HITACHI
Inspire the Next

NoSQLって結局どうなの？
～ HBaseを例に検証してみました ～

2017年3月10日

日立製作所　OSSソリューションセンタ
伊藤雅博

自己紹介

HITACHI
Inspire the Next

- 伊藤 雅博 (いとう まさひろ)
  - 所属： 日立製作所 OSSソリューションセンタ
  - 業務： Hadoop/Sparkを中心としたビッグデータ関連OSSの導入支援や検証、テクニカルサポート

Think ITの連載記事：
ユースケースで徹底検証！ Sparkのビッグデータ処理機能を試す
https://thinkit.co.jp/series/5747

はじめに

HITACHI
Inspire the Next

- IoT(Internet of Things) と NoSQL
  - 様々なセンサ機器が、膨大な量のデータを生成している
  - この膨大な量のデータを管理するために、NoSQLが注目を集めている

- なぜ HBase なのか？
  - NoSQLのひとつであるHBaseは、センサ機器が生成するような大量の時系列なデータを管理することに適している

- 本発表では、1,000万個のスマートメータのデータを使用したHBaseの性能検証の結果と、そこで得られた設計ノウハウを紹介する

発表内容

HITACHI
Inspire the Next

1. (今さらですが)NoSQLとは
2. HBaseの概要
3. スマートメータのデータを使用したHBaseの検証結果の紹介
   i. 検証シナリオ
   ii. 格納性能の検証
   iii. 圧縮性能の検証
   iv. 参照性能の検証
4. まとめ

HITACHI
Inspire the Next

1. (今さらですが) NoSQLとは

HITACHI
Inspire the Next

皆さんご存知のNoSQLですが、wikipediaを見てみると

NoSQL
https://ja.wikipedia.org/wiki/NoSQL

NoSQL（一般に "Not only SQL" と解釈される）とは、
関係データベース管理システム **(RDBMS) 以外**のデータベース管理システムを指すおおまかな分類語である。

関係データベースを杓子定規に適用してきた長い歴史を打破し、それ以外の構造のデータベースの利用・発展を促進させようとする**運動の標語としての意味合い**を持つ。

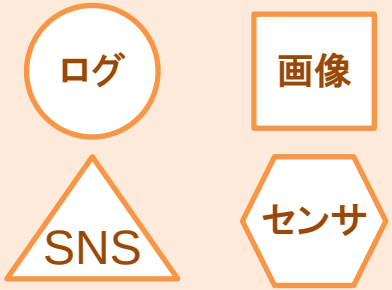
・・・技術用語かと思ったら運動の標語？
RDB以外のデータストアなら何でもよさそうです。

一般に言われるRDBの課題

HITACHI
Inspire the Next

RDBが苦手とするビッグデータの3つの【V】

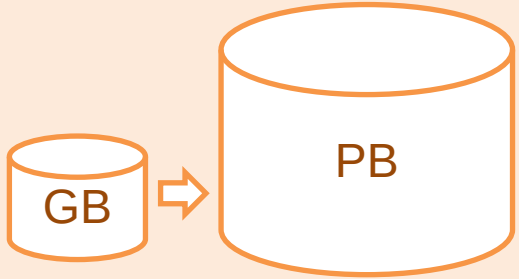
Variety（多様性）



様々な構造や大きさのデータを管理したい

事前定義する表形式とは相性が悪い

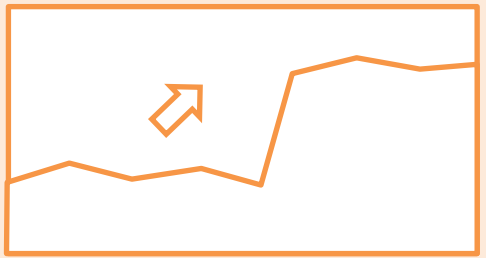
Volume（量）



膨大な量のデータを管理したい

データを分散させると整合性の管理が難しくなる

Velocity（頻度）



高頻度で発生するデータをリアルタイムに処理したい

整合性を維持するための排他処理に時間がかかる

RDBの特徴

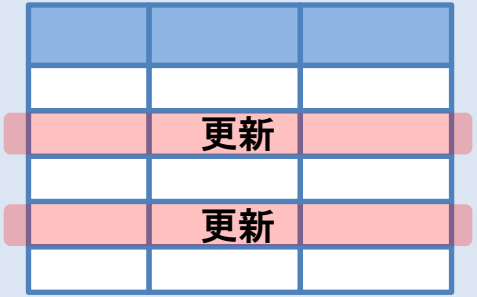
HITACHI
Inspire the Next

データモデル：表形式

日付	数値	文字

人間にわかりやすい形、
Excelが流行る理由

トランザクション



複数の人が同時更新しても
整合性担保（同時実行制御）

障害時はロールバックして
整合性を維持（耐障害性）

問合せ言語：SQL

```
SELECT score
FROM T1, T2
WHERE name='abc'
AND T1.id=T2.id
```

10

id	name
A	abc
B	xyz

id	score
A	10
B	5

英語っぽい問い合わせ言語

JOINなど複雑な条件が可能

RDBの特徴に対する性能対策(?)をしたものがNoSQL

HITACHI
Inspire the Next

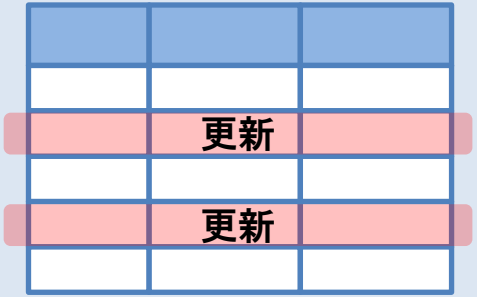
データモデル

日付	数値	文字

2列だけにしよう

行毎に列数が違ってもいいだろう

トランザクション



整合性は不要

障害時はバックアップから戻す

問合せ言語

http://・・・/score?id=A ➡ 10

id	name
A	abc
B	xyz

id	score
A	10
B	5

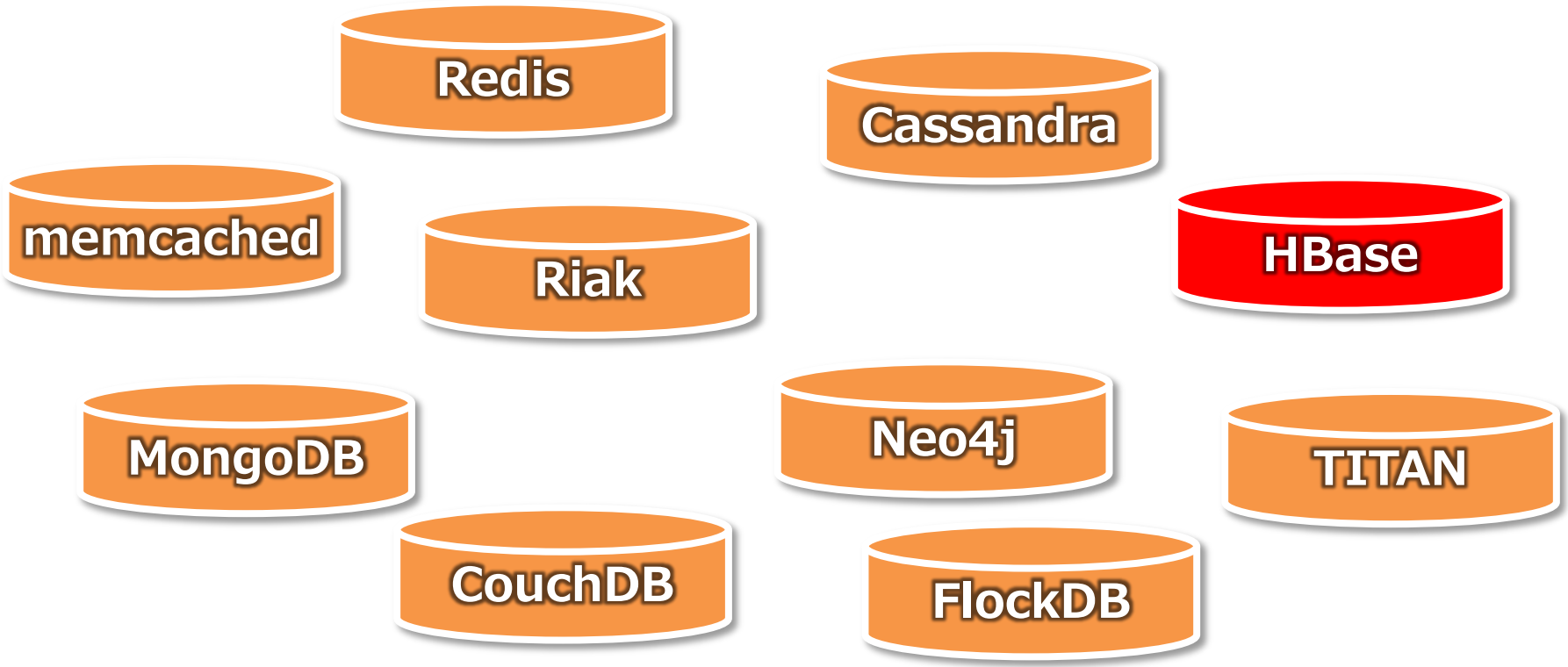
REST APIにしよう

Join不可に

RDBの機能を絞ったものが、NoSQL

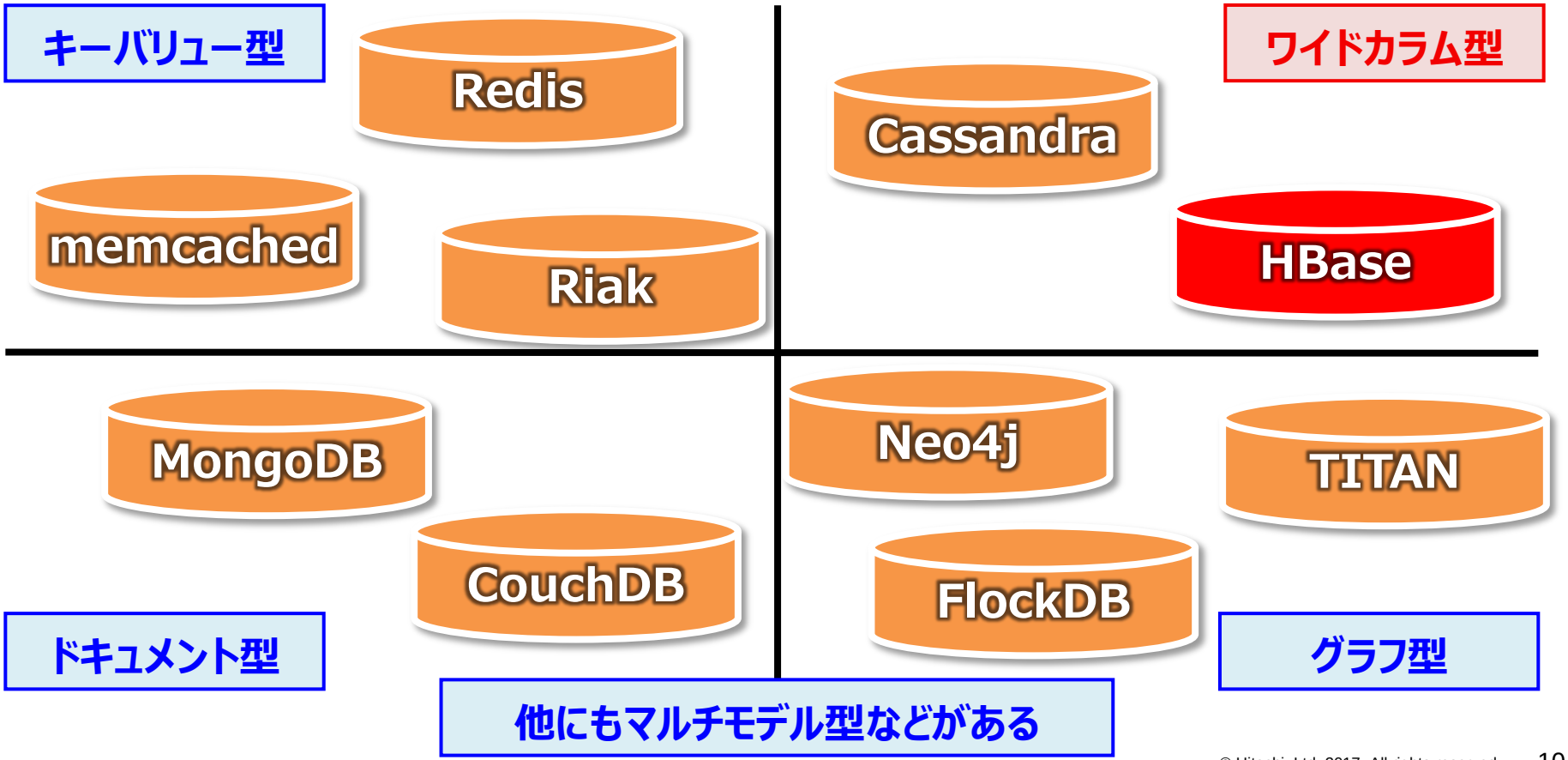
世の中には、たくさんのNoSQLがあります（他にも多数）

HITACHI
Inspire the Next



NoSQLは一般にデータモデルで分類されます

HITACHI
Inspire the Next



NoSQLは一般にデータモデルで分類されます

HITACHI
Inspire the Next

キーバリュー型

- キーバリューのペアというシンプルな構造
- データの分散が容易でアクセスも高速

ワイドカラム型

- 行ごとに異なる列数や型でデータを格納
- 列方向のアクセスが高速

- JSONやXMLなどの構造データを格納
- スキーマレスで柔軟なデータ運用

ドキュメント型

- データ間の関係をグラフ構造で格納
- 表形式での表現が難しいデータを扱える

グラフ型

HITACHI
Inspire the Next

2. HBaseの概要

HBaseとは

HITACHI
Inspire the Next

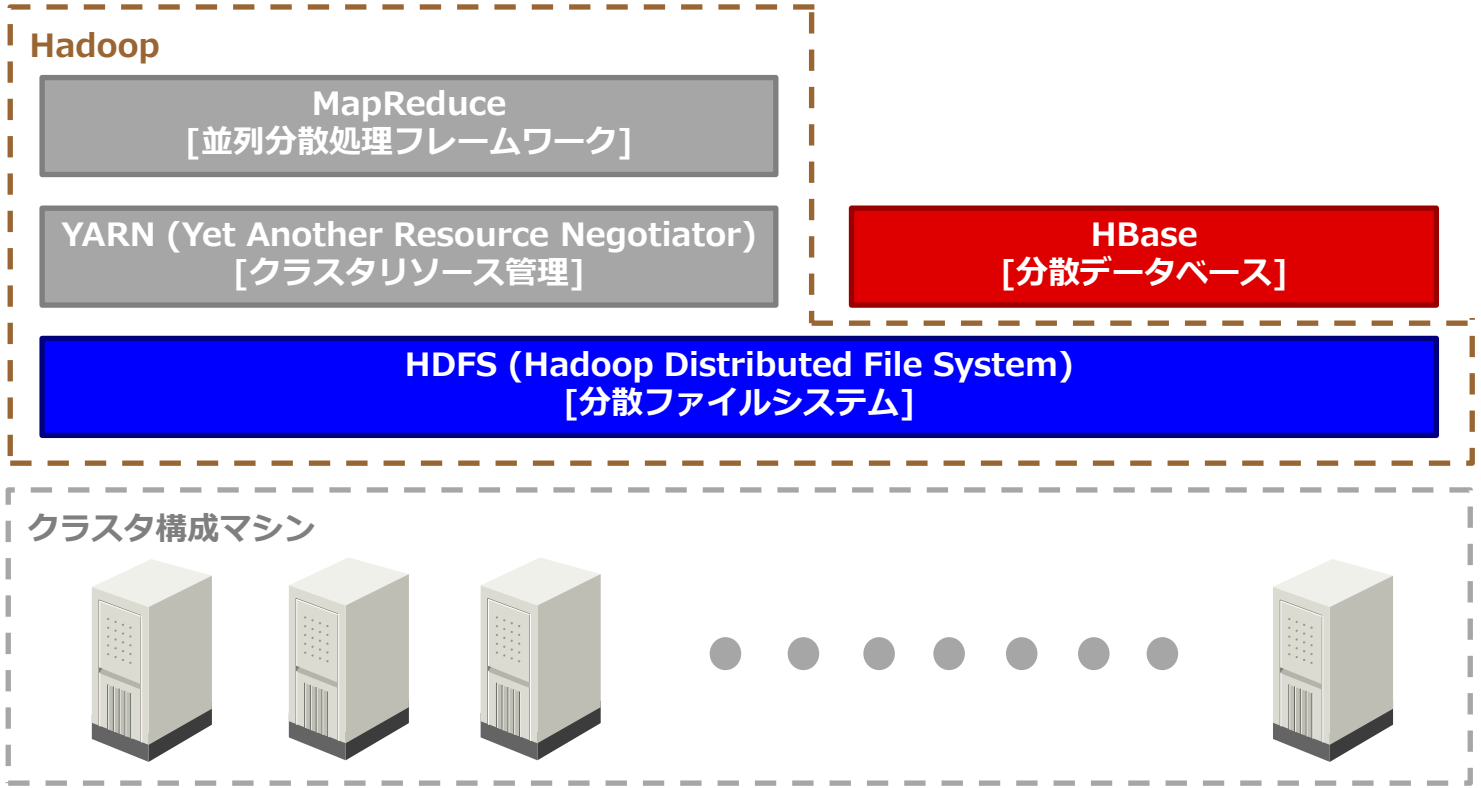
- HBaseはワイドカラム型のNoSQL
  - 行ごとに異なる列数を持ち、任意のデータ型（バイト列）で格納可能

- Googleの分散データベースBigtableをJavaで実装したOSSクローン
  - 分散処理基盤であるApache Hadoopのエコシステムを構成するOSSの1つ

HBase と Hadoop (HDFS) の関係

HITACHI
Inspire the Next

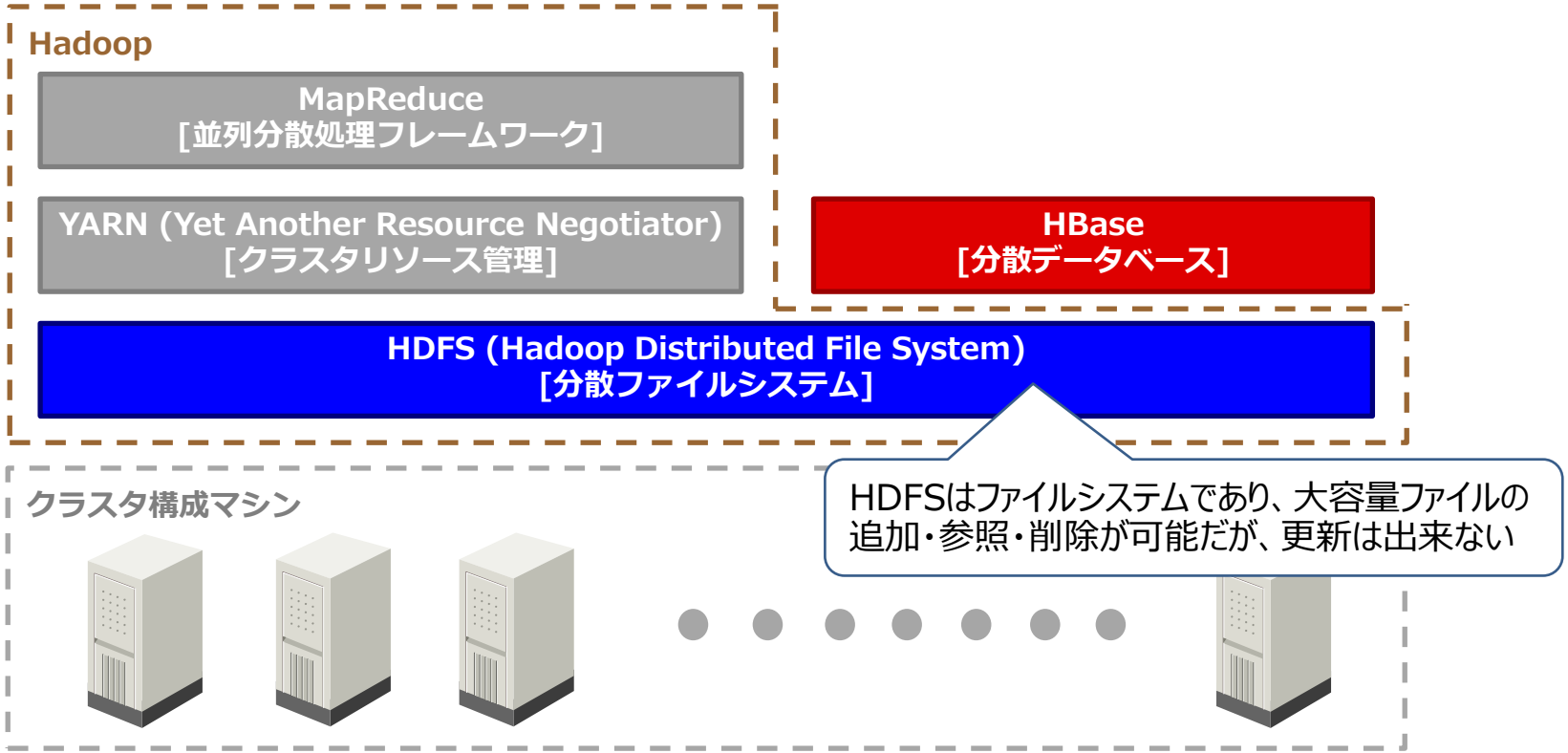
- HBaseは、Hadoopの分散ファイルシステムであるHDFS上に構築する



HBase と Hadoop (HDFS) の関係

HITACHI
Inspire the Next

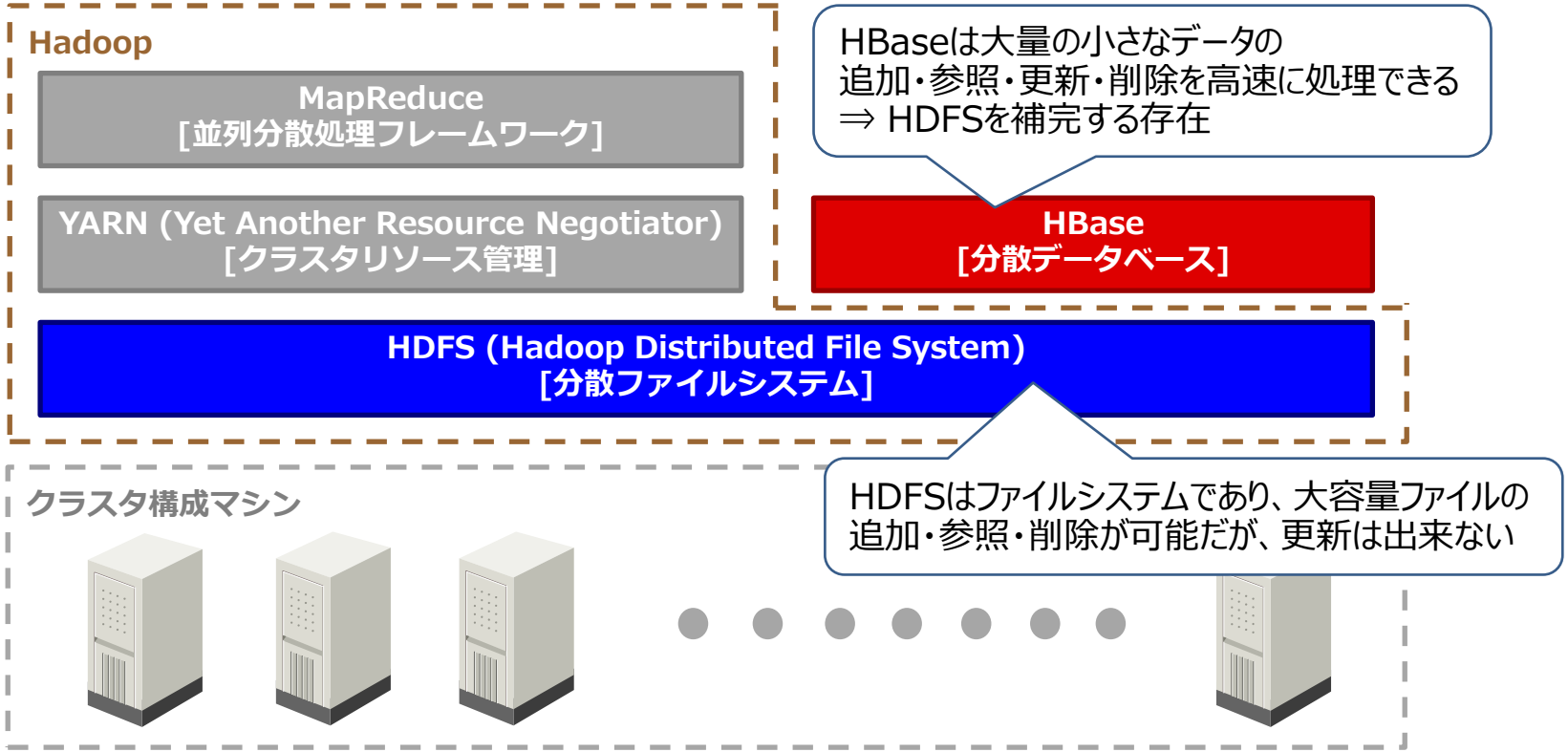
- HBaseは、Hadoopの分散ファイルシステムであるHDFS上に構築する



HBase と Hadoop (HDFS) の関係

HITACHI
Inspire the Next

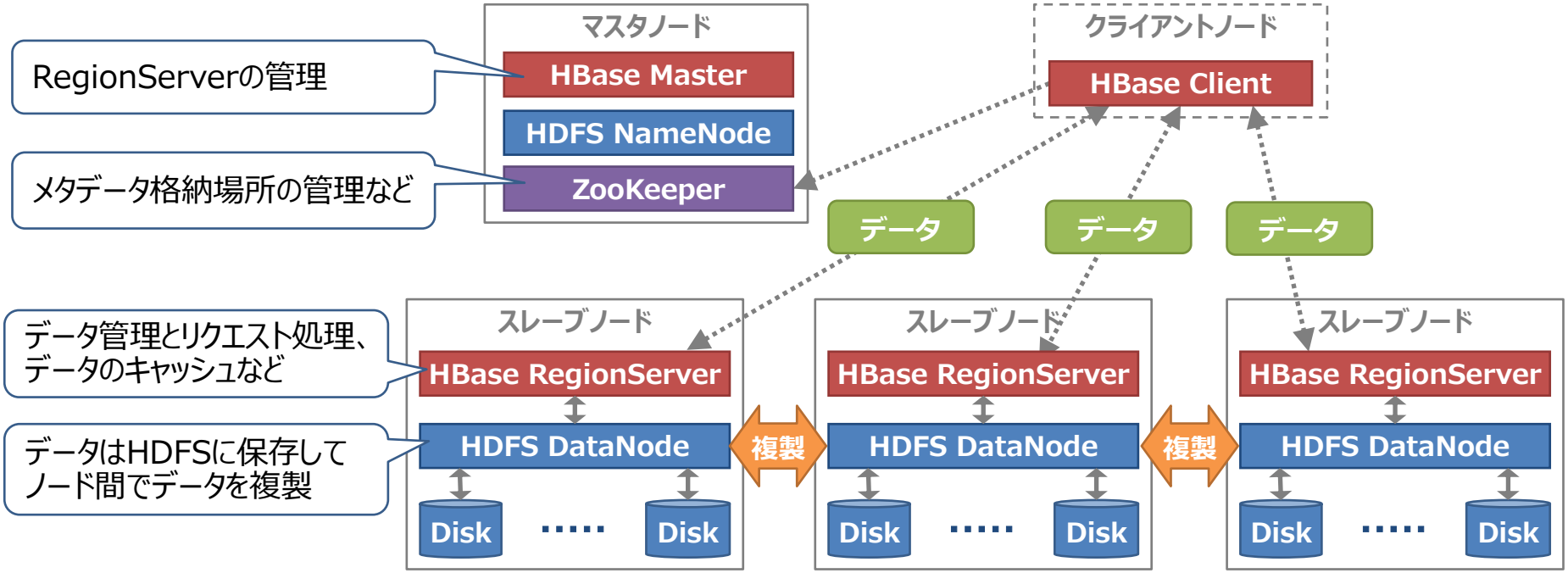
- HBaseは、Hadoopの分散ファイルシステムであるHDFS上に構築する



HBaseのシステム構成はマスタ・スレーブ型

HITACHI
Inspire the Next

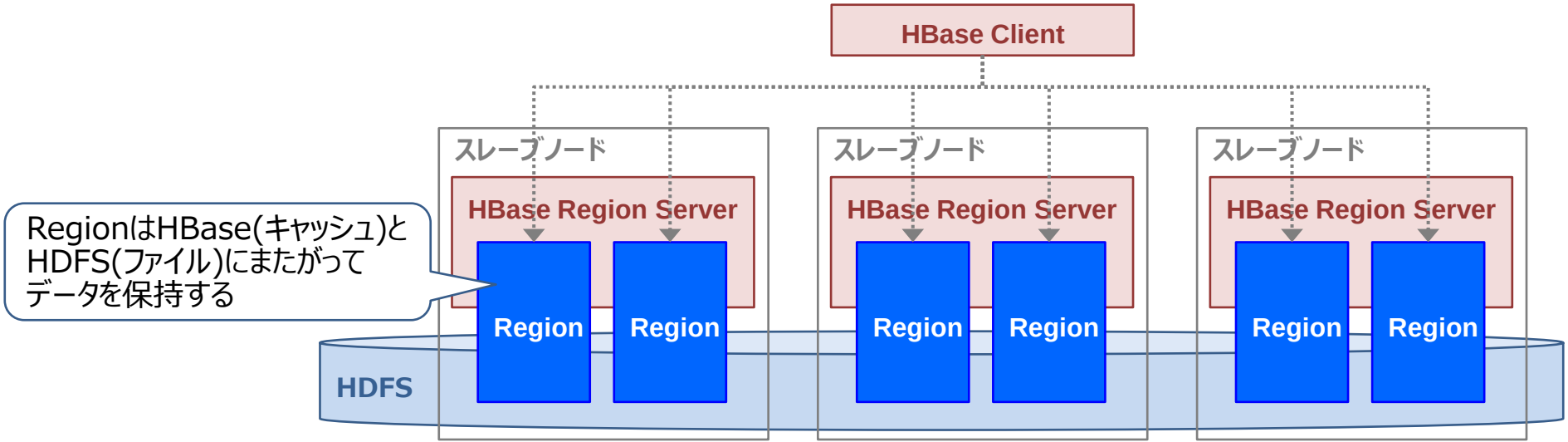
HBaseがデータ管理とリクエスト処理を行い、HDFSがデータを保存する



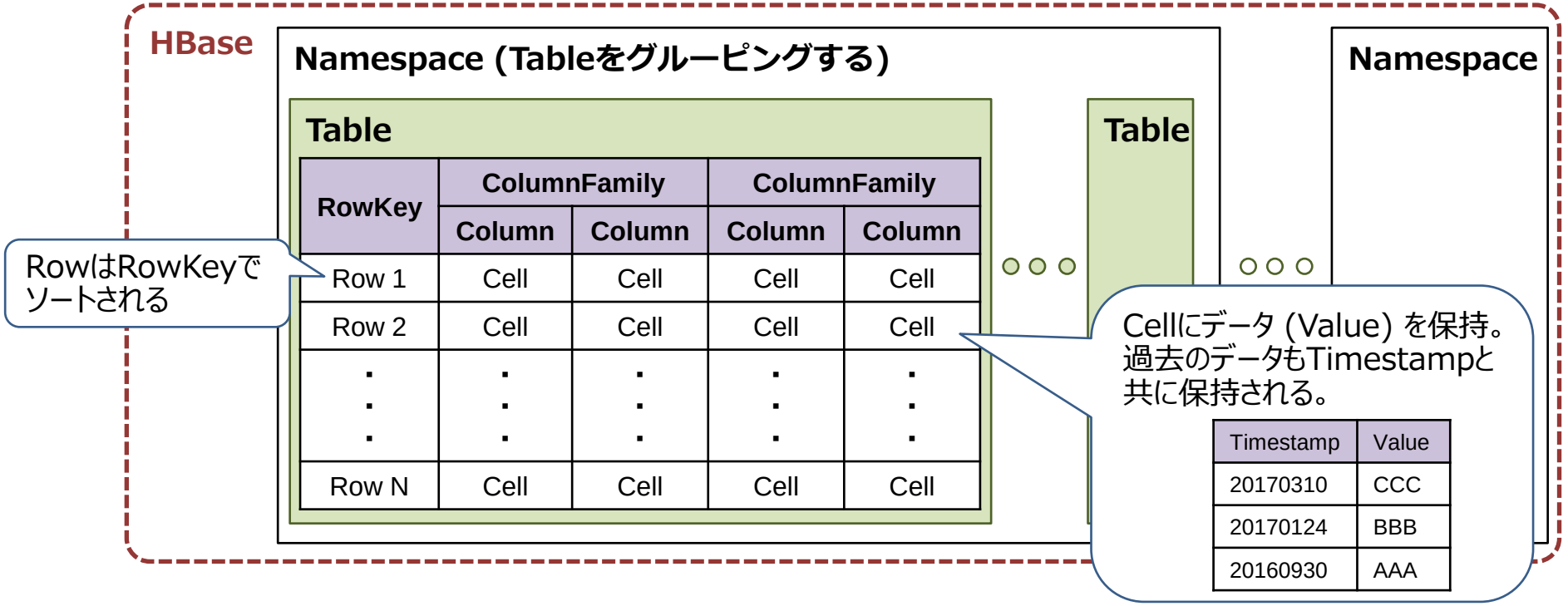
HBaseのデータ管理

HITACHI
Inspire the Next

- HBaseはRegionという単位でデータを分散する
  - Regionを複数台のスレーブノードに割り当て、書き込み/読み出しを分散処理
  - スレーブノードの追加で、処理性能とディスク容量が向上



HBaseのデータモデル（論理データ構造）



- Tableは一見するとRDBのような表形式に見えるが・・・

HBaseの物理データ構造

HITACHI
Inspire the Next

- **実際はキーバリューで格納される**
  - **キーは RowKey, ColumnFamily:Column, Timestamp の順番でソートされる**
  - RowKey, ColumnFamily：Column **に対するインデクスあり**

Tableの論理データ構造

RowKey	fam1		fam2	
	Col1	Col2	Col3	Col4
Row 1	Val1	Val2	Val3	Val4
Row 2	Val5	Val6	Val7	Val8

Tableの物理データ構造

Key　　Value

RowKey	ColumnFamily:Column	Timestamp	Type	Value
Row 1	fam1:Col1	20170310	Put	Val1
Row 1	fam1:Col2	20160310	Put	Val2
Row 1	fam2:Col3	20170215	Put	Val3
Row 1	fam2:Col4	20170309	Put	Val4
Row 2	fam1:Col1	20170310	Put	Val5
Row 2	fam1:Col2	20160104	Put	Val6
Row 2	fam2:Col3	20160925	Put	Val7
Row 2	fam2:Col4	20170310	Put	Val8

Put (追加/上書き), Delete (削除) を記録

Tableは RowKeyの範囲と ColumnFamilyの値で Storeに分割される

HITACHI
Inspire the Next

Table

ColumnFamily = fam1

ColumnFamily = fam2

Region (Row1-2)

Store

RowKey	ColumnFamily:Column	・・・	Value
Row 1	fam1:Col1	・・・	AAA
Row 1	fam1:Col2	・・・	BBB
Row 1	fam1:Col3	・・・	CCC
Row 2	fam1:Col1	・・・	DDD

Store

RowKey	ColumnFamily:Column	・・・	Value
Row 1	fam2:Col1	・・・	EEE
Row 2	fam2:Col2	・・・	FFF
Row 2	fam2:Col3	・・・	GGG

Region (Row3-4)

Store

RowKey	ColumnFamily:Column	・・・	Value
Row 3	fam1:Col1	・・・	HHH
Row 4	fam1:Col2	・・・	III
Row 4	fam1:Col4	・・・	JJJ

Store

RowKey	ColumnFamily:Column	・・・	Value
Row 3	fam2:Col1	・・・	KKK
Row 4	fam2:Col3	・・・	LLL
Row 4	fam2:Col5	・・・	MMM

HBaseのデータ操作

HITACHI
Inspire the Next

- Java API, HBaseシェル, REST APIなど。SQLは使えない（Phoenix経由なら可能）
- トランザクションは1つのRowの内部のみ（Omidなどを使えば可能）

#	リクエスト種別	リクエスト名	説明
1	更新	**Put**	**データの追加/上書き**
2		Append	1つのRowに値を追記する
3		Inclement	1つのRowに値を加算する
4		Delete	1つのRowのデータを削除する
5		MutateRow	1つのRowに対するPut/Deleteをアトミックに行う
6	参照	**Get**	**1つのRowのデータを取得する**
7		**Scan**	**連続した範囲のRowのデータを取得する**
8		Exist	Row/Columnの存在確認を行う
9	参照＋更新	batch	Put/Append/Inclement/Delete/Getを一度に実行する
10		checkAndPut	1つのRowの値を確認して条件に一致するなら上書きする
11		checkAndDelete	1つのRowの値を確認して条件に一致するなら削除する

HBaseではScanできるように
RowKeyを設計することが重要

HBaseのテーブル設計： Tall Table と Wide Tableの違い

HITACHI
Inspire the Next

Tall Table

RowKey	fam1
	Col1
Row 1	Val1
Row 2	Val2
Row 3	Val3
Row 4	Val4

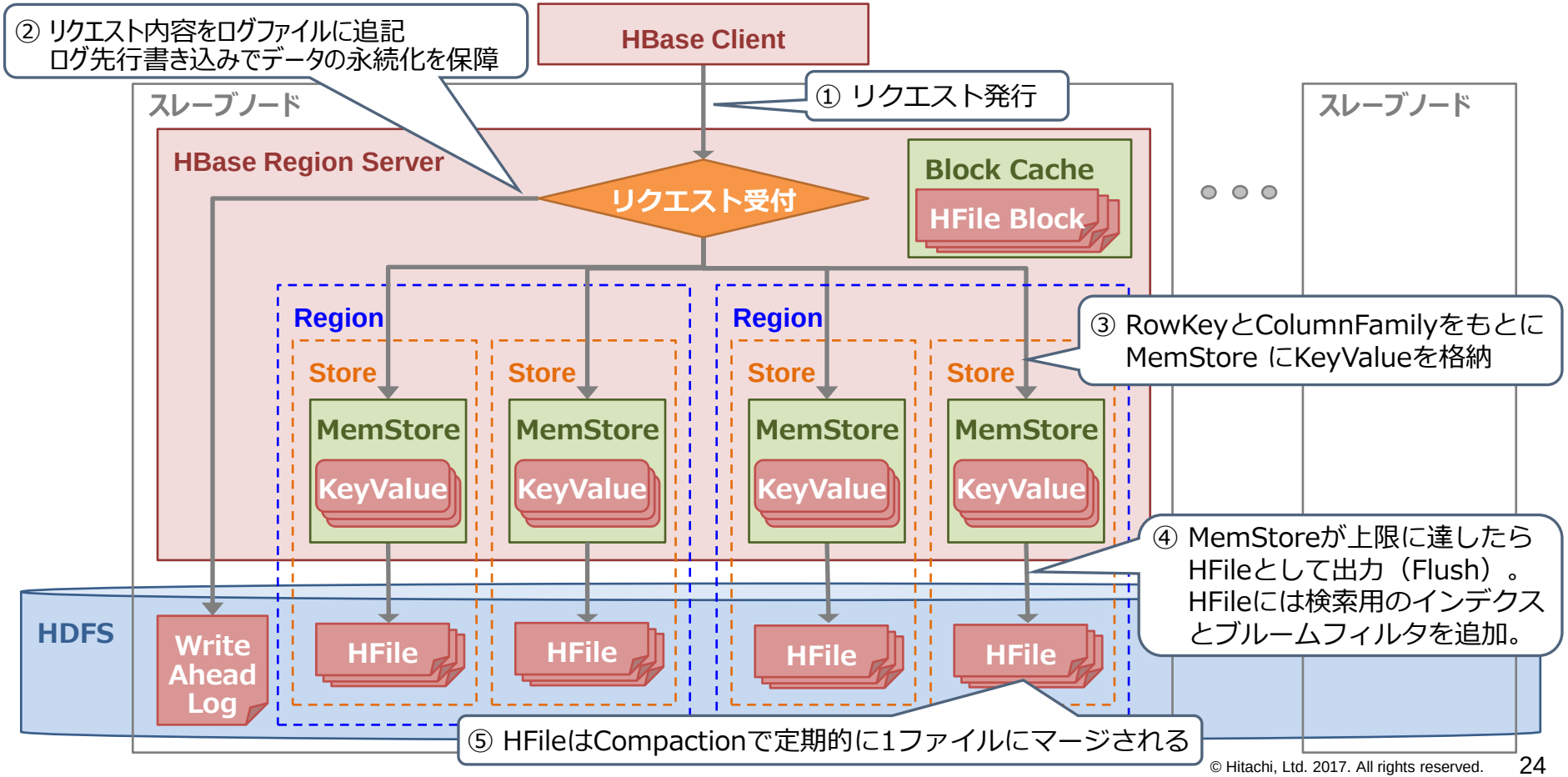
Wide Table

RowKey	fam1			
	Col1	Col2	Col3	Col4
Row 1	Val1	Val2	Val3	Val4

#	比較観点	Tall table	Wide table	理由
1	Scan(複数Rowの取得)	○	×	ScanはRowKeyの範囲を指定してRowを取得するため（Columnの範囲は指定できない）
2	トランザクション	×	○	トランザクションは1つのRow内でのみ可能なため
3	データの分割・分散	○	×	RegionはRowKeyの範囲で分割されるため、Row内にデータを詰め込みすぎると分割単位が大きくなる

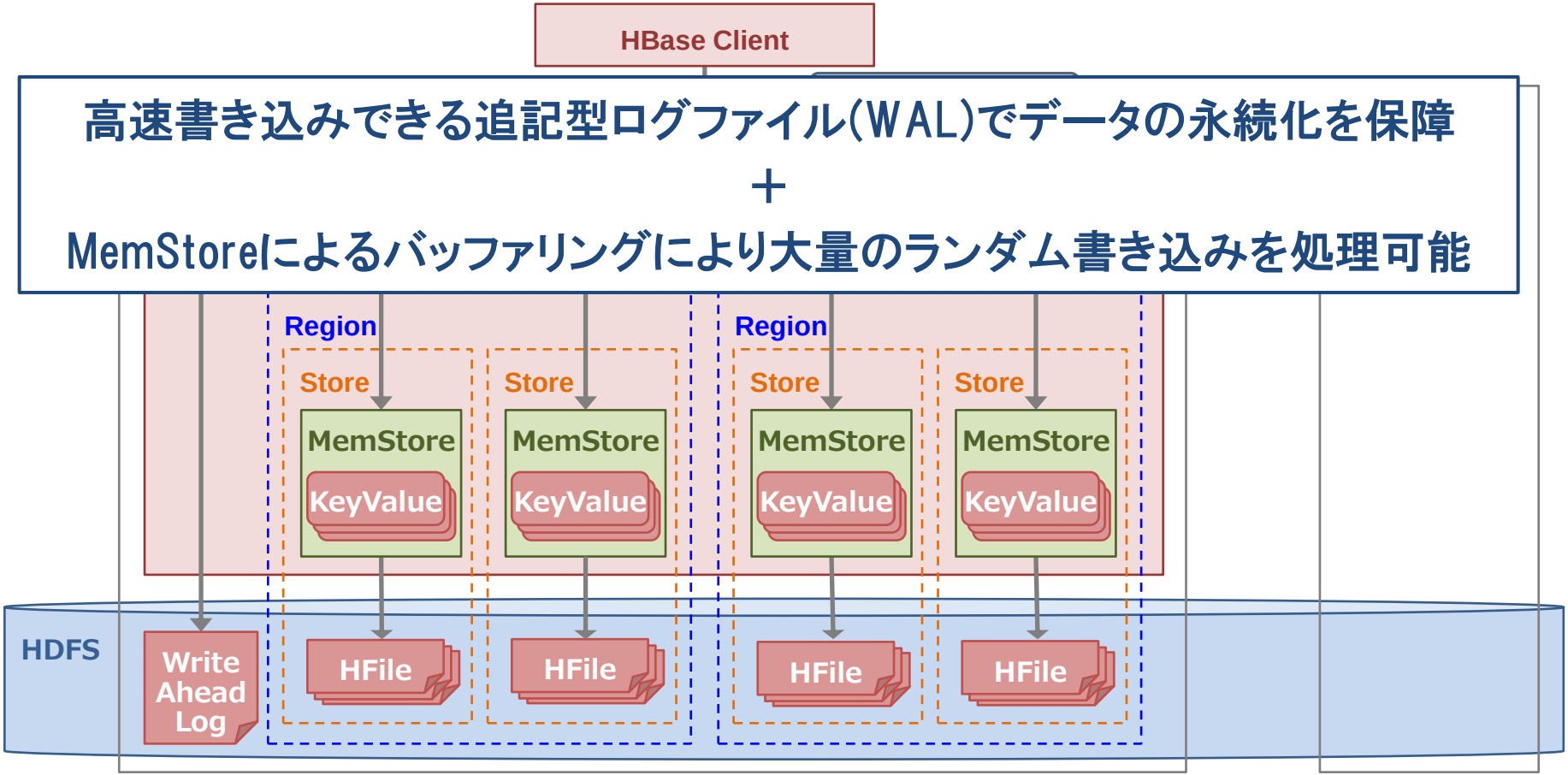
- 基本的には Tall Table でよい
- Row 内のトランザクション処理が必要な場合は Wide Table にする

HBaseの書き込み処理の流れ

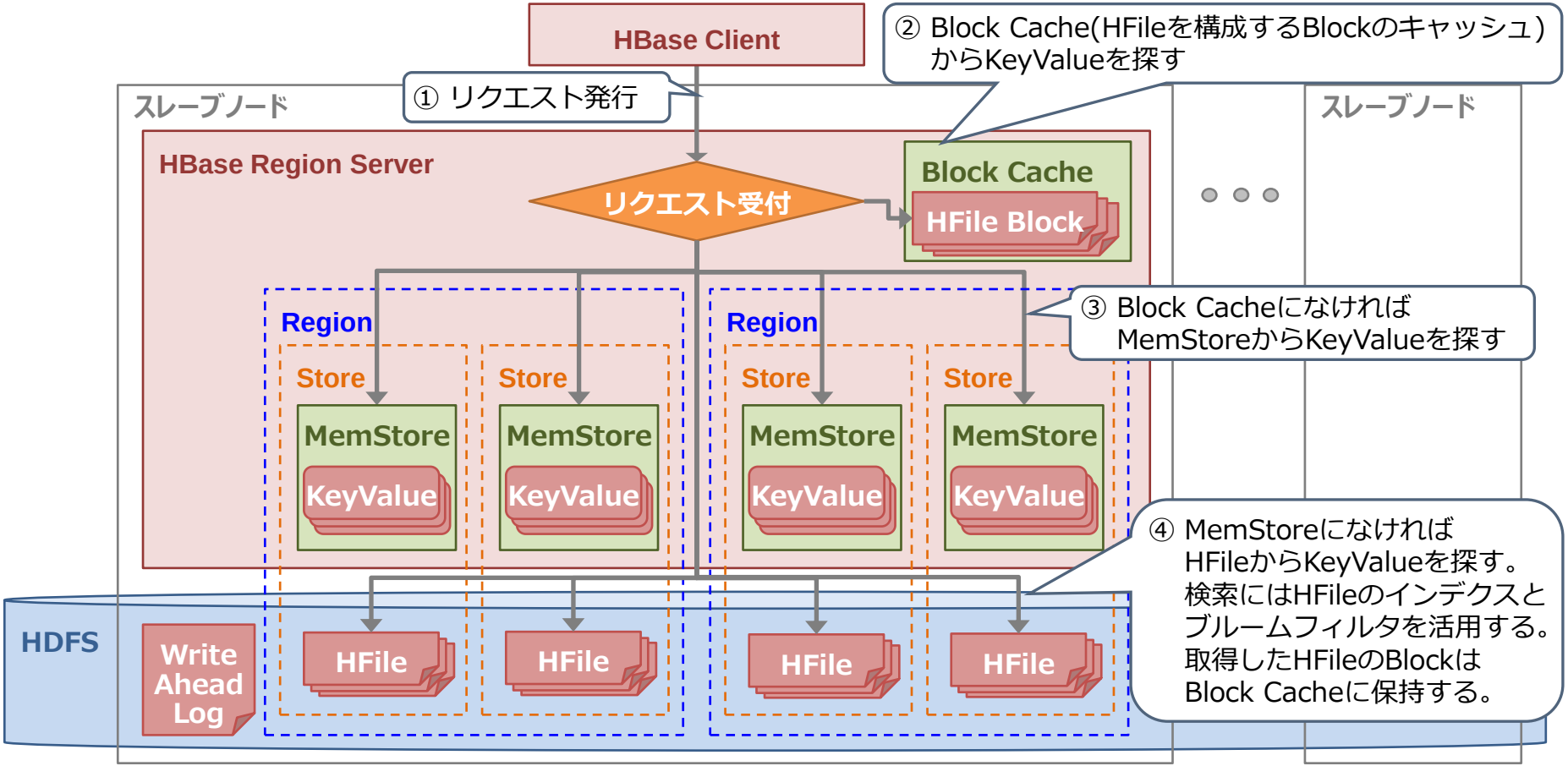


HBaseの書き込み処理の流れ

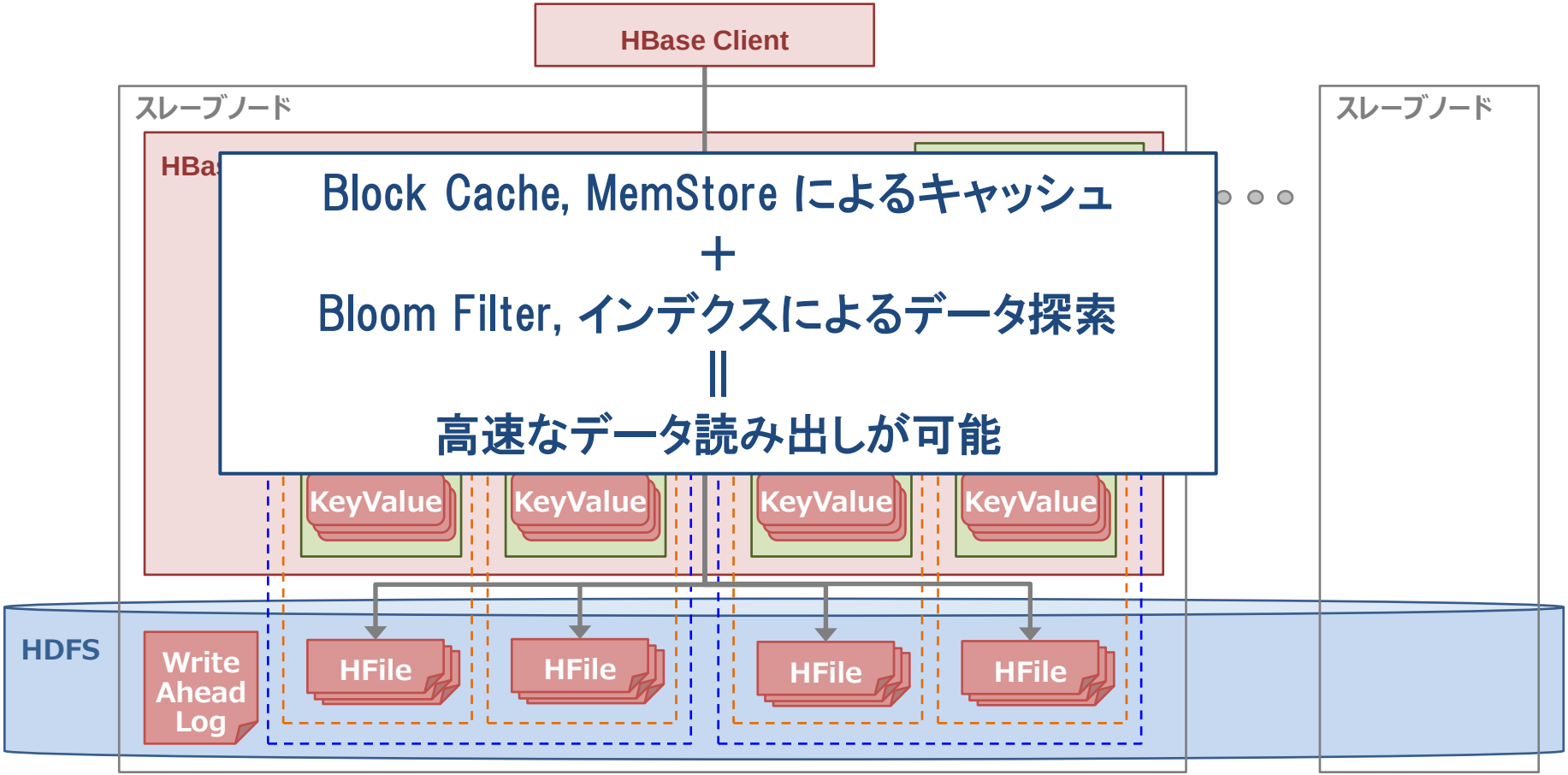
HITACHI
Inspire the Next



HBaseの読み出し処理の流れ



HBaseの読み出し処理の流れ



HBaseの概要まとめ

HITACHI
Inspire the Next

- Region単位でデータを分割して管理する
  - 複数台のスレーブノードにRegionを割り当て、書き込み/読み出しを分散処理
  - スレーブノードの追加で、処理性能とディスク容量が向上

- データをソートされたキーバリュー形式で保持する
  - キーによるランダムアクセス（Get）と、
    キーの範囲指定によるシーケンシャルアクセス（Scan）の
    両方に対応できる
    - うまくScanできるようにRowKeyを設計することが重要

- 制限されること
  - インデクスは RowKey と Column のみ
  - SQLは使えない（Phoenixなどを使えば可能）
  - トランザクションは1つの Row の内部のみ（Omidなどを使えば可能）

HITACHI
Inspire the Next

3. スマートメータのデータを使用したHBaseの検証結果の紹介

ⅰ. 検証シナリオ
ⅱ. 格納性能の検証
ⅲ. 圧縮性能の検証
ⅳ. 参照性能の検証

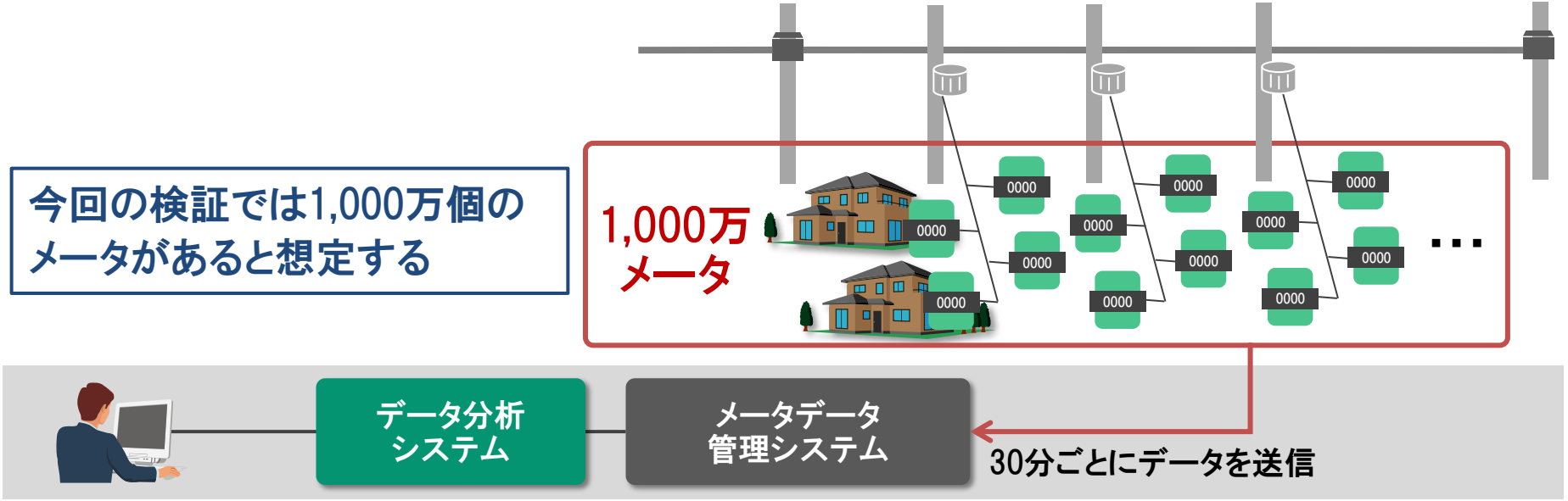
HITACHI
Inspire the Next

i. 検証シナリオ

検証シナリオ： スマートメータから収集したデータの管理

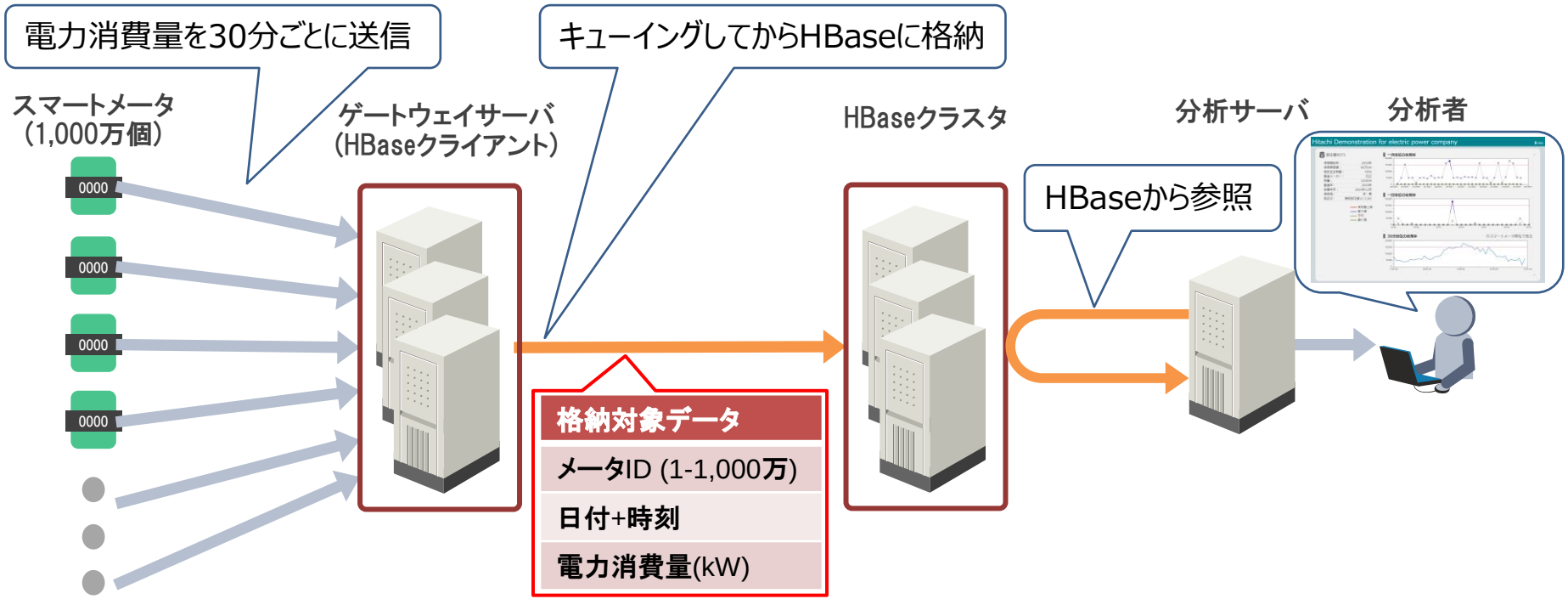
HITACHI
Inspire the Next

- スマーメータは、電力の需要家から電力消費量などのデータを収集
  - 各メータが収集したデータは、30分ごとに電力事業者へ送信
  - 料金計算や電力需要予測などの分析に利用される



想定するシステムとデータ

- 30分おきに発生する1,000万レコードをHBaseに格納する
- HBaseに格納したレコードを分析するために参照する

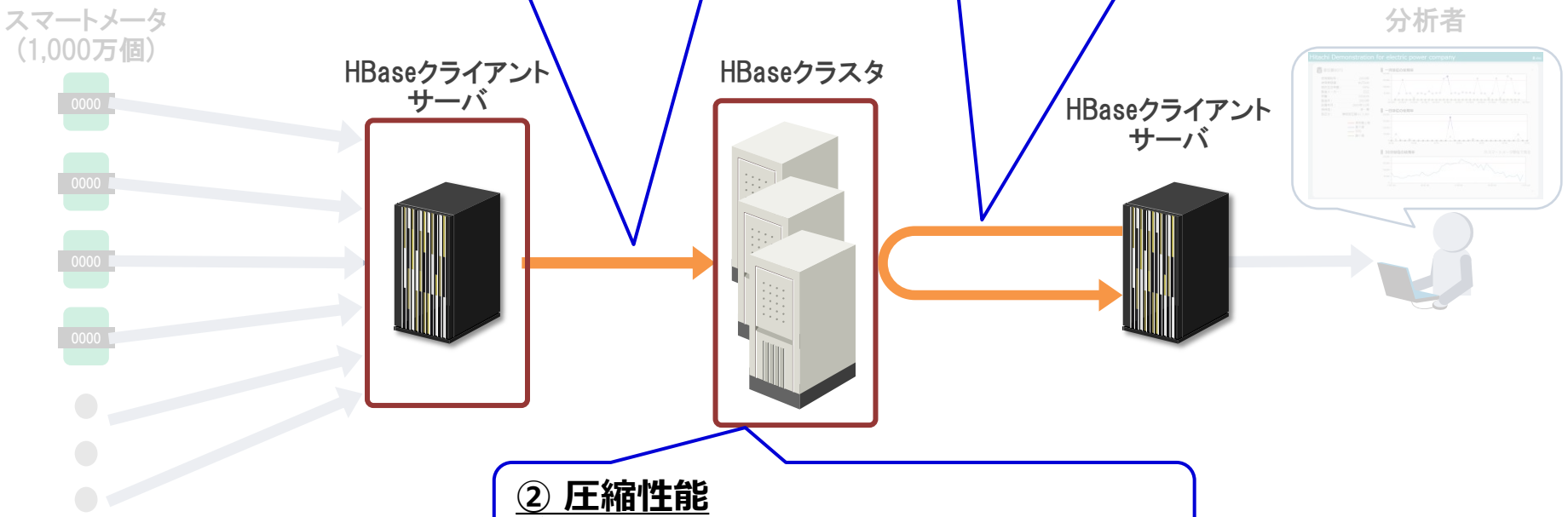


検証項目

HITACHI
Inspire the Next

① 格納性能
・1,000万件のレコードの書き込み時間を測定
・レコードはクライアントサーバで生成

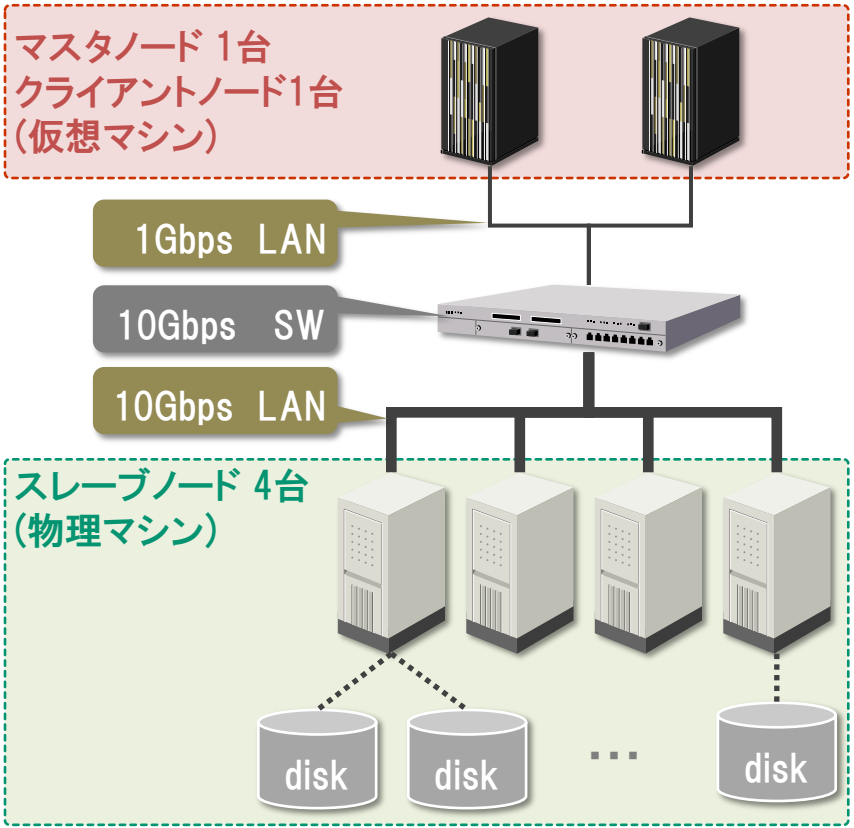
③ 参照性能
参照ユースケースごとの読み出し時間を測定



② 圧縮性能
データの圧縮率と、圧縮にかかる時間を測定

検証環境

ソフトウェア構成
CDH5.9 (HBase1.2.0 + Hadoop2.6.0)



	クライアントノード	マスタノード
CPUコア数	16 コア	2 コア
メモリ容量	12 GB	16 GB
ディスク台数	1 台	1 台
ディスク容量	80 GB	160 GB

	1ノード	4ノード合計
CPUコア数	32 コア	128 コア
メモリ容量	128 GB	512 GB
ディスク台数	6 台	24 台
1ディスク容量	900 GB	-
ディスク合計容量	5.4 TB (5,400 GB)	21.6 TB (21,600 GB)

テーブルの設計： Regionを事前に分割 + Tall Tableを使用

HITACHI
Inspire the Next

- Region数は400個（1RegionServerあたり100個）を初期値とする
  - デフォルト設定ではJavaヒープ(31GB)の40%(12.4GB)をMemStore用に割り当てるため、これを使い切るためには 12.4GB ÷ MemStoreのフラッシュサイズ128MB ≒ 100個となる
- テーブルを事前に400Regionに分割（分割キーは0001, 0002, …, 0399）

リージョン間でデータを分散するために、格納するレコードのRowKeyの先頭に0000～0399(メータID % 400)を付加する。Salt と呼ばれる手法

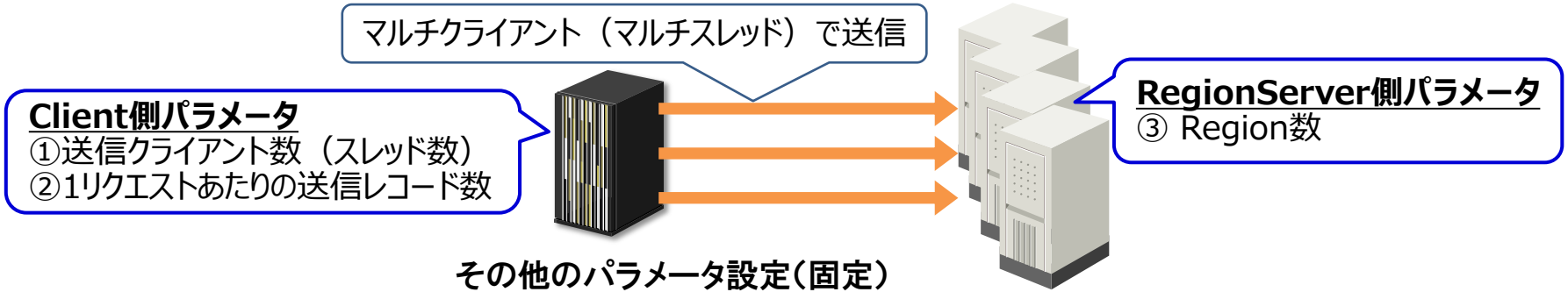
RowKey (<Salt>-<メータID>-<日付>-<時刻>)	ColumnFamily:Column	Timestamp	Type	Value (電力消費量)
0000-0000000001-20170310-1100	CF:		Put	3.241
0000-0000000001-20170310-1030	CF:		Put	0.863
…	…		Put	0.430
0000-0000000001-20160910-1100	CF:		Put	0.044
0001-0000000002-20170310-1100	CF:		Put	2.390
…	…		Put	1.432

HITACHI
Inspire the Next

ii. 格納性能の検証

格納性能の検証： 検証内容

- パラメータチューニングを行い、1,000万レコードの格納時間を測定する
  - A) HBase Java Client のパラメータチューニング
  - B) HBase Region Server のパラメータチューニング



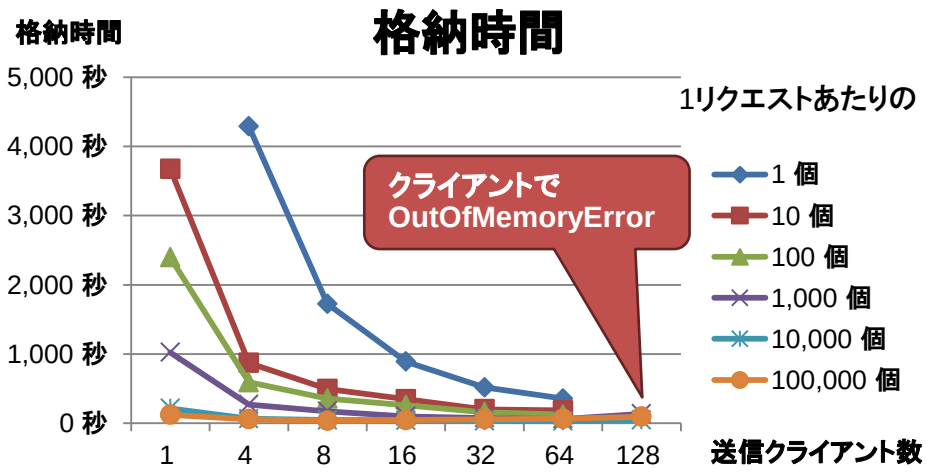
その他のパラメータ設定（固定）

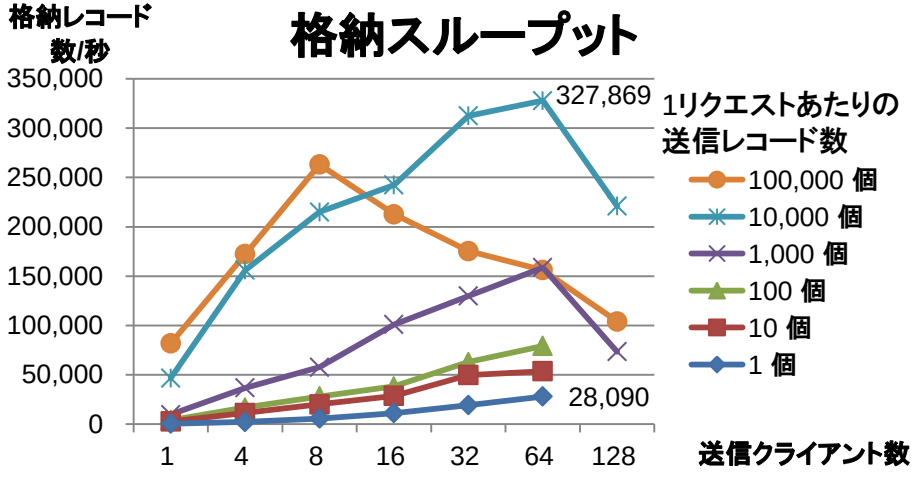
#	コンポーネント	パラメータ内容	パラメータ名	設定値
1	Client	Java ヒープサイズ	-Xmx	10 GB
2		送信バッファサイズ	hbase.client.write.buffer	32 MB (デフォルト2MB)
3	RegionServer	メジャーコンパクション実行間隔	hbase.hregion.majorcompaction	0 日（無効化）
4		Java ヒープサイズ	-Xmx	31 GB
5		MemStore のフラッシュサイズ	hbase.hregion.memstore.flush.size	128 MB
6		ハンドラ数(リクエスト処理スレッド数)	hbase.regionserver.handler.count	130 (デフォルト30)

A) Clientのパラメータチューニング結果

HITACHI
Inspire the Next

① 送信クライアント数 + ② 1リクエストあたりの送信レコード数 を変動させたときの格納時間と格納スループット(1秒当たりの格納レコード数)



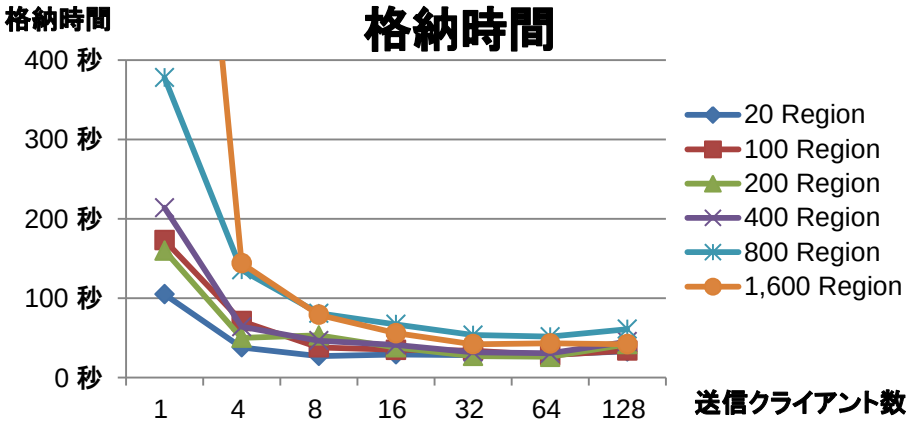


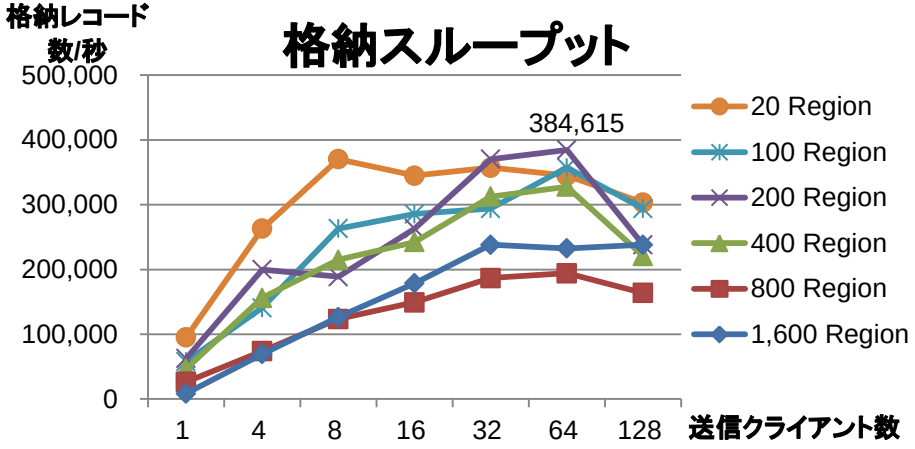
- 送信クライアント数が多いほど格納は速い (128クライアント以上は測定不能)
- 1リクエストあたりの送信レコード数は、10,000までは増やすほど速くなる
- ① 送信クライアント数=64, ② 1リクエストあたりの送信レコード数=10,000 で格納スループットは 約33万レコード/秒 となった

B) RegionServerのパラメータチューニング結果

HITACHI
Inspire the Next

- ③テーブルのRegion数を変動させたときの格納時間格納スループット
  - 1リクエストあたりの送信レコード数は10,000個で固定





- Region数が少ない方が格納スループットは高い傾向にある
  - ただし、送信クライアント数が増加すると、ある程度のRegion数が必要となる
- ③ Region数=200, 送信クライアント数=64 で格納スループットは約38万レコード/秒
  - ただし、Region数は保持データ量やMemStoreサイズも考慮して決めるべき

格納性能の検証結果まとめ

HITACHI
Inspire the Next

- HBaseクライアントのチューニングポイント
  - 1リクエストで複数レコードを格納する：1 ⇒ 10,000レコードでスループット約89倍
  - 送信クライアント数を増やす：1 ⇒ 64クライアントでスループット約7倍
    - マルチクライアント環境で性能を発揮するといえる

- RegionServerのチューニングポイント
  - Region数は少ない方がよい：400 ⇒ 200Regionでスループット約1.17倍
    - ただし、適切なRegion数はデータ量やメモリ容量を考慮して決める必要がある
      - 参考：140.2. Determining region count and size
        http://hbase.apache.org/book.html#ops.capacity.regions
        - 適切な Region 数： 1RegionServer あたり 20-200 個
        - 適切な 1Region のサイズ： 5-10 GB
        - 適切な MemStore のフラッシュサイズ： 128-256 MB（Region数が減るとメモリを使い切れない）

※ 各パラメータの最適値は、マシンスペックなど様々な要因に左右されます

HITACHI
Inspire the Next

iii. 圧縮性能の検証

エンコーディングと圧縮アルゴリズムの検証

HITACHI
Inspire the Next

- HBaseはキーバリューストアであるためデータ量が大きくなりやすい
  - 値ごとにキーが付くため、レコード数が増えやすい
  - キーが RowKey, ColumnFamily:Column, Timestamp などで構成されるため長くなりやすい
- 圧縮には、下記2種類の設定を組み合わせる

エンコーディング
キーを差分短縮する

#	エンコーディング	説明
1	PREFIX	RowKeyとColumnを圧縮
2	PREFIX_TREE	トライ構造を利用して圧縮
3	DIFF	RowKeyとColumnに加えてTimestampなども圧縮
4	FAST_DIFF	DIFFと同じだが長い値に対して圧縮率が高い

圧縮アルゴリズム（の一部）
データをブロック(デフォルト64KB)単位で圧縮する

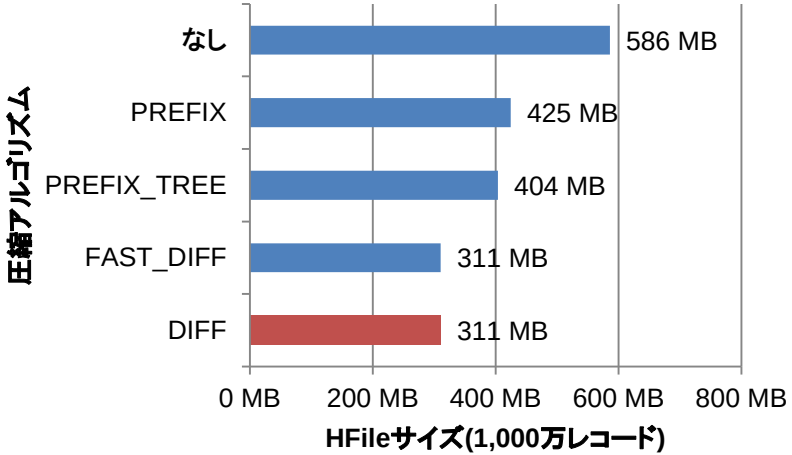
#	圧縮アルゴリズム	説明
1	SNAPPY	圧縮/解凍速度を重視
2	GZ (GZIP)	圧縮率を重視
3	LZ4	圧縮/解凍速度を重視

エンコーディングの検証結果

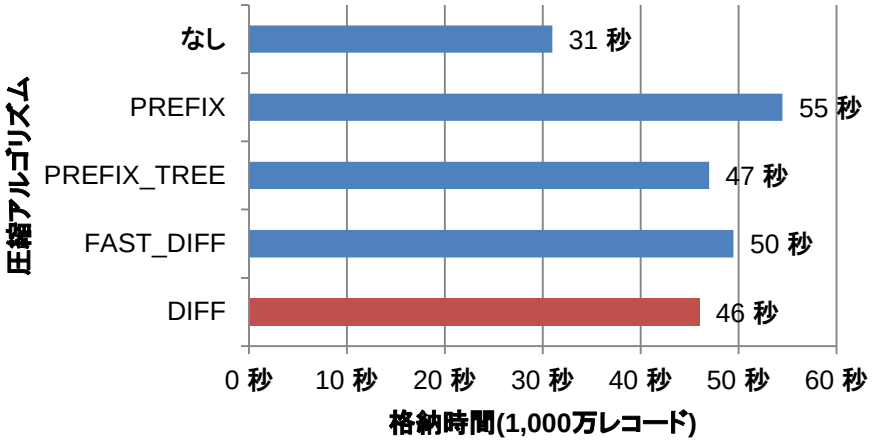
HITACHI
Inspire the Next

1,000万レコードをエンコーディングして格納した際のファイルサイズと格納時間

HFileサイズ



格納時間

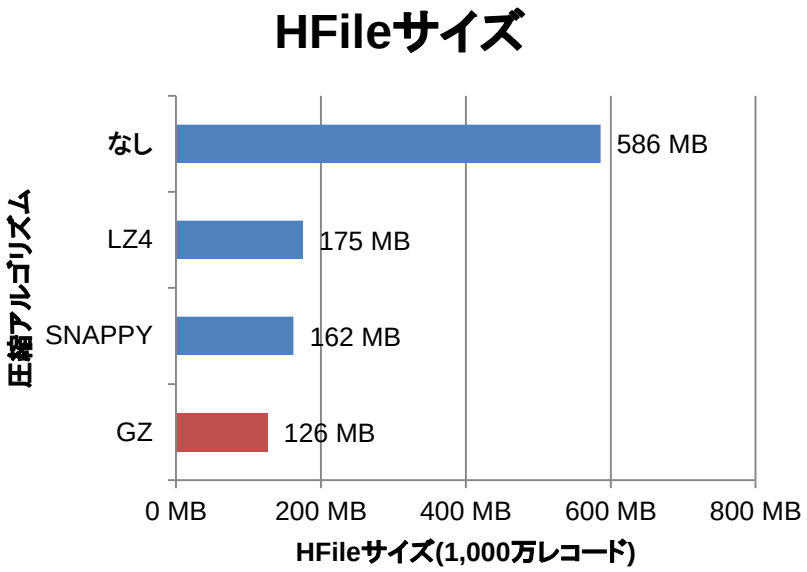


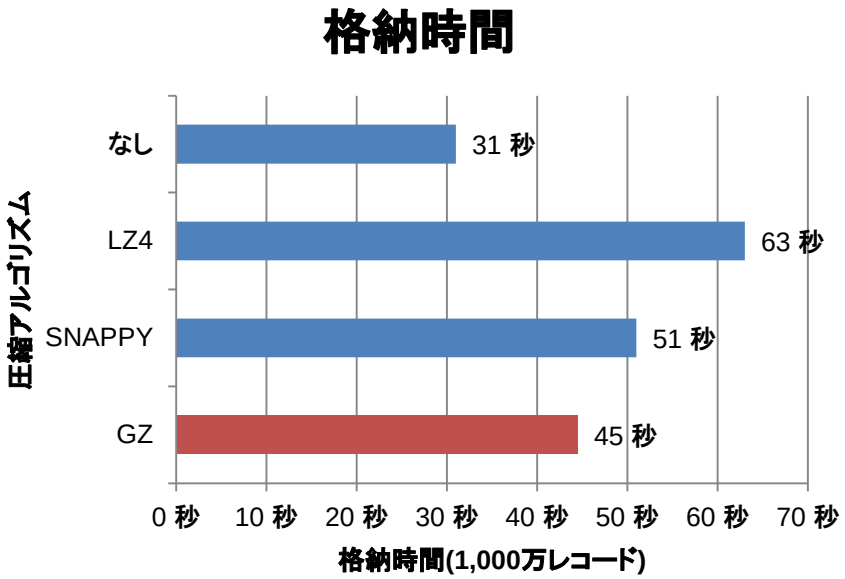
データ量は DIFF により 約53% まで減少したが、格納時間は 約48% 増加した

圧縮アルゴリズムの検証結果

HITACHI
Inspire the Next

1,000万レコードを圧縮して格納した際のファイルサイズと格納時間

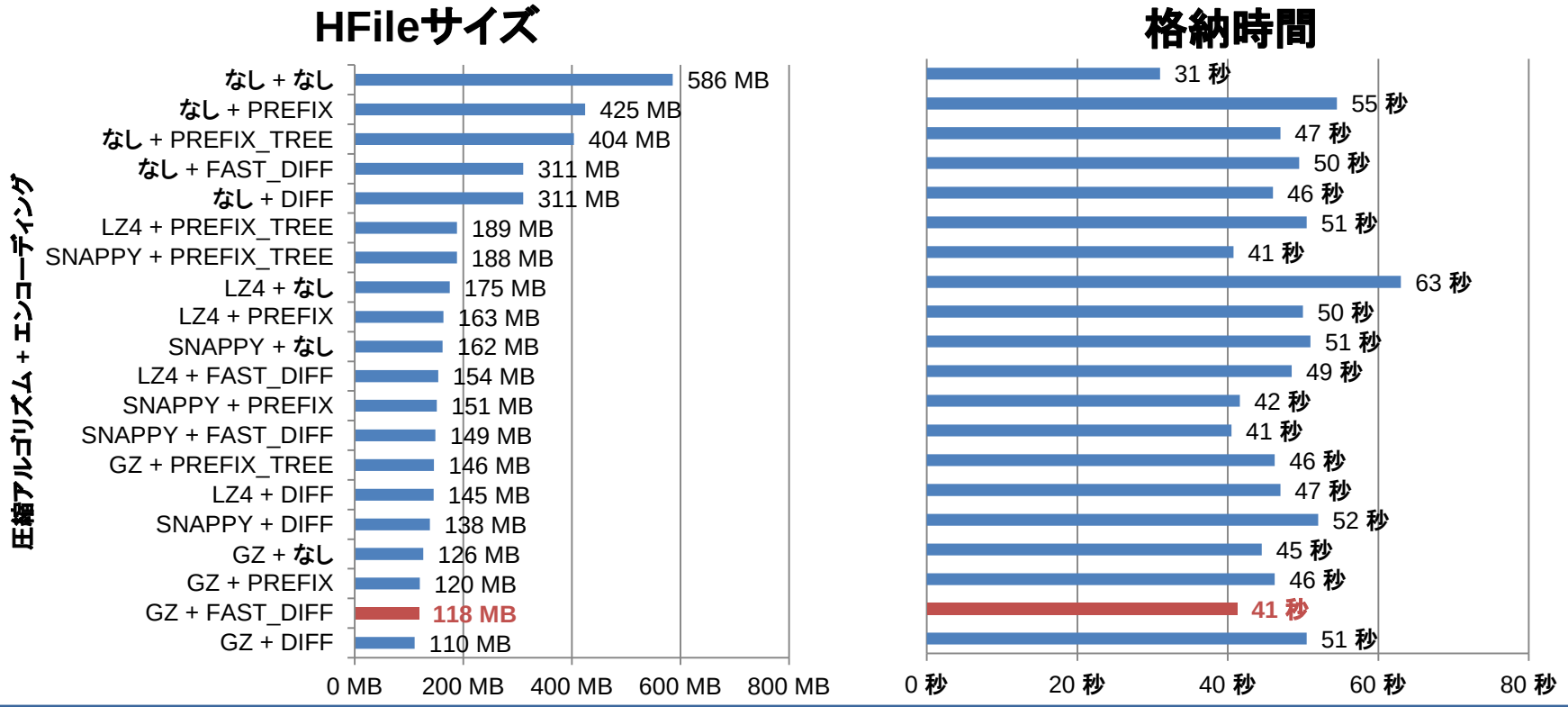




データ量は GZIP により約22%まで減少したが、格納時間は約68%増加した

エンコーディングと圧縮アルゴリズムを組み合わせた場合の検証結果

HITACHI
Inspire the Next



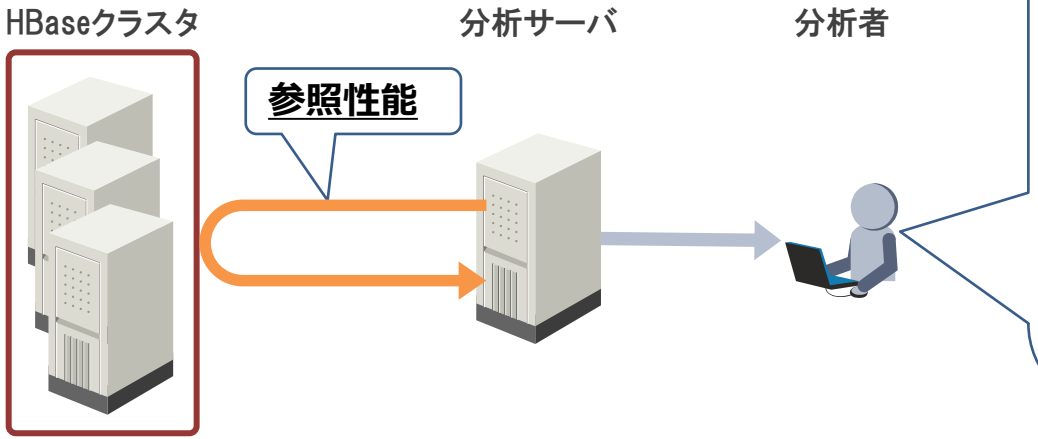
FAST_DIFFエンコーディング ＋ GZIP圧縮 を採用(サイズは19%に減少, 格納時間33%増加)

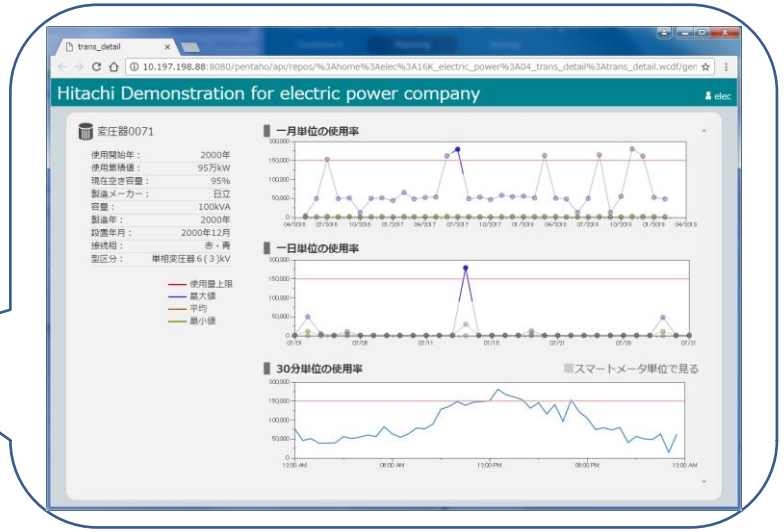
HITACHI
Inspire the Next

iv. 参照性能の検証

参照性能の検証： 検証内容

- **データ参照のユースケース：電力消費量の可視化**
  - ➢ **可視化に必要なデータの取得時間を測定する**
- **検証するユースケース**
  - A) **少数メータの時系列の電力消費量を取得する**
    - ➢ 各メータの電力消費量の推移のチャート表示を想定
  - B) **多数メータの最新の電力消費量を取得する**
    - ➢ 直近の電力消費量の合計値、平均値計算を想定





パラメータ設定

HITACHI
Inspire the Next

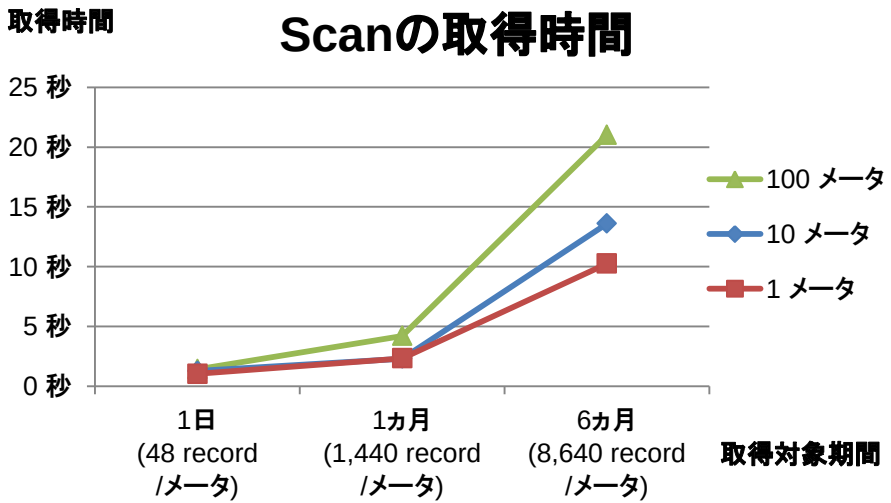
- キャッシュを無効化して、データを必ずディスクから読み出すように設定
  - HBaseとHDFSのキャッシュを無効化（表の#8,9）
  - 測定前にOSのページキャッシュをクリア
- データは圧縮済み（FAST_DIFFエンコーディング ＋ GZIP圧縮）

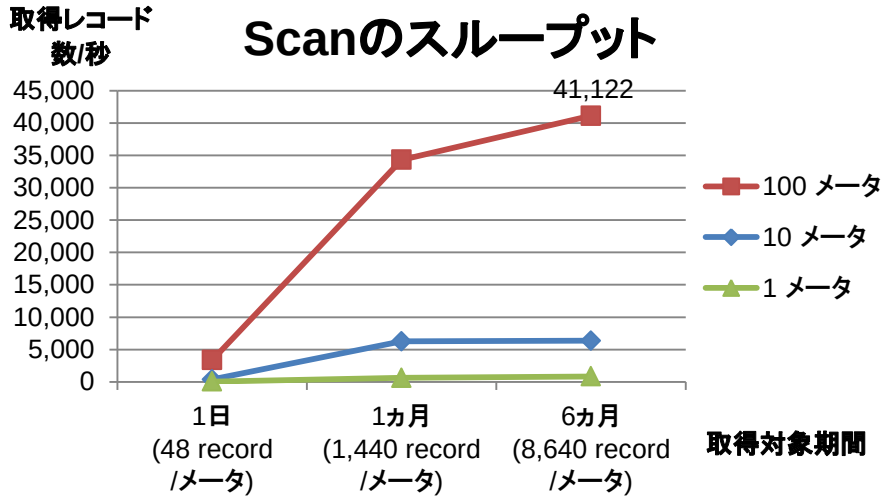
#	コンポーネント	パラメータ内容	パラメータ名	設定値
1	Client	Java ヒープサイズ	-Xmx	10 GB
2		送信バッファサイズ	hbase.client.write.buffer	32 MB (デフォルト2MB)
3	RegionServer	メジャーコンパクション実行間隔	hbase.hregion.majorcompaction	0 日（無効化）
4		Java ヒープサイズ	-Xmx	31 GB
5		MemStore のフラッシュサイズ	hbase.hregion.memstore.flush.size	128 MB
6		ハンドラ数(リクエスト処理スレッド数)	hbase.regionserver.handler.count	130 (デフォルト30)
7		Region数	テーブル作成時にコマンドで指定	400
8		JavaヒープのうちBlock Cacheに使用する割合	hfile.block.cache.size	0.0 (無効化)
9	HDFS	キャッシュに使用する最大メモリ	dfs.datanode.max.locked.memory	0 (無効化)

A）少数メータの時系列の電力消費量を取得（Scanで取得）

HITACHI
Inspire the Next

- 1-100メータの、1日-6ヶ月分のデータ取得にかかる時間を測定
  - 1回のScanで1メータの時系列データを取得



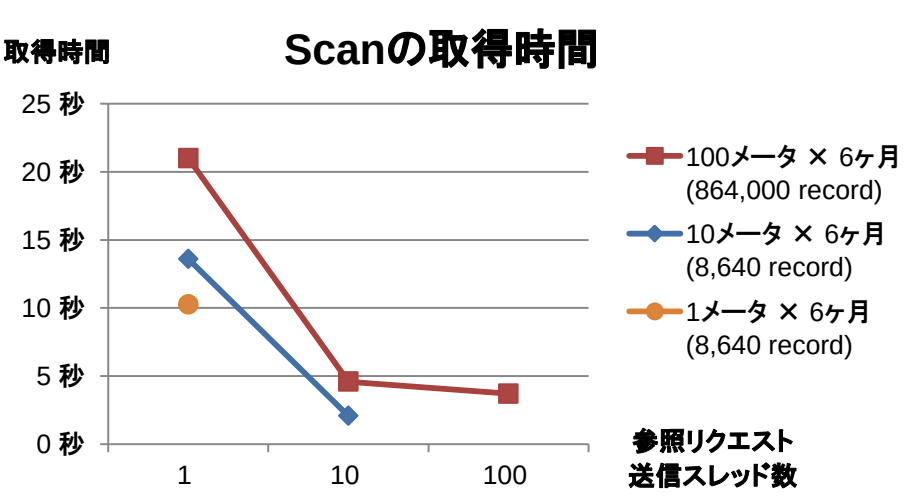


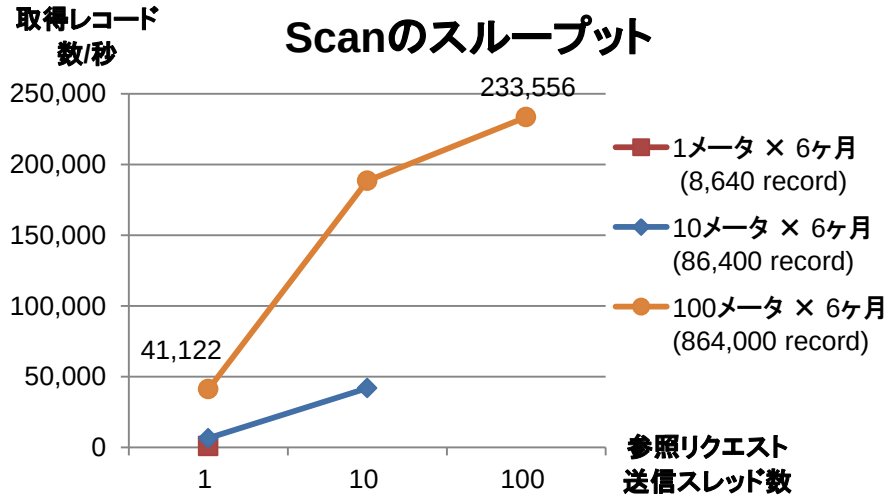
取得対象期間が長いほどスループットは向上する ⇒ Scanでまとめて取得できるため

A）少数メータの時系列の電力消費量を取得（Scan+マルチスレッドで取得）

HITACHI
Inspire the Next

- 1-100メータの、6ヶ月分のデータ取得にかかる時間を測定
  - 参照リクエストをScan数と同数のスレッドで並列実行（1Scan1スレッド）





- 複数メータをScanする場合、取得スレッド数を増やすとスループットが向上する
  - 約4万レコード/秒 ⇒ 約23万レコード/秒（約6倍）
  - これは、複数メータのScanを並列に行ったためと考えられる

A) まとめ：Scanでまとめて取得することで高いスループットを発揮

HITACHI
Inspire the Next

- HFile内のデータは Salt, メータID, 日付, 時刻 の順でソートされている
  - 各メータの時系列データをScanでまとめて取得すると、スループットは**約4万レコード/秒**
- 複数メータの時系列データは、複数のScanを並列実行して並列に取得可能
  - 100スレッドで並列実行すると、取得スループットは**約23万レコード/秒**

メータID=1の時系列データを
Scanでまとめて取得

メータID=201の時系列データを
Scanでまとめて取得

RowKey (<Salt>-<メータID>-<日付>-<時刻>)	…	Value (電力消費量)
0000-0000000001-20170310-1100		3.241
0000-0000000001-20170310-1030		0.863
…		…
0000-0000000001-20160910-1100		0.044
…		…
0200-0000000201-20170310-1100		10.390
0200-0000000201-20170310-1030		14.325
…		
0200-0000000201-20160910-1100		9.32
…		

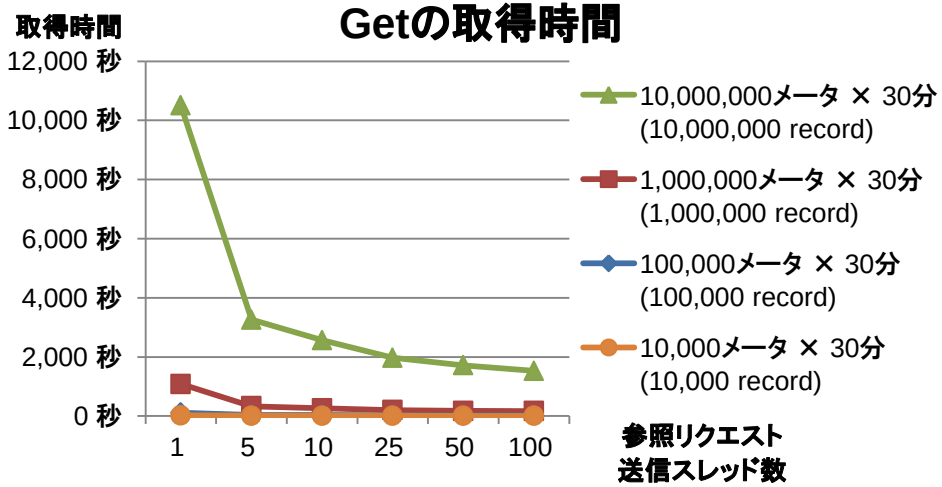
Region A

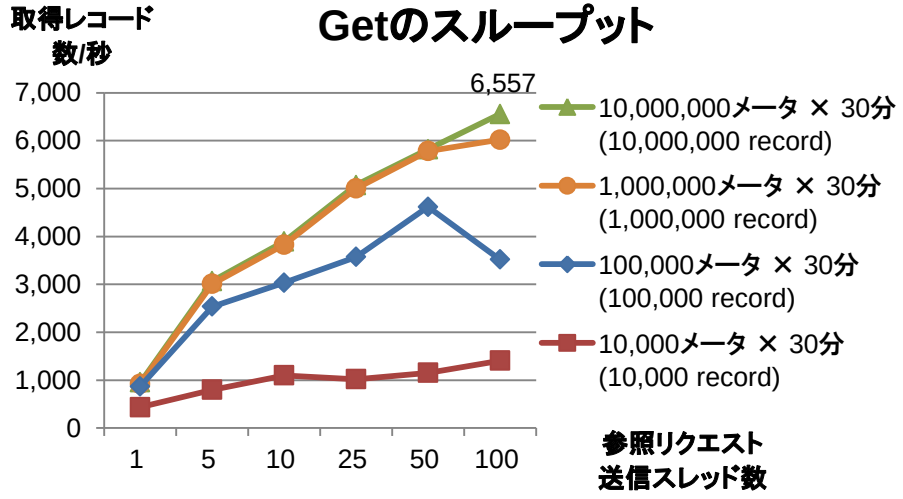
Region B

B) 多数メータの最新の電力消費量を取得する(Get+マルチスレッドで取得)

HITACHI
Inspire the Next

- 1万-1,000万メータの、最新時刻(30分間)のデータ取得にかかる時間を測定
  - RowKeyの並び上、Scanはできないため、Getリクエストを使用
  - 参照リクエストはマルチスレッドで実行し、かつ1リクエストで複数のGetをバッチ実行





- 取得の並列度を上げることでスループットは向上(約900 ⇒ 約6,500レコード/秒)
- それでも、Getのスループットは Scan(約23万レコード/秒)と比較すると低い

B) まとめ：Scanできない場合はランダムアクセスになり時間がかかる

HITACHI
Inspire the Next

- 各メータの特定時刻のデータは離れた位置に格納 ⇒ Scanできない
  - Getで1個ずつ取得するため、スループットは**約900レコード/秒**
- 複数のGetを並列に実行すれば、ある程度は高速に取得できる
  - 100スレッドでスループットは**約6,500レコード/秒**

各メータの最新時刻のデータに
1個ずつアクセスして取得

RowKey (<Salt>-<メータID>-<日付>-<時刻>)	…	Value (電力消費量)	
0000-0000000001-20170310-1100	…	3.241	Region A
…	…	…	
0000-0000000401-20170310-1100	…	0.863	
…	…	…	
0200-0000000201-20170310-1100	…	10.390	Region B
…	…	…	
0200-0000000601-20170310-1100	…	23.432	
…	…	…	

参照性能の検証結果まとめ

HITACHI
Inspire the Next

- Scanでまとめて取得できるようにRowKeyを設計することが重要
  - 本検証では、**ScanのスループットはGetの35倍以上**であった
    - Scan: 約23万レコード/秒　vs　Get:約6,500レコード/秒
    - これはデータ形式やデータサイズ、Scanするレコード数などに左右される

- 複数のScan/Getをマルチスレッドで実行することで取得性能が向上
  - 1 ⇒ 100スレッドで、取得スループットは Scanで約5.7倍、Getで約7.2倍
  - これはRegion数やマシンスペックなど様々な要因に左右される

HITACHI
Inspire the Next

4. まとめ

まとめ

HITACHI
Inspire the Next

- HBaseはワイドカラム型のNoSQLであり、キーバリューストアでもある
  - キーバリューをソートして格納するため、時系列データの格納に適しており、IoT(Internet of Things)と相性が良い
- HBaseの性能検証でわかったこと
  - データをScanでまとめて取得できるように、うまくRowKeyを設計する必要がある
  - 格納/参照ともに、マルチクライアント/マルチスレッド環境で性能を発揮する
  - データ量が大きくなりやすいため、データを圧縮すべき

他社所有商標に関する表示

- 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。
- その他記載の会社名、製品名は、それぞれの会社の商標もしくは登録商標です。