、KVS で SQL をサポートすれば、開発者は KVS 独自の API を改めて学ぶ必要がなくなる。
- **過去ソフトウェアの資産の流用が可能**　RDBMS 上で作られた過去のアプリケーションのうち、主キー以外でのアクセスがないテーブルなど、KVS に適する用途で利用されているものは、SQL をサポートすることで KVS に移行することが容易である。
- **別の KVS への移行が容易**　SQL をサポートしている KVS 間での移行が容易になる。例えば、クラウドサービスを乗り換える際に、アプリケーションをクラウドサービスの用いている KVS に合わせて書きなおす必要がなくなる。

そこで本論文では SQL を用いて各種 KVS に対して入出力を行うスキームを提案・試作し評価する。RDBMS はデータベースにアクセスするために、SQL 文を SQL パーサとプランナにより実行プランへ変換し、プラン実行部が実行プランに従って、データベースにアクセスするものが多い。 提案機構では、ここでプラン実行部の関数をフックし、KVS へアクセスするようにする。また、KVS には表形式のデータ構造を Value としてエンコードして保存する。これにより、ユーザが KVS へアクセスしていることを強く意識することなく、あたかも通常のテーブルのように使うことができる。プロトタイプとして、東芝社内で用いているデータベースである TinyBrace、およびオープンソースの KVS である Redis [1] を用いた実装を行った。このプロトタイプを用いて、Redis の API を直接用いる場合と比較し評価実験を行ったところ、SQL 処理によるオーバヘッドはレコードの挿入で 24.2%、主キー一致検索で 29.1%、テーブルスキャンで 16.1%程度であった。

1.1. 本論文で対象とする Key-Value Store

従来の RDBMS で大規模データを扱おうとした場合，大規模データを用いるアプリケーションでは性能が劣化してしまうことが知られている [6]。これは，従来の RDBMS が小規模の高頻度なトランザクションか，巨大だが書き込みをほとんど伴わないトランザクションに最適化されて設計されることが多いためである。大規模データを扱うアプリケーションとしては，インターネット検索のための文書のインデキシング，トラフィックの高いウェブサイトのサーバなどがある。これらのアプリケーションでは，高頻度で大規模な書き込みが必要である。

そこで、近年の KVS は大規模データを扱うための基盤となるよう設計されている。例えば、一貫性の完全保証を行わないように設計したり，分散アーキテクチャを採用することが多い。現在，Google の BigTable [4] や Amazon の Dynamo [5] が商用システムのベースとなる KVS として実用化されている。

本論文では キー（Key）を指定して，Key に関連付けられた値（Value）を設定・取得・削除することができるものを KVS と呼ぶ。ここで，Key は RDBMS の主キーに相当する。Key の値は KVS 全体にわたって一意である必要がある。KVS への値の設定・取得・削除はそれぞれ *SET*，*GET*，*REMOVE* 命令で行う。例えば，Key を k，Value を v としたとき，*SET(k,v)* とすると，k に v が関連付けられる。その後で *GET(k)* とすれば v を得ることができる。k と v の関連付けの削除は *REMOVE(k)* で行う。

なお、広義には KVS は NoSQL ストレージとしてグラフデータベースや XML データベースなど SQL を使わないデータベースを含んだ概念として扱われることがあるが、今回はこれらの DBMS は対象としない。

† （株）東芝 ソフトウェア技術センター, Corporate Software Engineering Center, TOSHIBA CORPORATION

1.2. 本論文の構成

第 2 章で関連研究について述べ、提案機構との違いを説明する。第 3 章では提案機構の基本的な設計について述べる。その後、第 4 章でプロトタイプ実装の詳細について述べる。第 5 章では評価実験を行い、プロトタイプ実装を評価する。最後に、第 6 章で本論文をまとめる。

2. 関連研究

KVS においては、基本的に SQL が使えず、データ検索のために独自の API を使う必要がある。これは利用者に対する大きな障壁として認識されており、この問題を解決するため様々な工夫がされている。例えば、MongoDB [2] は SQL との検索クエリ変換表を用意することで、独自のクエリ方式の学習を容易にしている。

この障壁をなくすため、KVS に対して SQL に類似したアクセス手段を提供しようとしている試みがいくつかある。例えば、Cassandra [3] では SQL に類似した ”CQL” というアクセス手段を提供している。Apache Hive [7] や HadoopDB [8] では MapReduce の操作を SQL から生成する手段を提供している。また、Oracle Coherence [9] も Coherence Query Language (CohQL) を提供している。これらのアクセス手段は SQL サブセットと独自の拡張文法からできており、各々の KVS に特有の部分が多い。

また、異種データベース連携のための手段で KVS への SQL アクセスを提供するアプローチがある。PostgreSQL の 外部データラッパー機能 [10] では Redis をアクセスできるようにした redis_fdw モジュール [11] が開発されている。しかし、この機能はデータを Key-Value ペアとしてのみ扱い、表形式データとしては扱えない。また、KVS へ保存したデータの参照のみしか行えない。

本研究で提案する機構では SQL エンジンがレコード単位で行うアクセスをフックし KVS へアクセスさせることと、入出力データを表形式データとして取り扱うことにより、標準的な SQL 操作をサポートできる。次章では提案機構の設計について説明する。

3.SQL エンジンによる KVS のアクセス

本論文では、SQL を用いて各種の KVS に対して入出力を行う機構を実装し評価する。SQL で KVS を扱えるようにすることにより、ユーザは KVS 独自の API を学ぶ必要なく KVS の高速性、レプリケーション、データ分散などの恩恵を受けられる。また、RDBMS と KVS のデータを結合した上での検索が可能になる。

本機構では KVS 上にあたかもテーブルが存在しているかのようにアクセスできるようにする。以下では、この仮想的なテーブルを KVS テーブルと呼ぶ。KVS テーブルでは、従来の RDBMS のテーブルと比較し大きく使い方が異なることはないようにする。これは、構造を持つデータを容易に保存できるようにしたいためである。このようにしたとしても、非構造化データは Key-Value ペアの形式で扱うこともできるので、大きな問題はない。

図 1 に提案機構のアーキテクチャを示す。まず、SQL パーサはクライアントから入力された SQL 文を解析し、構文木の形にしてプランナへ渡す。プランナは実行プランを作成しプラン実行部へ渡す。次にプラン実行部は、実行プランに基づき、ファイルシステム上のデータベースファイルへのアクセスを行う。ここで、SQL 文が KVS テーブルに対するアクセスであると判断した場合には、データベースファイルへのアクセスをフックし、KVS ドライバ上の関数を呼び出すことで KVS へのアクセスへ置き換える。なお、SQL 処理の多くを占める DML において、実行プランはレコード単位でのアクセスとして作成されることが多くなる。このため、KVS へはレコード単位でアクセスする。そして、目的のレコード操作や検索・集計が完了したら、操作の結果がクライアントへ返る。

このようにすることで、実行プラン部が KVS へのアクセスを行うようにするため、他の手法と異なり、ほぼフルセットの SQL をサポートすることができる。また、図 2 に示すように、KVS ドライバを追加することで、様々な KVS へ対応できる。

KVS へのレコードの保存は、Key をテーブルの主キー、その他の値を Value としてエンコードして保存する。そして、取得時に Key-Value ペアからレコードをデコードする。これにより、ユーザが KVS へアクセスしていることを強く意識することなく、あたかも通常のテーブルのように使うことができる。

以下では、SQL 文による KVS アクセスの実現について説明する。具体的には、まずプラン実行部のフック、および表形式データと Key-Value ペアの相互変換 (エンコードとデコード) について説明する。その後、制限点とその回避策について説明する。

3.1. プラン実行部のフック

提案機構では KVS テーブルを実現するため、プラン実行部にフックを挿入し、KVS ドライバを呼び出し、必要な処理を行う。フックを挿入するポイントを表 1 に示す。

まず、CREATE TABLE 文や DROP TABLE 文ではテーブル作成・削除処理をフックし、テーブルのメタ情報を KVS へ登録・削除する。これは SQL 処理部が異常終了したときに、メタ情報が消えてしまいテーブルにアクセスできなくなるのを防ぐためである。また、KVS 上にメタ情報を置くことで、SQL 処理部がボトルネックとなった時に処理プロセスの数を増やすことができるようになる。

第二に、INSERT 文、UPDATE 文で発生するレコード更新処理では、新しいレコードをファイルに書き込むところをフックし、ファイルの代わりに KVS へ書き込むようにする。この書き込みは KVS 上のデータの一貫性を保つために即座に行う。DELETE 文で発生するレコード削除についても同様の方法で実行する。

第三に、SELECT 文、UPDATE 文、DELETE 文で発生するレコード検索処理では、インデックス選択とカーソルオープン、クローズをフックする。インデックス選択では SQL 文の検索条件式に応じて KVS のスキャンの方法を決める必要がある。例えば、主キーの一

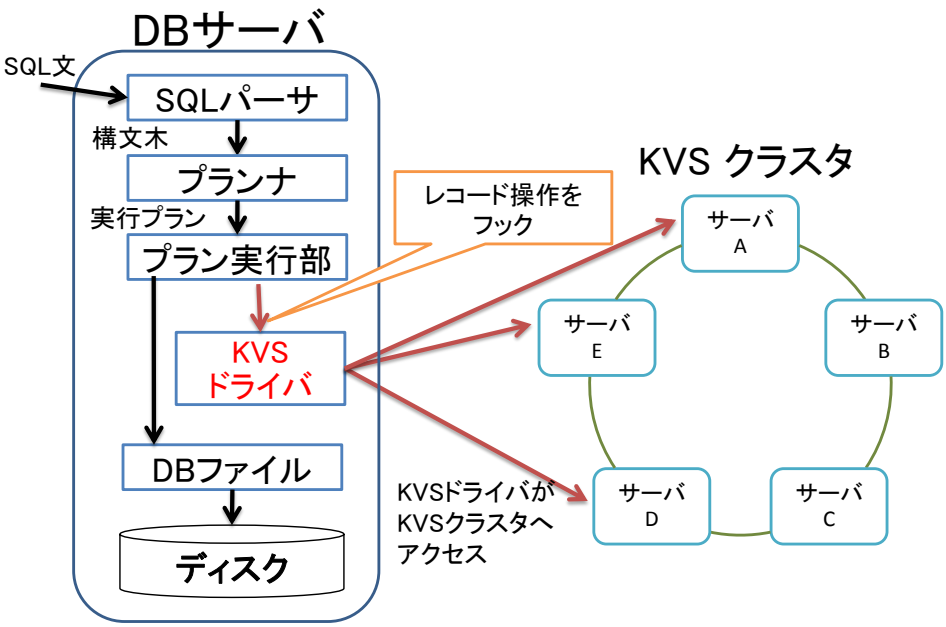

図 1: 提案機構のアーキテクチャ

致検索ならそのまま直接 KVS から GET 命令によってデータを読み出せばよい。一方、そうでなければ KVS テーブルの全データをスキャンし条件一致検索を行う必要がある。また、カーソル処理では、カーソルを定義できるよう、テーブル内でのカーソル位置と順序を定める必要がある。実装では、KVS が持つ KVS テーブルに関するキーを一旦全て取得し、その中で位置と順序を定めるようにしている。これは主キーがユーザによって設定されるため、どのような値を取るかわからないためである。もしアプリケーションが整数値の主キーを 1 から順に付けるという仮定が可能ならば、現時点での主キーの最大値を記憶しておくことでこの処理の負荷を軽減することができる。

最後に、SELECT 文、UPDATE 文でレコード取得処理を行うポイントにフックを挿入する。このフックの中では、KVS からデータを取得し、デコードを行い、プラン実行部の解釈可能な形にデータを整形してプラン実行部へ返す。ここで必要になるデータのエンコード・デコード方式については次節で説明する。

3.2. 表形式データと Key-Value ペアの相互変換

本機構では KVS にテーブルのレコードを Key-Value ペアとしてエンコードして保存する。これにより、ユーザが KVS へアクセスしていることを強く意識することなく、あたかも通常のテーブルのように使うことができる。

Key については、DB 名、テーブル名、KVS テーブルの主キーの 3 つをつなげたものを Key として用いる。例えば、DB 名 “test.db”、テーブル名 “table1” において、主キーが 1 であるレコードの Key は “test.db-table1-1” となる。こうすることにより、KVS のキー空間に多数のテーブルが別々に存在することができる。

Value については、レコード内の各カラムの情報を文字列に変換し、繋げることでエンコードを行う。デコードを正しく行えるようにするため、エンコード時に以下の 3 つの情報をつなげて文字列とする。

- **データの種類**　デコードの種類を決定する必要があるため、型を保存する。これはデータの先頭に 1 バイトの符号をつけることで行う。なお、データの種類についてはテーブルごとのメタデータとして別途保存することも可能である。しかし、今回は SQL 実行部が厳密な型チェックを行わないで済むよう、このようにした。
- **データの長さ**　レコードに含まれるカラムが可変長の場合は、カラム毎にデータ長を保存する必要がある。例えば、可変長テキストデータの場合は、テキストデータとともにその長さを保存する。これにより、任意の大きさのデータを保存することが出来る。
- **データ**　数値型やテキスト型についてはそのままテキストとして保存する。しかし、BLOB オブジェクトのエンコードには BASE64 などのテキスト形式になるエンコードを用いる必要がある。これは KVS 側の制限として任意のバイナリ列をそのまま保存はできないことが多いためである。実際、Redis ではバイナリデータをエスケープ文字によりエンコードして保存するため、保存するオブジェクトの数倍のデータ量を送信する必要がある。この問題を回避するには KVS 側を対応させる必要がある。ここで、BASE64 形式の保存であれば 1.3 倍程度で済むため、今回は BASE64 を選択した。

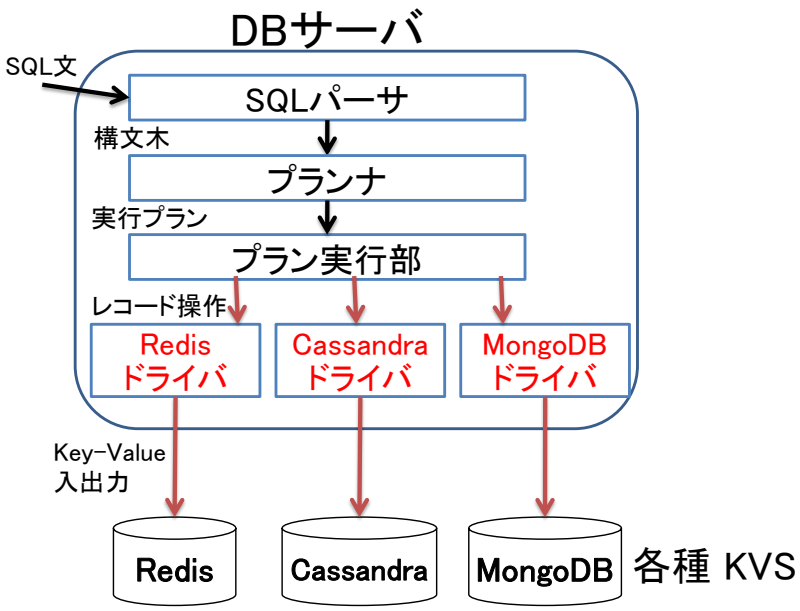

図 2: KVS ドライバの追加イメージ

表 1: KVS テーブルを実現するため、プラン実行部に挿入するフック

SQL 文の種類	フックするポイント	必要な処理
CREATE TABLE	実行時	KVS へメタ情報を与える
DROP TABLE	実行時	KVS 上のメタ情報を削除する
INSERT	レコード書き込み時	KVS 上にレコードデータを書き込む
UPDATE	レコード更新時	KVS 上のレコードデータを更新する
DELETE	レコード削除時	KVS 上のレコードデータを更新する
SELECT, UPDATE, DELETE	インデックス選択時	検索条件で KVS をどのように検索するか決定する
SELECT, UPDATE, DELETE	カーソルオープン時	KVS テーブルのカーソルを作る
SELECT, UPDATE, DELETE	カーソルクローズ時	KVS テーブルのカーソルを破棄する
SELECT, UPDATE	レコード取得時	KVS 上のレコードデータを取得する

表 2: 本論文での実装で用いたデータのエンコード形式

値の型	エンコード例
INTEGER	”i123456”
FLOAT	”f0.12345”
TEXT	”t4-text”
BLOB	”b12-BASE64STRING”

表 2 に本論文での実装で用いたデータのエンコード形式を示す。例えば 3 カラムに設定した KVS テーブルに `(1, 'text', 0.001)` というレコードを挿入した際には、`("test.db-table1-1", "i1t4-textf0.001")` という Key-Value ペアとしてシリアライズされる。このようにエンコードすることにより、1 つのレコードを 1 つの Key-Value ペアとして保存できる。ここでエンコードされたデータが主キーを持っているのは、実装の簡略化のためである。

3.3. 提案手法の制限点

KVS テーブルを実現するにあたっては、以下の制限点がある。

- **テーブルに主キーが必須である**　テーブル設計上、KVS テーブルには主キーを置くことが必須である。これは KVS のキーとして主キーを加工したものを用いるためである。しかし、RDBMS における通常のテーブルにおいては、テーブル設計上主キーを持つことが多いため、これは大きな問題にはならない。
- **SQL 文のパース処理・実行プランの作成処理がボトルネックになる**　提案機構では SQL 文をパースし、実行プランを作成する処理が必要となる。従って、KVS が SQL 処理部に対して極めて高速である場合には、これらの処理がボトルネックになってしまう。しかし、RDBMS では SQL 文を予めプリコンパイルしておき、実行プランになった段階で値をバインドしながら動作させる機能がある。従って、本機構でもこのようなプリコンパイル機構を用いることで、このボトルネックを回避できる。また、このような際には、第 3.1 節で述べたとおり、処理プロセスを分散することができる。

- **トランザクションが難しい**　KVS では完全に ACID 性を保証したトランザクションをサポートしないものが多い。これはデータへのアクセスを高速化する目的や、クラスタ上へのデータ分散をサポートする目的でそのようになっている。従って、KVS テーブルでもトランザクションを実施した場合には ACID 性を保証できない。KVS ドライバがトランザクション・マネージャとして動作することで ACID トランザクションを実装することは可能 [12] である。しかし、トランザクションの制御やデータのロックが必要になるため、KVS ドライバが複雑化し、低速化を招く可能性が高い。

4. 実装

東芝で作成しているデータベースである TinyBrace の SQL パーサ、プランナ、およびプラン実行モジュールを改変し、実装を行った。KVS はオープンソースの KVS である Redis を用いた。実装量は C 言語で 1,200 行程度であった。

4.1. 試作上の制限点

第 3.2 節、第 3.3 節で述べた以外に、実装の手間を軽くするための制限として、データベーストリガ、および複合主キーを使えない点を制限とした。これらの点については今後アプリケーションの要求に応じて対応する。

また、今回の実装は単純なもので、今後高速化の工夫を行う必要がある。考え得る高速化は 2 点ある。第一に、高速化のために INSERT による KVS への SET 命令をバッファリングし、ある程度まとめて出すようにする。このような最適化は書き込み速度が重要で読み込みが少なくなるユースケースでは特に有効である。しかし、GET 命令があった場合に読み出されるデータがちゃんと一貫性を保つようにする必要がある。第二に、テーブルスキャンのような GET 命令が連続することが予測される場合、大量の GET 命令を一回で出すようにする。これにより、テーブルスキャンを行わなければならなかった場合の速度の低下を抑えられる。

5. 評価実験

提案機構の性能を評価するため、作成したプロトタイプを用いてベンチマーク実験を行った。ベンチマーク実行環境を表 3 に示す。計測は同一マシン上で KVS サーバと SQL 実行部を同時に動作させ計測した。ベンチマークは

1. レコードの挿入の実行速度
2. 主キー一致検索の実行速度
3. テーブル内のレコードのスキャンの実行速度

について行い、同等の処理を直接 Redis の API で行った場合、および MySQL 5.5.23 で行った場合について比較した。なお、テーブルのレコード数は 10,000 件、同時接続するクライアント数は 50 とした。

用いた SQL 文を図 3 に示す。'?' の部分はプレースホルダであり、実行時にランダムな値が入る。ベンチマークプログラムは指定した SQL 文を指定した回数の実行する時間を測定する。具体的には、各クライアントは 1. を 10,000 回、2. を 10,000 回、 3. を 10 回行う。その後、全てのクライアントがそれぞれの SQL 文の処理にかかった時間を出力する。なお、図では SQL 文で示してあるが、実際 Redis の API で行った動作は GET と SET の組み合わせで同等の処理を行う。また、MySQL では主キーが重なったときにエラーにならないよう、INSERT 文ではなく REPLACE 文を使用した。測定は 10 回繰り返し、平均値を取った。

表 3: 評価実験の実行環境

機種名	Dell Vestro 460
CPU	Intel Core i7 3.4GHz (4 コア)
メモリ	16GB
HDD	2TB HDD (RAID1)
OS	Linux 3.3.0 (Fedora Core 16)
KVS	Redis Ver. 2.4.7
SQL エンジン	TinyBrace

ベンチマーク結果を表 4 に示す。提案機構は SQL 処理の実行プランを実行する部分について 30%近くのオーバヘッドがある。しかし、これにより SQL で操作が可能という利点がある。

提案機構、Redis 単体実行時共に テーブルスキャンの性能が MySQL に比較し大きく劣っている。これは MySQL が複数のレコードをページ単位でメモリにロードすることでスキャン時に必要なアクセス回数を抑えている [13] ためである。一方で、提案機構 や Redis では、1 レコード単位でアクセスしているため、アクセス数が多くなってしまうため性能が出ていない。これを解決するには、第 4.1 節で述べたようなテーブルスキャンの高速化を行うとよいと考えられる。

6. まとめ

KVS では、RDBMS における SQL のような共通のインターフェースは持たず、データアクセスのための独自の API を用いており、従って、データの検索・集計処理の多くは開発者がプログラムを書いて行う必要がある。そこで本論文では SQL を用いて各種 KVS に対して入出力を行うスキームの一つを提案した。SQL により KVS をアクセスできるようにすることで、複雑なプログラムを書くことなしに、柔軟なデータの検索・集計を行うことができる。本論文では、東芝で開発している RDBMS である TinyBrace と オープンソースの KVS である Redis を組み合わせたプロトタイプを実装し評価した。実験の結果、Redis の SQL による操作が行えていることを示した。今後はトリガー処理や複合主キー、および複数キーによるインデックスなど、実際のアプリケーションで必要とされる処理を作成する必要がある。

参考文献

[1] Redis. http://redis.io/.

[2] MongoDB. http://www.mongodb.org/.

```
CREATE KVS TABLE table1 (KEY INTEGER PRIMARY KEY, COLUMN1 TEXT);

-- 以下をそれぞれ繰り返す --
1. INSERT INTO table1 VALUES (?, '?')  -- 10,000 回
2. SELECT * FROM table1 WHERE KEY=? -- 10,000 回
3. SELECT * FROM table1 WHERE COLUMN1='?' -- 10 回
```

図 3: ベンチマークプログラムの実行する SQL 文

表 4: 実験結果

	提案機構	Redis	オーバヘッド (%)	MySQL (参考)
レコード挿入	109,844 レコード/秒	144,981 レコード/秒	24.2%	17,905 レコード/秒
一致検索	114,273 レコード/秒	161,158 レコード/秒	29.1%	115,830 レコード/秒
テーブルスキャン	12.0 回/秒	14.3 回/秒	16.1%	844.8 回/秒

[3] The Apache Cassandra Project. http://cassandra.apache.org/.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI '06)*, November 2006. http://labs.google.com/papers/bigtable.html.

[5] 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. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07)*, pp. 205–220, 2007.

[6] Rakesh Angrawal, Anastasia Ailamiki, Philip A. Bernstein, et al. The claremont report on database research. *Communications of the ACM*, Vol. 52, , June 2009. http://db.cs.berkeley.edu/claremont/claremontreport08.pdf.

[7] Apache Hive. http://hive.apache.org.

[8] Database Research at Yale University. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. . http://db.cs.yale.edu/hadoopdb/hadoopdb.html.

[9] 製品情報 - Oracle Coherence. http://www.oracle.com/technetwork/jp/middleware/coherence/overview/index.html.

[10] PostgreSQL: Documentation: devel: CREATE FOREIGN DATA WRAPPER. http://www.postgresql.org/docs/devel/static/sql-createforeigndatawrapper.html.

[11] dpage/redis_fdw - GitHub. https://github.com/dpage/redis_fdw.

[12] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. CloudTPS: Scalable transactions for Web applications in the cloud. Technical Report IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, February 2010. http://www.globule.org/publi/CSTWAC_ircs53.html.

[13] MySQL 5.1 リファレンスマニュアル 9.11.2. ファイル領域管理. http://dev.mysql.com/doc/refman/5.1-olh/ja/innodb-file-space.html.