HeteroTSDB: 異種混合キーバリューストアを用いた自動階層化のための時系列データベースアーキテクチャ

坪内 佑樹[†1,a)]　脇坂 朝人[†1]　濱田 健[†1]　松木 雅幸[†1]　阿部 博[†2,†3,†4]　松本 亮介[†5,†6]

概要：システム管理者からの需要の増加に伴い，システムの状態を時系列データとして収集するためのモニタリングシステムがサービスとして提供されるようになってきている．モニタリングサービスの時系列データベースには，高解像度な時系列データ，時系列データの長期間保存，高い書き込みスケーラビリティと高可用性が求められている．一方で，これまでサービスの機能追加のためにデータ構造の拡張性を考慮した時系列データベースは提案されていない．本研究では，インメモリのキーバリューストア (KVS) とオンディスクの KVS を組み合わせた異種混合 KVS により自動階層化する時系列データベースアーキテクチャを提案する．提案手法では，メモリとディスクのデータ構造を異なる KVS として分離することにより，書き込み効率とデータ保存効率を保ちつつ，用途ごとに最適化されたデータ構造へ書き込みを複製するといった拡張が容易となる．

HeteroTSDB: A Time Series Database Architecture for Automatically Tiering on Heterogeneous Key-Value Stores

Yuuki Tsubouchi[†1,a)]　Asato Wakisaka[†1]　Ken Hamada[†1]　Masayuki Matsuki[†1]
Hiroshi Abe[†2,†3,†4]　Ryosuke Matsumoto[†5,†6]

Abstract: A monitoring system has been provided as a monitoring service as the demand from system administrators increases. Providing a monitoring service requires high resolution time series, long-term retention, high write scalability, and high availability to a time series database (TSDB). On the other hand, TSDBs in consideration of the extensibility of the data structures for function addtions of a monitoring service has not been proposed so far. In this paper, we introduce a TSDB architecture for automatically tiering on heterogeneous Key-Value Stores (KVS) that combines with an in-memory KVS and an on-disk KVS. Our proposed architecture enables the TSDBs extensibility to duplicate the writes to new data structures optimized each use case without reducing writing efficiency and the data storage efficiency by unbundling the data structure on memory and disk as different KVSs.

1. はじめに

インターネットが普及し，当たり前に利用できるようになるにつれて，インターネットサービスを支える計算機システムに対するサービス利用者からの可用性と応答速度への要求が高まっている．したがって，システムに障害が発生したときに，システム管理者がいち早く問題を特定するためには，システムの状態を常に計測すること（以降，モニタリングとする）が必要である．しかし，モニタリングのためには，モニタリングを実現するためのシステム（以降，モニタリングシステムとする）が別途必要となり，システムの構築のための作業や継続的なハードウェアまたはソフトウェアの更新作業といったシステム管理者への負担が発生する．そこで，これらの負担を軽減するために提供されているサービス [1]（以降，モニタリングサービスとする）をシステム管理者が利用することがある．

†1 株式会社はてな
†2 株式会社レピダム
†3 ココン株式会社
†4 北陸先端科学技術大学院大学
†5 GMO ペパボ株式会社
†6 さくらインターネット株式会社
a) y_uuki@hatena.ne.jp

モニタリングシステムでは，対象のシステムからメモリ使用量などのシステムの状態を計測するための指標（以降，メトリックとする）を定期的に収集し，時系列データとしてデータベース (以降，時系列データベース [2] とする) に保存したのちにグラフとして表示する機能を提供することがある．グラフを利用し，発生中の障害を分析するためには，短時間での状態の変化を見逃さないように，メトリックを高解像度で時系列データベースに記録しておくことが必要となる．また，障害復旧後の原因分析や，未来の増強計画のための過去の負荷状況の分析のために，過去のデータを長期間遡れるようにしておくことが求められる．しかし，高解像度のメトリックをストレージに書き込むと，ディスクへの書き込み回数が増加し，高解像度のメトリックを長期間保存すると，ストレージの使用領域が増加するため，書き込み処理とデータ保存を効率化する必要がある．加えて，モニタリングサービスでは，利用者数の増加または利用者が管理するシステムの規模の拡大にともない，単位時間あたりに収集するメトリック数が増加するため，時系列データベースに高い書き込みスケーラビリティが求められる．さらに，モニタリングサービスが障害のために利用できない状態が続くと，その間利用者は自身が管理するシステムの状態を知ることができなくなるため，時系列データベースには高可用性が必要となる．

書き込みスケーラビリティと高可用性を実現するには，複数のサーバを用いて冗長化しつつ，スケールアウトさせる必要がある．先行手法である OpenTSDB[3] は，データの基本要素がキーと値のペア（キーバリューペア）であり，要素同士が互いに依存せず要素単位でデータを分散して配置しやすいために，書き込み処理をスケールアウトさせやすいデータベース管理システム (DBMS) であるキーバリューストア (KVS)[4] を利用している．先行手法は，単一の DBMS として構成されるために，新たなデータ構造を追加するなどの変更を加えるためには，当該 DBMS に対して変更を加えることになる．しかし，一般に，密結合したソフトウェアへの変更よりも，疎結合なソフトウェアへの変更の方が変更時の影響箇所が少なく変更が容易とされているため，各種機能が密結合した DBMS は変更しづらいという課題がある．モニタリングサービスの成長のためには，長期間のメトリックの分析などの単にグラフを表示する以外の機能を追加する必要があり，将来のデータ構造の追加などの変化に対応するための時系列データベースの拡張性が求められる．ここでの時系列データベースの拡張性とは，例えば，メモリとディスクそれぞれに対する書き込み処理の内容を変更し，用途ごとに最適化されたデータ構造を別々に用意し，それぞれのデータ構造に対して同一のデータを書き込むことを指す．我々の知る限りにおいて，データ構造の拡張性を考慮した時系列データベースは提案されていない．

本研究では，将来のデータ構造の拡張性を確保しつつ，書き込み処理効率とデータ保存効率を高め，書き込みスケーラビリティと高可用性を達成する時系列データベースアーキテクチャの実現を目的とする．まず，データ構造を拡張するために，DBMS 自体を変更するのではなく，拡張内容の特性に合わせた異なる DBMS を追加する疎結合なアーキテクチャをとりつつ，内部に複数の DBMS を利用していたとしても，時系列データベースの利用者からは，書き込みと読み込みのためのインターフェイスをそれぞれ 1 つに見えるように統合する．次に，OpenTSDB と同様に，書き込みスケールアウト性と高可用性のために，DBMS として KVS を利用する．さらに，ディスクへの書き込み回数を減らし，書き込み処理効率を高めるためには，ディスク上にデータ配置する KVS（以降，オンディスク KVS とする）ではなく，メモリ上にデータをすべて持つ KVS[5]（以降，インメモリ KVS とする）に着目する．しかし，時系列データベースでは，時間が経過するほど新しいデータが追加で書き込まれるため，短期保存と比較し，長期保存には大きなデータ保存領域が必要となるという前提があるため，ディスクと比較し，メモリは容量単価が大きいため長期保存するには向いていない．そこで，書き込み処理効率を高める役割をインメモリ KVS が担い，データ保存効率を高める役割をオンディスク KVS が担う，異種混合 KVS による疎結合な時系列データベースアーキテクチャを提案する．具体的には，インメモリ KVS で書き込みを受け付けつつ，メモリ上の古いデータをオンディスク KVS に移動することにより，メモリ使用量を抑え，インメモリ KVS の高い書き込み処理効率の利点とオンディスク KVS のデータ保存効率の利点を得られる，KVS 間の自動階層化を実現する．本研究で提案するアーキテクチャを実現することにより，メモリとディスクへの書き込み処理が特定の DBMS から分離されるため，当該書き込み処理の部分を変更するのみで，データ構造の拡張が可能となる．

本論文の構成を述べる．2 章では，関連研究の調査と課題を提示する．3 章では，提案手法のアーキテクチャおよび実装の詳細を示す．4 章では，実験環境で提案手法の有効性を評価する．5 章では，実験結果を考察する．6 章では，株式会社はてなのモニタリングサービス Mackerel[*1] の本番環境への提案手法の適用を示す．7 章では，本論文をまとめ，今後の展望を述べる．

2. 関連研究

モニタリングサービスの時系列データベースには，書き込み処理効率，データ保存効率，書き込みスケーラビリティ，高可用性およびデータ構造の拡張性が求められる．しかし，先行手法は，データ構造の拡張性の観点において

[*1] https://mackerel.io/

課題が残る．そこで，先行手法の特徴と問題点を述べる．

OpenTSDB[3] は，ディスク上にデータを配置する KVS である HBase にデータを格納しており，HBase の分散機構による書き込みスケールアウト性と高可用性をもつ．HBase のストレージエンジンは LSM-Tree(Log Structured Merge Tree)[6] を利用した Bigtable[7] と同様に，ログ先行書き込み，MemStore と呼ばれるメモリ上のデータ構造，および HStore と呼ばれるディスク上のデータ構造で構成されており，MemStore へ蓄積されたデータを HStore へまとめて書き込むことにより，ディスクへ直接書き込み方式と比較し，単位時間あたりのディスク書き込み回数を小さくしている．さらに，OpenTSDB では，時系列データの読み出し効率を高めるために，MemStore 上でメトリック系列ごとにタイムスタンプの順にソートされるようにスキーマが設計されており，メトリック系列単位で範囲検索が可能となっている．

Gorilla[8] は，直近のデータをすべてメモリ上に保持することにより，読み込み性能に特化させたインメモリの時系列データベースであり，HBase を利用した ODS(Operational Data Store) と呼ばれるストレージ層に長期間のデータを保持する．Gorilla と ODS の両方に同時にデータが書き込まれるため，ODS への書き込み処理を削減するわけではなく，あくまで高速にデータを読み出すことに焦点を置いている．

InfluxDB[9] は LSM-Tree を時系列データに最適化した Time Structured Merge Tree(TSM) を実装しており，HBase 同様にメモリ上のデータ構造に蓄積したデータをまとめてディスクに書き込み，さらに Gorilla で提案されている差分符号化手法などを取り入れているため，書き込み処理効率とデータ保存効率が高い．

データ構造の拡張性の観点では，OpenTSDB と InfluxDBB は各種構成要素が密結合な単一の DBMS として構成されており，データ構造の拡張性が低いと言える．Gorilla は，HBase では満たせない要求のために，インメモリのデータ構造を拡張しているといえるが，読み取り処理の高速化以外の拡張については言及がない．

3. 提案手法

2 章で述べた先行手法の課題である拡張性を確保するために，特定の DBMS のアーキテクチャに依存せず，KVS の特徴である書き込みスケールアウト性と高可用性を持ち，書き込み処理効率とデータ保存効率を両立するために，メモリとディスクのデータ構造をそれぞれ異なる KVS に分離し疎結合化した時系列データベースアーキテクチャ HeteroTSDB を提案する．

3.1 HeteroTSDB アーキテクチャ概要

HeteroTSDB アーキテクチャは，将来の拡張性を高めるために，密結合な DBMS を拡張するのではなく，拡張内容の特性に合わせた異なる DBMS を追加し，各 DBMS を一つの DBMS にみえるように統合した疎結合なアーキテクチャをとる．その上で，書き込みスケールアウト性と高可用性を達成するために，分散構成をとりやすい KVS を利用する．さらに，書き込み処理効率とデータ保存効率を実用上問題ならない程度に高めるために，インメモリ KVS にデータを書き込み，インメモリ KVS 上に蓄積された古いデータをメトリック系列単位でまとめてオンディスク KVS へ移動する自動階層化を実現する．インメモリ KVS は，メモリ上のデータが揮発し消失するという課題がある．そこで，インメモリ KVS に書き込む前にインメモリ KVS とは異なるサーバ上のディスクにデータ点をログとして書き込んでおき，インメモリ KVS が故障しデータが揮発したときに，ログを元にデータ点をインメモリ KVS に再度書き込むことにより，揮発したデータを復旧させるログ先行書き込み [10] により解決する．受信したメッセージを保存し予め購読している購読者へメッセージを送信するメッセージブローカー [11] を経由して時系列データを書き込むことにより，メッセージをログとして捉えたログ先行書き込みを実現する．メッセージブローカーは，ディスクにシーケンシャルにメッセージを書き込むため，書き込み効率が高く，時系列データベース全体での書き込み処理効率を大幅に低下させるわけではない．

HeteroTSDB アーキテクチャの仕組みを図 1 に示す．まず，書き込み処理は次のような流れとなる．

(1) クライアントがメッセージブローカーに書き込み，メッセージブローカーに書き込みを完了した時点でクライアントにレスポンスが返却される
(2) クライアントの書き込みとは非同期に，メッセージブローカーを購読する MetricWriter がインメモリ KVS に書き込む
(3) バックグラウンドでインメモリ KVS からオンディスク KVS へ一定以上古いタイムスタンプをもつデータを移動させる

次に，読み込み処理は次のような流れとなる．

(i) クライアントは MetricReader へメトリック名とタイムスタンプの範囲を含んだ問い合わせを送信する
(ii) MetricReader は問い合わせの条件に基づき各 KVS からメトリック系列を読み出す
(iii) MetricReader は各 KVS から読み出したメトリック系列を結合し，クライアントへ返却する

3.1.1 時系列データ構造

HeteroTSDB アーキテクチャでは，書き込みスケールアウト性と高可用性を確保するために KVS 上に時系列データを格納するため，キーバリュー形式で時系列データを格納する必要がある．以下に示す時系列データベースとしてデータを保存および取得するだけの機能をもつ KVS 上の

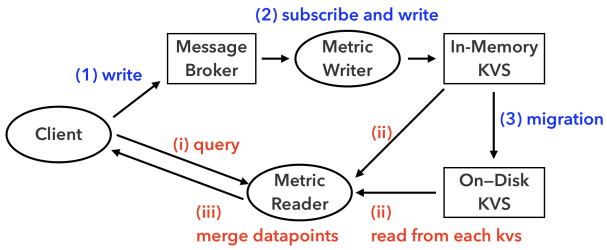


図 1: HeteroTSDB アーキテクチャ

時系列データ構造を提案する．

(1) 64 ビット整数の UNIX 時間により表現されるタイムスタンプと 64 ビット浮動小数点数の組（以降，データ点とする）の系列を保存可能

(2) メトリック名とタイムスタンプの開始時刻と終了時刻を入力として，該当するメトリック系列を取得可能

Redis[12] のようないくつかの既存の KVS は，与えられたキーに対してハッシュ計算したのちに順序を維持せずにデータを格納しているため，範囲検索をサポートしていないことがあり，タイムスタンプによる範囲検索ができない．ただし，Bigtable や HBase のように，内部的にキーをソートした状態で保存することにより，効率的に範囲検索できる KVS も存在するが，提案手法では範囲検索をサポートしていない KVS であっても実現できるデータ構造をとる．そこで，データ点を書き込む前に，タイムスタンプを解像度の倍数に揃える（以降，アラインメントとする）ことにより，タイムスタンプの範囲をもとに範囲内に含まれるタイムスタンプの一覧を静的に特定できるため，それらのタイムスタンプをキーとして参照すればよい．例えば，解像度を 60 秒とし，元のタイムスタンプが 1482699025 であれば，60 で割った剰余を差し引いた値である 1482699000 に変更したのちに，KVS に書き込む．問い合わせに含まれるタイムスタンプの範囲が 1482699000 から 1482699180 であるとすると，1482699000，1482699060，1482699120，1482699180 をそれぞれキーに含む 4 回の KVS への参照となる．ただし，アラインメントにより，元のデータをそのまま保存できないことと，解像度の秒数の間隔よりも短い時間間隔で新たなデータ点が書き込まれる場合，一つ前の古いデータ点が上書きされるという制限が発生する．

さらに，一つのキーバリューペアに一つのデータ点を格納するのではなく，一つのキーバリューペアに同一メトリック系列内の複数のデータ点を格納することにより，KVS への参照回数を削減する．しかし，このデータ構造では，メトリック系列内のデータ点が追記されるたびにキーバリューペアのサイズが増大し，KVS の制限に達する可能性がある．そこで，メトリック系列を固定幅のタイムウィンドウに分割することにより，制限されたサイズ内にデータを収める．KVS のキーバリューペアのサイズ制限については，例えば Amazon DynamoDB[13] は 400KB となっている．

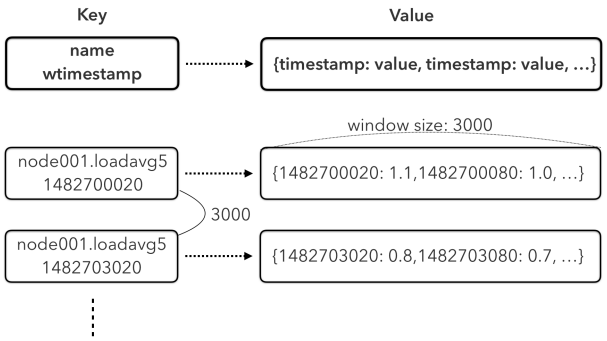


図 2: 時系列データ構造

KVS への書き込み処理中に一時的なエラーが発生したときのデータ消失を防ぐために，書き込み処理を再試行する必要があり，同じデータを重複して書き込まないように，書き込みに対してべき等となるようにする．そこで，1 つの一意キーに 1 つの値を紐付け，キーに対応する値を参照するためのデータ構造であるハッシュマップを利用する．具体的には，ハッシュマップのキーをタイムスタンプ，バリューをタイムスタンプに紐づく値とすると，同じタイムスタンプをもつデータ点であれば上書きされるため，べき等性を担保できる．

以上のタイムスタンプアラインメント，タイムウィンドウ，およびハッシュマップを組み合わせたキーバリュー形式の時系列データ構造を図 2 に示す．図 2 の name はメトリック名を表す文字列，timestamp はデータ点の UNIX 時間を表す 64 ビット整数型の数値，value はデータ点の値を表す 64 ビット浮動小数点型の数値，および wtimestamp はタイムウィンドウの開始時刻を表す UNIX 時間である．例えば，ウィンドウサイズを 3000 とすると，wtimestamp の値は 1482699000，1482702000，1482705000... のように 3000 の倍数となる．データ点の書き込みでは，wtimestamp の値は書き込むデータ点のタイムスタンプを 3000 で割った剰余を差し引いた値となり，データ点の読み込みでは，入力されたタイムスタンプの範囲の開始から終了までに含まれる 3000 の倍数をすべて wtimestamp の値とし，各 wtimestamp に対応するタイムウィンドウからデータ点を読み込み，メトリック系列として結合する．

3.1.2 KVS 間のデータ移動

HeteroTSDB アーキテクチャでは，新規の書き込みを受けつつ，インメモリ KVS からオンディスク KVS へ古いデータのみを移動する必要がある．3.1.1 節で述べた時系列データ構造により，KVS 上には，書き込まれたメトリック系列の個数に対して線形にキーバリューペアが増加するため，メトリック系列数の増加に対して，データ移動時間がスケールするデータ移動手法が必要となる．そこで，タイマー手法とカウント手法を提案する．

図 3 にタイマー手法を示す．タイマー手法では，KVS のキーバリューペア単位でタイマーを設定し，タイマーが

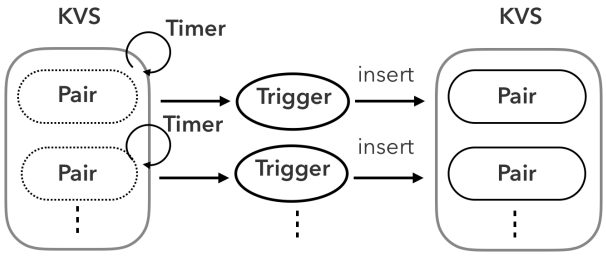


図 3: KVS 間のデータ移動: タイマー手法

0 になると予め登録したトリガーに当該キーバリューペアを引き渡し，トリガーが当該キーバリューペアのデータ移動処理を実行する．タイマー手法により，各キーバリューペアに紐づくトリガー処理では一つのキーバリューペアのデータ移動を処理すればよいため，互いの処理に依存関係がなく独立して処理できるため，並行処理が容易となる．したがって，キーバリューペア数が増大したとしても，その分だけスレッドなどのデータ移動のための処理単位を増加させることにより，各キーバリューペアの合計移動時間の増大を低減できる．さらに，各トリガー処理中にエラーが発生したとしても，3.1.1 節で述べたようにべき等性をもつデータ構造であるため，再試行によりデータの一貫性は保たれる．タイマーの実装として，キーバリューペア単位で設定可能な TTL(Time To Live) を用いることを想定している．

一方で，TTL が実装されていない KVS を利用できない場合のために，メトリック系列のデータ点が一定間隔で書き込まれることに着目したカウント手法を提案する．カウント手法では，MetricWriter がデータ点をインメモリ KVS に書き込むときに，タイムウィンドウ内のデータ点数を計上し，一定個数以上であれば，インメモリ KVS からタイムウィンドウを読み出し，オンディスク KVS に書き込んだのちにインメモリ KVS から削除することにより，データ移動を実装する．ただし，データ点が書き込まれなくなったメトリック系列は，オンディスク KVS に移動されず，インメモリ KVS 上にデータ点が残り続けるため，MetricWriter の処理とは独立して，定期的なバッチ処理により残留データ点をオンディスク KVS へ移動させる．データ点が書き込まれ続ける系列はオンディスク KVS へ逐一移動されるために，バッチ処理により移動させなければならない系列の数は小さくなるため，実行時間を短くでき，系列の数の増大に対するスケーラビリティの問題は解消される．エラー処理の問題についても，バッチ処理の実行時間が短ければ、失敗時に最初からやり直しやすい．3.1.1 節で述べたようにべき等性をもつデータ構造であるため，一連の処理の流れの中で一時的なエラーが発生した場合は，処理の最初から再試行することにより，KVS 間のデータ整合性が担保される．カウント手法は，タイマー手法と比較し，データ点数の計上と，インメモリ KVS からのデータ読み出しのためのオーバーヘッドがあり，さらに部分的にバッチ処理する必要があるが，TTL の実装されていない KVS を利用する場合に有効な手法となる．

3.1.3 データ構造の拡張

HeteroTSDB アーキテクチャにおいて，データ構造を拡張するための手法を示す．まず，3.1.1 節で述べた時系列データ構造をもつインメモリ KVS およびオンディスク KVS 以外に，異なるデータ構造を備えた異なる DBMS（以降，追加 DBMS とする）を用意する．次に，書き込まれたデータを複製し，追加 DBMS に書き込む処理を追加する．最後に，MetricReader に追加 DBMS を参照するための処理を追加する．

複製書き込みのための処理を追加する手法は，次の 2 点のいずれかとなる．

(a) リアルタイム書き込み: MetricWriter と同様にメッセージブローカーを購読する処理単位を並置させ，追加 DBMS にデータ点を書き込む．
(b) バッチ書き込み: カウント手法とタイマー手法のいずれにおいて，オンディスク KVS に書き込む処理の後に，オンディスク KVS に書き込んだデータ点と同一の内容を追加 DBMS に書き込む．

リアルタイム書き込みでは，クライアントから時系列データベースに書き込まれたデータ点は即時追加 DBMS に書き込まれ，参照可能となる．バッチ書き込みでは，追加 DBMS に書き込むタイミングがインメモリ KVS 上にデータ点を蓄積する間だけ遅延する一方で，オンディスク KVS と同様に追加 DBMS に対する書き込み回数を削減できる．さらに，いずれの手法においても，追加 DBMS に対する書き込み処理がべき等となるように工夫しておくことで，いずれの DBMS への書き込み処理中にエラーになったとしても，処理を最初から再試行することにより，各 DBMS 間のデータの一貫性が担保される．

本手法により，追加 DBMS に対する書き込み回数と追加 DBMS へのデータの反映遅延時間のトレードオフに対して，追加 DBMS の性質に合わせて，リアルタイム書き込みまたはバッチ手法のいずれかを選択できる．

3.2 実装

HeteroTSDB アーキテクチャの一実装として，AWS(Amazon Web Services) を利用した実装を示す．本実装では，書き込み処理効率と容量単価の異なる KVS を 3 階層に構成し，2 階層の構成と比較してデータ保存効率を高めている．また，異種混合 KVS とメッセージブローカーを運用することは単一の DBMS を運用することと比較し，サーバ上で動作する OS と各種アプリケーションの環境構築，ソフトウェアのバグや脆弱性の対応のための更新作業，負荷の増大に合わせたサーバのスケールアウト作業といった負担が発生する．そこで，本実装では，これらの負担をクラウド事業者が自動的に処理することにより，利用者は

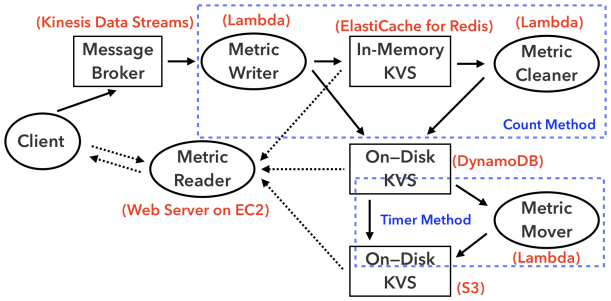


図 4: システム構成

API(Application Programming Interface) を実行するのみで作業を容易に自動化できるサーバレスプラットフォーム [14] を利用する．

図 4 にシステム構成を示す．Amazon Kinesis Data Streams[15] は，メッセージブローカーとして動作し，受信したメッセージを利用者が追加でプログラムを書くことなく Lambda 関数の引数に渡せる．Kinesis Data Streams におけるストリームはシャードと呼ばれる単位の集合体であり，シャード数の増加により，メッセージの書き込み処理をスケールアウト可能である．MetricWriter は，Kinesis Data Streams と連携した Lambda 関数であり，Kinesis Data Streams のシャードの個数だけ Lambda 関数の処理単位が起動するため，シャード数の増加によりスケールアウトできる．インメモリ KVS の Amazon ElastiCache for Redis[16] は，書き込みを分散できる Redis Cluster として利用し，稼働中のクラスタに対してノードの追加が可能であり，ノード間でデータを均等に分散する機能をもつ．オンディスク KVS の Amazon DynamoDB は，SSD(Solid State Drive) 上で構築された KVS であり，秒間の読み取り回数，書き込み回数，およびキーバリューペアのサイズを基に独自定義されたキャパシティユニットという単位を増加させることにより，クラスタのスケールアウトが可能となっている．Redis と DynamoDB は，ハッシュマップをサポートするため，3.1.1 節で述べた時系列データ構造を実装できる．Amazon S3[17] は DynamoDB と比較し，約 1/10 の容量単価となるオブジェクトストレージであり，ファイルパス名をキー，ファイルコンテンツをバリューとすると KVS とみなせる．MetricCleaner は，インメモリ KVS 上に残留するデータをバッチ処理でスキャンし，オンディスク KVS へ移動させる Lambda 関数であり，Amazon CloudWatch Events[18] により定期的に Lambda 関数を起動する．MetricReader は，Go 言語 [19] で実装した Amazon EC2[20] のインスタンス上の Web サーバであり，処理に必要なデータは各 KVS 上に保存されており，MetricReader 自体は状態を持たないため，ロードバランサを利用することによりスケールアウトできる．

DynamoDB から S3 へのデータ移動では，DynamoDB は DynamoDB Triggers と呼ばれる機能により，TTL が 0 になりキーバリューペアが削除されると Lambda 関数を起動できるため，サーバレスプラットフォーム上で容易にタイマー手法を実装できる．ここでの Lambda 関数 (MetricMover) の処理は，削除されたキーバリューペアを受け取り，単に S3 へ書き込むのみとなる．一方で，Redis から DynamoDB へのデータ移動では，Redis には DynamoDB のように Lambda 関数と連携する機能はないため，実装要件であるサーバレスプラットフォーム上で実現するために，カウント手法をとった．

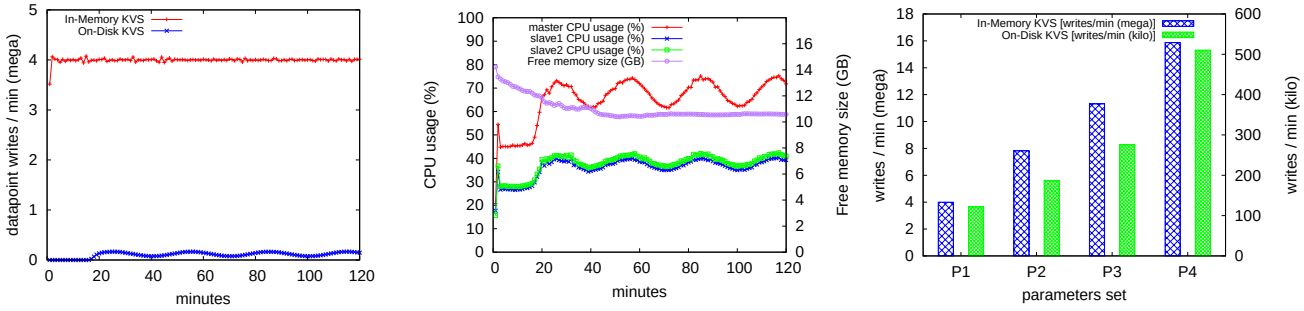
表 1: 実験環境

ロール	項目	仕様
Benchmark Client	EC2 インスタンス	
	タイプ	c5.4xlarge
	OS,Kernel	Amazon Linux 2, 4.14
Message Broker	Kinesis Data Streams	
	シャード数	32
MetricWriter	Lambda	
	メモリ量	1600(MB)
	ランタイム	Node.js 8.10
In-Memory KVS	ElastiCache for Redis	
	タイプ	cache.r4.large
	バージョン	3.2.10
On-Disk KVS	DynamoDB	

4. 実験

HeteroTSDB アーキテクチャの有効性を確認するために，3.2 節にて示した実装の一部が動作する環境にて，書き込み処理効率，データ保存効率，および書き込みスケールアウト性を評価した．実験では，表 1 に示す実験環境を構築し，メッセージブローカーに対してメトリックを書き込むベンチマークを実施する．実環境を模すために，収集したメトリックを定期的にモニタリングシステムへ送信するためのエージェントがホスト上で動作している状況を想定し，エージェントの個数，エージェントのメトリック送信間隔および 1 回の送信に含まれるメトリックの個数を指定可能なベンチマークプログラムを作成した．ベンチマーチプログラムは，Go 言語で実装されており，指定したエージェントの個数分のスレッドを作成し，各スレッドが自動生成したメトリックを定期的に送信している．以下のベンチマークでは，メトリックの送信間隔を 1 分，エージェントの同時送信メトリック数を 100，MetricWriter がインメモリ KVS 上に同一メトリック系列内のデータ点を蓄積する個数を 15 とする．これらの固定のパラメータ値を除いた，ベンチマークごとに可変なパラメータの組み合わせを表 2 に示す．WCU(Write Capacity Units) とは，AWS により定義されている DynamoDB のテーブルに対する書き込みのためのキャパシティ量であり，テーブル上の 1 項目に対する 1KB 以下の 1 秒あたり 1 回の書き込みが 1 WCU となる．

まず，書き込み処理効率を評価するために，オンディス



(a) 分間書き込み回数　**(b)** インメモリ KVS のリソース使用量　**(c)** 書き込みスケールアウト性能

図 5: 実験結果

表 2: スケールアウト性能ベンチマークのパラメータ

	エージェント数	インメモリ KVS のノード数	WCU
P1	40,000	3	6,000
P2	80,000	6	12,000
P3	120,000	9	18,000
P4	180,000	12	24,000

ク KVS に直接書き込む場合と比較し，インメモリ KVS で書き込みを受けることにより，オンディスク KVS への書き込み回数が減少しているかを確認する．次に，データ保存効率を評価するために，書き込みを続けた結果，インメモリ KVS の空きメモリ量が 0 になることなく，オンディスク KVS へデータを移動できているかを確認する．表 2 の P1 のパラメータを用いてベンチマークプログラムを 2 時間動作させ，インメモリ KVS とオンディスク KVS への分間書き込み回数変化，インメモリ KVS の空きメモリ量と CPU 利用率の変化を観察し，グラフ化した．図 5(a) に分間書き込み回数の遷移，図 5(b) に空きメモリ量と CPU 利用率の遷移を示す．

最後に，書き込みスケーラビリティを評価するために，各 KVS のキャパシティを増加させたときに，分間書き込み回数がスケールアウトするかどうかを確認する．表 2 のパラメータの各組み合わせについて，それぞれベンチマークプログラムを 2 時間動作させ，分間書き込みデータ点数の平均値をとり，グラフ化した．図 5(c) にインメモリ KVS とオンディスク KVS のそれぞれのスケールアウト性能を示す．ただし，ベンチマークの最初の 15 分間はインメモリ KVS にデータ点を蓄積しており，オンディスク KVS の書き込み回数は 0 となるため，オンディスク KVS の書き込み回数が不当に小さくならないように最初の 15 分間の値は平均値の計算対象から除外した．

図 5(a) では，インメモリ KVS の分間書き込み回数は約 4M の値で一定であり，オンディスク KVS への分間書き込み回数は最初の 15 分間では 0 のまま遷移し，それ以降は 70k から 170k の間の値をとりつつ遷移している．4M という値は，ベンチマークプログラムにて指定した分間書き込みデータ点数と一致する．オンディスク KVS への分間書き込み回数は，インメモリ KVS の分間書き込み回数の約 1/20 程度となっている．

図 5(b) では，インメモリ KVS の空きメモリ使用量は最大容量の 16GB からベンチマーク時間が増加するにつれて 50 分経過するまで徐々に減少し，その後約 10.5GB 前後で一定の値をとりつつ遷移している．図 5(b) の master はクラスタ内のマスタノードを示しており，slave1 と slave2 はそれぞれ 2 台のスレーブノードを示している．マスタノードの CPU 利用率は，15 分経過するまで約 45%で一定となっているが，その後約 75%まで増加し，約 60%から約 75%の間を波形状に値をとりつつ遷移している．MetricWriter が一部の読み取り処理を分散させているスレーブノードの CPU 利用率は，グラフの形状はマスタノードと同様だが，15 分経過以降は，最大値として約 40%，最小値として約 35%の値をとる．

図 5(c) の parameters set は表 2 のパラメータの組み合わせを示しており，Redis Cluster のマスタノード 1 台，スレーブノード 2 台を一つのグループとし，3 台ずつ増加させている．図 5(c) では，インスタンス KVS とオンディスク KVS への分間書き込みデータ点数がノード数および WCU の増加に対して線形に増加しているため，スケールアウトしていると言える．

5. 考察

オンディスク KVS へ直接書き込む方式で同様のベンチマークを実施したとすると，提案手法にてインメモリ KVS が受け付けていた書き込みをインメモリ KVS の代わりにオンディスク KVS が受け付けることになるため，オンディスク KVS への分間書き込み回数が提案手法におけるインメモリ KVS への分間書き込み回数が同程度になるはずである．したがって，4 章で示した結果から，オンディスク KVS へ直接書き込む方式と比較して，提案手法はオンディスク KVS への書き込み回数を約 1/20 に削減していると言える．また，実験に利用したベンチマーククライアントは，同じメトリック名かつ同じタイムスタンプをもつデータ点

を 1 回のみ送信するため，インメモリ KVS 上ではデータが追記されるのみとなり，メモリ使用量は時間経過に対して単調増加するはずである．4 章で示した結果から，インメモリ KVS のメモリ使用量が時間経過に対して変化しないため，提案手法は，オンディスク KVS へデータを移動させることにより，インメモリ KVS のメモリ使用量を一定に保つことがわかる．以上により，実験では，書き込み処理効率の高いインメモリ KVS にデータを書き込みつつも，容量単価の大きいメモリの使用量の増加を抑え，容量単価の小さいオンディスク KVS へデータを移動していることを確認できたため，提案手法は書き込み処理効率とデータ保存効率を両立できていると考える．

図 5(b) において，ベンチマーククライアントは一定のスループットでデータ点を送信しているにも関わらず，時間経過するとインメモリ KVS の CPU 利用率が増加している．MetricWriter のインメモリ KVS に対する処理が時間変化に対して増える要素は，インメモリ KVS 上に一定個数以上蓄積したメトリック系列をオンディスク KVS へ書き込むときに，インメモリ KVS から系列を読み出し，削除する処理以外に実装上存在しないため，これらのオーバーヘッドにより，CPU 利用率が増加していたと推測できる．インメモリ KVS からオンディスク KVS へのデータ移動について，実験にて採用したカウント手法ではなくタイマー手法を採用することにより，インメモリ KVS から系列を読み出す処理が不要になるため，CPU 利用率の増加を抑えられると考える．

6. 実環境への適用

株式会社はてなのモニタリングサービスである Mackerel の実環境に HeteroTSDB アーキテクチャを適用した例を示す．具体的には，本手法を適用した 2017 年 8 月から 2018 年 8 月までの 1 年間の時系列データベースに関する障害と故障についての対応を記述する．実環境におけるシステム構成は，図 4 と同等だが，開発当時に ElastiCache for Redis がクラスタのサイズの拡張に対応しておらず，スケールアウトが困難であったために，実環境では EC2 インスタンス上の Redis Cluster を採用している．

上記の期間に発生した障害は 2 件ある．1 件目では，インメモリ KVS の特定のノードに書き込み負荷が集中し，当該ノード内のメモリ消費が上限に達し，OS がプロセスを強制停止した結果，複数の特定のメトリックのデータを一時的に消失した．そこで，Kinesis Data Streams に残留しているデータに対して，消失した時刻よりも前の時刻から Lambda 関数により再処理させることにより，消失したデータを復旧した．インメモリ KVS の故障時のデータ永続性の課題をメッセージブローカーにより解決できたと言える．2 件目では，同一メトリック名かつ同一タイムスタンプをもつ想定以上の個数のデータ点が短時間で書き込まれ，インメモリ KVS に対する書き込みクエリのサイズが実装上の上限を超えエラーを返し，Lambda 関数の実行が再試行され続け，MetricWriter 全体の処理が遅延した．MetricWriter にて，同一メトリック名かつ同一タイムスタンプをもつデータ点の重複を除去するようにプログラムを修正し，解決した．

同期間に発生した故障として，インメモリ KVS のノード故障が 2 件ある．ただし，AWS の利用者からは本システム構成の要素のうち EC2 インスタンス以外のコンポーネントの故障を確認できないため，故障記録の対象から除外している．1 件目では，クラスタを構成するノードのうち 1 個の EC2 インスタンスが意図せず再起動したが，Redis Cluster の故障検知機能により，クラスタから自動で除外されたため，サービス利用者に影響はなかった．EC2 インスタンスの起動時に同一データの複製をもつ他のノードからデータを取得するようにしているため，データの消失なく自動で復旧した．2 件目では，クラスタ内のノードのうち同時に 2 個の EC2 インスタンスが意図せず停止したが，1 件目と同様にクラスタから自動で除外されたため，クラスタから除外されるまでエラーが短時間発生し，一部のメトリックの書き込みが遅延するのみの影響に留まった．以上により，故障が発生したとしても，全体として処理を継続できていたことから，可用性を確保できていると言える．

7. まとめと今後の展望

本研究では，モニタリングサービスの要件を踏まえ，データ構造の拡張性を高めるために異種混合 DBMS 構成を前提とし，インメモリ KVS とオンディスク KVS を自動階層化する時系列データベースアーキテクチャの HeteroTSDB を提案した．AWS のサーバレスプラットフォーム上で実装することにより，複数の DBMS の構築やスケールアウト作業の負担を低減できるため，異種混合 DBMS 構成をとりやすくなり，HeteroTSDB アーキテクチャを実環境へ適用しやすくしている．実験により，拡張性のために疎結合なアーキテクチャをとりつつも，実用に耐えられる程度の書き込み処理効率，データ保存効率，および書き込みスケールアウト性をもつことを確認した．また，HeteroTSDB アーキテクチャを実環境へ適用した結果，故障時に全体の処理を継続できなくなることがなく，可用性が問題になることがないことがわかった．

今後の展望としては，まず，他の時系列データベースとの比較実験をした上で HeteroTSDB アーキテクチャが性能面で実用的であることを示す必要がある．次に，データ構造の拡張の実装を進め，HeteroTSDB アーキテクチャの拡張性を評価できるように研究を進めていく．モニタリングのために有用な機能を追加する予定であり，具体的には，問い合わせ時に必要なデータが存在する KVS のみ参照するためのデータ構造を追加することにより，低速な KVS

への参照を減らし，読み込み処理速度を高めていく．さらに，OpenTSDB や InfluxDB のようにメトリック系列に対して属性情報を付与することにより，豊富なデータ表現が可能となるように拡張していく．最後に，KVS 間のデータ移動のためのタイマー手法を汎用化し，処理とデータとタイマーを結びつけたデータパイプラインのための新たなアーキテクチャを考えていく予定がある．

参考文献

[1] Meng S, Liu L: Enhanced Monitoring-as-a-Service for Effective Cloud Management, *IEEE Transactions on Computers*, Vol. 62, No. 9, pp. 1705–1720 2013.
[2] Jensen S K, et al.: Time Series Management Systems: A Survey, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 29, No. 11, pp. 2581–2600 2017.
[3] OpenTSDB, http://opentsdb.net.
[4] Cattell R: Scalable SQL and NoSQL Data Stores, *ACM SIGMOD Record*, Vol. 39, No. 4, pp. 12–27 2011.
[5] Zhang H, et al.: In-Memory Big Data Management and Processing: A Survey, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 27, No. 7, pp. 1920–1948 2015.
[6] O'Neil P, et al.: The Log-Structured Merge-Tree (LSM-tree), *Acta Informatica*, Vol. 33, No. 4, pp. 351–385 1996.
[7] Chang F, et al.: Bigtable: A Distributed Storage System for Structured Data, *ACM Transactions on Computer Systems (TOCS)*, Vol. 26, No. 2, pp. 4:1–4:26 2008.
[8] Pelkonen T, et al.: Gorilla: A Fast, Scalable, In-Memory Time Series Database, *41st International Conference on Very Large Data Bases (VLDB)*, Vol. 8, No. 12, pp. 1816–1827 2015.
[9] InfluxData: InfluxDB, https://www.influxdata.com/time-series-platform/influxdb/.
[10] Mohan C, Levine F: ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging, *ACM International Conference on Management of Data (SIGMOD)*, pp. 371–380 1992.
[11] John V, Liu X: A Survey of Distributed Message Broker Queues, arXiv preprint arXiv:1704.00411 2017.
[12] Sanfilippo S, Noordhuis P: Redis, https://redis.io.
[13] Amazon Web Services: Amazon DynamoDB, https://aws.amazon.com/dynamodb/.
[14] Amazon Web Services: Serverless Computing and Applications, https://aws.amazon.com/jp/serverless/.
[15] Amazon Web Services: Amazon Kinesis Data Streams, https://aws.amazon.com/jp/kinesis/data-streams/.
[16] Amazon Web Services: Amazon ElastiCache for Redis, https://aws.amazon.com/elasticache/redis/.
[17] Amazon Web Services: Amazon S3, https://aws.amazon.com/jp/s3/.
[18] Amazon Web Services: Amazon CloudWatch, https://aws.amazon.com/documentation/cloudwatch/.
[19] Donovan A A, Kernighan B W: *The Go Programming Language*, Addison-Wesley Professional, 1st edition 2015.
[20] Amazon Web Services: Amazon EC2, https://aws.amazon.com/jp/ec2/.