Bullet:分散キーバリューストアにおける列指向クエリエンジン

西山 達也[1] 内山 寛之[1]

概要：分散キーバリューストアは、単純なデータ構造を持ち、高いスケーラビリティと信頼性を持つことから、Web サービスなどの様々なサービスでの利用が進んでいる。分散 KVS に蓄積されたデータに対して分析を行う際、既存のインタラクティブクエリ基盤では応答性能が低く、分析用途での利用において課題となっている。
本論では、分散 KVS 上のデータに対して SQL クエリを発行可能とし、高速な応答を実現する分散クエリエンジン (Bullet) を提案する。分散 KVS が投入されたデータを列ごとに格納する点に着目し、列指向 DB のクエリエンジンの持つ遅延実体化や軽量圧縮等の高速化技術を取り込みつつ Bullet の実装を行った。
性能測定実験において、Bullet が既存技術に比べて一桁高速であること示した。また、複数の圧縮方式についての測定結果を示し、圧縮なしの場合に比べて高速であることを確認した。

Bullet: Distributed query engine for distributed key value store

Tatsuya Nishiyama[1] Hiroyuki Uchiyama[1]

1. はじめに

近年、様々なサービスにおいて分散キーバリューストア (KVS) が利用されている。分散 KVS は、キーとバリューという単純なデータ構造を持ち、キーによる値の追加、更新、削除、取得という単純な操作に特化することで、高いスケーラビリティを実現し、大量のデータ管理を可能にしている。蓄積された大量のデータを分析し、企業の意思決定やサービスの改善に利用されている。

分散 KVS に対するデータ分析は、MapReduce[1] のような低レイヤの分散処理フレームワークなどが用いられてきた。しかしながら、アドホッククエリによる分析において、何度も低レイヤのプログラム言語でクエリ処理を記述することは分析の効率が悪化する。つまり、クエリの記述容易性は極めて重要であり、SQL によって検索するニーズは高い。このようなニーズに対して、分散 KVS である HBase[2] 上のデータに対し、HiveQL と呼ばれる宣言型言語で検索可能としたクエリ処理エンジンとして、Hive[3] や Impala[4] などが提案されている。

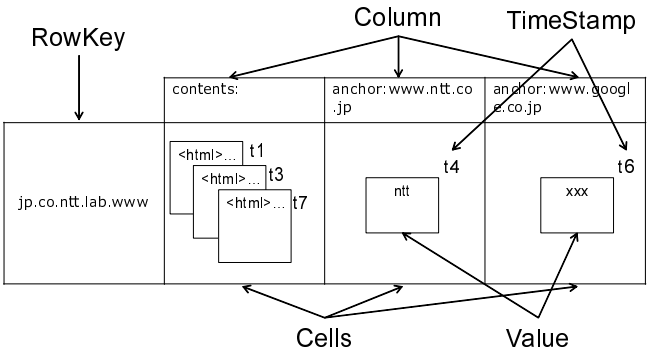
[1] NTT ソフトウェアイノベーションセンタ
NTT Software Inovation Center

これらのプロダクトにより、クエリの記述は容易となった。しかし、以下に述べる理由により、検索応答の高速化に限界がある。HBase に投入されたデータは、列ごとに格納されているが、Hive や Impala のクエリエンジンへのデータロードにおいて、行として実体化される。そのため、列指向データベースで有効であることが示されている遅延実体化 [5] といった最適化の適用が困難である。

本論では、分散 KVS の特徴である列ごとに格納されたデータに対して、列指向 DB で提案されている技術を利用した高速な分散クエリエンジン Bullet を提案する。Bullet は、以下に挙げる特徴を持つ。

- 分散 KVS のストレージエンジンにおいて、クエリエンジンのオペレータを実行することで検索処理のプッシュダウンを実現。
- 列で格納されたデータをできる限り列として扱うことで、遅延実体化を実現。
- 軽量圧縮 (RLE、 Dictionary、 Bitvector) を利用可能とし、圧縮されたデータを効率的に処理するオペレータ群の実現。

上記方針に従ってアーキテクチャを設計し、プロトタイ



Cell := (Rowkey, Column, TimeStamp, Value)

図 1　テーブル構造とデータの格納

プの実装を行った。性能測定の結果、TPC-H No.5 (Scale Factor = 1) において、Hive の 33 倍の高速化を確認した。

本論の構成は、以下のとおりである。2 節にて、Bullet で用いる分散 KVS Billy について述べ、3 節で列指向 DB について述べる。4 節では、Bullet の設計と実装について述べ、5 節において TPC-H ベンチマークを用いて Bullet と Hive の比較実験結果を示す。6 節では関連研究について紹介し、7 節にて本論のまとめと今後の課題について述べる。

2. 分散 KVS Billy

本節では、Bullet で用いる分散 KVS Billy[6] について述べる。Billy とは、Bigtable[7] と同様のアーキテクチャを採用している分散 KVS である。

Billy は、以下に示すようなソート済みマップを提供する。

(Rowkey, Column, TimeStamp) → Value

よって、(Rowkey, Column, TimeStamp) を指定することで Value を特定できる。このように、Billy は、内部的には KVS であるが、ユーザに対しては以下に述べるようなテーブル構造を提供している。テーブルは、ユーザが設定するスキーマ情報に基づいて定義される。テーブル構造の概略を図 1 に示す。Rowkey、Column は、それぞれ行、列を特定する。Rowkey と Column を指定することで、Cells を特定する。Cells は複数の Cell からなっており、(Rowkey, Column, TimeStamp,Value) を Cell と定義する。TimeStamp は、Value のバージョンを表している。

テーブルを構成するデータを複数のサーバによって分散管理するため、テーブルを Area という単位に分割する。ここで、Area は図 2 に示すように、テーブルを水平方向に分割したテーブルの部分集合として定義される。テーブルにデータが追加されていくと、複数の Area に分割されてクラスタ上のサーバに配置される。

図 3 に Billy のアーキテクチャ概要を示す。Billy では、

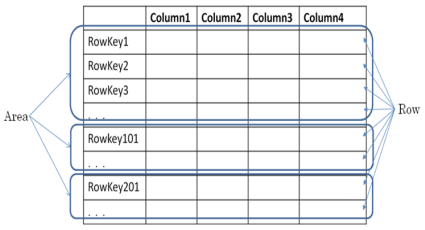


図 2　テーブル構造と Area

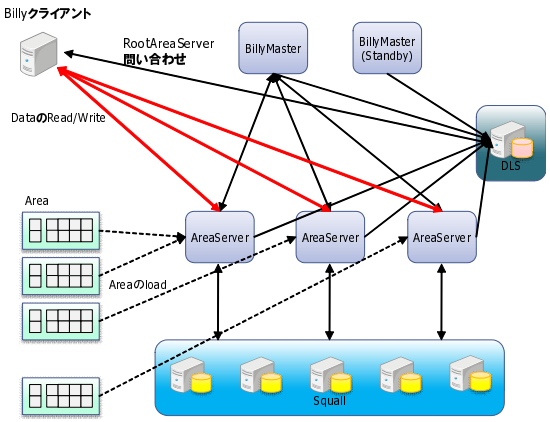


図 3　Billy のアーキテクチャ

外部システムとして、分散ファイルシステム Squall と分散ロックシステム DLS を使用する。Squall[6] と DLS[6] は、それぞれ GFS[8]、Chubby[9] のアーキテクチャを参考に構築されたプロダクトである。

Billy は、MasterServer プロセスと AreaServer プロセスから構成される。MasterServer では、AreaServer の管理や Area の割り当て管理、スキーマ管理などを行う。AreaSerer では、割り当てられた Area に対する更新・参照処理を行う。

図 4 を用いて、Billy の追加・更新・削除における AreaServer の動作について述べる。この図は、ある Area を AreaServer が管理することを示した図である。Bigtable のアーキテクチャを採用する Billy は追記型のシステムであり、更新・削除では既に格納済みの Cell は変更されず、更新や削除を意味するフラグを付与した新たな Cell を追加する。図中 (a) において、コミットログに書き込みを行い、(b) において Cell をメモリ上のバッファ領域 (セルバッファと呼ぶ) に格納する。(c) セルバッファに格納されたデータサイズが、一定の閾値を超えると Cell を Rowkey でソートし、ファイルとして Squall 上に書き出し永続化を行う (MinorCompaction[7] と呼ぶ)。このファイルをセルファイルと呼び、一度書き出されたセルファイルは部分的な更新はされない。MinorCompaction で作成されたセルファイルには、追加・更新・削除といったオペレーションによって追記された全てのセルが含まれている。

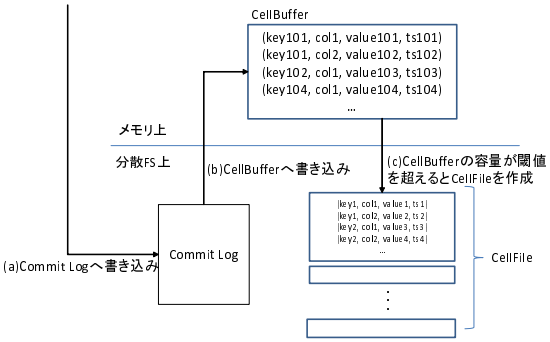


図 4　追加・更新・削除処理の概要

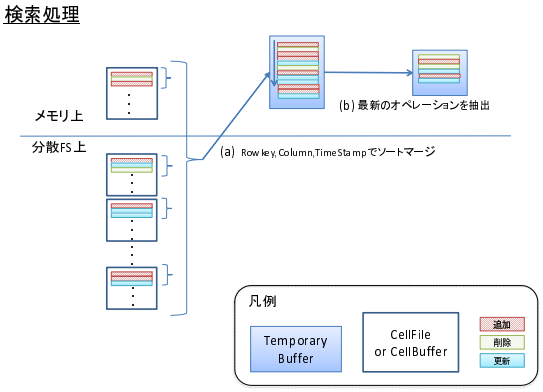


図 5　検索処理の概要

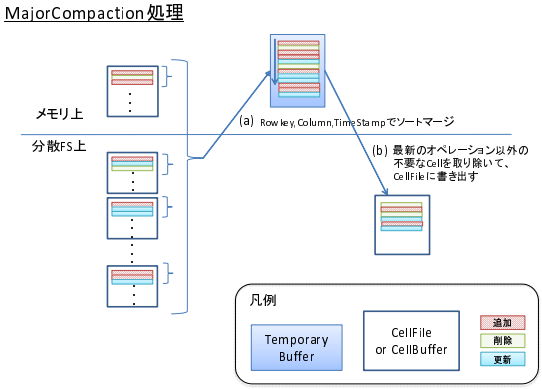


図 6　MajorCompaction 処理の概要

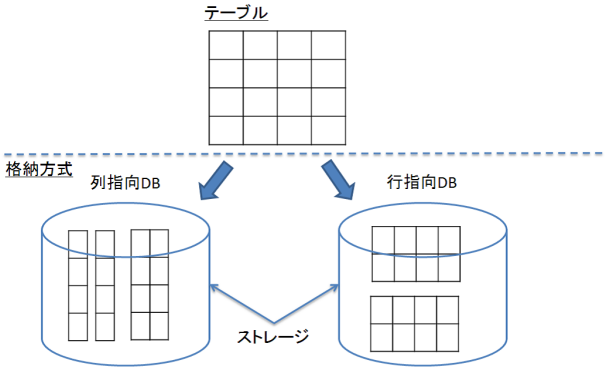


図 7　列指向 DB と行指向 DB の格納構造

ある Area に対するオペレーションを全順序とするため、一意な単調増加の値 (リビジョン) を Cell に付加している。リビジョンの必要性を以下に述べる。追記型であるため、(Rowkey, Column, TimeStamp) に対して、複数の Cell が存在する。そのため、検索時には最新のオペレーションを検索結果に適用するために、リビジョンは利用される。

図 5 を用いて、検索における AreaServer の動作について述べる。1 つの Area に対して、Rowkey でソートされたセルファイルが複数存在する。ソートされているため、(a) 各セルファイルとセルバッファを先頭から読み込み (Rowkey, Column, TimeStamp) でソートマージ処理を行う。(b) マージした Cell の集合に対して、リビジョンをもとに更新・削除により不要となった Cell を取り除くことで、同一の Rowkey において最新の Cell のみを返却する。

追記型であることから、データが追加・更新・削除されるにつれて、セルファイルの数が増加する。その結果、セルバッファとセルファイルのマージ処理に関するオーバヘッドが大きくなり、検索時に性能低下を引き起こす。そこで、Area のサイズが一定の閾値を超えると、図 6 に表すように (a) セルバッファとセルファイルをすべてマージし、(b) 不要な Cell をすべて取り除いた新たな一つのセルファイルを作る処理が実行される (MajorCompaction[7] と呼ぶ)。

3.　列指向 DB

本節では、Bullet のクエリエンジンにおいて利用される、遅延実体化や軽量圧縮等の技術を導入した列指向 DB について述べる。

列指向 DB は、一般的な RDB(行指向 DB) がとるデータを行ごとに格納する構造と異なり、列ごとに格納することで OLAP(On Line AnalyticalProcessing)[10] 向けに最適化されたシステムとして提案された。図 7 に格納構造の差異を示す。このように列方向に格納することで、数多くの列を持つテーブルから、比較的少ない列を抽出する場合、必要な列に対してのみ IO が発生することから IO を減らすことができる。また、列には類似性の高い値が格納されるため圧縮効率が高いという特徴を持つ。一方で、結果出力時には列を結合してテーブルに戻す必要があるため列を結合するオーバヘッドがかかる。

列指向 DB の代表的なものとして C-store[11]、MonetDB/X100[12] がある。

列指向 DB は、データを列ごとに格納することによって、OLAP におけるクエリ処理の高速化において以下のような利点がある。

(1) 多数の列からなるテーブルに対して、必要最小限の列だけを読み込むことが可能であり、ディスク IO が行

指向 DB に比べて小さくなる。
(2) 列方向に圧縮することで圧縮効率が高い。また、展開せずに直接処理が行える圧縮方式が提案されている。
(3) レコードの遅延再構築 (Late Materialization) と呼ばれる、不要なデータの読み込みをスキップし、ディスク IO を削減する手法が提案されている。
(4) 列指向 DB の基本形は列ごとに値を格納するものであるが、頻繁に一緒に出力される列に対して複数の列をまとめて格納することで処理を高速化する最適化手法が提案されている。

以降で述べる、Bullet では上記のうち 1、2、3 の特徴をアーキテクチャに取り込んだ設計となっている。

4. Bullet

本節では、本論で提案する Bullet のアーキテクチャについて説明し、Bullet に導入した Timelimited Consistency の概念、及び計算可能な圧縮方式について述べる。最後に、実装した Bullet が提供している機能の概要について述べる。

4.1 アーキテクチャ

Bullet のアーキテクチャを図 8 に示す。Bullet は一つの GlobalProcess と複数の AreaServer から構成されている。GlobalProcess はクライアントからの要求の受付と処理の分散・集約を担当する。AreaServer は Billy の AreaServer をベースにしており、自身の管理する Area 内データの処理を担当する。

GlobalProcess、AreaServer はともに後述する Parser、Rewriter、 Planner、Executor からなるクエリエンジンを内部に持ち、クライアントから要求のあったクエリ処理を分散実行するアーキテクチャ構成となっている。

本アーキテクチャの詳細についてクエリ処理の実行過程の記述も含めて述べる。クライアントから発行された SQL クエリは GlobalProcess に渡される。GlobalProcess は Parser によって SQL クエリの文を解釈し、クエリエンジンの内部表現であるパース木を生成し、Rewriter に渡す。Rewriter では、SELECT 文に含まれる「*」の展開や型情報の付与といったパース木の書き換えを行う。書き換えられたパース木は GlobalProcess に渡されると同時に AreaServer にも送信される。

GlobalProcess の Planner では、AreaServer の処理結果を集約するためのプランを生成する。プランはパース木を解析し、クエリを実行するために必要な一連の処理の流れを木構造で表したもので、プラン木と呼ばれる。Planner には統計情報によりプランの実行コストを計算しながら、最適なプランを生成するコストベースの Planner と、予め決められた規則によりプランを生成するルールベースの Planner がある。Bullet ではルールベースの Planner を採用し、フィルタ処理のプッシュダウンや遅延実体化を行うプランを生成している。GlobalProcess の Planner は AreaServer からの局所的な処理結果を受け取り集約処理を行うためのプランが生成される。GlobalProcess の Executor では、Planner から渡されたプラン木の処理を実行するためのオペレータ木に変換する。GlobalProcess の Exectuor では AreaServer からの処理結果を受け取り、受け取った処理結果に対してオペレータを適用することで実行する。

次に、AreaSever 側の処理について述べる。GlobalProcess の Rewriter 後のパース木を AreaServer 側で受け取り、パース木を Planner に渡す。そこで、AreaServer 側の実行プランが生成され、Executor にてオペレータ木に変換される。オペレータの実行では AreaServer が管理している Area 内のデータを読み込みオペレータの処理を行う。オペレータの最終的な処理結果は GlobalProcess に返却される。その後、GlobalProcess にて各 AreaServer の処理結果を集約し、ユーザに結果を返却することで処理を完了する。

このように、GlobalProcess 側では、AreaServer で処理した局所的な結果をもとに集計、ソートと言った処理を担当する。また、テーブルの結合には AreaServer に対して局所的な結合を実行させるために必要なデータの分配やクエリの書き換えを行う。

また、AreaServer 側では、自身の持つデータに対する局所的なフィルタ、集計、ソートを担当し、テーブルの結合時には GlobalProcess から受け取ったクエリとデータをもとに局所的な結合を行う役割を担当する。

4.2 リレーショナルテーブルと KVS のマッピング

2 節で述べたように、Billy では Cells は複数の Cell を含む。リレーショナルモデルでは、行と列が確定した場合には、値は一意に決定する。Billy 上のデータに、SQL で検索を行うために、このデータモデルに関する差異を以下のように吸収した。

- TimeStamp は、ユーザからは設定不可とし、TimeStamp 値は Bullet において固定値を設定している。これにより、Cells の中には高々 1 つの Cell しか含まれないことを保証する。
- Rowkey は行の復元のために利用するため、行に対して一意な値として追加されることを制約としている。

以上により、KVS としてのセマンティクスを残しつつ、SQL クエリを発行可能としている。

4.3 列ごとの格納

Billy では、Bigtable[7] と同様に、ColumnFamily-Group(CFG) を指定できる。本機能を用いることで、Bullet においてテーブル作成時に指定される列ごとにセルファイルを作成することが可能となる。

まず、セルファイルの構造について述べる。セルファイ

ルはヘッダ情報と複数のブロックから構成されており、ブロックは列方向のいくつかの値をまとめたものである。すなわち、列は水平方向にブロック単位で分割されており、いくつかのブロックを含むセルファイルとして Squall に書き込んでいる。ヘッダ情報には、ブロックの種類やセルファイルのサイズ、リビジョンといった情報が入っている。また、後述する圧縮はブロック単位で行っており、それぞれの圧縮フォーマットでブロックを表現し、セルファイルに書き込んでいる。

SQL での検索時にはセルファイルを読み込むが、セルファイルとセルバッファをマージするためのオーバヘッドがかかり、検索処理においてボトルネックとなる。特に圧縮されたブロックの場合、読み込んだセルファイルに含まれる圧縮されたブロックを一度展開し、セルバッファとマージするため、大きなオーバヘッドとなる。

マージ処理をなくす手段として前述した MajorCompaction がある。MajorCompaction が実行されるとすべてのセルファイルとセルバッファをマージして、一つのセルファイルにするため MajorCompaction 直後であればマージ処理が必要なく、高速な検索が可能である。しかし、MajorCompaction はすべてのセルファイルを読み込み、書き込むため、非常に重い処理である。そのため更新のたびに頻繁に実行すると更新にかかる時間が増大し、更新性能が低下する。すなわち、検索性能と更新性能は MajorCompaction の実行頻度において、下記の表で示すようなトレードオフの関係にあると言える。

表 1 MajorCompaction(MC) の実行頻度による検索・更新の性能変化

MC 頻度 / 性能	低	高
検索性能	低下	向上
更新性能	向上	低下

表の一列目は MajorCompaction の頻度を下げるとマージが必要となる一方で更新により発生する MajorCompaction が減少するため更新性能が向上することをあらわしている。表の二列目は MajorCompaction の頻度を上げることにより、マージのオーバヘッドがなくなり検索性能が向上するが、一方更新のたびに発生する MajorCompaction により更新性能が低下することを表す。

Bullet では更新と検索の両者の性能を両立するため、MajorCompaction によって生成されたセルファイルのみを検索の対象とすることで高速化を行う。しかし、この手法では MajorCompaction 後に追加された Cell がマージされないため、MajorCompaction 以降のオペレーションの結果が検索結果に反映されない。その結果、検索結果において以下で述べるように一貫性が満たされなくなる。

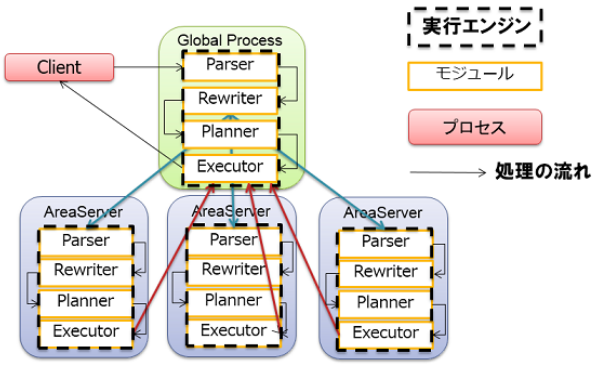


図 8 Bullet のアーキテクチャ

本論で述べる検索結果の一貫性は次の通りである。ユーザが検索した結果が時刻 t における追加・更新・削除といったオペレーションが反映されているとする。このとき、検索結果のいずれのデータも t までに実行されたオペレーションが反映され、かつ t 以降に実行されたオペレーションが反映されていない状態を時刻 t に関して、一貫性が満たされていると定義する。

MajorCompaction で生成されたセルファイルを検索対象とする手法において一貫性が満たされない要因は、MajorCompaction は Area ごとに実行されるため、Area によってどの時刻までのオペレーションが反映されているかが異なるためである。例えば、ある Area では 1 時間前に MajorCompaction が実行され、その時刻までのオペレーションを含んだセルファイルが生成されているとする。この時、この Area のデータは 1 時間前までのオペレーションの結果が反映されている。一方で、他の Area で 1 2 時間前に MajorCompaction が実行されていた場合、この Area では 1 2 時間前までのオペレーションしか反映されていないため両者の Area にまたがった検索では一貫性が取れない結果が出力される。このように一貫性が満たされない場合、列の値を集計するような処理においてユーザが期待する結果と異なる結果が返され、不都合が生じる。

OLAP のユースケースの場合、常に最新の情報で分析を行う必要性は低いと考えられる。例えば、OLAP 向けのキューブを用いたシステムでは、分析する情報を一度ロードするため、最新ではない。また、分析処理においては売上集計など、ある特定の時刻までのデータを使って集計を行えば目的が達成されることが多い。このような場合、その時刻までのデータがあれば良く、最新のデータである必要がない。そこで、最新のデータの取得では高速性が犠牲になる一方で、特定の時間までのデータに対しては、一貫性を満たしつつ高速な検索を可能にする概念として、次節で述べる Timelimited Consistency を提案する。

4.4 Timelimited Consistency

ここでは、Bullet が導入する Timelimited Consistency

の概念について述べる。Timelimited Consistency とは、時刻に t に関して一貫性を保証した高速な検索処理を可能にするというものである。ユーザは指定する時刻を変動させることで、最新のデータの取得を優先するか、検索性能を優先するかについて選択が可能となる。

Timelimited Consistency の実現方法について述べる。検索処理において、高速化を行うために MajorCompaction 済みのセルファイルのみを対象に読み込むことでセルバッファ、及び複数のセルファイルとのマージ処理をなくしている。この時、前述した通り従来の分散 KVS のアーキテクチャでは Area ごとに MajorCompaction のタイミングが異なってくるため一貫性が満たされないという課題があった。そこで、現在の Bullet ではすべての Area に対して強制的に MajorCompaction を実行する Vacuum コマンドを実装し、Vacuum 実行時の時刻で一貫性が保証された高速な検索が可能である。

以下では、今後の取り組みについて述べる。ユーザが設定した任意の時間間隔 I の間に各エリアで MajorCompaction が実行する仕組みを取り、検索時に現時刻から t 時刻前にあたる t' で Cell をフィルタすることで t' までの一貫性を満たしつつ高速な検索を可能にする方式を取る。図 9 は 3 つの Area における上記方式の概念を示す。MajorCompaction の実行タイミングが Area によって異なるため、図中の枠で表すようにオペレーションが反映される区間が Area ごとに異なる。そこで、全 Area においてオペレーションが必ず反映されている t' でフィルタすることで、一貫性を満たしている。

実装においては時刻について TimeStamp を利用し、各 AreaServer は同期がとれた時刻を TimeStamp として Cell に付与する。検索時のセルファイルの読み込みにおいて Cell の TimeStamp を t' 以下であるものを出力するようにフィルタする。この区間をユーザが調整することで任意の時刻における一貫性を満たす高速な検索を提供する。最新のデータを取得する場合、I の間隔を短くすることで可能であるが、その場合は MajorComapction が頻発するため性能が低下する。I をユーザが許容できる範囲内において長く取ることで一貫性と検索処理の高速性を提供するというのが TimelimitedConsistency の利点である。

4.5 計算可能な圧縮フォーマット

ここでは、Bullet の実装する圧縮方式について述べる。C-store のような列指向データベースでは、圧縮された状態のままオペレータによる処理が可能な軽量な圧縮手法が導入されている [5]。Bullet ではその中でも基本的な軽量圧縮の方式である、RLE、Dictionary、BitVector について実装を行った。

Bullet の軽量圧縮の実装について述べる。Bullet では前述した通り、ブロックごとに圧縮が行われており、ブロック

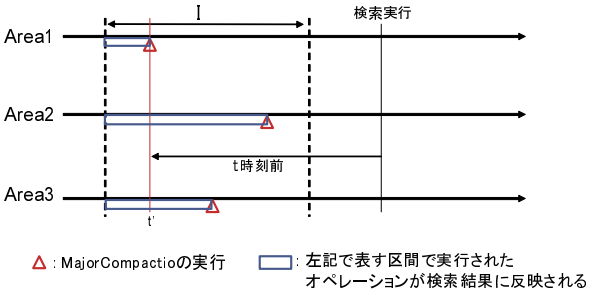


図 9　Timelimited Consistency の概念図

の長さは圧縮のフォーマットごとに異なる。例えば、RLE では連続する値が一つのブロックに含まれるようにしており、列において値の連続が途切れた箇所でブロックを区切っている。BitVector や Dictionary では固定長の長さを用いている。圧縮されたブロックが生成されるタイミングは MajorCompaction 時であり、追加された値をセルバッファ中では従来通り Cell の形で持っている。

評価結果については後述するが、現在の実装では大幅な性能改善は見られなかった。その要因として、セルファイルのヘッダ部分でリビジョンが含まれており、値ごとに付けられたリビジョンを保持するため、ブロック部分を圧縮することができてもリビジョンが含まれたヘッダ部分が大きいためセルファイルのサイズが小さくなっていない。更にセルファイルの読み込みにおけるリビジョンのエンコーディング処理にも時間がかかることで性能低下を起こしている。

リビジョンはセルバッファとのマージ処理において必要であったが、Timelimited Consistency の導入によりマージ処理が実行されないため、リビジョンをすべて持っている必要がなくなっている。そのため、圧縮ブロックのフォーマットからリビジョンを取り除くことで性能改善を図る。

4.6 機能概要

Bullet は以下の機能をサポートする。

- データ型として、文字列、Int、Double、Date を提供
- SELECT、FROM、WHERE による列の抽出とフィルタ処理が可能
- 表の内部結合と直積が可能
- 集約関数、group by、 having による処理が可能
- INSERT、 DELETE、 CREATE TABLE、 DROP TABLE によるテーブル操作が可能
- 列の圧縮機能 (RLE、 Dictionary、 Bitvector) を提供
- 数値演算処理が可能

上記の通り、基本的な SQL 文をサポートする段階まで実装を完了している。今後、OLAP 関数を始めとした一般的な RDBMS が提供する関数について実際のユースケー

スにおいて重要度が高いものから実装を行っていく予定である。

5. 実験結果

本節では、設計したアーキテクチャの評価を目的として、実装した Bullet の性能測定の結果を示す。Bullet は C++で実装されており、コンパイラーには gcc 4.8 を使っている。

測定には CPU として Intel Xeon 2.83GHz 4 コアプロセッサ x2、メモリ 48G バイト、CentOS6.4 のマシンを 18 台使用し、1 台をマスターの役割をするサーバとして利用し、17 台をスレーブの役割をするサーバとして利用している。ネットワーク構成は 17 台のマシンが 1Gbps のスイッチによって接続されている。

測定において、各クエリの実行前にカーネルキャッシュをクリアするコマンドを実行し、システムを再起動することでシステムのキャッシュもクリアしている。

5.1 Hive との比較

大規模データのクエリ処理において、広く利用されている Hive との比較を行った。ここでは、Hive は HDFS 上の CSV ファイルに対してクエリの実行を行う。

ベンチマークセットには、対象とするユースケースである分析分野での利用を想定し、TPC-H[13] のデータとクエリを使用した。

TPC-H のクエリ 1、3、5、10 に対して scale factor を 1、10、100 と変えて両者を測定した結果を図 10、図 11、図 12 に示す。図の縦軸はクエリの応答時間を表しており、横軸に実行した TPC-H のクエリ番号を Bullet と Hive のそれぞれについて並べている。

測定結果について sf=1,10 では、Hive と比較して Bullet は 1 桁高速な結果となっているが、sf=100 ではもっとも早いクエリでも 2.5 倍程度となっており、応答速度の差が小さくなっている。この要因として、Bullet の AreaServer の実行エンジンは現時点で並列化されておらず、各サーバのマシンリソースを完全に使い切っていないことが結果として現れていると考えている。

5.2 圧縮フォーマットの比較

Bullet で実装した軽量圧縮の効果について測定結果を示し、評価を行う。

TPC-H の lineitem テーブルに対して shipinstruct をソートの第一キー、shipmode を第二キーとしてテーブルをソートし、shipinstruct、shipmode の 2 つの列を対象に RLE、Dictionary、BitVector で圧縮したテーブルを用意した。

これらのテーブルに対して以下のクエリを実行した応答時間の測定結果を図 13 に示す。図 13 の縦軸はクエリの応答時間を表しており、横軸は各圧縮を表している。横軸のうち Vector となっているものは無圧縮のものを表している。

```
SELECT shipinstruct FROM lineitem WHERE shipmode = 'AIR'
```

テーブルを shipinstruct、shipmode でソートしている理由は、データの生成に TPC-H のジェネレータを使用しているため完全にランダムな値が生成されてしまうためである。ランダムな値では RLE の効果が見られないためため、ソートすることで RLE にとって最も理想的な状態の測定結果を出している。

測定結果を見ると RLE、Dictionary、BitVector の圧縮によりそれぞれおよそ 2 倍程度の高速化といった結果となっており、想定していたほどの高速化がされなかった。要因として、前述したリビジョンが圧縮フォーマットに含まれているからであると考えている。実際に圧縮によるデータサイズの変化を調査したところ、最も圧縮効果の見られた RLE でもおよそ半分程度のサイズにしかなっておらず、その圧縮されたデータフォーマットの九割をリビジョンが占めていることを確認している。そのため、リビジョンをフォーマットから取り除くことで大幅な性能が以前が見込めると考える。

6. 関連研究

大規模にスケールする分散クエリエンジンとして、Dremel[14] がある。Dremel は、SQL を書き換えながらノードに分配し、分散実行することで大規模にスケールするアーキテクチャとなっており、半構造データを扱えることも特徴として挙げられる。Dremel は Bullet が対象とする分散 KVS 上のデータに対するクエリ処理を想定しない。

Dremel 登場以降、多くの分散クエリエンジンが開発され、Drill[15] や Impala、Phoenix[16]、Presto[17] といったシステムがある。

Drill は Dremel と同様のアーキテクチャを取るクエリエンジンである。一方で、SQL だけでなく他の宣言的な記述ができるインターフェースの提供や多様なストレージへの対応など高機能なクエリエンジンの実現を目指している。しかし、未だ開発途中ということもあり、実行できるクエリが少なく、実用的な段階には至っていない。

Imapla、Presto については、着実に機能を増やしていっているものの現状これらのシステムの利用は HDFS 上のデータに対する処理が主であり、HBase 等の分散キーバリューストア上のデータに関してはパフォーマンスが大幅に低下する。

Phoenix は、HBase にクエリエンジンを搭載し、HBase 上に格納したデータに対して SQL による検索を行えるものである。思想として Phoenix は Bullet と近いものであるが、HBase の標準インターフェースを利用してデータを読み込んでおり、この点において分散キーバリューストア内部にエンジンを組み込む Bullet と違いがある。Bullet で

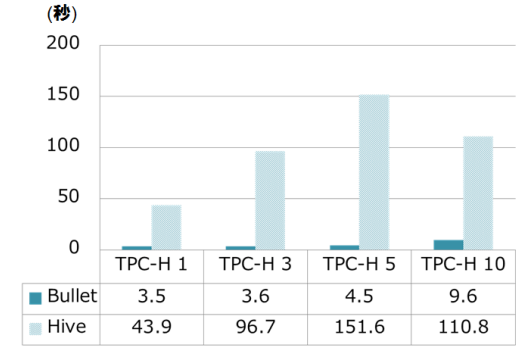


図 10　TPC-H による測定結果 (sf=1)

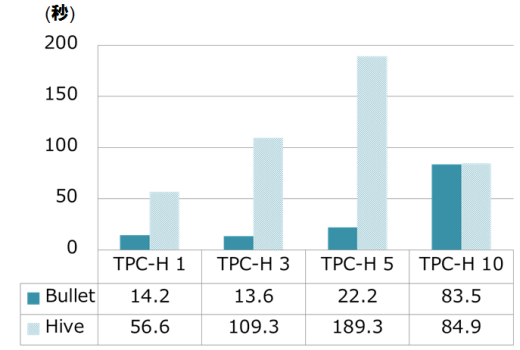


図 11　TPC-H による測定結果 (sf=10)

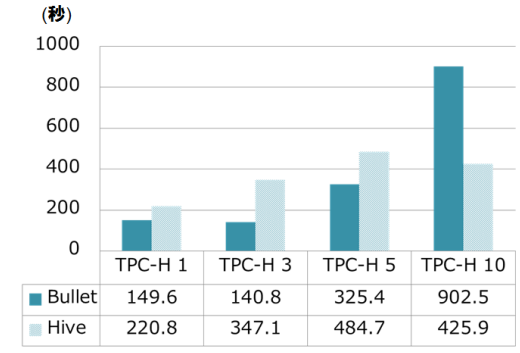


図 12　TPC-H による測定結果 (sf=100)

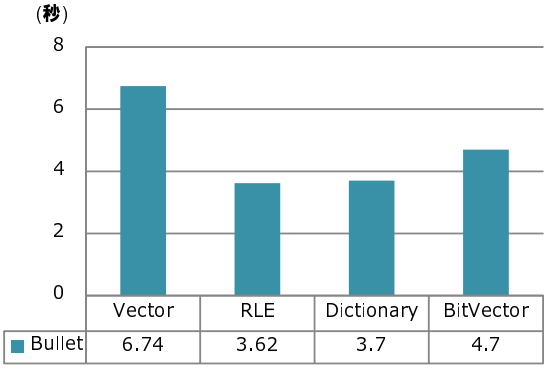


図 13　圧縮方式による性能変化

は内部に実行エンジンを持つことで実行エンジンが処理するブロック構造をファイルに書き出し、直接読み込むことで実行効率を上げている。

7. まとめと今後の課題

本論では、分散 KVS 上のデータに対するクエリによる検索を高速化するアーキテクチャを提案し、その実装である Bullet について述べた。Bullet は分散 KVS として利用可能であり、かつ高速なクエリ処理の実行が可能なものを目指している。しかし、4.2 節で述べたように RDBMS とのデータモデルの差異を吸収するため、セマンティクスにおいて差異がある。また、性能測定においては sf=1 では Hive と比較して高速な結果が出ているが、sf を上げるに連れて性能差が小さくなっている。以降では、この両者の課題について述べる。

まず、Rowkey と TimeStamp に関するセマンティクスの差異について述べる。Rowkey に関するセマンティクスの差異については、Bullet では Rowkey を行の復元に利用しているため、行に対して一意な値を設定するという制約は必要である。この制約について、Rowkey とは別に行の復元に使う値を内部に持つことでセマンティクスを合わせることが可能であるが、次に述べる理由から Bullet では、この制約を残す方針である。Rowkey を行の復元に利用するキーとすることで、Billy のテーブルが内部的に Rowkey でソートされ、Rowkey による値の取得が高速であるという特徴を活かした高速化が可能である。具体的には、テーブル同士の結合やソート処理において有効である。そのため、Rowkey を上手く設定することで上記で述べた高速化が見込めることから、Rowkey をチューニングの手段としてユーザに提供することを狙っている。Rowkey は現在、KVS のインタフェースから設定できるものの、クエリによる検索では列としてユーザに公開していない。今後は、列として Rowkey を公開し、上記で述べた高速化を可能にする。

TimeStamp については、RDBMS のデータモデルにない概念であり、クエリによる検索時の TimeStamp の扱いについて規定する必要がある。具体的には、検索時にどの TimeStamp の値を出力するかを設定値や SQL の拡張といった形で指定する仕組みを提供する必要がある。また、Timelimited Consistency の実装において TimeStamp を利用するため、TimeStamp の代替として Cell の追加時刻を付与する仕組みが必要である。この両者に対応することで TimeStamp に関するセマンティクスの差異をなくすことが可能である。

最後に、sf が大きくなるに連れて性能がスケールしていない課題について述べる。この課題の要因として、以下が挙げられる。

(1) AreaServer のクエリエンジンは処理が並列化されておらず、マシンリソースを使い切っていない。Hive では処理は並列化されているため、スケールするようになっている。

(2) セルファイルに含まれるリビジョンのため、実際のデータサイズよりも内部的にはサイズが大きくなっているため、読み込みの IO が大きくなっている。また、圧縮においてはリビジョン部分が圧縮されないため、圧縮による効果が小さくなっている。

スケール性能の改善については、まずこの両者の課題に取り組むことから行っていく予定である。

参考文献

[1] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 2008.

[2] Apache HBase. `http://hbase.apache.org/`.

[3] Apache Hive. `http://hive.apache.org/`.

[4] Cloudera. Impala. `http://www.cloudera.com/content/support/en/documentation/cloudera-impala/cloudera-impala-documentation-v1-latest.html`.

[5] D. J. Abadi. Query Execution in Column-Oriented Database Systems. MIT PhD Dissertation, 2008. PhD Thesis.

[6] 鷲坂, 中村, 高倉, 吉田, 冨田. 大量データ分析のための大規模分散処理基盤の開発. NTT 技術ジャーナル, pp. 22–25, 2010.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pp. 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pp. 29–43, New York, NY, USA, 2003. ACM.

[9] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[10] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. 1997.

[11] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pp. 553–564. VLDB Endowment, 2005.

[12] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pp. 225–237, 2005.

[13] TPC-H. `http://www.tpc.org/tpch/`.

[14] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pp. 330–339, 2010.

[15] Apache Drill. `http://incubator.apache.org/drill/`.

[16] Phoenix. https://github.com/forcedotcom/phoenix.

[17] Presto. `http://prestodb.io/`.