日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

Cassandraを使ったCMSのPCクラスタを使ったスケーラビリティの検証

玉城 将士　　河野 真治

現在, 数ある分散 Key-Value ストアの中でも Cassandra が注目を集めている. Cassandra は Consitency level の変更が可能であり、スケーラビリテイを高めるための使い方には工夫が必要である. 本研究では, Cassandra 上で動作する CMS を実装し学科のクラスタ上で動作させる. 特に, CoreDuo などの安価だが非力なマシンの振舞を調べることを行なった. そしてその環境上でスケーラビリティを確認する実験手法に関して考察する.

1 はじめに

インターネットやスマートフォンなどの普及に伴い，インターネット上のサービスを使用するユーザーが急速に増え続けている. サービスを利用するユーザーが増えると，いままでのシステムでは膨大なアクセスに対応できなくなり，サービスの品質を維持することができなる. 品質を維持するためには，使用するサーバー性能の向上を測ればよい. しかし，性能の良いサーバーを揃えるには膨大なコストを必要とし，これをスケールアップと呼ぶ. そこで，安価なサーバーを複数用意し，連携させることによって性能を向上させる方法があり，これをスケールアウトと呼ぶ. この方法では，従来使用してきたソフトウェアを複数のサーバーに移動するだけではうまく動作しない. 複数のサーバーを強調させるのは難しく，データの整合性や通信速度，負荷分散など様々な考慮をしなければならないためである. Cassandra は複数のサーバーで動作を想定した分散データベースである. 本研究では，実際に分散させることによって高価なサーバーを超えることが出来る性能を出すことが出来るのか，また，どの様に Cassandra 上で動くソフトウェアを開発することによって性能を発揮することが出来るのかを，９０台の PC クラスタ上でベンチマークを取り検証する.

Shoshi TAMAKI, Shinji KONO, 琉球大学工学部情報工学学科, Dept. of Information Engineering, Ryukyu University.

2 分散データベース Cassandra

Cassandra は, FaceBook が自社のために開発した分散 Key-Value ストアデータベースである. 2008 年にオープンソースとして公開され, 2009 年に Apache Incubator のプロジェクトとなった. 2010 年には Apache のトップレベルプロジェクトとなり, 現在でも頻繁にバージョンアップが行われている.

2.1 ConsictencyLevel

Cassandra には, ConsistencyLevel が用意されている. これは, 整合性と応答速度どちらを取るか選ぶためのパラメータであり, リクエストごとに設定することが出来る. また, Read と Write で ConsistencyLevel の意味は異なる. この ConsistencyLevel を適用するノードの台数を ReplicationFactor といい, Cassandra の設定ファイルで設定することが出来る.

Read

1. ConsistencyLevel::ZERO
   サポートされていない.
2. ConsistencyLevel::ANY
   サポートされていない.
3. ConsistencyLevel::ONE
   一番最初に返答したノードの値を返すが値が最

新のものであるかは保証できない. 整合性の調査は常に非同期で行われており, 再度読み出しを行うときに結果が変わっている可能性がある.

4. ConsistencyLevel::QUORUM
   すべてのノードにリクエストを送信し, 取得した値のタイムスタンプを比較し, 最も多数のノードが返した値のうちで最新のタイムスタンプを持つ値を返す.
5. ConsistencyLevel::ALL
   すべてのノードにリクエストを送信し, もっともタイムスタンプの新しいノードの値を返す.

Write

1. ConsistencyLevel::ZERO
   何も保証しない, 書き込みは非同期的に行われる.
2. ConsistencyLevel::ANY
   別のどこか他のノードに書き込まれることを保証する.
3. ConsistencyLevel::ONE
   最低 1 つのノードのログとメモリテーブルに書き込まれていることを保証する.
4. ConsistencyLevel::QUORUM
   (ReplicationFactor/2) + 1 のノードに書き込むことに書き込みを終えてからクライアントにレスポンスを返す.
5. ConsistencyLevel::ALL
   ReplicationFactor のノード数に書き込みを終えてからレスポンスを返す.

2.2 コンシステント・ハッシュ

Cassandra は複数のノードにデータを分散して格納する. その為に使用されているのがコンシステント・ハッシュである. 普通, n 台で構成されたノードにデータを分散する場合, HASH(key) mod n で分散させる. この場合だと, ノードが追加・削除された場合すべてのデータの位置を再計算する必要があり面倒である.

そこで, 図 1 のようなものを考える. 図 1 はハッシュ関数が取りうる値を範囲としたリングである. このリング上に構成するノードを配置していく. この図の場合, アルファベットがノードで数字がデータ, 矢印が担当するノードである. 次に, ハッシュ関数により計算された値をリングの上に配置する. このとき, リングを右回りに周り一番最初にあたったノードがデータを担当するノードとする. こうすると, ノードが追加・削除された場合に, 全体を再計算する必要はなく, 担当するノードがいなくなったデータのみを再計算し, 次の担当するノードに移せばよい. Cassandra では, 右回りに回ったとき担当するノード数を複数にする場合, ReplicationFactor で調整することが出来る.

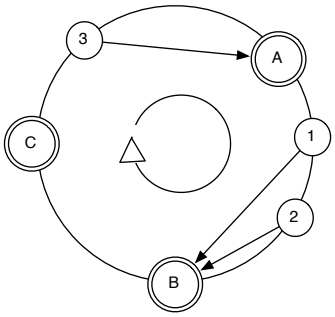

図 1 コンシステントハッシュ

2.3 SEDA

SEDA(Staged Event-Driven Architecture) は, Cassandra で使用されているアーキテクチャである. 処理を複数のステージに分解しタスクキューとスレッドプールを用意し処理を行う. 処理の様子を図 2 に示す. タスクが各ステージのタスクキューに入ると, スレッドプールにどれかのスレッドがタスクキューの中からタスクを取り出し処理を行う. 処理が終わるとそのタスクを次のステージのタスクキューに入れる.

このアーキテクチャはマルチスレッドベースなためマルチコアな PC と多数のタスクがある状況で性能を発揮することができる. しかし, あまりにもスレッドプールやタスクが多すぎると, コンテキストに切り替えに時間がかかり性能は低下する. そのため, Cassandra では最低 4 コアを搭載した計算機で動作させることを推奨している.

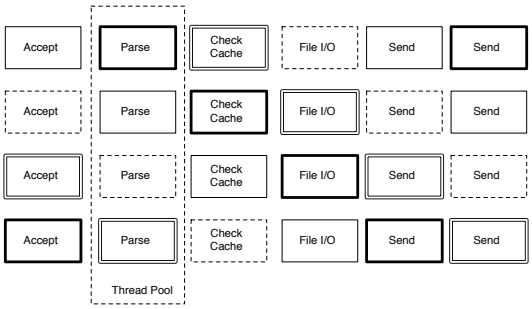

図 2 SEDA

2.4 Cassandra 上でのステージの構成

Cassandra は主に以下のステージにより構成されており, concurrent. StageManager を参照すると見つけることが出来る.

- READ STAGE
- MUTATION STAGE
- STREAM STAGE
- GOSSIP STAGE
- RESPONSE STAGE
- AE SERVICE STAGE
- LOADBALANCE STAGE
- MIGRATION STAGE

実際にはもっと多数のステージが存在し, この他にもクライアントの接続を待つスレッドプールや MemTable の Flush を行うスレッドプールがあり, 全部で 40 個程度のスレッドが動作している.

2.5 YukiWiki on Cassandra

今回の検証のため, CMS のである Wiki クローンの YukiWiki を Cassandra 上で動作するように改造した．YukiWiki は文書の管理に TIEHASH を使用しており，Cassandra 用の TIEHASH を作成することで簡単に実装することが出来る.
Cassandra 上で動作するため，この Wiki で複数のサーバー上でデータを共有することが出来るようになった.

3 実験

本研究では, Cassandra のスケーラビリティの検証の為にベンチマークテストを行う. 実験環境は以下のとおりである.

3.1 実験環境

1. クラスタ (クライアント)
   - CPU : Core Duo
   - Mem : 1GB
   - O S : CentOS 5
2. 実験用サーバー 1(MacMini)
   - CPU : Core2 Duo
   - Mem : 4GB
   - O S : OSX SnowLeopard
3. 実験用サーバー 2
   - CPU : Core i7 950 @3.0GHz
   - Mem : 16GB
   - O S : CentOS 5

3.2 実験方法

1. クライアントクラスタ管理ツールの Torque を使用し, 使用するノード数を指定してクラスタにジョブを投げて PHP スクリプトを実行させる. この PHP スクリプトは Cassandra と MySQL に 10000 回リクエストを送信するスクリプトである.
2. Cassandra Cassandra 0.6.3 を使用した.
3. MySQL MySQL 5.5 を使用した．Cassandra と似たデータ構造を持たせるために表 1 のような構造でテーブルを作成した.

表 1 テーブルの定義

フィールド名	データタイプ	備考
NAME	VARCHAR(100)	UNIQUE
VALUE	VARCHAR(100)	-
TIMEUUID	LONG	-

4 実験結果と考察

4.1 単純なベンチマーク

はじめに, 単純なベンチマークを行った. 単体のクライアントとサーバーを用意し, Cassandra と MySQL の実行時間の比較を行った. 結果を表**??**に示す. この時の Cassandra の ConsistencyLevel は ONE である.

結果を見てみると, MySQL より Cassandra のほうが高速に動作していることが分かる. MyySQL は C++で記述されているが Cassandra は Java であるため, 動作が遅い. よって, 単純な使用方法では Cassandra より MySQL の方が優れていると言える, 普通の方法では Cassandra の性能を引き出すことは出来ない.

表 2 単純なベンチマークの結果 (Read)

	Cassandra	MySQL
サーバー 1	13.72s	5.94s
サーバー 2	12.56s	3.99s

表 3 単純なベンチマークの結果 (Write)

	Cassandra	MySQL
サーバー 1	11.75s	5.7s
サーバー 2	9.62s	5.3s

4.2 コア数の少ないサーバー上でのベンチマーク

次に, クライアントを並列化しての実験を行う. ここでは, コア数の少ないサーバー 1 を用いる. クライアントの並列化はスクリプトを指定した時間に同時起動するようにして実装した. 実験結果を図 3 と図 4 に示す.

Read は両方とも, 同じような推移の仕方をしているが, Cassandra の方が遅い. しかし, Write は Cassandra の方が断然速く動作している. この実験では, Cassandra の動作を基準に考えたため書き込みのコマンドに REPLACE を使用した. REPLACE は置き換えるようなコマンドである. そのため, INSERT に比べて多少遅くなる. それがこのグラフに出ているのではないかと考えられる. SEDA は複数のスレッドで動作しているためコア数が少ないサーバーでは性能が出にくいことがわかる.

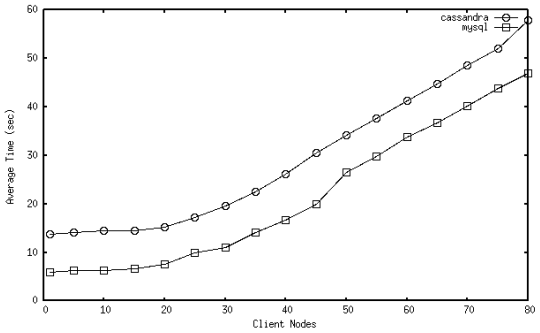

図 3 サーバー 1 上でのベンチマーク (Read)

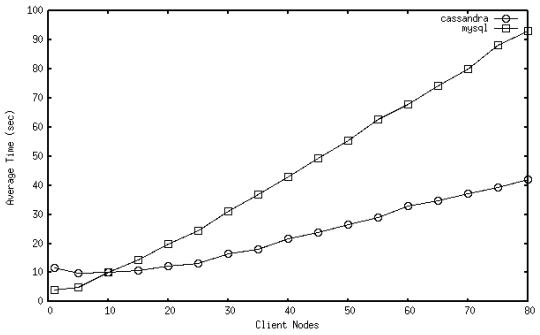

図 4 サーバー 1 上でのベンチマーク (Write)

4.3 コア数の多いサーバー上でのベンチマーク

クライアントを並列化した状態で, コア数の多いサーバー 2 を用いたベンチマークを行う. 実験結果を図 5 と図 6 に示す.

Read/Write 共に MySQL の性能を超えることに成功した. Read においてはコア数が少ない場合に超えることが出来なかったが, 並列度が 70 度付近で MySQL を超えることが出来ている. Cassandra の平均時間は並列度がましても, MySQL よりは平均時間の増加度が少ない. これは, SEDA の特徴である, 多くのタスクを並列に実行すると性能がでるという部分が確認することが出来た. また, SEDA はマルチスレッド前提であるため, コア数が少ないサーバー 1 で

は性能が出ず, コア数の多いサーバー 2 で性能が発揮できるということがわかった.

つまり, Cassandra は負荷が高いときに MySQL を超える性能を出すことが出来る. 負荷がかかっても性能の劣化が少ないことを考えると考えると遅延をあまり考慮しなくても済むのではないだろうか.

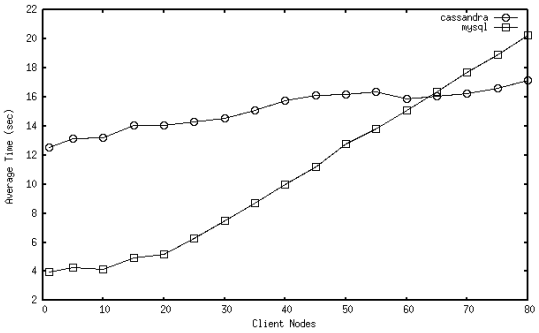

図 5　サーバー 2 上でのベンチマーク (Read)

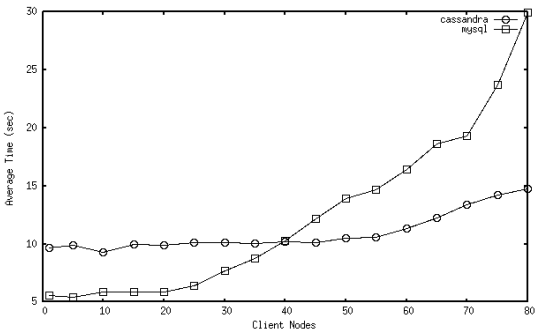

図 6　サーバー 2 上でのベンチマーク (Write)

4.4　複数ノードで構成した Cassadra のベンチマーク

最後に分散しなかった Cassandra と複数ノードで構成した Cassandra の比較を行う. サーバーはサーバー 1 を 5 台使用して行った. 実験結果を図 7 と図 8 に示す.

Read/Write ともに, 今回の場合は分散を行わなかったほうが性能を引き出せてることが分る. これは, 実験に使用したデータが Read/Write 共に 1 つだけで, 結局は同じノードにリクエストが転送されている. そのため, リクエストは 1 台のノードに集中する. よって, 性能が出ないのではないかと考えられる. Cassandra をただ増やすだけでは性能は得ることが出来ず, データも分散させて実験を行わないといけないことがわかった.

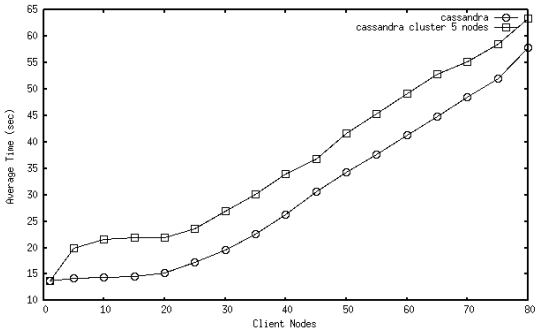

図 7　サーバー 1 を複数ノードにしたベンチマーク (Read)

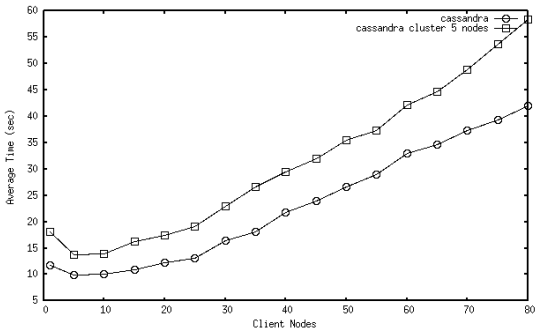

図 8　サーバー 1 を複数ノードにしたベンチマーク (Write)

5 まとめ

今回の実験で, Cassandra を使用するには従来の使用方法ではいけないということがわかった．Cassandra はコア数が少ない場合, Read は MySQL より遅いがほぼ同し推移の仕方をする．Write は，コア数が少なくてもクライアントの並列度を高く設定すれば，MySQL に勝つことがある．コア数が多い場合，Read・Write 共に，初めはやはり MySQL の方が動作が早いが，グラフの傾きは MySQL の方が大きく Cassandra はかなり緩やかである．特に Cassandra の White の性能は高く, MySQL を大きく上回っている．また, 単純に Cassandra のノード数を増やしても性能は高くならない．これは, データも綺麗に分散させてあげないとデータを読み込む際に一定のノードに集中してしまい，他のノードにアクセスを分散しても結局は保持しているノードに聞きに行かないといけないことになるからである．データもある程度分散させなければならないため，汎用的な HASH 関数では性能が発揮できなく, そのアプリケーション専用の関数が必要だと思われる．格納されるデータを決めるのに Partitioner というものがあり, それを利用することで実装できると思われる．

6 今後の課題

今後は, Partitioner を拡張し複数のデータをノードに分散させた環境下でベンチマークを行い, その結果を Cassandra 単体でのベンチマーク結果と比較したいと考えている．他にも, 沖縄東京間などの離れた地域での分散を Cassandra でどの様に行なっていくか実験していきたい．

参 考 文 献

[1] Benchmarking Cloud Serving Systems with YCSB
[2] The Staged Event-Driven Architecture for Highly-Concurrent Server Applications
[3] SEDA : An Architecture for Well-Conditioned , Scalable Internet Services
[4] Bigtable : A Distributed Storege System for Structured Data