Vol.2011-OS-116 No.9
2011/1/25

読み出し性能と書き込み性能を選択可能なクラウドストレージ

中村俊介[†1]　首藤一幸[†1]

読み出し性能重視となるか書き込み性能重視となるかは、分散データストアにおいてもストレージエンジンが決めると予測した。もしそうであれば、読み出し性能と書き込み性能を、データストア自体の使い分けではなく、ストレージエンジンを差し替えることで調整ができる。そこで実証のために、ストレージエンジンの差し替えが可能な分散データストア MyCassandra を開発し、クラスタ上で元の Cassandra とストレージエンジンを MySQL とした場合を比較した結果、書き込み比率の高いワークロードでは前者の書き込み遅延が後者より 41.4%小さく、読み出し比率の高いワークロードでは後者の読み出し遅延が前者より 49.4%小さくなることを確認した。

A Cloud Storage Adaptable to Read-Intensive and Write-Intensive Workload

Shunsuke Nakamura[†2] and Kazuyuki Shudo[†2]

We expect that a storage engine determines whether a cloud storage is read-optimized or write-optimized.

It means that a single cloud storage can be both of them just by replacing its storage engine with another one. It is not necessary to use another cloud storage to adjust a balance of read and write performance.

The expectation was confirmed by performance comparison of Bigtable-based storage engine of the original Cassandra and MySQL storage engine. Write latency of the former is 41.4% lower than the latter with a write-heavy workload, and read latency of the latter is 49.4% lower than the former with a read-heavy workload.

†1 東京工業大学
†2 Tokyo Institute of Technology

1. はじめに

RDBMS が満たしきれない要求に応える分散データストアとして NoSQL や Key-Value Store といったクラウドストレージが注目されている。これらは従来の RDBMS と比べてデータモデルやクライアント側の機能を制限し、また一貫性についての条件を緩めることで、負荷の分散を比較的容易にし、ノード数をスケールアウトさせやすいという点が共通の特徴である。

しかしながら、それぞれのクラウドストレージは様々な点で異なっている。例えば、データモデルには key-value 形式や多次元マップ (multi dimentional map) 形式があり、分散の構成は master/worker 型やツリー型や非集中分散型があり、データを全てメモリに保持することで遅延を抑えたものや永続化が可能なもの、複製の配置方法やそれを同期/非同期で行うかなどというように様々な設計方針がある。また、従来のクラウドストレージには書き込み性能重視のものと、読み出し性能重視のものがある。例えば、Apache Cassandra[1)2)] や Apache HBase[3)4)] は書き込み性能重視であるが、一方、Yahoo Sherpa[5)] や sharded MySQL[6)](MySQL で sharding) は読み出し性能重視である。

クラウドストレージの利用者がこのような多くのクラウドストレージの中から利用用途に忠実に沿うものを選択するのは困難である。例えば、利用者が Redis のようなキューやスタックを扱えるデータモデルで、かつ Cassandra のように単一故障点の無い高い可用性を持ったデータストアを利用したい場合はアプリケーションを Cassandra のデータモデルに無理やり適用を強いるか、自身で新しいクラウドストレージを作ることになるのが現状である。

そこで我々は既存クラウドストレージを「分散のための機構」と「ストレージエンジン」に分離し、後者の「ストレージエンジン」を同一データストア内で差し替えることで、全く異なる読み出し/書き込み性能の性質を得られるということを提案する。我々はこの提案を示すために Apache Cassandra をベースに、MyCassandra というモジュラーな分散データストアを開発した。

本稿では、MyCassandra の設計方針を説明するとともに、実際にストレージエンジンを差し替えることで読み出し性能重視か書き込み性能重視かを選択できることをベンチマークを用いて示す。また、異なるストレージの組み合わせによるクラスタを構成することにより、読み出し性能と書き込み性能の両立ができる MyCassandra Cluster を提案する。

本稿の構成は以下のとおりである。２章でまず提案手法とその実装ある MyCassandra を

表 1 既存クラウドストレージの特徴
Table 1 Charactics of existing cloud storages

	Cassandra, HBase	Sherpa, sharded MySQL
書き込み	Diff Sequential	Key lookup Update
読み出し	Diff Merge	Single Read
性能	書き込み性能重視	読み出し性能重視
ストレージエンジン	Bigtable 由来	MySQL

説明した後に MyCassandra の実装に用いた Apache Cassandra と MySQL などの各ストレージエンジンについて簡単に述べる。３章で性能評価に用いた YCSB ベンチマークについて述べた後に、性能測定を結果示して考察を行う。４章でモジュラーなクラウドストレージに関する関連研究について紹介する。５章で本研究の貢献をまとめ、また、読み出し性能と書き込み性能を両立する MyCassandra Cluster を提案する。

2. 提案手法・設計

クラウドストレージを用いる大規模なシステムでは、次にどのデータが参照・更新されるかを予測することは難しく、全てのデータがメモリ上に収まらない限り、読み出し時にディスクへのランダム I/O が発生してしまう。書き込み時にランダム I/O を行うよりも、ディスクへのシーケンシャル I/O のみでログを記録する方が、高いスループットを実現できるが、一方このようなログ記録方式では、読み出し時はログから一連の更新結果を拾い集めなければならないので効率が悪い。つまり、読み出し性能と書き込み性能は、常にトレードオフの関係がある。

このような理由から、既存の永続型クラウドストレージはあらゆるワークロードで優れた性能を提供しているのではなく、その設計は書き込み性能・読み出し性能どちらか一方に偏る。幾つかのクラウドストレージの性質を**表 1** に示す。Cassandra や HBase は書き込み性能を重視した設計がされているために、書き込み比率の高いワークロードに向いており、一方、Sherpa や sharded MySQL(MySQL で sharding) は読み出し性能を重視した設計がされているために、読み出し比率の高いワークロードに向いている。クラウドストレージの設計方針は１章で述べた通り、データストアによって様々であるが、我々はこの読み出し/書き込み性能の性質は主にストレージエンジンの違いによるものであると予測した。なぜならば、従来のデータストアのボトルネックになりがちなのは、データモデルの定義の仕方、分散の機構、トランザクションや JOIN といった機能のサポートの有無といった部分よりも、むしろ各ノードでのディスク入出力部分であり、それは分散のための機構に依存せず分散データストアにも当てはまると考えられる。実際に Cassandra や HBase は Bigtable のストレージエンジン (2.2.2 節参照) を採用することでディスクへの書き込みを差分のみシーケンシャルに書くことでランダム I/O が発生せず、書き込みを高速に行うことができるが、読み出し時には差分のマージ処理が必要となり、複数回のランダム I/O が発生するため読み出し性能が犠牲になる。一方、MySQL や Sherpa は従来のバッファプール方式により読み出しは１回の I/O で最新のレコードを引き出すことができるが、書き込みには古いレコードの読み出しが必要なため、ランダム I/O が発生するため書き込み性能が犠牲になる。

ところで RDBMS の１つである MySQL の設計はコネクションやデータの分散アルゴリズム、クエリパーサといった部分とストレージエンジン部分は独立したコンポーネントとして構成されている。そうすることで、利用者は欲しい性能や機能に応じたストレージエンジンを選択することでデータモデルやノードの分散構成などを変更することなく性能の調整を行うことができる。例えば、デフォルトの MyISAM の他にもトランザクション機能が欲しい場合は InnoDB、データを全てメモリ上で扱いたい場合は MEMORY、更新が必要ない場合は Archive というように MySQL には十種類を超えるストレージエンジンが用意されており、利用者はこれを容易に選択することができる。しかし、既存のクラウドストレージはそのような設計はなされておらず、利用者はアプリを特定のデータストアのデータモデルに無理やり適用させるか、足りない機能を他のソフトウェアと併用したり、もしくは自身で新しいクラウドストレージを実装しているのが現状である。

そこで本研究では Apache Cassandra をベースとして、同じシステム内で MySQL をはじめ、様々なストレージエンジンから選択が行えるようなクラウドストレージ MyCassandra を開発した。

以下ではまず MyCassandra のアーキテクチャを説明した後に、Apache Cassandra のアーキテクチャと、評価の為に MyCassandra に追加した各ストレージエンジンについて簡単に述べる。

2.1 MyCassandra

MyCassandra は、Cassandra をベースとし、「分散のための機構」と「ストレージエンジン」に分離した分散データストアである。これにより、後者のストレージエンジンを他の部分に影響を与えることなく、差し替えることができる。

図 1 は Cassandra と MyCassandra での各ノード上での読み書きに関係する部分を示している。MyCassandra には Cassandra のリクエスト受理部分と各ストレージエンジンの間

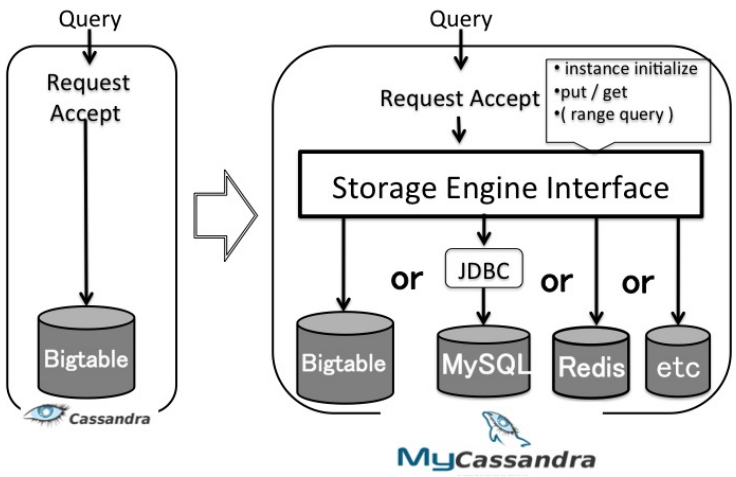

図 1 MyCassandra の Storage Engine Interface
Fig. 1 Storage Engine Interface of MyCassandra

```
1  public class KVSStorageEngine implements StorageInterface {
2    StorageInstance client;
3    String keyspace, cf;
4    final String KEYSEPARATOR = ":";
5
6    public Instance(String keyspace, String cf) {
7      client = new KVSClient(<HOST>,<PORT>);
8      this.keyspace = keyspace;
9      this.cf = cf;
10   }
11
12   public int put(String rowKey, ColumnFamily newcf) {
13     String key = keyspace+KEYSEPARATOR+cf+KEYSEPARATOR+rowKey;
14     ColumnFamily cf = (client.exists(rowKey) ? updateCF(rowKey, newcf) : newcf);
15     return client.set(key, serialize(cf));
16   }
17
18   public ColumnFamily get(String rowKey) {
19     String key = keyspace+KEYSEPARATOR+cf+KEYSEPARATOR+rowKey;
20     return deserizlize(client.get(key));
21   }
22 }
```

図 2 Storage Engine Interface の実装コード例
Fig. 2 Code example implementing Storage Engine Interface

に Storage Engine Interface を設けた。このインタフェースが規定する以下の関数を実装することで、MyCassandra に新たなストレージエンジンを追加できる。

- ストレージエンジンの初期化 (インスタンス生成、ネットワーク接続)
- データの put 関数と get 関数

利用者はこれらを JDBC 等のデータストアにアクセスする API を用いて実装できる。**図 2** は set/get API を持つ key-value store を MyCassandra のストレージエンジンとして追加するコードを表している。

Cassandra はスキーマレスの多次元モデルをデータモデルとして採用しており、これはそのままでは RDBMS や key-value store には格納できない。そこで、データの形式を変換する必要がある

図 3 は Cassandra のデータモデルを RDBMS である MySQL と key-value store である Redis にそれぞれマッピングしている例を示している。MySQL の場合は ColumnFamily を Table に対応付けており、スキーマレスなデータモデルを構成するために key とその行の複数カラムをシリアライズした value という key-value のペアでストアする方法をとっている。一方、Redis は同じく key-value のペア形式であるが、複数の表を管理するモデルになっていないために key に ColumnFamily 名を prefix として付加している。

MyCassandra は Cassandra 由来の分散のための機構を様々なストレージエンジンと共に用いることができるため、ストレージエンジンの異なるノード群を混在させることもできる。この詳細については 6 章で説明する。

2.2 Apache Cassandra

Apache Cassandra は、Facebook 社が開発し、Apache Project としてオープンソース化したクラウドストレージである。複数のデータセンターにまたがる数百台のノードで運用可能なスケーラビリティや、非集中で単一故障点を持たないことによる高い可用性などを特徴とする。

2.2.1 Consistent Hashing

Cassandra はデータの分散アルゴリズムとして Amazon Dynamo[7] を参考にした Consistent Hashing という非集中な分散アルゴリズムを用いている。

Cassandra は Amazon Dynamo にならった方式でデータを分散させる。つまり、consis-

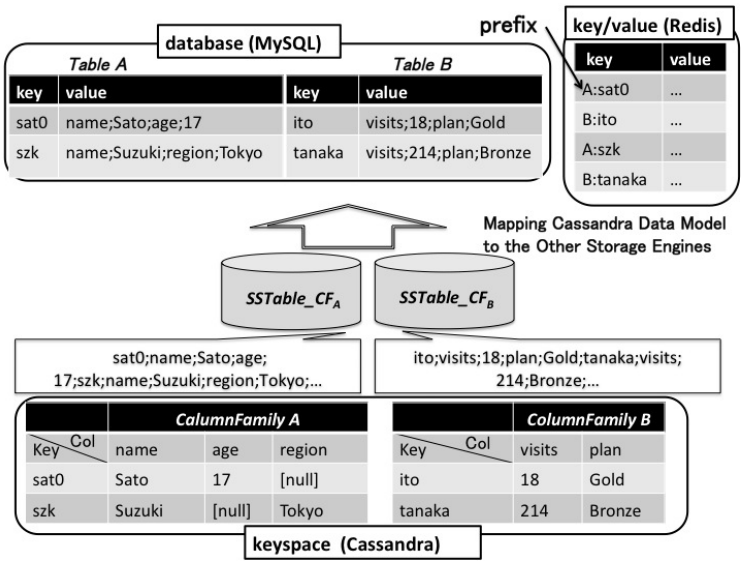

図 3 Cassandra データモデルのマッピング
Fig. 3 Mapping Cassandra's data model to RDBMS and key-value store

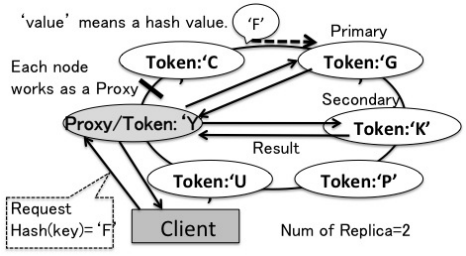

図 4 Consistent hashing
Fig. 4 Consistent hashing

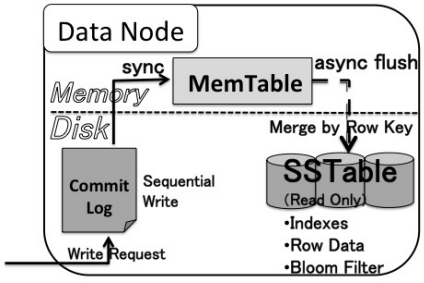

図 5 Cassandra の書き込みの流れ
Fig. 5 Write process in a Cassandra node

tent hashing というアルゴリズムでデータの担当ノードを決める。consistent hashing では、リング状の ID 空間（**図 4**）を考え、ノードとデータの双方に ID を割り当てる。ID が n ビットの場合、値は $0 \sim 2^n - 1$ となる。例えば図 4 の状況において、キーが 'F' である行は、'F' のハッシュ値から時計回りに最もノード ID が近い Token 'G' のノードが担当し、保持する。Cassandra ではノード ID を Token、担当ノードをプライマリノードと呼ぶ。

Consistent Hashing の主な特徴は ハッシュ関数を用いることで各ノードが担当するデータ数が比較的均等になり易く、負荷分散が容易であることと、データの担当範囲を集中管理するサーバーが必要ないので、単一故障性の無い高い可用性を提供できることである。各ノードは他ノードの位置を定期的に Gossip Protocol により情報交換している為、任意ノードがプロキシとしてクライアントのリクエストに応じることができる。

2.2.2 Bigtable のストレージエンジン

Cassandra のストレージエンジンは Bigtable と同じアーキテクチャを採用しており、それは Commit Log、MemTable、SSTable の 3 つのコンポーネントから構成される。Cassandra の書き込み処理の流れを**図 5** に示す。書き込みはまず永続用にディスク上の Commit Log に差分をシーケンシャルに書き、次に読み出しのパフォーマンスの為にメモリ上の MemTable に< Key, Data >というマップ形式で既にあるキーに関して更新を行い、クライアント側に書き込み成功のリプライを返す。MemTable のデータサイズが閾値を超えると、古いデータから順に非同期でディスク上の SSTable に単位サイズごとにキーでソートされたファイルとして書き出される。

この方式の利点は、ディスクへの書き込みが常にシーケンシャルであることと、一度ディスクに書きこまれた内容は変更されることが無いため、書き込みロックが不要で常に書き込みが可能という点である。一方、欠点としては読み出し時に、指定されたキーを持つ差分データを複数の SSTable から読み出してマージする処理が必要となる為に読み出し性能が犠牲になることである。ノードが落ちて MemTable の内容が消えても、ノード再起動時に Commit Log 上にある差分データを全て SSTable にフラッシュすることでデータの永続性を保つことができるようになっている。

2.2.3 Replication Strategy

Cassandra は複数のデータセンターを跨いだノード間でクラスタを構成することが想定されているため、ノード間でのレプリケーション配置方法として以下の 3 つが用意されている。

- Rack Unaware: リング上 ID 空間においてプライマリノードの右隣 N-1 個のノードに

配置

- Rack Aware: プライマリノードと異なるデータセンターに 1 個、異なるラックに N-2 個配置
- Datacenter Aware: 異なるデータセンターに均等に配置

(データセンター、ラックの比較は、それぞれ IP アドレスの 2nd, 3rd octet で判別)。例えば、レプリカ数が 2 で、Rack Unaware を選択した場合は、図 4 のようにハッシュ値が'F' の行は Token'G' とその隣の Token'K' に配置される。

2.3 MySQL / InnoDB

今回、ストレージエンジンの 1 つとして MySQL を MyCassandra に組み込んだ。MySQL へのアクセスには JDBC API を使用し、MySQL 内のストレージエンジンとしては InnoDB を選択した。

InnoDB はトランザクション処理のために設計されたストレージエンジンであり、デフォルトの MyISAM と比べるとロールバックによりクラッシュからの自動リカバリ機能がサポートされているなど機能面の充実性から、実サービスで幅広く使われている。

また、InnoDB はマルチバージョンの並行性制御 (MVCC) を使うことで高い並行性を実現し、SQL の標準分離レベル 4 つを全てサポートしている。

InnoDB のデータはクラスタ化インデックスに基づいて構成される。そのインデックス構造は他のほとんどの MySQL ストレージエンジンのものと大きく異なっており、主キーによる非常に高速なルックアップが実現できることが性能面での主な特徴である。

2.4 Redis

Redis[8] は memcached と同様に Key と Value のペアをメモリ上に保持する key-value store である。Redis の大きな特徴はメモリ上のデータをプライマリとしつつ、非同期で定期的にディスクへ書き出すことができる点であり、特定時点での永続性が保証される。データをオンメモリで扱うため、読み出し・書き込み両方共に高速に行えることが利点ではあるが、実メモリに乗りきらないデータ量は扱えないという問題があり、利用できる用途が限定的であった。しかし、新しいバージョンでは、独自の仮想メモリ機構を実装しており、実メモリに乗りきらないデータをディスクへ書き出す仕組みが取り入れられている。

3. ストレージエンジンの性能比較

MyCassandra で使うことのできるストレージエンジンのうち、元の Cassandra が用いる Bigtable 由来のストレージエンジン、MySQL、Redis という 3 種の性能を比較する。性能測定には Yahoo!'s Cloud Serving Benchmark(YCSB)[9] を用いる。

表 2 YCSB ワークロード
Table 2 YCSB workloads

Workload	Read-ratio	Update-ratio	Record Selection	App. Example
Update-Only	0%	100%	Zipfian	Log
Update-Heavy	50%	50%	Zipfian	Session Store
Read-Heavy	95%	5%	Zipfian	Photo tagging
Read-Only	100%	0%	Zipfian	Cache

3.1 YCSB

YCSB は、Yahoo! Research が様々なクラウドストレージを公平に評価することを目的として実装されたオープンソースのベンチマークフレームワークであり、実アプリに近いコアワークロードが用意されている。

YCSB では、読み出し処理と書き込み処理の回数の比率をユーザが指定できる。YCSB がデータストアに対して読み出しと書き込みを実行し、ワークロード全体のスループットと、各処理に要した時間、つまり遅延を集計する。

YCSB が行う処理は次の 3 フェースから成る。

(1) load phase: 初期データをロード
(2) warm phase: ワークロードを実行してデータストアのキャッシュをウォームアップ
(3) transaction phase: ワークロードを実行してスループットと遅延を集計

表 2 に、今回の測定で用いる 4 種類のワークロードを示す。書き込み比率が高い Update-Only と Update-Heavy、読み出し比率が高い Read-Only、Read-Only を用意した。各ワークロードに対応する実アプリの例を右側に示している。各ワークロードにおいて、アクセス対象データの分布として Zipfian 分布を用いる。Zipfian 分布とは、データ鮮度とは関係なく人気によってアクセス頻度が決まるようなアプリのデータアクセス分布を確率としてモデル化したものであり、一部のデータがデータが常にヘッドであり、大部分がテールとなるような分布である。

表 3 に実験パラメータを、**表 4** に実験環境を示す。

3.2 評価と考察

図 6 はクライアント数を調整して 1 秒あたり 5,000 回のクエリを発行した際の読み出しと書き込みの遅延をワークロードごとに示している。Bigtable ストレージエンジンと MySQL の結果を比較すると、書き込み遅延は Bigtable の方が小さく、最大でも MySQL の 41.4%で

表 3 実験パラメータ
Table 3 Experiment parameters

データノード	実マシン× 6
YCSB クライアント	実マシン× 1
ロードレコード	2,400 万件
単一レコード	10 カラム合計 1KB
ウォームアップ命令数	5 万件

表 4 実験環境
Table 4 Machine configurations

OS	2.6.35.6-48.fc14.x86_64
CPU	2.40 GHz Xeon E5620 × 2
Mem	32GB RAM
Disk	1TB SATA HDD × 2
JVM	Java SE 6 Update 21

あり、読み出し遅延は MySQL の方が小さく最大でも Bigtable の 49.4%であるという結果が得られた。

図 7 はクライアント数を 40 として負荷をかけたときのスループットをワークロードごとに示している。書き込みが多いワークロードでは、Bigtable の方が高く MySQL の最大 5.32 倍であり、読み出しが多いワークロードで MySQL の方が高く Bigtable の 2.35 倍という結果が得られた。また、同一ストレージエンジンについて各ワークロードでの結果を比較すると、Bigtable では書き込み比率が高いほどスループットが高く、MySQL では読み出し比率が高いほどスループットが高くなっていることが確認できる。Bigtable では write-only ワークロードのスループットが read-only の 11.7 倍であり、MySQL では read-only ワークロードのスループットが write-only の 5.8 倍となっている。ストレージエンジンごとに、読み出し性能の高いものと書き込み性能の高いものがはっきりと分かれた。

この通り、データストア自体を置き換えずともストレージエンジンを差し替えることで読み出しと書き込みの性能の傾向が異なるデータストアを得ることができた。

Redis は全データをオンメモリで扱うがゆえに、いずれのワークロードでも読み出しと書き込み両方の操作において高速に行うことができる。

4. 関 連 研 究

Anvil[10] は、粒度の細かいコンポーネント dTable を組み合わせて構成されるデータストアである。アプリケーションのデータアクセスパターンに応じたデータストアを構成できる。データストアをモジュラーに構成しようという点が本研究と共通しているが、分散データストアを対象としているわけではない。

Cloudy[11] は、クラウドストレージをモジュラーに構成しようという提案である。ストレージエンジン以外にも、ルーティング処理やロードバランスの方式もコンポーネント化し、選択可能とすることを提案している。性能は報告されていない。

Amazon Dynamo[7] は、Amazon 社がウェブ上で提供するサービス向けに開発した key-value store である。ストレージエンジンとして Berkeley DB や MySQL、メモリ上のバッファを用いることができる。ストレージエンジンを選択する際の指標の例として、格納データのサイズが挙げられている。Dynamo で用いることのできるストレージエンジンでは、読み出し性能と書き込み性能を調整することはできない。

本研究は読み出しと書き込みという対比について分散環境での定量的な評価を行えている。

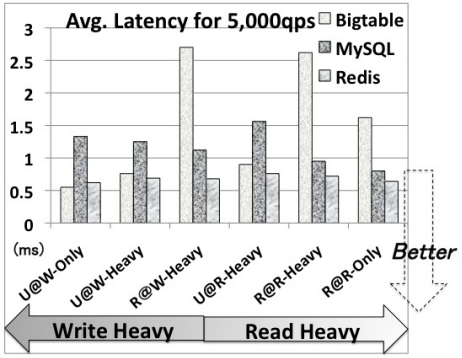

図 6 ワークロードごとの遅延
Fig. 6 Latency for each workload

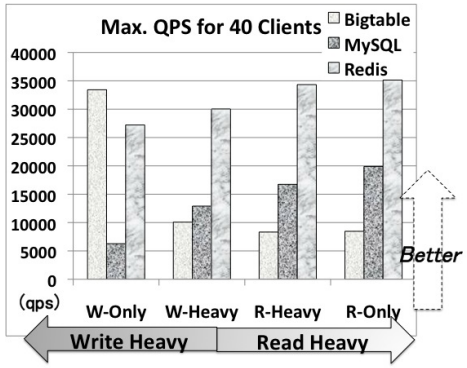

図 7 ワークロードごとのスループット
Fig. 7 Throughput for each workload

5. まとめと今後

本研究では、同一システム内のストレージエンジンを差し替えることで読み出し/書き込み性能を調整できるようなクラウドストレージの提案と実装を行い、ベンチマークにより実際にストレージエンジンごとに大きく性能が異なることを確認した。

5.1 SSD 上での評価

今回の実験環境は HDD であったが、今後は SSD 上でも評価を行う。
SSD は HDD と比較するとランダム I/O の性能が優れているため、各ストレージエンジンが苦手とする処理の性能が向上し、HDD との結果と比べると Bigtable と MySQL の性能差は縮まるものと予測される。

5.2 MyCassandra Cluster(読み出し性能と書き込み性能の両立)

また次のステップとして、ストレージエンジンの異なるノードを組み合わせて用いることで、単一のクラスタで、読み出し性能と書き込み性能を両立させることを狙う。現在、性能

Vol.2011-OS-116 No.9
2011/1/25

評価を進めている。

具体的には、**図 8** の通り、読み書き性能についての性質が異なるノード群に複製を配置する。そして、書き込みは Bigtable と Redis に対して同期的に行い、読み出しには MySQL と Redis に対して同期的に行うというように、それぞれのストレージエンジンが得意とする処理を同期的に行い、得意でない処理についてはリクエストへの応答とは独立して非同期で行うよう、プロキシ側でリクエストの振り分けを行う。非同期の読み出しはクライアントにデータを返すためではなく、複製の間で整合性をとるために行われるので、遅延が大きくても読み出し性能には影響しない。これらを行うためには、各ノードが全ノードのストレージエンジンの種類を知っておく必要があるので、Cassandra のノード生存確認に利用される Gossip Protocol にストレージエンジンに関するメタ情報を加える。

このとき、レプリカ間の一貫性が課題となる。例えば、書き込んだデータをすぐ読み出す場合、非同期で書き込みが行われたノードから読み出しを行うと古いデータが得られてしまう可能性がある。しかし、Cassandra は、直前に書き込まれたデータを読み出せる、ということを Quorum（多数決）で保証できる。例えば、$N = 3$、$R = W = 2$ として Quorum を用いる場合、3 つの複製のうち、2 つへの書き込み、2 つからの読み出しが成功した時点で、クライアントに返答を返す。

MyCassandra には、その他に次の課題がある。

- ネットワーク近接性 (proximity) とストレージエンジンに応じた書き込み先ノード選択の両立
- 負荷分散

ネットワーク近接性、つまりラックやデータセンターを考慮しつつも、ノードごとに読み出し性能と書き込み性能のどちらに優れるのかも考慮して、複製を配置する必要がある。また、ノードごとに読み書きのどちらが得意なのかが異なるため、同期書き込みを処理するノード、同期読み出しを処理するノード、両方を処理するノード、と分かれ、負荷分散に問題が生じかねない。この課題に対しては、単一のサーバに複数のノードを動作させることで対処できる。その際、同一サーバに複数の複製が作られないように配慮する必要がある。

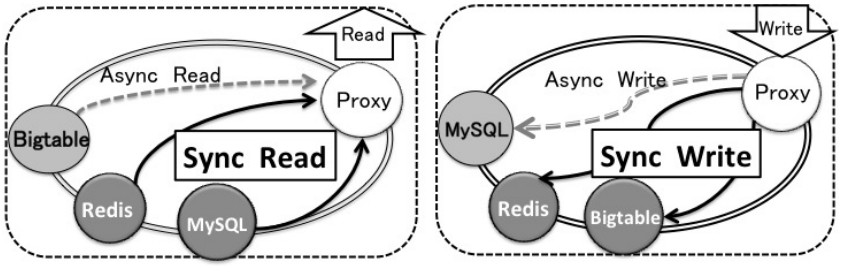

図 8 MyCassandra Cluster
Fig. 8 MyCassandra Cluster

参 考 文 献

1) The Apache Software Foundation: Cassandra, `http://cassandra.apache.org/` (2010).
2) Lakshman, A. and Malik, P.: Cassandra - A Decentralized Structured Storage System, *Proc. LADIS '09* (2009).
3) The Apache Software Foundation: HBase, `http://hadoop.apache.org/hbase/` (2010).
4) 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, *Proc. OSDI '06*, Vol.7, pp.205–218 (2006).
5) Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D. and Yerneni, R.: PNUTS: Yahoo!'s Hosted Data Serving Platform, *Proc. VLDB '08* (2008).
6) Oracle: MySQL, `http://www.mysql.com/`.
7) 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, *Proc. SOSP '07* (2007).
8) Redis: Redis, `http://code.google.com/p/redis/` (2010).
9) Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R. and Sears, R.: Benchmarking Cloud Serving Systems with YCSB, *Proc. SOCC '10* (2010).
10) Mammarella, M., Hovsepian, S. and Kohler, E.: Modular Data Storage with Anvil, *Proc. SOSP '09* (2009).
11) Kossmann, D., Kraska, T., Loesing, S., Merkli, S., Mittal, R. and Pfaffhauser, F.: Cloudy: A Modular Cloud Storage System, *Proc. VLDB '10* (2010).