情報処理学会論文誌 Vol.62 No.3 818–828 (Mar. 2021)

HeteroTSDB：異種分散KVS間の自動階層化による高性能な時系列データベース

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

受付日 2020年6月22日, 採録日 2020年12月1日

概要：クラウド上のシステムの大規模化にともない，CPU 利用率などのシステムの各構成要素の状態を把握するために，大量の時系列データを保存する必要がある．そのために，時系列データを保存するデータベースにはデータの挿入処理とデータ保存の効率化と挿入スケールアウト性の向上が求められる．既存技術は，挿入スケールアウト性を高めるために広く利用されているディスクベースの分散 KVS（Key Value Store）を利用する．しかし，ランダム I/O が低速なディスクへ書き込むという前提があることから，メモリ上でキーを整列させながら挿入可能な平衡木が利用されるが，キーの挿入時に系列数に対して対数時間を要する．すべてのデータをメモリ上に保持するメモリベース KVS であれば，ハッシュ表に基づくデータ構造の利用により定数時間の挿入が可能となる．しかし，メモリは容量単価が大きいことから，データを長期間保存するには不向きである．本論文では，メモリベース KVS とディスクベース KVS を階層化する高性能な時系列データベース HeteroTSDB を提案する．HeteroTSDB は，メモリベース KVS 上に系列名をキーとして，系列本体をバリューとしたハッシュ表を構成することにより，系列数に対して定数時間でデータを挿入する．加えて，系列を格納するキーに TTL（Time To Live）によるタイマを設定し，古くなったデータを系列単位でまとめてディスクベース KVS へ移動させることにより，データ保存のための容量単価を低減させている．実験の結果，ディスクベース KVS を利用した既存の時系列データベースである KairosDB と比較し，HeteroTSDB は 3.98 倍の挿入スループット向上を達成した．

キーワード：key value store，monitoring，time series database，redis，cassandra

HeteroTSDB: A High Performance Time Series Database for Automated Data Tiering in Heterogeneous Distributed KVSs

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

Received: June 22, 2020, Accepted: December 1, 2020

Abstract: As the scale of systems on the cloud grows, there is a need to store a significant amount of time series data to understand the status of each component of systems, such as CPU utilization. Time series databases require high insertion scale-out property and high efficient utilization of hardware resources for insertion and storage. Existing technologies utilize disk-based distributed KVSs (Key Value Stores), which is widely used to achieve insertion scale-put property. Since random I/Os to disk are slower than to memory, the disk-based KVSs adopt an equilibrium tree, that can be inserted while aligning the keys in memory. However, it takes logarithmic time for the number of time series. Memory-based KVSs, where all the data are stored in memory, adopts data structure based on a hash table, which is possible to insert an key in constant time. However, it is not suitable for long-term storage because the unit cost of memory is higher than disk. In this paper, we propose HeteroTSDB, a high performance time series database that tiers a memory-based KVS and a disk-based KVS. HeteroTSDB accelerates the inserting process by constructing a hash table with a series name as a key and series data as a value on the memory-based KVS. Automatic tiering reduces the unit cost of storage capacity by moving the old data from a memory-based KVS to a disk-based KVS. Benchmark experimentation demonstrates that, compared with the existing time series database on a disk-based KVS KairosDB, HeteroTSDB achieves performance improvement of up to 3.98 times in insert throughput.

Keywords: key value store, monitoring, time series database, redis, cassandra

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，インターネットを介して利用するクラウド上のシステムに対して，利用者が高可用性および高速な応答速度を要求するようになってきている．そこで，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，システムの状態をつねに計測すること（以降，モニタリングとする）が必要となる [8]．モニタリングを実現するために，モニタリングのためのシステム（以降，モニタリングシステムとする）をシステム管理者が別途構築する．モニタリングシステムでは，対象のシステムから CPU 利用率や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 [9], [18] とする）に保存したのちに時系列グラフとして可視化する機能を提供することがある．

発生中の障害を分析するためには，障害発生前の状態から現在までの変化と短時間での状態の変化を見逃さないように，メトリックを定常的にかつ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，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．したがって，メトリックの挿入とデータ保存のためのハードウェアリソース利用を効率化する必要がある．加えて，モニタリングシステムでは，モニタリング対象の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，ホスト数を増加させることにより線形に挿入性能が向上するという性質，すなわち挿入スケールアウト性が求められる．

挿入スケールアウト性を高めるために，データの基本要素がキーと値のペアであり，要素どうしが互いに依存せず要素単位でデータを分散して配置することから，挿入スケールアウト可能な分散 KVS [15] が一般に利用されている．分散 KVS 上に時系列データを扱うためのアプリケーションを構成する既存の技術 [16], [22] は，ディスク上にデータを格納する前提のディスクベース KVS [11], [13] を利用している．これらのディスクベース KVS は，書き込み要求を受信するとメモリ上のデータ構造に書き込み，メモリ上のデータサイズが閾値を超えると非同期にディスクへフラッシュ書き込みを行う．ディスクはメモリと比較してランダム I/O が低速であるから，フラッシュ書き込み時に順次的に書き込む必要があるために，メモリ上では要素が整列した状態を維持可能な平衡木などの局所性を持つデータ構造が利用される．しかし，メモリへの書き込み速度だけを追求するのであれば，定数時間で書き込み可能なハッシュ表などのデータ構造を利用することが望ましい．すべてのデータをメモリ上に保持するメモリベース KVS [21], [23] であれば，メモリへの読み書きに最適化したデータ構造を採用しやすい．しかし，ディスクと比較し，メモリは容量単価が大きいことから，時系列データをメモリのみに長期間保存するには不向きである．

本論文では，多数のメトリックを定常的にかつ高解像度で収集可能なモニタリングシステムを実現するために，メモリベース KVS とディスクベース KVS を階層化することにより，データの挿入とデータ保存に消費されるハードウェアリソース使用を効率化する時系列データベース HeteroTSDB を提案する．以降では，個々のメトリックの時系列データを単に系列とする．系列は，系列名とデータ点列により構成され，データ点列はタイムスタンプと，当該タイムスタンプに対応するメモリ使用量などの数値の組の列を含む．また，タイムスタンプと数値の組をデータ点とする．HeteroTSDB は，メモリベース KVS 上に系列名をキーとして，系列名に対応するデータ点列をバリューとしたハッシュ表を構成することにより，定数時間でデータ点を挿入可能とする．さらに，系列を格納するキーに TTL（Time To Live）によるタイマを設定し，古くなったデータ点を系列単位でディスクベース KVS へ移動させることにより，データ保存のための容量単価を低減させている．加えて，データ移動時にメモリベース KVS 上のキーに蓄積されたデータ点列を単一の挿入要求の中に集約することにより，ディスクベース KVS への挿入効率を向上させている．実験の結果，1 m 個の系列を含むデータセットに対して，ディスクベース KVS を利用した既存技術である KairosDB と比較し，HeteroTSDB は処理ホスト数が 8 のときに最大で 3.98 倍の単位時間あたりに挿入されたデータ

[1] さくらインターネット株式会社さくらインターネット研究所
SAKURA internet Research Center, SAKURA internet Inc., Osaka 530–0011, Japan
[2] 京都大学情報学研究科
Graduate School of Infomatics, Kyoto University, Kyoto 606–8501, Japan
[3] 株式会社はてな
Hatena Co., Ltd., Kyoto 604–0835, Japan
[4] Nature 株式会社
Nature Inc., Shibuya, Tokyo 150–0042, Japan
[5] 株式会社野村総合研究所
Nomura Research Institute, Ltd., Chiyoda, Tokyo 100–0004, Japan
[6] トヨタ自動車株式会社
Toyota Motor Corporation, Chiyoda, Tokyo 100–0004, Japan
[a] y-tsubouchi@sakura.ad.jp
[b] y-tsubouchi@net.ist.i.kyoto-u.ac.jp

本論文の暫定版は，2019 年の 7 月に国際会議 IEEE 43rd Annual Computer Software and Applications Conference（COMPSAC 2019）にて出版済みである．暫定版に対して本論文では，提案手法の要点をより明確化し，関連研究を充実させ，既存の他手法との比較実験に関する記述を追加している．

点数（以降，挿入スループットとする）の向上を達成した．

本論文の構成を述べる．2 章では，関連技術の調査と課題を提示する．3 章では，HeteroTSDB の詳細を示す．4 章では，実験環境において HeteroTSDB の有効性を評価する．5 章では，実験結果と HeteroTSDB の制限を考察する．6 章では，株式会社はてなのモニタリングサービス Mackerel [5] 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. モニタリングのための時系列データベース

2.1 時系列データベースの定義と要件

本論文では，Bader らの調査 [9] にならい，(1) タイムスタンプ，値，オプションタグ（任意の名前と値の組あるいは単一の文字列）からなるデータの行（時系列データ）を格納し，(2) 時系列データを複数行にまとめて格納し，(3) データの行に対して問合せを実行し，(4) 問合せにタイムスタンプや時間範囲を含めることができるデータベースを時系列データベースと呼ぶ．**図 1** に (1) と (2) の構造を示す．(1) の値として，モニタリングでは CPU 利用率やメモリ使用量，単位時間あたりのリクエスト数やエラー数などの数値を記録する必要があるため，既存の時系列データベース [10], [16], [20], [22] では単精度あるいは倍精度の浮動小数点数をサポートする．浮動小数点数以外に整数値，文字列をサポートする時系列データベース [17] も例外的に存在する．オプションタグを結合した文字列は，系列名として機能する．

読み取りを効率化する観点では，データ点列を行として連続領域に保存することにより，1 つの I/O 操作で指定したデータ点列を読み取れることが望ましい．同様に，挿入処理の効率化の観点では，データ点列を 1 つの I/O 操作で連続領域に書き込めることが望ましい．そのためには，時系列データベースの外部のデータ送信元が，一定個数のデータ点列を一時保存しておき，まとめて時系列データベースに送信しなければならない．しかし，一時保存されているデータ点にはアクセスできなくなるため，システム管理者が取得時間から最新の値を知るまでの間の遅延が増加する．この遅延を低減させるために，モニタリングシステム [5], [10] では，データ送信元でデータ点を一時保存せずに，即時に送信する．以上により，データ点列を 1 つの I/O 操作で挿入することは難しいため，それ以外の手法により挿入効率をいかに向上させるかが重要となる．

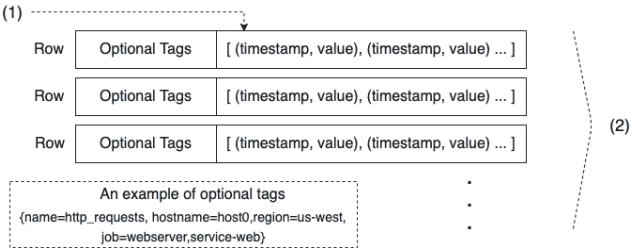


図 1 時系列データベース上の論理的なデータ構造
Fig. 1 Logical data layout of a time series database.

続いて，モニタリングシステムにおける時系列データベースの具体的な要件を述べる．第 1 に，実システムにおいて，どの程度の挿入数が要求されるかを定量的に見積もる．モニタリング対象数を n，対象あたりの平均系列数を m，メトリック取得のインターバル秒数を i とすると，秒間のデータ点の平均挿入数 w は，$w = \frac{nm}{i}$ となる．対象あたりの系列数の目安として，企業で利用される代表的なモニタリングシステムである Prometheus [10] は，Linux ホストのメトリックをデフォルトで 500 系列以上のメトリックを収集する．Linux ホスト上にデータベースや Web サーバのようなミドルウェアをインストールしている場合，それらが公開するメトリックを追加すると，対象あたりの系列数は 1,000 を超えることがある．次に，インターバル秒数として，Prometheus ではデフォルトで 15 秒を採用している．たとえば，クラウド上で管理する仮想サーバやコンテナの個数が 1,000 であれば，n は 1,000 となり，m を 1,000 とし，i を 15 とすると，w は 66,666 となる．大規模な事例として，Facebook では，分間 7 億個（秒間 10 m 個）の挿入スループットが要求されている [20]．

第 2 に，どの程度のデータ保存量が要求されるかを見積もる．さらに，モニタリングシステムをサービス提供している Mackerel では時系列データの保存期間として 460 日が要求されている．データ点のサイズを d バイト，保存期間秒数を t 秒とすると，時系列データの保存容量 s は最低でも，$s = \frac{nmdt}{i}$ バイトとなる．タイムスタンプが UNIX 時間として 64 ビット整数値をとり，値を 64 ビット浮動小数点数とすると，d は 16 バイトとなる．n を 1,000，m を 1,000，i を 15 として，460 日の保存要求を満たすとすると，時系列データの保存容量 s は 4.1（TB）となる．

第 3 に，挿入スケールアウト性に関する要求を整理する．Goldschmidt らの既存の時系列データベースに対する負荷実験 [14] によると，最小サイズのクラスタにおけるデータ点挿入スループットは秒間 40 k 個にとどまる．したがって，システム管理者が管理する対象の仮想サーバやコンテナの個数の増大に対して，ノードの増加に対して負荷を分散可能とする必要がある．Facebook では，最低でも 2 倍の年間成長率が要求されている [20]．

2.2 既存の時系列データベース技術

2.2.1 時系列データベースアプリケーション

既存の時系列データベース技術として，第 1 に，時系列データ以外の幅広い用途にも利用可能な既存のデータベース上に時系列データを効率良く扱うためのアプリケーションを構成する時系列データベースアプリケーションがある．

伝統的なリレーショナルデータベース（Relational DataBase，RDB）のスキーマとしてプライマリキーに系列名，セカンダリキーにタイムスタンプを指定することにより，すべての系列を単一のテーブルに保存できる．RDB はテーブル内外のレコード間の一貫性を保証するために，トランザクションを利用できる．しかし，挿入処理をスケールアウトさせるために，テーブルをキー単位で分割し，分割した各キーを複数のホストに配置する際に，一貫性を保証するために，ホスト間で同期をとる必要がある．したがって，多くの伝統的な RDB は，挿入スケールアウトのためのクラスタリング機構をサポートしない．

分散 KVS では，異なるホスト上の複数のキーに対するトランザクションをサポートしないことにより，挿入スケールアウト可能となる．モニタリングのための時系列データでは各系列は独立しており，複数の系列に対する一貫性が必要ないことから，既存技術の OpenTSDB [22] と KairosDB [16] は，それぞれ分散 KVS の HBase [13] と Cassandra [11] 上に時系列データを扱うアプリケーションを構成する．これらの分散 KVS は，ディスク上のデータ構造と同等の構造をバッファとしてメモリに配置するのではなく，memtable と呼ばれるメモリ常駐のデータ構造と，SSTable（Sorted String Table）と呼ばれるディスク常駐のデータ構造では異なるデータ構造を採用する [12]．データの追加・更新時は，ディスク上のログへの追記書き込みの後にメモリ上の memtable に書き込むのみである．memtable のサイズがあらかじめ定められた閾値を超えるか一定時間が経過するタイミングで，ディスク上の SSTable にまとめて書き込まれる．

キーの探索を効率化するために，SSTable はディスクへはキー名で整列された状態で格納されている．memtable 上のインデックス構造として，ハッシュ表のように局所性の小さいデータ構造を利用すると，SSTable へのフラッシュ書き込み時に整列処理のオーバヘッドが発生する．整列せずに済ませるためには，平衡木のように整列された状態を維持可能なデータ構造を利用することになるが，ハッシュ表と比較すると要素数が大きいときに低速となる．このように，ディスクベース KVS はメモリと比較してランダムアクセス性能の低いディスクへ書き込む前提があることから，メモリ上のデータ構造に局所性が必要となり，挿入時の効率を最適化しにくい．

2.2.2 時系列データ専用データベース

第 2 に，時系列データ専用のストレージエンジンを備える時系列データ専用データベースがある．

Gorilla [20] 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メモリベースの時系列データベースであり，HBase の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挿入されるため，ODS への挿入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 [17] や VictoriaMetrics [6] は時系列データに最適化したストレージエンジンを実装しており，HBase や Cassandra 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挿入することにより，挿入処理効率を高めている．さらに Gorilla で提案されている差分符号化手法を適用するため，データ保存量を低減できる．

時系列データの別形態として，タイムスタンプと文字列を行とするテキストログがある．Hayabusa [7], [24] は，ネットワーク機器やサーバソフトウェアが生成するテキストログを収集し，収集したログのすべての系列に対して時間範囲を指定した高速な全文検索を可能とする．

3. 異種混合キーバリューストアの自動階層化手法

本研究では，クラウド環境で KVS が広く利用されていることから，KVS を基にした時系列データアプリケーションを設計することを前提に，モニタリングシステムにおける時系列データベースに求められる高い挿入スループット，高い挿入スケールアウト性，データ保存のための低いディスク使用量あるいは低い容量単価を満たすことを目指す．モニタリングにおいて，収集する系列数が増大している昨今の背景をふまえると，系列数に対する時間計算量が小さいほうが望ましい．しかし，既存のディスクベース KVS はランダム I/O が低速なディスクへ書き込むという前提があることから，ハッシュ表のような定数時間で挿入可能だが空間局所性を持たないデータ構造よりも，平衡木のような局所性を持つデータ構造が望ましい．メモリベース KVS であれば，ディスク上にデータを移動しないため，メモリへの書き込み速度に最適化されたデータ構造を利用しやすいが，容量単価が相対的に大きいため，データの長期保持には不向きである．

そこで，メモリベース KVS とディスクベース KVS を階層化する時系列データベースアプリケーション HeteroTSDB を提案する．HeteroTSDB は，メモリベース KVS 上に系列名をキーとして，系列をバリューとしたハッシュ表を構成することにより，系列数に対して，定数時間で挿入処理可能とする．さらに，系列を格納するキーに TTL を設定し，古くなったデータ点を系列単位でディスクベース KVS へ移動させることにより，データ保存のための容量単価を低減できる．これにより，ディスクベース KVS への挿入回数を低減できるため，挿入処理が低速なディスクベース KVS も利用しやすくなる．

3.1 設計

図 2 は，Ingester モジュール，Flusher モジュール，Querier モジュール，メモリベース KVS およびディスク

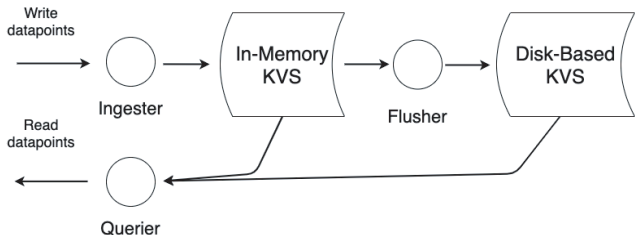


図 2 HeteroTSDB のアーキテクチャ
Fig. 2 HeteroTSDB architecture.

ベース KVS からなる HeteroTSDB のアーキテクチャを示している，各モジュールと分散 KVS は OS のプロセス単位で分散しており，それぞれ TCP/IP 上のプロトコルを介して通信する．Ingester モジュールは，受信したデータ点をメモリベース KVS に書き込む．Flusher モジュールは，メモリベース KVS 上のデータ点を系列単位で定期的に取得し，ディスクベース KVS に書き込む．Querier モジュールは，クエリを受信および解析したのちに，必要な系列を取得するために，メモリベース KVS とディスクベース KVS にそれぞれアクセスし，結果を統合して，クエリ送信者に返却する．

メモリベース KVS 上でのデータ構造は，系列名をキーとして，系列名に対応するデータ点列をバリューに格納する．データ点列は，64 ビット整数のタイムスタンプと 64 ビット浮動小数点数の値の組が連続したバイナリである．データ点の挿入時には，バイナリの末尾にデータ点を追記する．データ点がタイムスタンプ順に整列されていることは保証されないため，読み出し時にデータ点を整列する．モニタリングの場合，時系列データベースに対して基本的にタイムスタンプの昇順にデータ点を送信するため，データ点列はほぼ整列された状態となる．したがって，多くの要素がすでに整列済みの状態のときに高速となる挿入ソートなどの整列アルゴリズムにより高速に整列する．

HeteroTSDB は，メモリベース KVS からディスクベース KVS へデータを移動させるために，TTL によりキー単位でタイマを設定する．Ingester モジュールはメモリベース KVS にデータ点を挿入するときに，当該系列を格納するキーに TTL を設定し，TTL の値が 0 になると，その通知を受けた Flusher モジュールが当該キーとキーに紐づく系列をディスクベース KVS へ移動させる．

挿入スループットを向上させるために，データの移動の際，Flusher モジュールはメモリベース KVS 上の各データ点列をそれぞれ単一のトランザクションとしてまとめて，ディスクベース KVS 上のキーへ挿入する．そのため，メモリベース KVS に対して 1 つのデータ点につき 1 つのトランザクションという対応付けに対して，ディスクベース KVS に対してその時点でメモリベース KVS 上に存在するデータ点列につき，1 つのトランザクションという対応付けとなる．これにより，ディスクベース KVS へのトランザクション単位のスループットがメモリベース KVS へのそれよりも小さくても，トランザクションではなくデータ点単位の挿入スループットの観点では，ディスクベース KVS への挿入スループットが向上する．その結果，ディスクベース KVS への挿入スループットがシステム全体の挿入処理を律速を防げるため，メモリベース KVS のメモリ使用量が上限に到達することを避けられる．

HeteroTSDB は，メモリベース KVS とディスクベース KVS に対して，残りの TTL を利用して，クエリを処理するために必要な時間範囲のデータがいずれの KVS にあるかを判定する．Querier モジュールが受信したクエリに含まれる時間範囲の区間が，クエリの受信時刻と受信時刻から当該レコードのタイマの経過時間を差し引いた時刻の間の区間を超えないのであれば，クエリを処理するためのデータ点はメモリベース KVS にしかないことが保証される．よって，当該区画の条件を満たした場合は，メモリベース KVS のみに問い合わせる．必要な時間範囲のデータ点がメモリベース KVS 上には存在せず，ディスクベース KVS 上にのみ存在する場合は，Querier モジュールは，ディスクベース KVS へ問い合わせたとしても，メモリベース KVS 上の TTL を直接知る術がないことから，データ点がディスクベース KVS 上にのみ存在するかどうかを判別できない．そこで，メモリベース KVS とディスクベース KVS の両者へ同時に問い合わせ，両者の応答を統合したのちに，問い合わせ元に返却する．タイマの経過時間を利用することにより，必要なデータ点がいずれの KVS にあるかを管理するためのデータ構造を追加せずに済む．

3.2 実装

本実装では，メモリベース KVS としてクラスタ管理機能を持つ Redis Cluster（Redis 6.0.4）およびディスクベース KVS として広く利用されている Cassandra（Apache Cassandra 3.0.20）を採用する．Redis はキーとバリューを格納するためのデータ構造として，ハッシュ表を利用しているため，系列数に対して定数時間で書き込めるため，HeteroTSDB の要件を満たす．

Redis 上に保持するデータ量を低減させるために，Redis のキーとして，系列名をそのまま格納するのではなく，系列名に対するハッシュ値を格納する．Redis の文字列型は，Redis のクライアント側で事前に文字列に変換しなくても，どのような種類のデータも格納できるため，Redis の文字列型のバリューの末尾に数値の組であるデータ点を追記する．データ点の追記操作のために，データをバリューの末尾に追記するための APPEND コマンドを利用する．Querier モジュールは，クエリに指定された時間範囲内のデータ点がメモリベース KVS 上にあると判定すると，当該系列内のデータ点を Redis からすべて取得したのちに，指定範囲外のデータ点を取り除く．

TTL によりデータ移動するために，APPEND コマンドの直後に当該系列に TTL が設定されていなければ，SET コマンドにより TTL を設定する．TTL がすでに設定されている場合，TTL コマンドにより TTL の残り秒数を取得し，取得した値があらかじめ設定した閾値を下回っていれば，Redis 上のキューとして利用可能なデータ型である Redis ストリームに当該系列を挿入する．Flusher モジュール内の複数のワーカースレッドが当該ストリームを購読しておき，いずれかのワーカースレッドが通知内容を受信したのち，Cassandra へ書き込む．

本論文の執筆時点（2020 年 6 月 8 日）で，HeteroTSDB の実装を Xtsdb と名付け，GitHub で開発を行っている[*1]．

4. 実験

HeteroTSDB の有効性を確認するために，3.2 節で示した実装が動作する環境で，挿入スケールアウト性および挿入スループットを評価した．実験ホストのハードウェア構成は**表 1** のとおりである．実験環境をさくらのクラウド[*2]上の仮想ホストを用いてクライアントを 1 台，データベースは実験に応じて同一構成のホストを複数個用意した．各実験ホストの OS はすべて Ubuntu 18.04.4 Kernel 4.15.0 であり，各仮想ホスト間の帯域は 1 Gbps である．比較対象となる KairosDB のバージョンは 1.2.2，KairosDB が依存する Cassandra のバージョンは 3.0.20 である．実験に利用した各設定ファイルは，GitHub リポジトリ[*3]に公開している．KairosDB [16] はクエリを受信するアプリケーションプロセスとデータを格納する Cassandra プロセスの 2 つのプロセスにより構成される．ハードウェアリソース条件を同一にするために，HeteroTSDB は Ingester，Flusher，Redis および Cassandra の各プロセスを単一のホスト上に同居する形で配置し，KairosDB についても同様にアプリケーションプロセスと Cassandra プロセスを同居させた．

KairosDB では，ともに Java 仮想マシン（以降，JavaVM とする）で動作するアプリケーションプロセスと Cassandra プロセスのヒープメモリ量をそれぞれデフォルト値の 512 MB と 1,024 MB とした．HeteroTSDB では，同居する Redis と Cassandra のうち，Redis により多くのメモリを利用させ，より多くの蓄積されたデータ点を集約することにより，ディスクベース KVS への挿入スループットを向上させることができる可能性がある．そのため，Cassandra のヒープメモリ量をデフォルト値よりも小さい 512 MB とした．

表 1 実験環境
Table 1 Experiment environment.

	項目	仕様
クライアント	CPU	Intel Xeon CPU E5-2650 v3 2.30 GHz 1core
	Memory	2 GBytes
	Disk	40 GBytes SSD
データベース	CPU	Intel Xeon CPU E5-2650 v3 2.30 GHz 2core
	Memory	4 GBytes
	Disk	40 GBytes SSD

[*1] https://github.com/yuuki/xtsdb
[*2] https://cloud.sakura.ad.jp/
[*3] https://github.com/yuuki/xtsdb-experiments (commit: 49f679a)

実験では，実環境のデータセットを模した時系列データ専用のベンチマークツール influxdb-comparisons[*4]が比較対象の時系列データベースに対して負荷を与えることにより，性能を計測した．influxdb-comparisons では，モニタリング対象ホスト数を表すスケール変数（-scale-var オプション）を設定することにより，生成するデータセットの系列数を変化させられる．スケール変数が 1 増えるごとに系列数が 100 増加する．前の試行の結果が影響しないように，各計測の試行前に各 KVS に蓄積されたデータを初期化した．各試行においてベンチマークの計測時間は 30 分とした．また，ベンチマーク計測中にデータ移動の負荷を発生させるために，HeteroTSDB の TTL を 30 分より十分に小さい 10 分に設定した．HeteroTSDB と KairosDB に対してデータ点の送信するためのプロトコルとして，TCP 上のテキストベースの Graphite プロトコル[*5]を採用した．

4.1 挿入スケールアウト性能

データの挿入効率を評価するために，HeteroTSDB と KairosDB それぞれについて同一ハードウェアリソース条件の挿入スループットを計測した．さらに，挿入スケールアウト性能を評価するために，処理ホストが増加した場合にスループットがどの程度向上するかどうかを実験した．スケールアウトのために複数のホストでクラスタを構成する際に，Redis と Cassandra の各クラスタでは，データの複製を持たないように設定した．よって，Redis はリーダノードのみのクラスタとなり，リーダの複製データを持つフォロワノードを持たない．また，HeteroTSDB と KairosDB の両者について，前述の同居構成のホストを増加させ，各ホスト上の Redis プロセスと Cassandra プロセスをそれぞれホストをまたいでクラスタ構成をとるように設定した．さらに，influxdb-comparisons は複数の接続先を指定できないため，各ホストに対して接続を分散させるために，オープンソースのロードバランサソフトウェアである HAProxy[*6]をクライアントサーバ上に配置した．

HeteroTSDB と KairosDB それぞれについて，スケール変数を 10,000 に設定した合計 100 万系列のデータセットに対して，処理ホスト数を 1 台から 8 台まで増加させたときの挿入スループットの変化を**図 3** に示す．Redis は 2 ノー

[*4] https://github.com/influxdata/influxdb-comparisons
[*5] https://graphite.readthedocs.io/en/latest/feeding-carbon.html
[*6] http://www.haproxy.org/

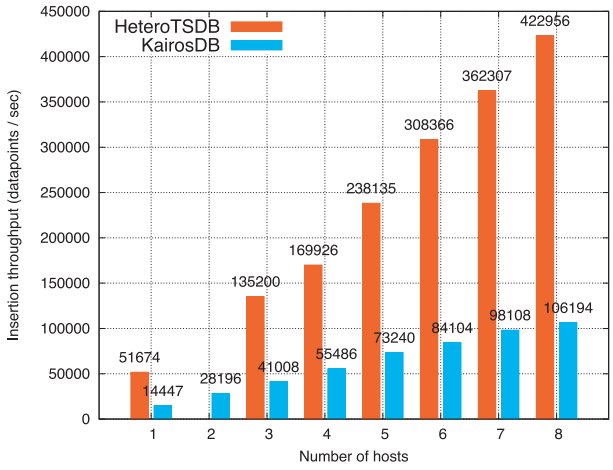


図 3 ホスト数に対する挿入スループットの変化
Fig. 3 Insertion throughput with the number of hosts.

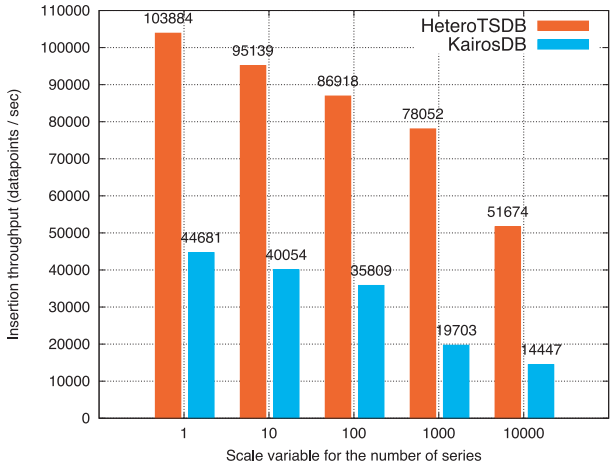


図 4 系列数に対する挿入スループットの変化
Fig. 4 Insertion throughput with the number of series.

ドでのクラスタを構成できないため，処理ホスト数が 2 台のときに HeteroTSDB の挿入スループットは計測していない．ホストの台数を増やすに従い，スループットが増加し，8 台のホストのスループットは，HeteroTSDB が 420 k（datapoints/sec），KairosDB が 11 k（datapoints/sec）となり，HeteroTSDB は KairosDB に対して 3.98 倍高速に動作した．

4.2 系列数を変化させたときの挿入スループット

系列数の増加に対する挿入効率を評価するために，HeteroTSDB と KairosDB のそれぞれについて，データセットの系列数を変化させたときの挿入スループットを計測した．スケール変数を 1 から 10,000 まで 10 倍ずつ増加させたときの処理ホスト 1 台での挿入スループットの変化を図 4 に示す．スケール変数すなわち系列数を増加させると，HeteroTSDB と KairosDB のいずれもスループットが低下した．HeteroTSDB のスループットは KairosDB と比較してスケール変数 1 のときに 2.3 倍，スケール変数 10,000 のときに 3.6 倍となったことから，HeteroTSDB は KairosDB と比較して系列数の増加に対する挿入スループットの低下率が小さい．

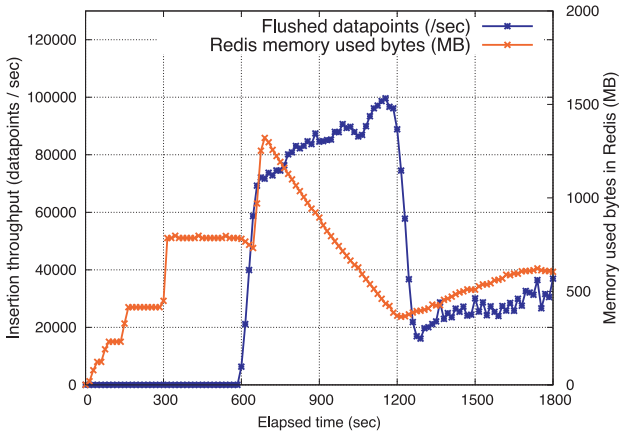


図 5 挿入スループットとメモリ使用量の時間変化
Fig. 5 Insertion throughput and memory used bytes.

4.3 Flusher モジュールの性能

メモリベース KVS からディスクベース KVS へのデータ移動，すなわち Flusher モジュールの処理性能を評価するために，ディスクベース KVS への挿入スループット，および，メモリベース KVS のメモリ使用量のそれぞれの時間変化を示す．**図 5** に，ホスト数が 1 のときのベンチマークの計測開始時から終了までの経過時間を横軸とした計測値を示す．図 5 より，ディスクベース KVS への挿入スループットは 600 秒後の TTL が 0 となるまでは 0 のまま推移したのちに，最大 100 k（datapoints/sec）のスループットとなり，データ移動開始後から計測終了時点までの平均スループットが 52 k（datapoints/sec）となった．これらのスループットは，図 3 のホスト数が 1 のときのスループットである 51 k（datapoints/sec）を超えており，ディスクベース KVS へのデータ移動のスループットがメモリベース KVS へ流入するスループットよりも大きいことを示す．また，Redis のメモリ使用量は 600 秒後までは増加を続け，そこから 645 秒後まで減少し続けた後に 690 秒後に 1,385 MB まで増加し続け，1,215 秒後の 382 MB まで減少し続けた．これらの結果より，Flusher モジュールは，メモリベース KVS のメモリ使用量を上限値まで増加させ続けることなく，データ点をディスクベース KVS へ移動したといえる．

5. 考察

5.1 挿入スループットの評価

4.1 節のスケールアウト実験において，HeteroTSDB の秒間挿入スループットは，ホスト 1 台のときに 52 k，ホスト 8 台のときに 420 k となっている．したがって，ホスト数を 8 倍にした結果，スループットがおおむね 8 倍となっていることから，HeteroTSDB はホスト数に対して線形に

挿入スケールアウト可能といえる．

4.3 節の実験結果により，ディスクベース KVS 単体の挿入スループットの最大値がメモリベース KVS のそれを超える結果となった．この結果をふまえると，KairosDB などのディスクベース KVS 単体の時系列データアプリケーションと比較し，HeteroTSDB はディスクベース KVS への挿入効率を向上させることができていることから，ディスクベース KVS 自体は低速なストレージを利用することもできる．さらに，本実験では 10 分に設定した TTL 値を増加させることにより，メモリベース KVS 上のデータ使用量と引き換えに，ディスクベース KVS への系列の挿入回数を低減可能である．したがって，ディスクベース KVS のストレージとして I/O が低速だが，容量単価の小さいものを利用しやすい．

HeteroTSDB が KairosDB よりも高スループットとなった要因について，各 KVS 上のインデックス構造の違いがあげられる．Redis はキーのアクセスにハッシュ表を採用していることから，系列数に対して定数時間で挿入・更新可能な一方で，Cassandra は平衡二分探索木を利用することから，挿入・更新に対数時間を要する．実際，4.2 節の系列数を変化させる実験において，HeteroTSDB が KairosDB に対してスループット低下率が小さいという結果になった．しかし，Redis は系列数に対して定数時間アクセス可能であるならば，HeteroTSDB は系列数増加に対してスループットが変化しないはずである．この結果の要因は，HeteroTSDB の KVS 間のデータ移動時に系列の個数分だけ Redis からデータを取得・削除処理が実行されることから，系列数が増加すると Redis 上の挿入処理をブロックする時間が増加することではないかと考えている．

図 5 における Redis のメモリ使用量の変化を示すグラフについて，300 秒後までに，Redis のメモリ使用量が短時間で急激に増加している．その理由は，メモリ確保のための処理オーバヘッドを低減させるために，Redis は必要な分だけそのつどメモリを確保するのではなく，直近のメモリ使用量の増加量に応じた量のメモリを先んじて確保するためであると考える．

図 5 における Redis のメモリ使用量の変化を示すグラフについて，300 秒後までに，Redis のメモリ使用量が短時間で急激に増加している．その理由は，メモリ確保のための処理オーバヘッドを低減させるために，Redis は必要な分だけそのつどメモリを確保するのではなく，直近のメモリ使用量の増加量に応じた量のメモリを先んじて確保するためであると考える．また，Redis のメモリ使用量が 690 秒後から即座に減少せずに，1,200 秒後まで漸減した理由は，3.2 節で示した実装で，突発的なデータ移動による負荷上昇を避けるために，TTL をすべてのキーで同じタイミングに 0 にせず，TTL にゆらぎを加えるためである．また，データ移動を開始するタイミングである 600 秒後から 1,200 秒後と，1,200 秒後から 1,800 秒後まで 2 つの期間を比較すると，各計測地の変動傾向が異なっている．

5.2 メモリベース KVS とディスクベース KVS の選択性

HeteroTSDB 自体はどのキーをどのノードに紐付けるかを管理するわけではないことから，各 KVS にキーの紐付け管理を含むクラスタ管理機能が必要となる．

HeteroTSDB で選択するメモリベース KVS は，到着するデータ点を順次書き込む必要があることから，キーに対してデータ点を表すバイナリの追記書き込みが可能である必要がある．3 章で示したように，キー単位で TTL を設定することによる KVS 間のデータ階層化を実現していることから，メモリベース KVS にはキーに対して TTL を設定できるようにする必要がある．ただし，モニタリングでは基本的にデータ点を固定のインターバルで受信するため，キー単位で系列内のデータ点の個数を計上しておき，個数の閾値と現在の個数の差分をタイマと見なすこともできるため，TTL を設定可能であることは必須条件ではない．しかし，OSS（Open Source Software）の範疇においては，開発されているメモリベース KVS の種類が少ないことから，実質的に Redis が有力な選択肢となる．

ディスクベース KVS については，メモリベース KVS 上のキーに格納された系列をディスクベース KVS 上のキーにそのまま格納すればよい．したがって，RDB やファイルシステム上のファイルやオブジェクトストレージ上のオブジェクトと系列を 1 対 1 対対応させて格納することも可能である．データ階層化により，挿入回数を低減できていることから，挿入処理効率の低いストレージを利用しやすい．現時点の実装では，実装の利用者が利用するストレージに対応するには，Flusher モジュールを直接修正しなければならない．プラグイン形式により，利用者が各ストレージに対応する実装を疎結合に拡張できることが望ましいため，プラグイン化の実装は今後の課題とする．

5.3 実運用上の制約

メモリベース KVS を利用するうえで，ノードあるいは OS プロセスの停止時に，メモリ上のデータが揮発することにより，メモリベース KVS 上のデータを消失する危険性がある．この危険性を回避するための 1 つの方法は，コミットログが永続化されるメモリベース KVS を選択することである．一部のメモリベース KVS は，コミットログのみをディスク上に永続化したうえで，データの本体をメモリ上に保持する．たとえば，HeteroTSDB の実装に使用した Redis は，AOF（Append Only File）[*7]と呼ばれるデータベースのコミットログを永続化するための機能を備えている．永続化されたコミットログにより，メモリ上のデー

[*7] Redis Persistence - https://redis.io/topics/persistence

タが消失したとしても，コミットログからメモリ上にデータを読み込むことにより，データを復旧できる．

データ消失を回避するためのもう 1 つの方法は，レプリケーション機能を持つメモリベース KVS を選択することである．Redis などの一部のメモリベース KVS は，複数のノード間でデータをリアルタイムに複製するためのレプリケーション機能[*8]を備えている．これにより，ある 1 つのノードに障害が発生したとしても，別のノードが複製を持つため，データの消失を避けられる．以上により，メモリベース KVS のデータの消失の危険性を回避するために，コミットログ永続化機能あるいはレプリケーション機能を持つメモリベース KVS を選択する必要がある．

KairosDB が利用する Cassandra では，memtable 全体のサイズの増加を監視することから，メモリ上のデータがサーバの搭載メモリ量を超えないように制御しやすい．一方で，HeteroTSDB では系列単位で TTL を設定することから，系列がメモリ上に滞在する時間を設定できるものの，合計のメモリ量をもとにデータ移動頻度を制御することはできない．しかし，定期的に Redis のメモリ使用量を監視し，TTL 値を動的に増減させるなどの工夫は可能である．メモリ使用量を超過させないように制御しながら，高速にディスクへ移動するためのデータ階層化の実現については，今後の課題とする．

HeteroTSDB は異種の分散 KVS を利用することを前提とするため，システム管理者は各分散 KVS のクラスタ管理に習熟しなければならないという欠点がある．しかし，メモリベース KVS の有力選択肢である Redis は Web サービス企業を中心に広く利用されている KVS であることから，システム管理者は Redis にすでに習熟している可能性が高い．さらに，ディスクベース KVS は HBase や Cassandra だけでなく，既存の RDB，ファイルシステムやオブジェクトストレージからも選択可能であることから，システム管理者がすでに習熟している KVS を選択可能である．また，クラウド事業者が提供している KVS のクラスタを自動で構築・管理するマネージドサービス [1], [2], [3] を利用することにより，システム管理者の負担を低減できる．

5.4 検索性能の議論

HeteroTSDB はメモリベース KVS 上にタイムスタンプが新しいデータ点を配置するため，直近のデータ点を検索する限りでは，ディスク上に配置されたデータ点を検索するよりも高速となる．しかし，3.1 節で述べたように，HeteroTSDB の KVS 上のキーに格納されるデータ点の整列順はデータ点の到着順となるため，タイムスタンプにより整列されている保証はない．そのため，KVS からデータ点を取得したのちに整列する必要がある．したがって，既存のディスクベース KVS のようにメモリ上のデータ構造に平衡木を採用することにより，タイムスタンプ順であることを保証する場合と比較し，整列処理の分だけ検索性能が低下する．

本論文で着目するモニタリングシステムにおける検索は，システム管理者がシステム異常の調査のために時系列グラフを表示するタイミングで発生する．あるいは一部のメトリックの異常を検知しシステム管理者に通知するために，一部のメトリックの時系列データを 1 分ごとなどの頻度で定期的に検索する．そのため，すべてのメトリックに対して定常的に発生する挿入処理の負荷と比較し，検索処理に要する負荷は相対的に小さいといえる．

しかし，今後のモニタリングシステムにおいて，機械学習を利用した時系列データの解析などの要求が生まれることを想定すると，繰返しの学習のために検索回数や検索対象のメトリック数が増加する可能性があることから，検索のための負荷を低減させる必要がある．したがって，HeteroTSDB の検索性能の評価とその向上については今後の課題とする．

6. 実環境への適用

2017 年 8 月に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事例を示す．

3.2 節で示した実装と実環境における実装との間には次のような差異がある．

- メモリベース KVS のクラスタ全体の故障時にデータが消失する可能性に対処するために，メッセージブローカ [19] を Ingester モジュールの前段に配置した．
- ディスクベース KVS には Amazon DynamoDB [1] を採用し，その後段にオブジェクトストレージの Amazon S3 [3] を採用し，3 階層の構成をとった．
- Ingester モジュールと Flusher モジュールは，AWS Lambda [4] 上で動作させた．

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は 2 件ある．1 件目では，メモリベース 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．その後，メッセージブローカ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 Ingester モジュール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．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持つ想定以上の個数のデータ点が含まれた挿入 API 呼び出しが複数回誤って発行された．その結果，Redis Cluster に対する挿入クエリは，本来であれば少数のデータ点のみを含むはずのところを大量のデータ点を含むようになったことから，Redis Cluster に対する

[*8] Redis Replication - https://redis.io/topics/replication

挿入クエリのサイズが実装上の上限を超えたためにエラーを返した．そのため，Ingester モジュールからの実行が再試行され続け，Ingester モジュール全体の処理が遅延した．Redis Cluster へのデータ点の挿入前に，同一メトリック名かつ同一タイムスタンプを持つデータ点の重複を除去したうえで，Redis Cluster への挿入クエリを構築するようにプログラムを修正することにより解決した．

7. まとめ

本論文では，多数のシステム構成要素からメトリックを収集する際に，高挿入効率と高挿入スケールアウト性を達成するために，メモリベース KVS とディスクベース KVS を階層化する時系列データベース HeteroTSDB を提案した．実験の結果，1 m 個の系列を含むデータセットに対して，既存技術である KairosDB と比較し，HeteroTSDB が同一ハードウェアリソース条件下で処理ホスト数が 8 のときに 3.98 倍の挿入スループット向上を達成した．

本論文の Flusher モジュールの性能評価では，ディスクベース KVS に対する挿入トランザクションをまとめることによる高速化の要因，OS カーネルのページキャッシュによる性能への影響，および，HDD ではなく SSD を利用することの妥当性が明確となっていない．これらの評価を追加することを今後の課題とする．

今後は，HeteroTSDB のオープンソースでの開発を進め，広く利用可能な実装として成熟させるつもりである．また，自動階層化に必要な TTL などのパラメータを各モジュールの負荷状況に応じて自動で調整可能なアーキテクチャを考えていく予定である．

参考文献

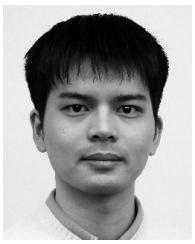
[1] Amazon DynamoDB, available from ⟨https://aws.amazon.com/dynamodb/⟩.
[2] Amazon ElastiCache for Redis, available from ⟨https://aws.amazon.com/elasticache/redis/⟩.
[3] Amazon S3, available from ⟨https://aws.amazon.com/jp/s3/⟩.
[4] AWS Lambda, available from ⟨https://aws.amazon.com/jp/lambda/⟩.
[5] Mackerel, available from ⟨https://mackerel.io/⟩.
[6] VictoriaMetrics, available from ⟨https://victoriametrics.com/⟩.
[7] Abe, H., Shima, K., Sekiya, Y., Miyamoto, D., Ishihara, T. and Okada, K.: Hayabusa: Simple and Fast Full-Text Search Engine for Massive System Log Data, *12th International Conference on Future Internet Technologies* (*CFI*), pp.1–7 (2017).
[8] Aceto, G., Botta, A., De Donato, W. and Pescapè, A.: Cloud Monitoring: A Survey, *Computer Networks*, Vol.57, No.9, pp.2093–2115 (2013).
[9] Bader, A., Kopp, O. and Michael, F.: Survey and Comparison of Open Source Time Series Databases, *Datenbanksysteme für Business, Technologie und Web (BTW) – Workshopband*, pp.249–268 (2017).
[10] Brazil, B.: *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*, O'Reilly Media, Inc. (2018).
[11] Carpenter, J. and Hewitt, E.: *Cassandra: The Definitive Guide: Distributed Data at Web Scale*, O'Reilly Media, Inc. (2020).
[12] 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, *ACM Trans. Computer Systems*, Vol.26, No.2, pp.4:1–4:26 (2008).
[13] George, L.: *HBase: The Definitive Guide: Random Access to Your Planet-Size Data, 1st edition*, O'Reilly Media, Inc. (2011).
[14] Goldschmidt, T., Jansen, A., Koziolek, H., Doppelhamer, J. and Breivold, H.P.: Scalability and Robustness of Time-Series Databases for Cloud-Native Monitoring of Industrial Processes, *IEEE 7th International Conference on Cloud Computing*, pp.602–609 (2014).
[15] Han, J., Haihong, E., Le, G. and Du, J.: Survey on NoSQL database, *2011 Sixth International Conference on Pervasive Computing and Applications* (*ICPCA*), pp.363–366 (2011).
[16] Hawkins, Brian, KairosDB: Fast Time Series Database on Cassandra, available from ⟨https://kairosdb.github.io/⟩.
[17] InfluxData, InfluxDB, available from ⟨https://www.influxdata.com/time-series-platform/influxdb/⟩.
[18] Jensen, S.K., Pedersen, T.B. and Thomsen, C.: Time Series Management Systems: A Survey, *IEEE Trans. Knowledge and Data Engineering*, Vol.29, No.11, pp.2581–2600 (2017).
[19] John, V. and Liu, X.: A Survey of Distributed Message Broker Queues, arXiv preprint arXiv:1704.00411 (2017).
[20] Pelkonen, T., Franklin, S., Teller, J., Cavallaro, P., Huang, Q., Meza, J. and Veeraraghavan, K.: 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).
[21] Sanfilippo, Salvatore and Noordhuis, Pieter, Redis, available from ⟨https://redis.io⟩.
[22] The OpenTSDB Authors, OpenTSDB: The Scalable Time Series Database, available from ⟨http://opentsdb.net⟩.
[23] Zhang, H., Chen, G., Ooi, B.C., Tan, K.-L. and Zhang, M.: In-Memory Big Data Management and Processing: A Survey, *IEEE Trans. Knowledge and Data Engineering*, Vol.27, No.7, pp.1920–1948 (2015).
[24] 阿部 博，島 慶一，宮本大輔，関谷勇司，石原知洋，岡田和也，中村 遼，松浦知史，篠田陽一：時間軸検索に最適化したスケールアウト可能な高速ログ検索エンジンの実現と評価，情報処理学会論文誌，Vol.60, No.3, pp.728–737 (2019).

坪内 佑樹 （正会員）

2013 年大阪大学大学院情報科学研究科博士前期課程中途退学．同年株式会社はてな入社．2019 年よりさくらインターネット株式会社さくらインターネット研究所研究員．京都大学情報学研究科博士後期課程所属．クラウドのシステム運用技術を研究している．IEEE，ACM 各会員．

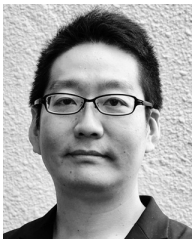
脇坂 朝人

株式会社はてなアプリケーションエンジニア，シニアエンジニア．



濱田 健

株式会社はてなアプリケーションエンジニア．



松木 雅幸

2014 年株式会社はてな入社を経て，2019 年より株式会社 Nature 取締役 CTO．

小林 隆浩

株式会社野村総合研究所．

阿部 博 （正会員）

トヨタ自動車株式会社．

松本 亮介 （正会員）

2015 年京都大学大学院情報学研究科博士課単位取得認定退学．同年 GMO ペパボ株式会社入社を経て．2018 年よりさくらインターネット株式会社さくらインターネット研究所上級研究員．京都大学博士（情報学）．OS やサーバソフトウェア，インターネットの運用技術やセキュリティ等に興味を持つ．IEEE，ACM 各会員．