HITACHI
Inspire the Next

分散型データストアApache Kuduで大量のIoTデータを処理してみた

2018年12月14日

株式会社日立製作所　OSSソリューションセンタ
伊藤雅博

自己紹介

- **伊藤 雅博 (いとう まさひろ)**
  - **所属： 株式会社日立製作所　OSSソリューションセンタ**
  - **業務：** Hadoop/Spark/Kafkaなど**ビッグデータ関連OSSの導入支援やテクニカルサポート**

  - **去年の発表（オープンソースカンファレンス** 2017.Enterprise）
    - **めざせ！**Kafka**マスター ～**Apache Kafka**で最高の性能を出すには～**
      - **検証結果の詳細**(Qiita): Apache Kafka**の概要とアーキテクチャ**

目次

1. Apache Kuduの概要とアーキテクチャ
2. 性能検証のシナリオ
3. 検証①: データ移行処理の検証
4. 検証②: オンライン処理の検証
5. まとめ

1. Apache Kuduの概要とアーキテクチャ

Apache Kuduとは

- 高いスケーラビリティを持つ分散型のデータストア
  - 多数のマシンでクラスタを構成することで大量のデータを扱うことができる
  - 高頻度のトランザクション処理と、大量データの分析処理の両方に優れている HTAP (Hybrid Transactional/Analytic Processing) という分野のデータストア
- Cloudera社が開発して2015年にOSSとして公開
- 基本的には、Hadoopと組み合わせて使用する
  - ビッグデータ処理の分野ではHadoopを中心としたOSSがデファクトスタンダード
  - KuduもHadoopエコシステムに適合するように設計
  - HDFSやHBaseと使い分け

HDFS / HBase / Kudu の比較

- Hadoop向けデータストアである HDFS, HBase, Kudu の比較

機能	HDFS	HBase	Kudu
データモデル	テーブル形式 (メタストア使用時)	ワイドカラム形式 （キーバリュー形式）	テーブル形式
データの更新単位	テーブル/パーティション単位 (ファイルごと入れ替え)	レコード単位	レコード単位
トランザクション	なし	行単位のACID トランザクションのみ	行単位のACID トランザクションのみ
得意なワークロード （性能面の比較）	大容量データの格納・参照 （バッチ処理向き）	高頻度なデータ格納・参照 （リアルタイム処理向き）	高頻度なデータ格納/ 大容量データの参照 （リアルタイム格納+バッチ参照）

- Kuduは性能面において、HDFSとHBaseの両方の特性を持つ
- 高頻度に発生するデータを格納しつつ、蓄積したデータを分析用に大量参照する用途向き

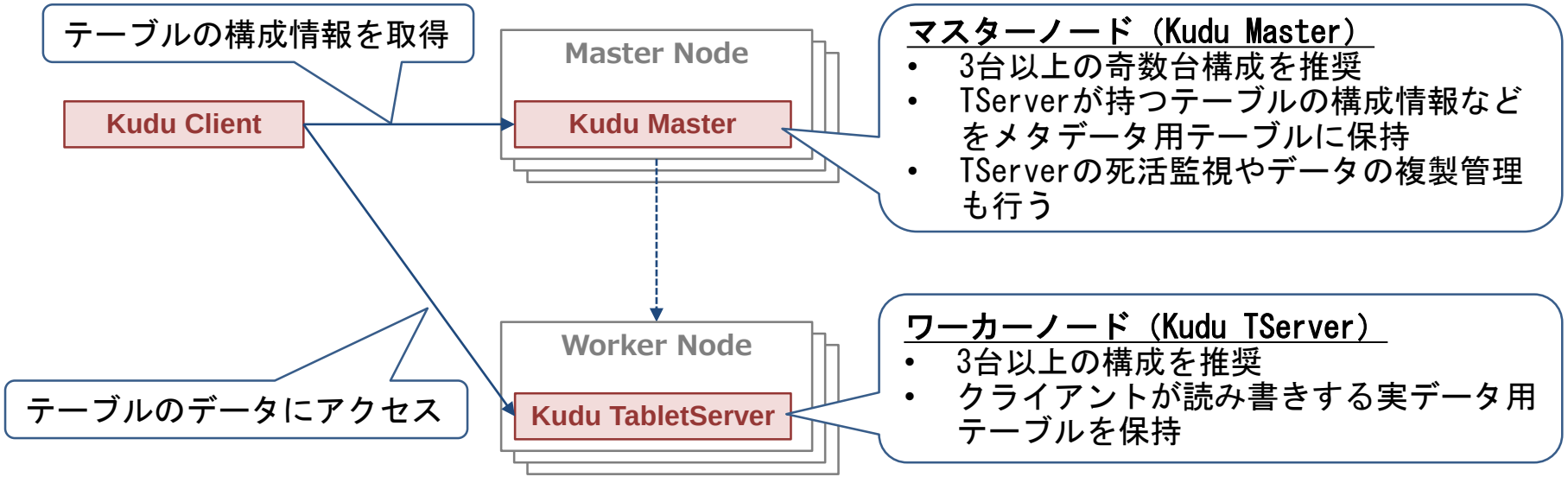
Kuduのインタフェース

- 厳密にいえば、Kuduはストレージエンジンであり、データの格納と参照のみを担当する
  - RDBMS のようにSQLでデータを集計・変換するような機能はない
    - HDFSやHBaseも同様
  - C++ / Java / Python のAPIによるデータの追加/更新/削除/スキャンのみ可能

- データの集計・変換を行うためには、クエリエンジン（Impala）や並列分散処理フレームワーク（Spark, MapReduce）を組み合わせる
  - Impalaを経由してKuduにJDBC/ODBC接続が可能
    - これにより様々なBIツールからKuduにアクセスできる

Kuduのシステム構成

- **マスターノードとワーカーノードでクラスタを構成**
  - **クラスタ構成により、大容量・高性能・高可用性のシステムを実現**
  - **ノード障害による動作停止やデータロストを避けるため、**
    Master / TabletServer (TServer) **共に最低でも3台以上の構成を推奨**



Kuduの論理データモデル: テーブル

- テーブルの各列は完全に型付け
- テーブルのレコードは必ず主キーを持ち、主キーでソートされた状態で保持

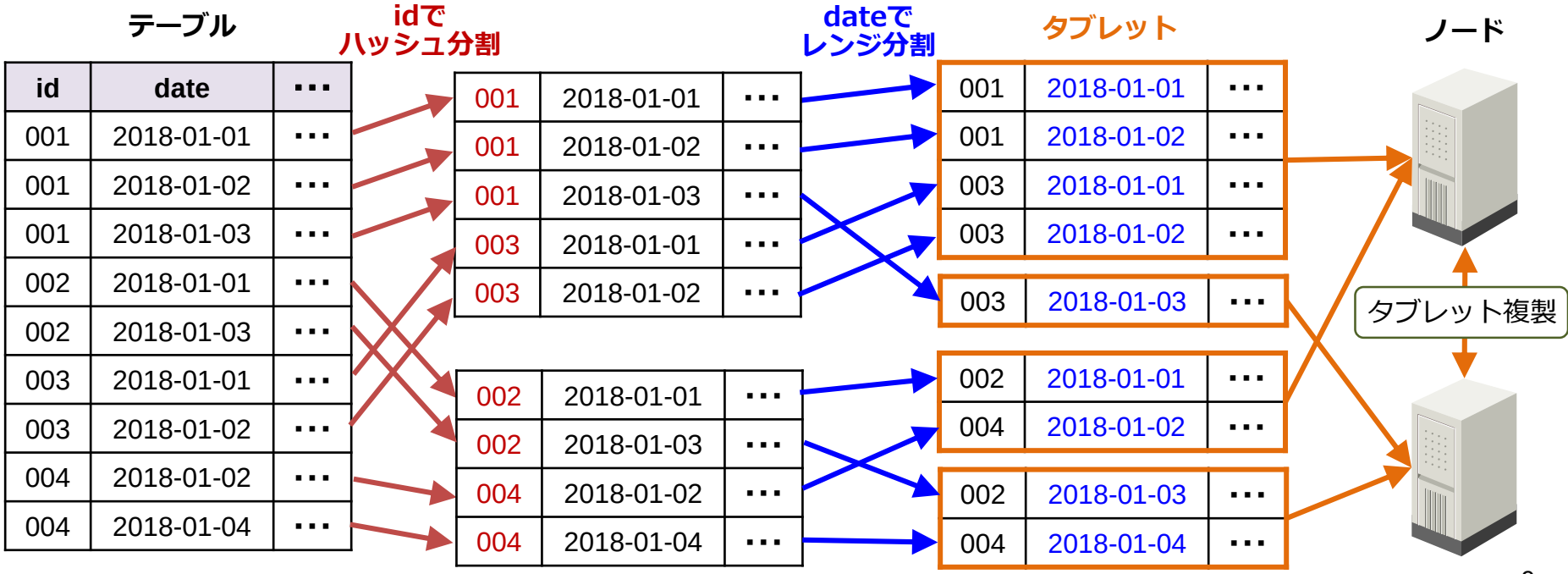
テーブル

Id (主キー1)	date (主キー2)	usage	cost	complete
001	2018-01-01	20.86	22,360	True
001	2018-01-02	124.23	182,345	True
001	2018-01-03	22.53	736	False
002	2018-01-01	30.01	5,842	True

- **レコードの書き込み操作**: Insert / Update / Delete / Upsert
  - 行単位のACIDトランザクションに対応。複数行にまたがる処理のコミットやロールバックは不可
- **レコードの読み出し操作**: Scan
  - ソートされた主キーの範囲指定で効率的にスキャン可能
  - 更新・削除前のレコードも保持しており、指定時刻のスナップショットを参照可能

Kuduの物理データモデル: タブレット

- 各テーブルを ハッシュパーティション × レンジパーティションでタブレットという単位に分割
  - このタブレットを各ノードに分散配置することで、分散処理が可能となる
  - さらに各タブレットをノード間で複製して可用性を確保
    - 各タブレットは1個のリーダーレプリカと、その複製である0個以上のフォロワーレプリカで構成
    - 実データ用テーブルはTServer同士、メタデータ用テーブルはMaster同士で複製



2. 性能検証のシナリオ

検証シナリオ: 電力消費量データの蓄積と集約

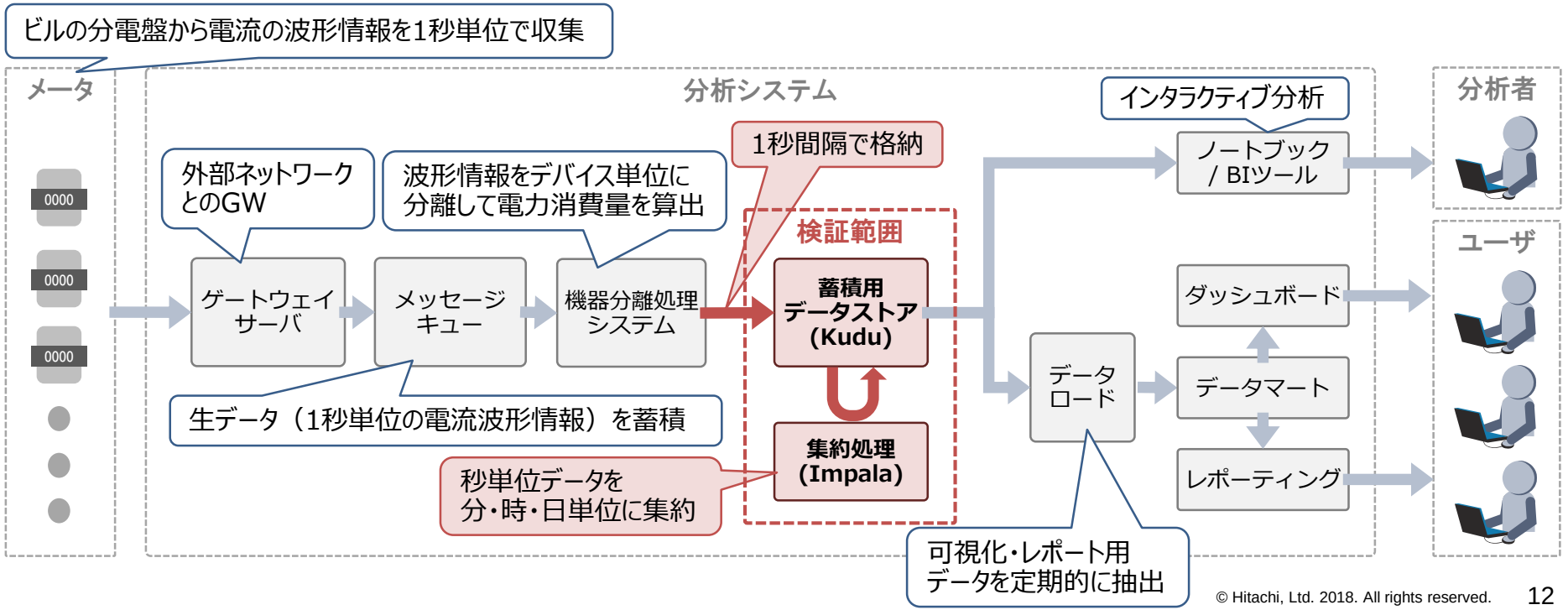
- 背景
  - 建物の分電盤から得られる電流の波形情報を分離することで、電力を消費する設備や家電機器ごとの動作状態を推定する「ディスアグリゲーション」技術が実用化
  - 電力消費量の見える化や生活行動の分析といったサービスが生まれている

- 課題
  - 機器単位の正確な動作状態を把握するためには、かなり短い間隔で電流データをサンプリングする必要あり
  - ディスアグリゲーション技術では1秒間隔のデータを使用することが多く、データ量は膨大になる

- Kuduは高頻度なデータ格納と、大量データの参照に優れているため、このようなデータを扱うのに適していると考えられる

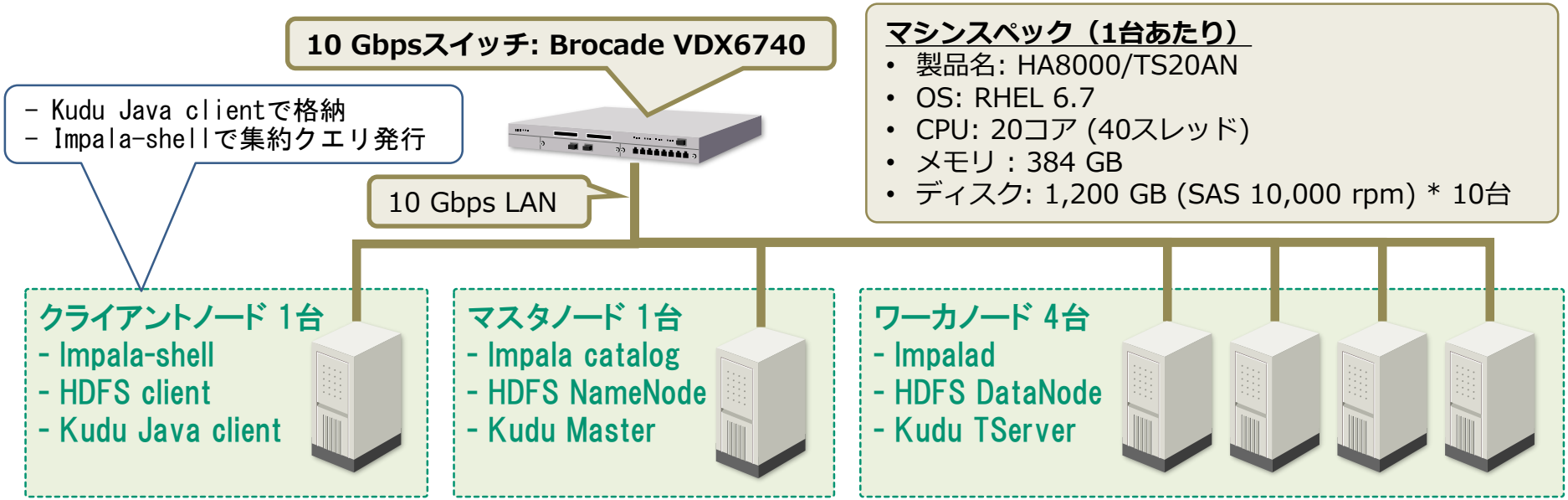
検証で想定するシステムの全体像

- ビル内の機器の電力消費量を蓄積し、その集約結果を分析や可視化に使用
  - このうち電力消費量データの蓄積と集約に Kudu + Impala を使用
  - 今回の検証範囲： Kudu に対するデータ格納と、Impalaによる集約処理の読み書き



検証環境

- クライアントノード、マスタノード、ワーカノードは同一スペックの物理マシン(HA8000／TS20AN)を使用
- **KuduはWAL用のディスクにSSDの仕様を推奨しているが、今回はすべてのディスクにHDDを使用**
- クライアントノード、マスタノード、ワーカノード間は10Gbps回線で接続
- 今回は性能検証であるためマスタノードの冗長化はしていない
- Cloudera Managerを用いて Kudu + Impala + HDFS クラスタを構築(CDH 5.12 / Kudu1.4 を使用)



テーブル設計

- 1秒、1分、1時間、1日間隔のデータを格納するテーブルを用意

テーブル一覧

#	テーブル名	説明
1	load_per_sec	機器の1秒単位の電力消費量レコードを格納
2	minutely_load	load_per_secテーブルのレコードを1分単位で合計したものを格納
3	hourly_load	minutely_loadテーブルのレコードを1時間単位で合計したものを格納
4	daily_load	hourly_loadテーブルのレコードを1日単位で合計したものを格納

テーブルスキーマ (全テーブル共通)

#	カラム名	キー	データ型	サイズ	説明
1	building_id	○	int32	4 bytes	建物を示すID
2	floor_id	○	int32	4 bytes	建物内の階層を示すID
3	device_id	○	int32	4 bytes	電力を消費する設備や機器を示すID
4	time_stamp	○	unixtime_micros	8 bytes	電力測定時の時刻
5	device_load		int64	8 bytes	電力消費量(Active power [watts])
6	device_type		int32	4 bytes	設備や機器の種別
				32 bytes	

各テーブルの想定データ量とパーティション定義

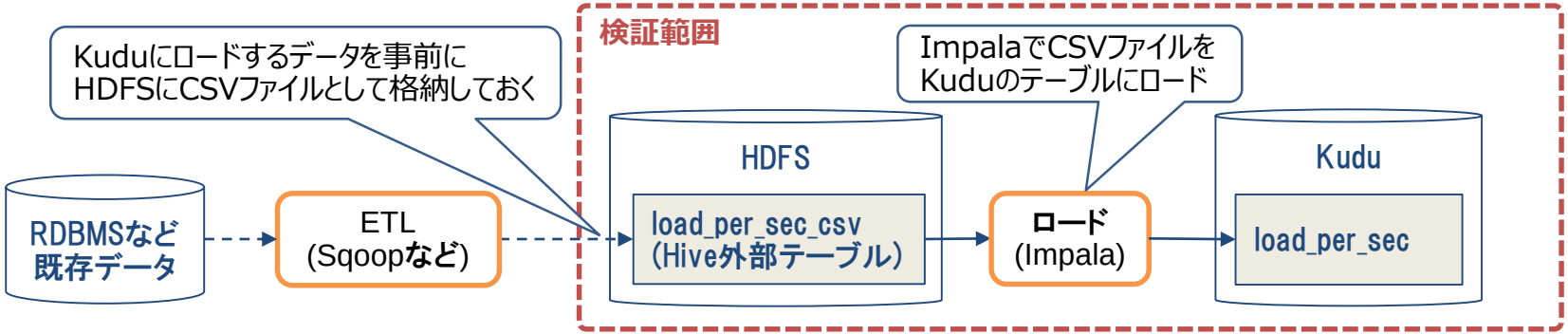
- 想定する最大データ量からパーティション定義を決定
  - 機器数: ビル10棟 × 1ビルあたり50フロア × 1フロアあたり200機器 = **10万機器** と仮定
  - 1タブレットあたりの最大データ量（圧縮後）が5GB程度となるようパーティションを定義

#	テーブル名	最大保持期間 (想定)	最大レコード数 (想定)	最大サイズ (想定)	パーティション設定	最大タブレットサイズ (想定)	最大タブレット数 (想定)
1	load_per_sec	秒単位 * 30日分	2,592億	定義上: 7,725 GB 圧縮後: 2,467 GB	ハッシュ: 16個 レンジ: 1日単位	5.1 GB	480個
2	minutely_load	分単位 * 5年分	2,628億	定義上: 7,832 GB 圧縮後: 2,501 GB	ハッシュ: 8個 レンジ: 1月単位	5.1 GB	480個
3	hourly_load	時単位 * 10年分	87億6,000万	定義上: 261 GB 圧縮後: 83 GB	ハッシュ: 4個 レンジ: 1年単位	2.1 GB	40個
4	daily_load	日単位 * 10年分	3億6,500万	定義上: 11GB 圧縮後: 4 GB	ハッシュ: 4個 レンジ: なし	0.9 GB	4個

3. 検証①: データ移行処理の検証

データ移行処理の検証内容

- 既存のシステムにあるデータをKuduに移行することを想定
  - まずデータをCSVファイルに書き出してHDFSに格納しておく
  - HDFSに格納したCSVファイルに対してHiveの外部テーブルを定義して、そのテーブルのデータをImpalaでKuduのテーブルにInsertする時間を測定

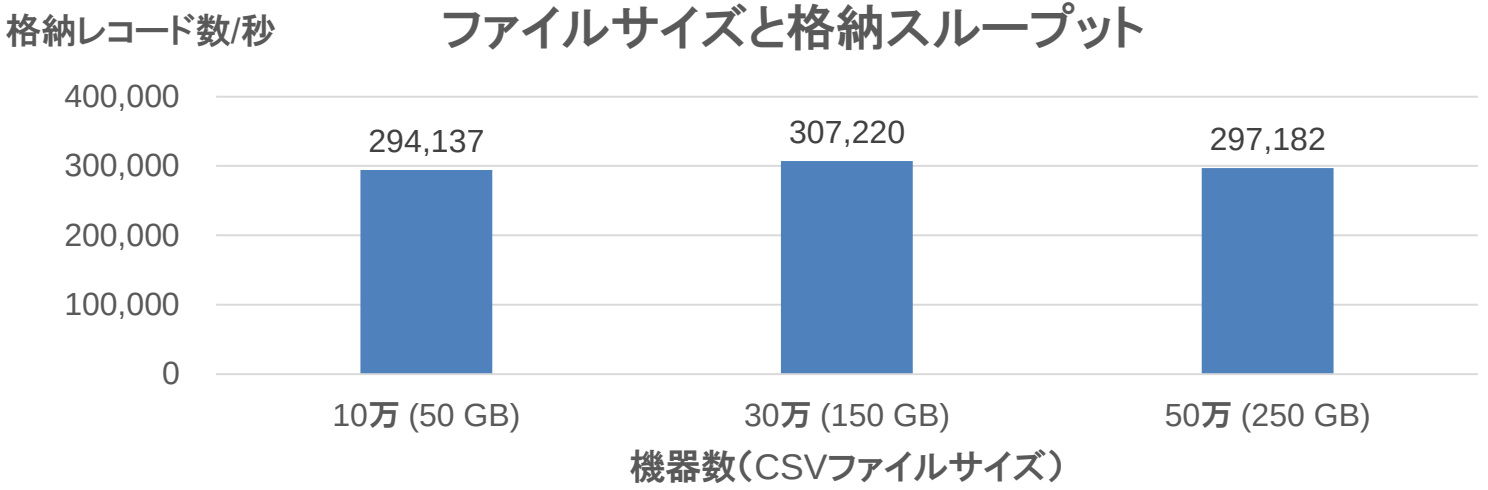


検証データサイズ

#	機器数	期間	レコード数	論理サイズ（バイナリ）	CSVサイズ（テキスト）
1	10万	6時間 (21,600秒)	21.6億	64.4 GB	50.2 GB
2	30万	6時間 (21,600秒)	64.8億	193.1 GB	150.5 GB
3	50万	6時間 (21,600秒)	108.0億	321.9 GB	250.8 GB

データ移行処理の検証結果: CSVファイルサイズと格納スループット

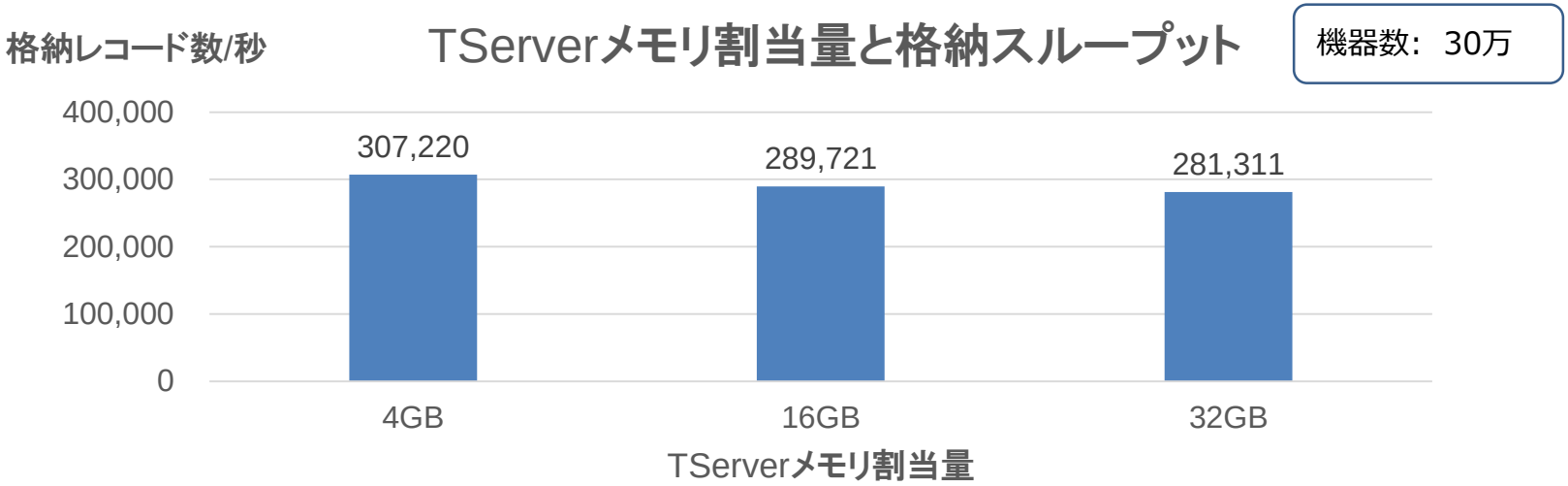
- CSVファイルをImpalaでKuduにInsert
- **機器数10万、30万、50万×6時間分のCSVファイルの格納時間を測定**



格納スループットは秒間約30万レコードでほぼ一定

データ移行処理の検証結果：TServerメモリサイズと格納スループット

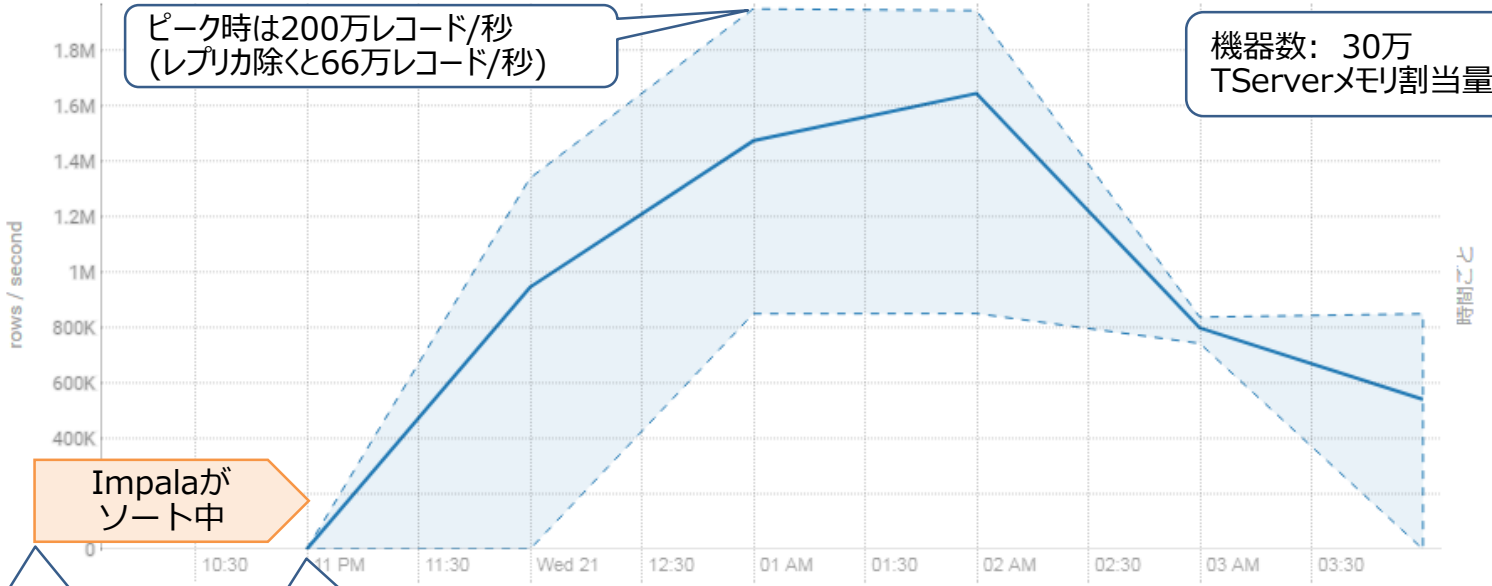
- チューニング：各TServerのメモリ割当量をデフォルト設定の4GBから16GB（推奨値）、および32GB（推奨値の2倍）まで増やしてみた



メモリ割当量を増やすと格納スループットが若干下がる傾向にあったが、
秒間約30万レコードでほぼ一定

データ移行処理の検証結果: 格納スループット(秒間格納レコード数)の推移

- 以下のスループットはレプリカの分も含むため、実際の格納スループットは1/3となる



- Impalaはメモリ上でデータをソートしてからKuduに格納するため、
  最初の73分間はKuduへの格納が発生していない？
- 平均格納スループットは30万レコード/秒だが、実際には一定ではないことがわかる

データ移行処理のチューニングポイント

- **基本的にKudu / Impala共チューニングは必要ない**

- Kudu
  - **Kuduのメモリ割当量を増やしても効果はなかった**

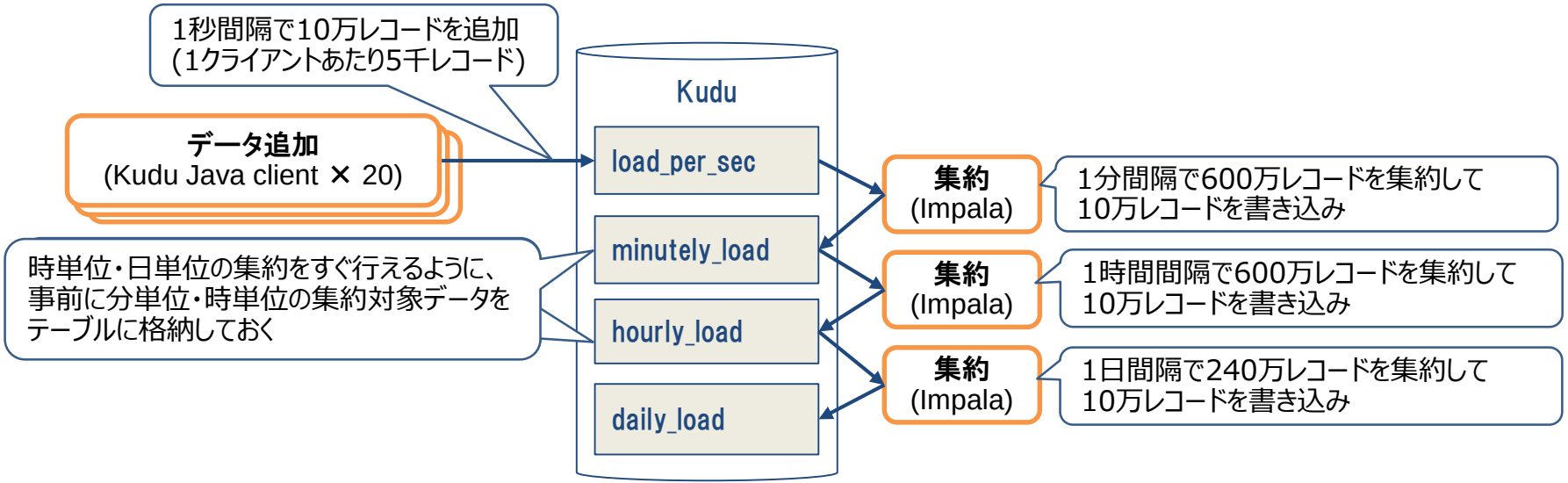
- Impala
  - **Impalaはデータをメモリ上でソートしてからKuduに格納するため、Kuduよりメモリを多く使用する**
  - **しかしImpala のメモリ割当量はCloudera Managerが多めに設定してくれるため、基本的に変更する必要はない**

4. 検証②: オンライン処理の検証

オンライン処理の検証内容

- 1秒間隔の格納処理と、1分/1時間/1日間隔の集約処理を同時に実行
  - 処理を一定期間実行して、格納および集約の遅延・失敗が発生しなければOKとする
  - 毎秒の格納レコード数を増やしていき、どこまで耐えられるかを確認

- 毎秒10万レコードを処理する場合の流れ：

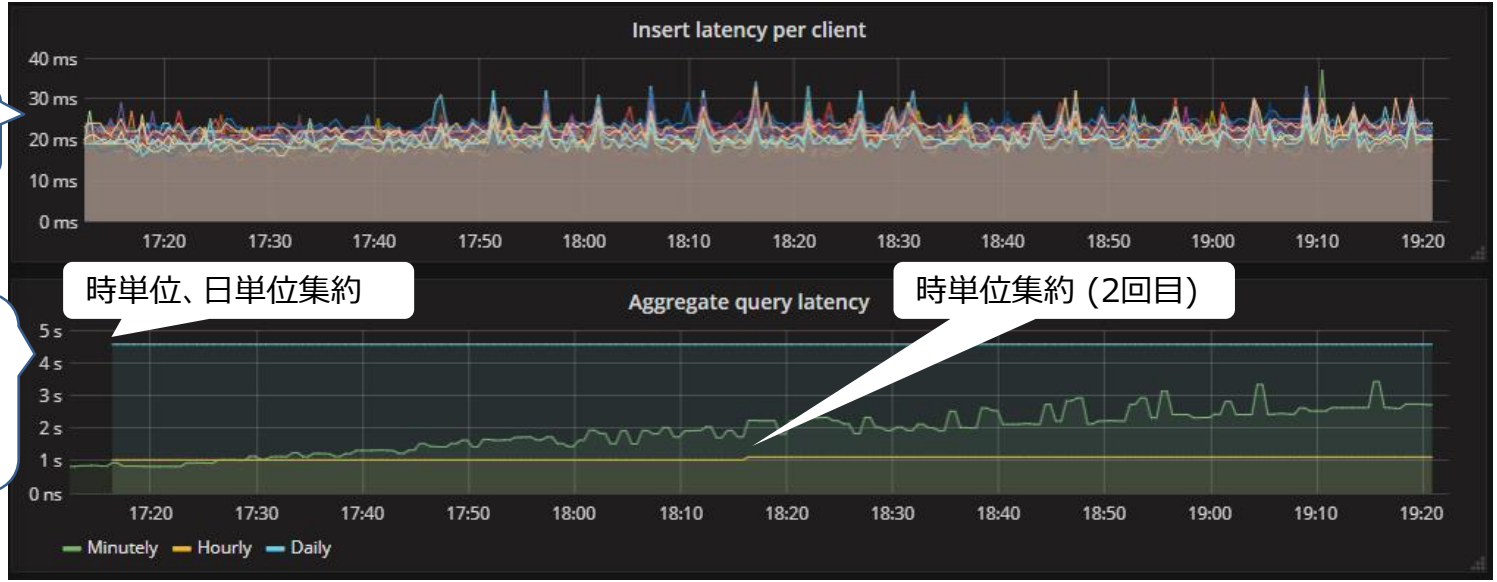


オンライン処理の検証結果: 毎秒10万レコード

- 処理を2時間流し続けて、格納リクエストと集約クエリのレイテンシを確認
  - 横軸は測定時の実時間(17:11– 19:21)であり、レコードのTimestamp(23:55 - 02:05)ではないことに注意
  - レコードの集約はTimestampに従って実行される

格納リクエストのレイテンシ
20-30msでほぼ一定

集約クエリのレイテンシ
- 分単位集約は約3秒まで徐々に増加
- 時単位集約は約1秒
- 日単位集約は約4.5秒

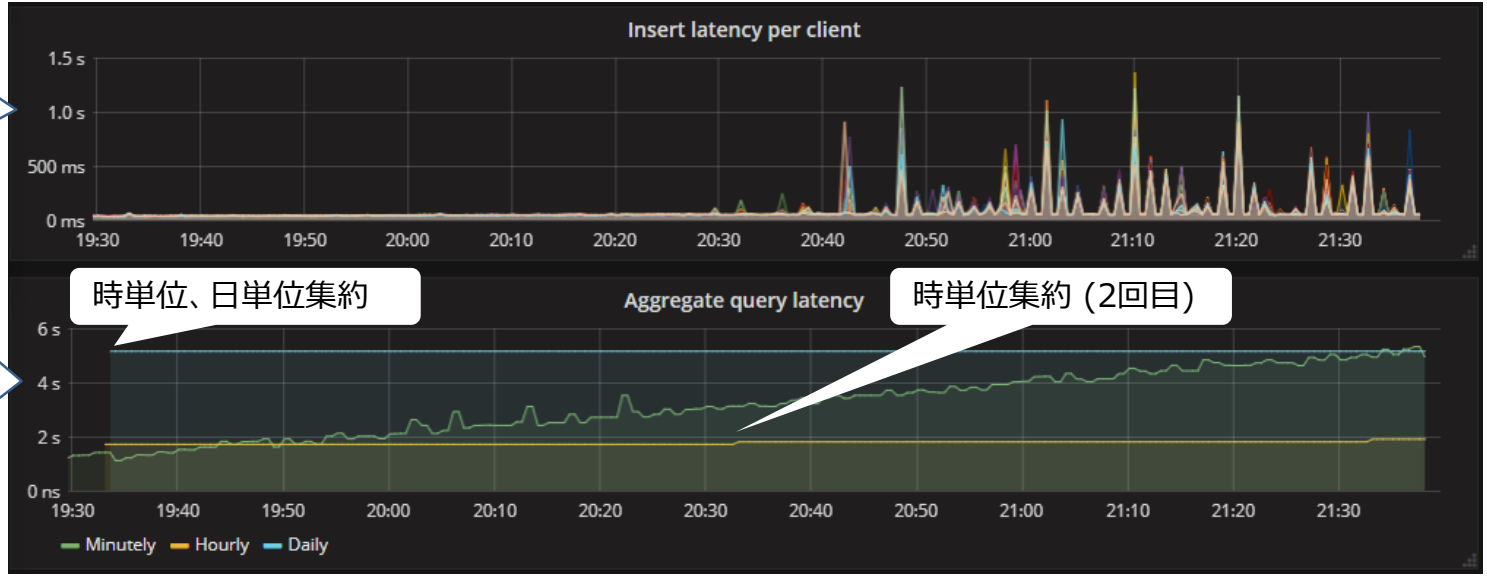


- 格納リクエスト・集約クエリともにレイテンシは1分以内であるため問題なし
- 格納リクエスト・集約クエリともにエラーや失敗、レコードの欠損なし

オンライン処理の検証結果: 毎秒20万レコード

格納リクエストのレイテンシ
まれに1秒を超えるがすぐに回復

集約クエリのレイテンシ
- 分単位集約は約5秒まで徐々に増加
- 時単位集約は約2秒
- 日単位集約は約5秒



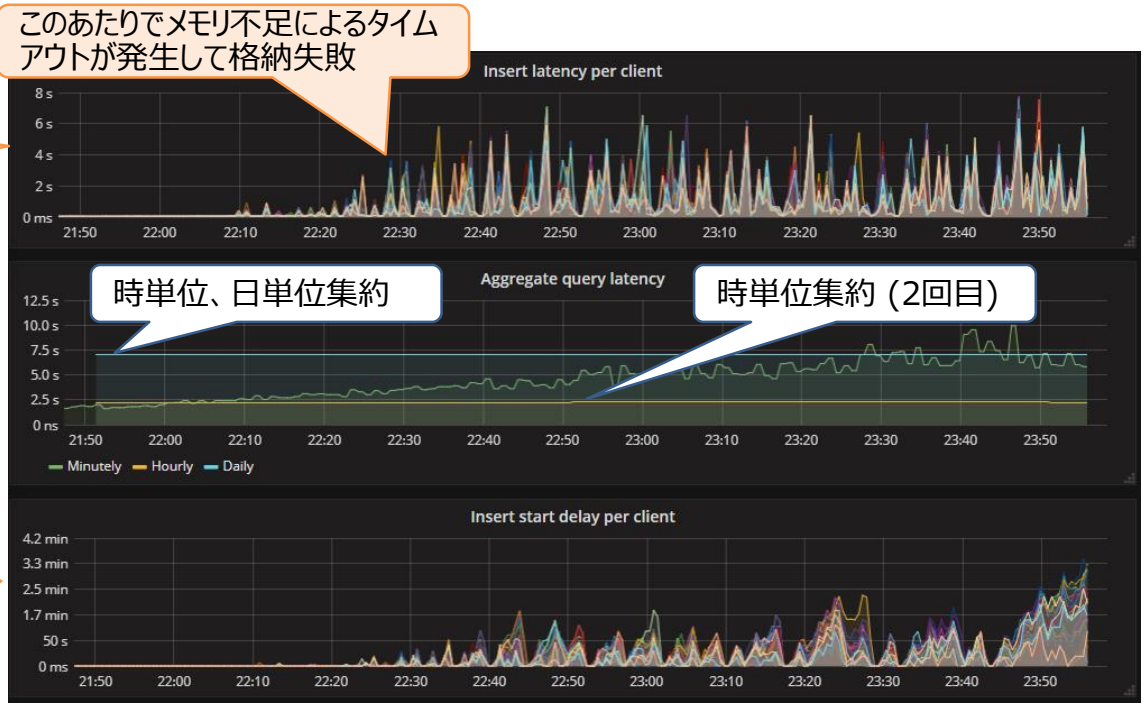
- 格納リクエストのレイテンシはスパイク的に1秒を超えるがすぐに回復するため問題なし
- 分単位集約クエリのレイテンシが徐々に増加していったが1分以内であるため問題なし
- 格納リクエスト・集約クエリともにエラーや失敗、レコードの欠損なし

オンライン処理の検証結果: 毎秒30万レコード

格納リクエストのレイテンシ
スパイク発生で8秒を超えることもある

集約クエリのレイテンシ
- 分単位は約10秒まで増加
- 時単位集約は約2.5秒
- 日単位集約は約7秒

格納開始の遅延時間
格納リクエストの遅延が1秒を超えるため、
格納開始が徐々に遅延していく



- 一部の格納リクエストがメモリ不足によるエラーでレコードの格納に失敗
- 格納リクエストのレイテンシは1秒を超える場合が多く、格納の開始が徐々に遅延
- 以上の結果から、毎秒30万レコードの処理は失敗といえる

オンライン処理の検証結果: 初期設定における検証結果まとめ

#	毎秒格納レコード数	格納遅延	集約遅延	格納エラー	集約エラー
1	100,000	なし (2h経過時)	なし (2h経過時)	なし (2h経過時)	なし (2h経過時)
2	200,000	なし (2h経過時)	なし (2h経過時)	なし (2h経過時)	なし (2h経過時)
3	300,000	あり (0.8h経過時)	なし (2h経過時)	あり (0.7h経過時)	なし (2h経過時)

- 毎秒20万レコードまでの処理は、2時間実行した時点では問題なし
- 毎秒30万レコードの処理は、TServerのメモリ不足エラーによる格納失敗と格納開始の遅延が発生
- 以上の結果から、初期設定だと秒間20万レコードの処理が限界

TServerのメモリ割当量を増やして検証: メモリ割当量16GB / 毎秒30万レコード

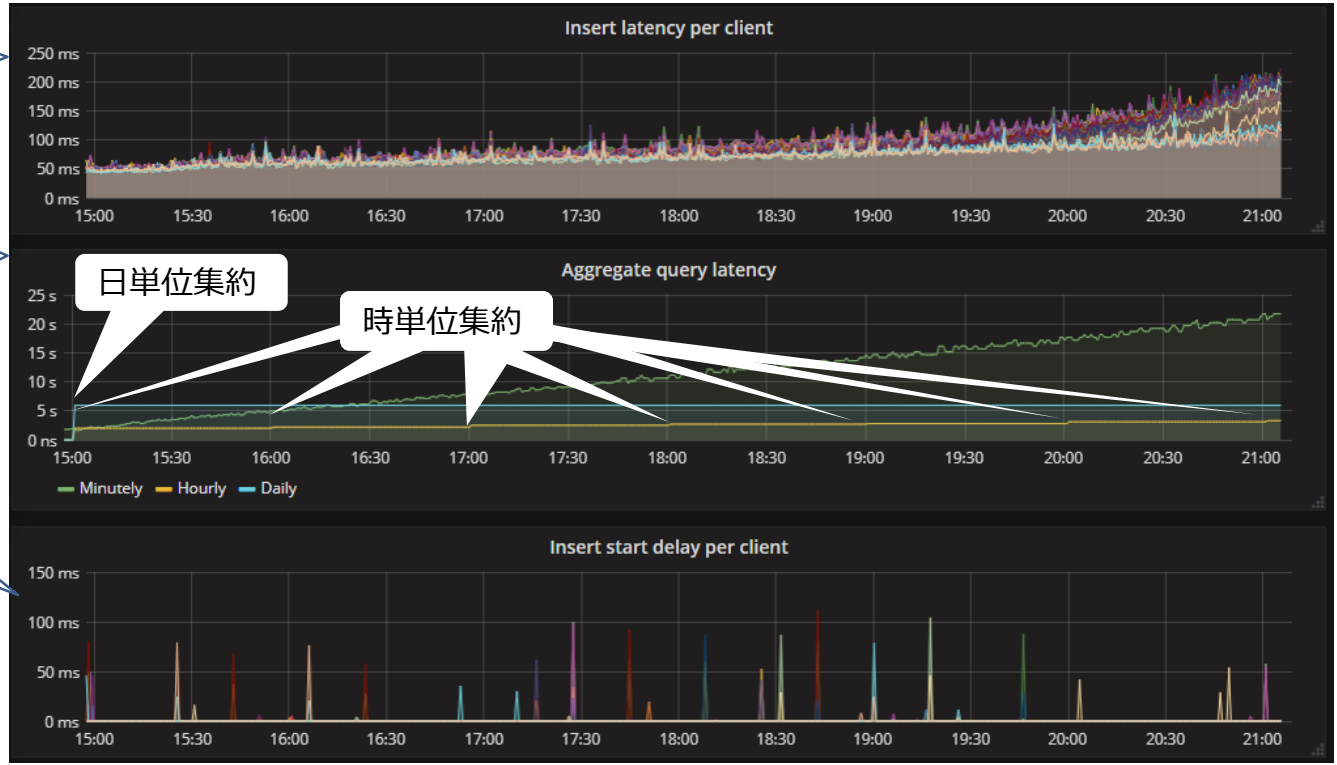
- TServerのメモリ不足エラーを解消するため、メモリ割当量をデフォルトの4GBから16GBに増量
- 処理期間も6時間に延長

格納リクエストのレイテンシ
50msから徐々に増加して
250msまで増加

集約クエリのレイテンシ
- 分単位は約22秒まで増加
- 時単位集約は約4秒
- 日単位集約は約6秒

格納開始の遅延
ほぼなし

格納/集約の遅延失敗なし



TServerのメモリ割当量を増やして検証: メモリ割当量16GB / 毎秒40万レコード

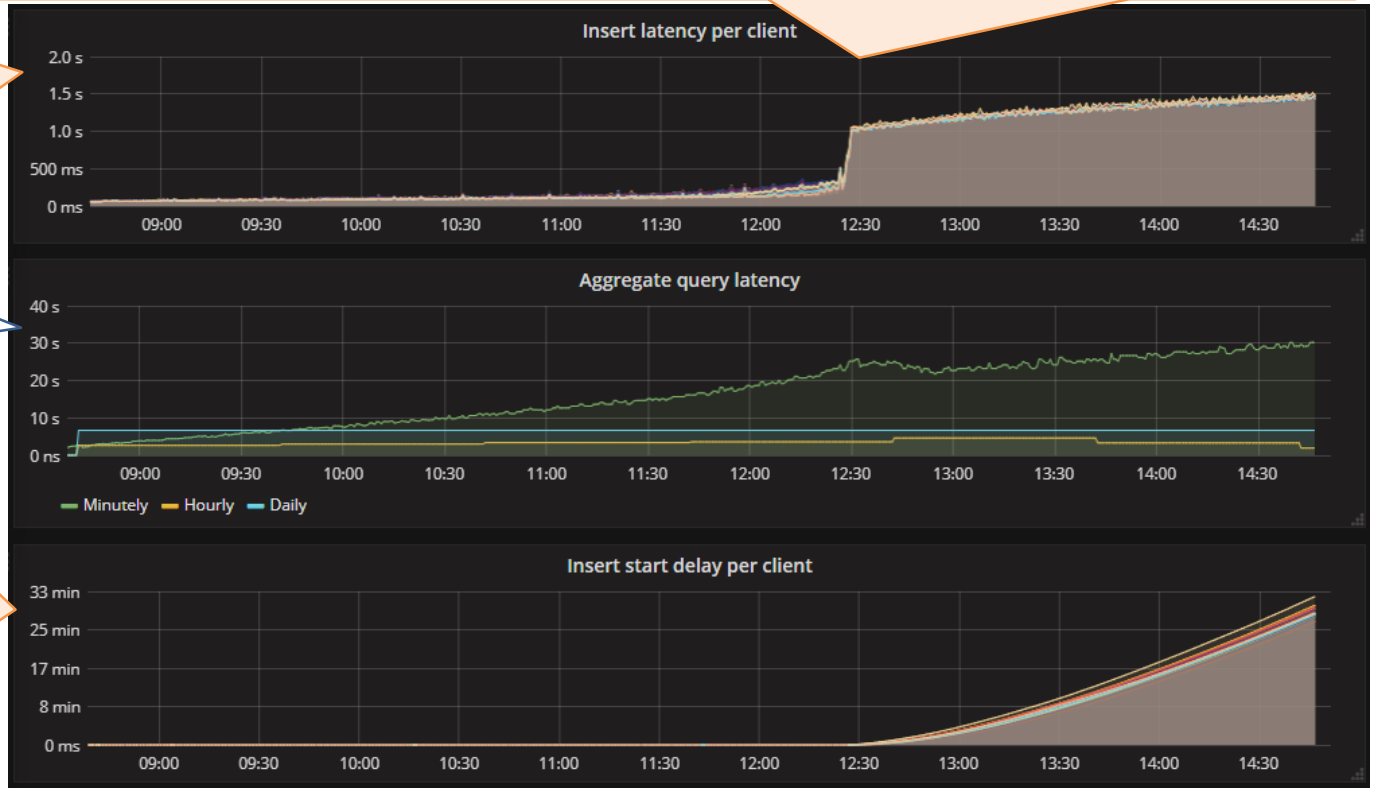
INFOレベルのログ出力発生：
Call kudu.tserver.TabletServerService.Write from 10.196.215.209:36356
(ReqId={client: 041195b2ae1e476da433acc5ec084f68, seq_no=220968, attempt_no=1}) took 1053ms.

格納リクエストのレイテンシ
測定開始から3.8時間ほど経過した時点でレイテンシが急上昇し1秒超え

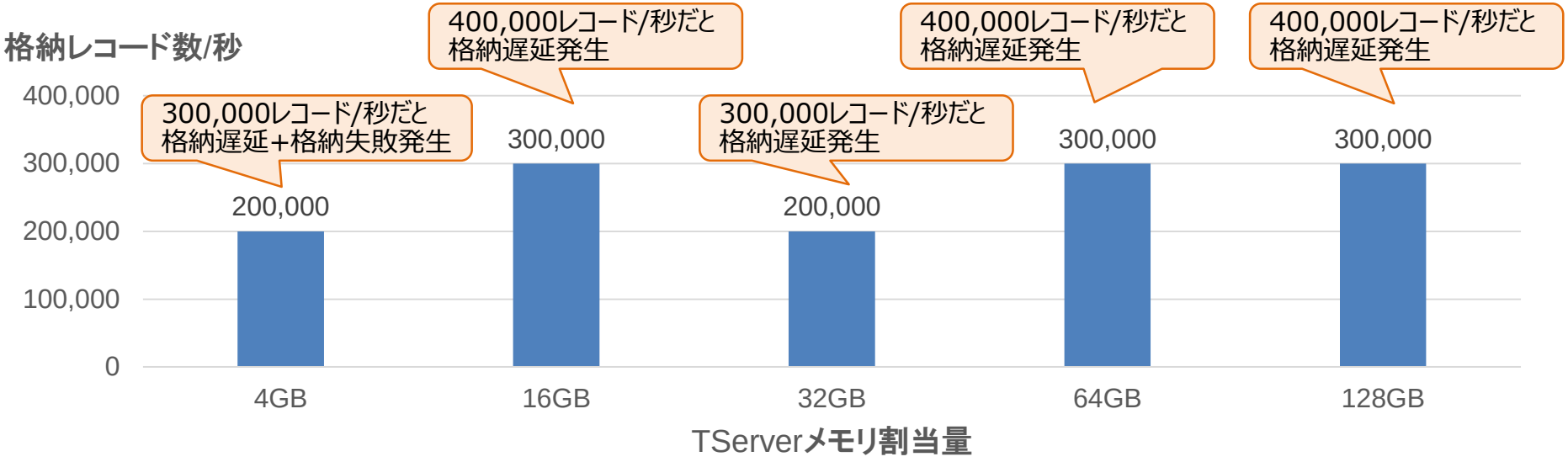
集約クエリのレイテンシ
- 分単位は約30秒まで増加
- 時単位集約は約5秒
- 日単位集約は約7秒

格納開始の遅延
格納リクエストのレイテンシが1秒を超えたため、格納開始の遅延が拡大

格納の遅延が発生



TServerのメモリ割当量を 4GB から 128GB まで増やした測定結果



- メモリ割当量を16GBに増やすことで、メモリ不足エラーによる格納失敗は解消
- しかし格納遅延は発生するため30万レコード/秒が限界
- 今回は6時間処理を実行したが、格納リクエストと集約クエリのレイテンシは増加し続けているため、より長時間処理を行った場合は問題となる可能性がある

レンジパーティション間隔を24h -> 1h単位に変更して検証

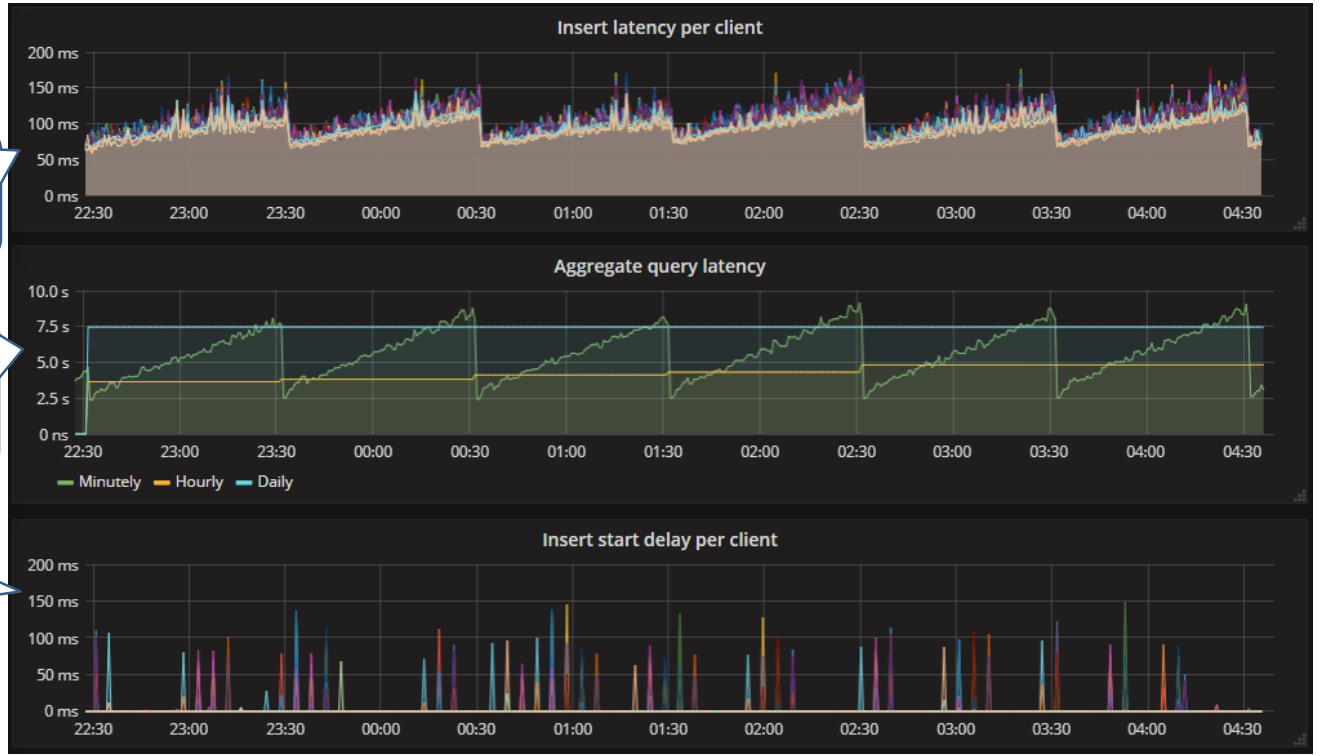
- ここまでの検証ではタブレット数が固定のため、毎秒の格納レコード数が増加すると、1タブレットあたりの最大データ量も増加していた
- 1タブレットのサイズを減らすことで、レイテンシの増加を抑えられる可能性がある
  - ハッシュパーティション数を増やすか、レンジパーティションの期間を短くする
- そこで、load_per_secテーブルのレンジパーティションを1日単位から1時間単位に短縮して測定してみる
  - 他のテーブルのレンジパーティション間隔はそのまま
  - メモリ割当量は 32GB, 64GB, 128GB でそれぞれ測定

レンジパーティション間隔1hで検証: メモリ割当量32GB / 50万レコード

格納リクエストのレイテンシ
100msから150msまで増加するが
タブレット切り替えのタイミングで元に戻る

集約クエリのレイテンシ
- 分単位は約8秒まで増加してリセット
- 時単位集約は約5秒
- 日単位集約は約7.5秒

格納開始の遅延
遅延はほぼなし



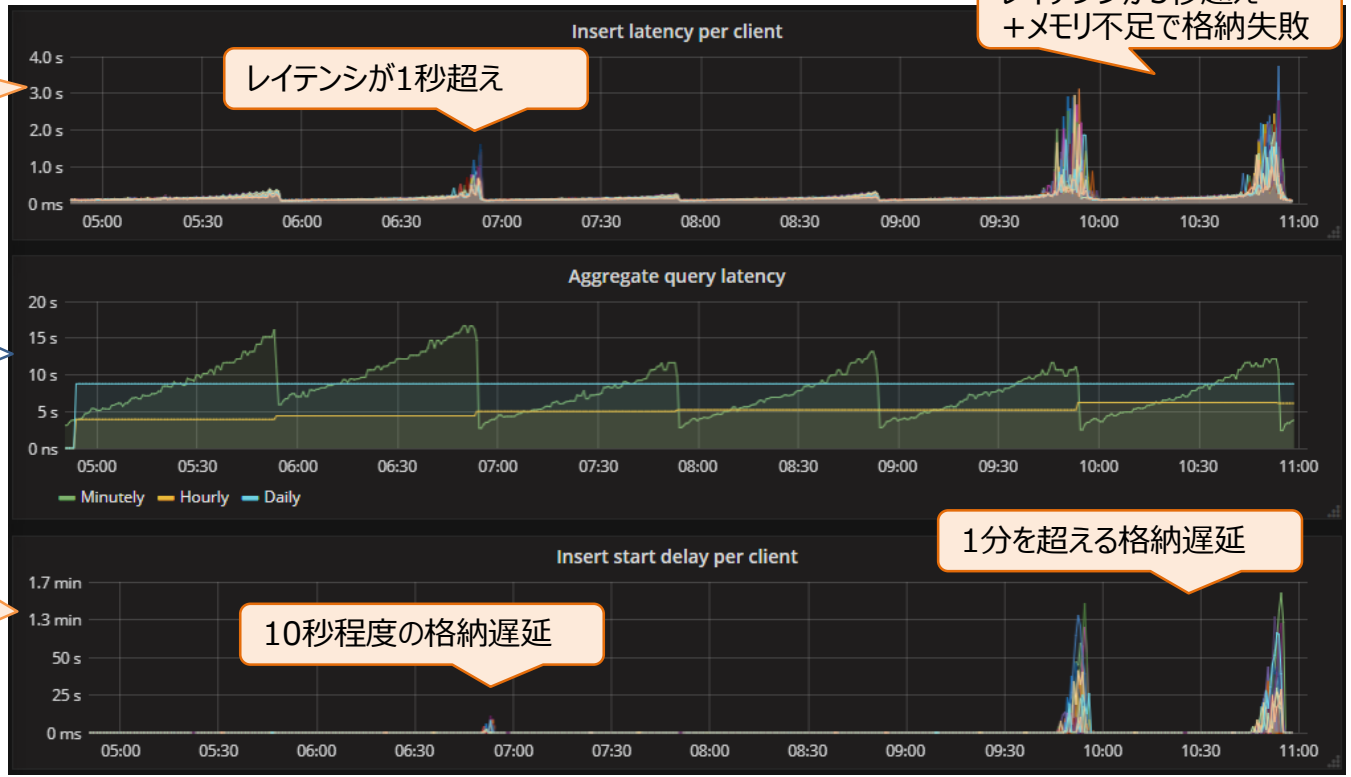
- 1時間ごとのタブレットが切り替え時に、格納/集約共にレイテンシ増加が毎回リセット
- 格納/集約の遅延/失敗もなし

レンジパーティション間隔1hで検証: メモリ割当量32GB / 60万レコード

格納リクエストのレイテンシ
5時間経過後から3-4秒まで急上昇

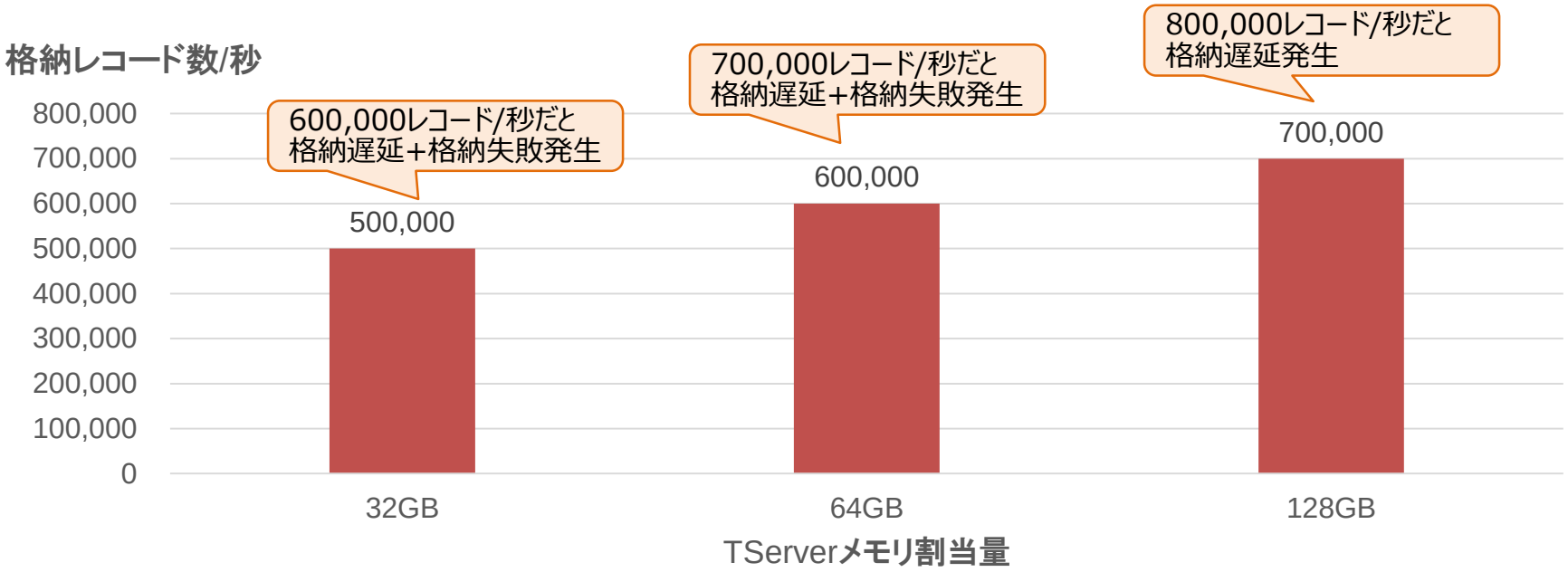
集約クエリのレイテンシ
- 分単位は約15秒まで増加してリセット
- 時単位集約は約6秒
- 日単位集約は約9秒

格納開始の遅延
格納リクエストのレイテンシが1秒を超えたため、格納開始の遅延が拡大



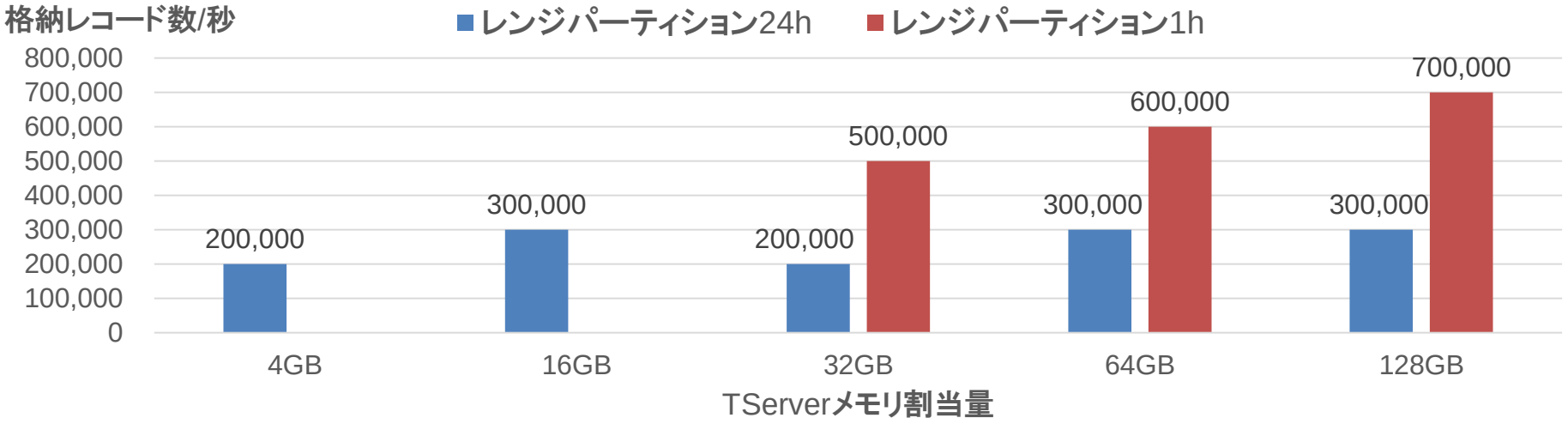
1時間ごとのタブレット切り替え前に格納レイテンシが急上昇し、格納遅延と格納失敗が発生

TServerのメモリ割当量を 32GB から 128GB まで増やした測定結果



レンジパーティション間隔を1hに短縮して1タブレットの最大データ量を抑えることで、格納・集約レイテンシの増加を抑制

オンライン処理の検証結果まとめ



- TServerのメモリ割当量を増やすことで、メモリ不足エラーによる格納失敗は解消
  - ただし、格納レイテンシの急上昇による格納遅延に対しては効果なし
- タブレットサイズを抑えることで、格納・集約レイテンシの増加を抑制

オンライン処理のチューニングポイントと注意点

- **オンライン処理のチューニングポイント**
  - TServerのメモリ割当量を増やす
    - メモリ不足によるエラーが発生してレコードの格納に失敗することがある
  - 1タブレットの最大サイズを抑える
    - ハッシュパーティション数を増やすか、レンジパーティションの期間を短くする
    - タブレットを頻繁に切り替えることで、クエリのレイテンシ増加を抑制可能
- **オンライン処理の注意点**
  - 今回は2-8時間処理を実行し続けて測定したが、より長期間だとどうなるか？
  - レイテンシなどのメトリクスを監視して長期間の性能負荷試験を実施すべき

5. まとめ

まとめ

- Apache Kuduとは
  - データのリアルタイムな格納と大量データの参照（分析）が得意なデータストアであり常時稼働が必要なオンライン処理システムに活用される
- 性能検証の結果
  - メモリ割当量とパーティション割当方法のチューニングを行うことで、8時間にわたって毎秒70万レコードの格納とその集約処理が可能であることを確認
- 検証結果から見えてきた課題
  - ただし、より長い期間処理を実行し続けた際に問題が発生しないとはいえない
  - 今回は全ディスクにHDDを使用したが、KuduはSSD向けに作られているため、SSDを使用すればより性能が出るはず

他社商品名、商標等の引用に関する表示

- Apache Kudu, Apache HBase, Apache Hadoopは、Apache Software Foundationの米国およびその他の国における登録商標または商標です。
- Clouderaは、Cloudera Inc.の米国およびその他の国における登録商標または商標です。
- Javaは、Oracle Corporation及びその子会社、関連会社の米国およびその他の国における登録商標です。
- HITACHIは、株式会社 日立製作所の商標または登録商標です。
- その他記載の会社名、製品名などは、それぞれの会社の商標もしくは登録商標です。