

分散処理アプリ演習 第14回

HBaseスキーマ設計

(株)NTTデータ



講義内容

1. HBase スキーマ設計

- スキーマ設計のポイント、行キー設計、列ファミリ設計

2. Twitterのツイートログ

- ツイートログの紹介、【演習】ツイートログを格納するテーブルを設計

3. HBase の Java API

- Java APIの紹介、【演習】ツイートログをHBaseに格納するアプリ演習



1. HBase スキーマ設計

HBase の特徴から見るスキーマ設計

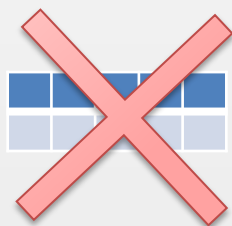
■ HBaseの特徴から見るスキーマ設計のポイント

■ 単純なクエリのみ → HBaseではテーブルを結合した検索ができない

- 検索処理に必要なデータをすべて結合したテーブルを作る
- RDBMSのように正規化せず、重複して持たせる

■ インデックスは行キーのみ

- 検索条件となる要素はできるだけ行キーに入れる
- 「少ない行、多くの列」より、「多くの行、少ない列」



■ トランザクションは行単位

- 1つのトランザクションで扱いたいデータのペアがあれば、同一行に入るように設計



スキーマ設計のポイント

- 設計箇所ごとにポイントを説明
 - 行キー設計のポイント
 - 列ファミリ設計のポイント



行キー設計のポイント

■ 行キー設計のポイント

- リージョン分割の仕組みを理解し、データの書き込みの際、対象となるRegionServerが偏らないように設計する
 - 単調増加、単調減少を行キーの先頭にしない
- データ読み込み時のHFile読み込みロジックを理解し、より短時間で目的のデータに辿りつけるように設計する。
 - 検索条件となる要素はできるだけ行キーに入れる
 - 行キーでソートした際、よく検索される行がはじめに来るようにする



行キー設計 -書き込みRegionSeverを分散- (1/5)

- 行キーをツイート IDにした場合を考えてみる
 - ツイート ID: ツイートごとに払い出されるID。単調増加する。

- ツイート IDを行キーにした場合の tweet テーブル

ツイート ID	ツイート
1	1つ目のつぶやき
2	2つ目のつぶやき
3	3つ目のつぶやき
4	4つ目のつぶやき
5	5つ目のつぶやき



6

6つ目のつぶやき

新たなつぶやきがあると...

行キー設計 -書き込みRegionSeverを分散- (2/5)

- はじめは、Regionが1つであり、全データが1つのRegionServer#1上にある
- Clientからの書き込みはすべてこのRegionServer#1が処理する

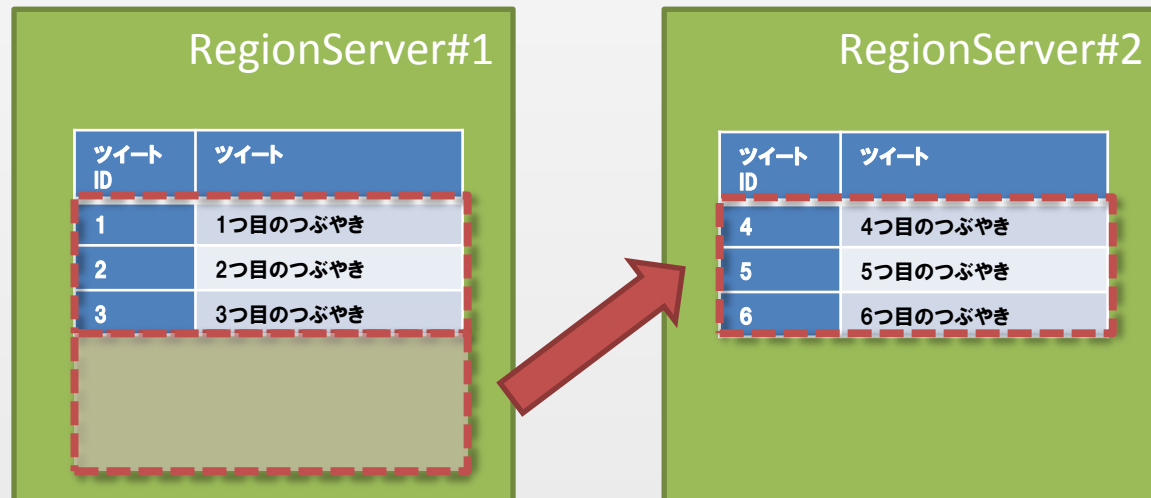




行キー設計 -書き込みRegionSeverを分散- (3/5)

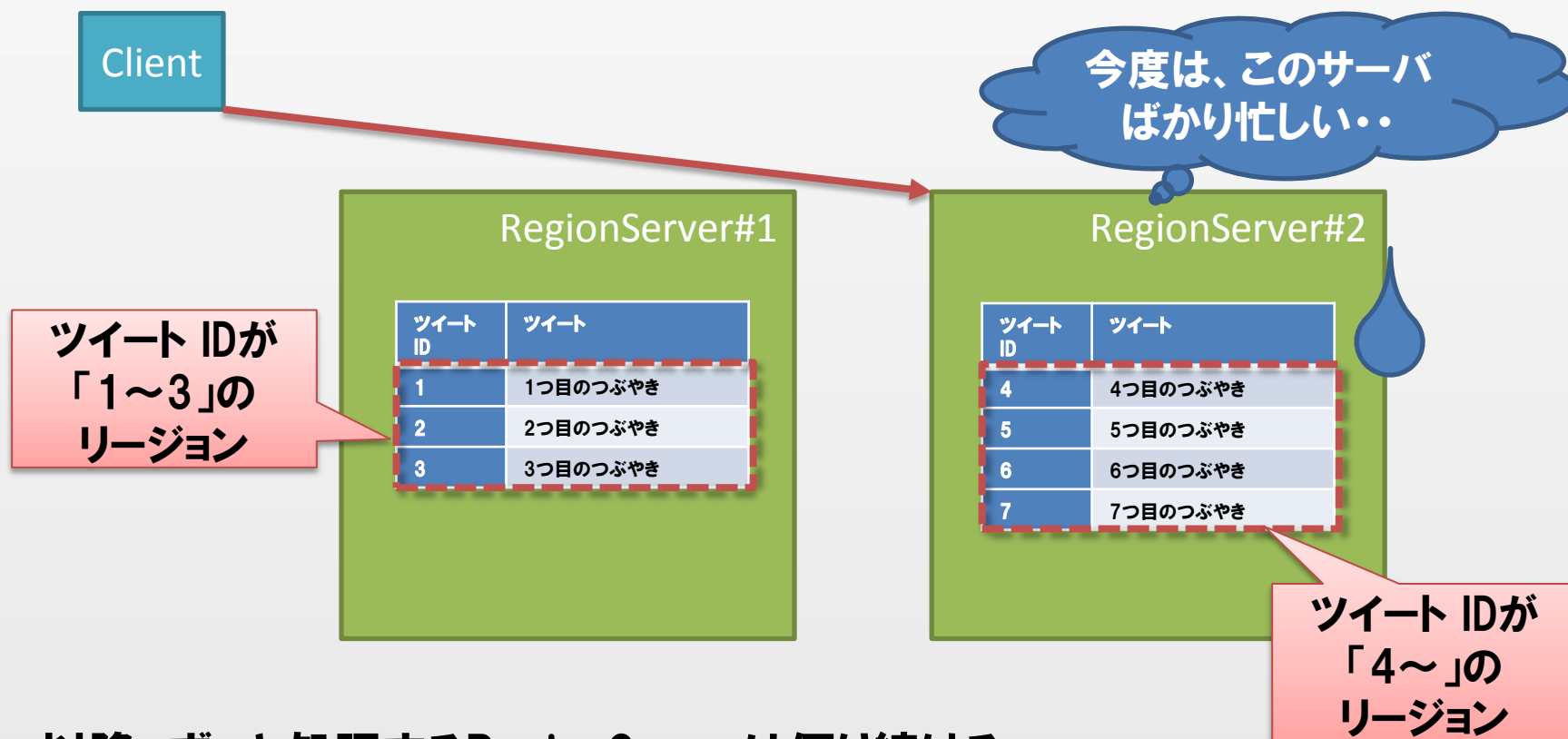
- どんどんつぶやかれ、ストアのサイズがしきい値を超えると、スプリットされる

Client



行キー設計 -書き込みRegionSeverを分散- (4/5)

- ツイート IDは、単調増加であるため、7つ目以降のつぶやきは、すべて RegionServer#2で処理される



- 以降、ずっと処理するRegionServerは偏り続ける...



行キー設計 -書き込みRegionServerを分散- (5/5)

- 行キーに単調増加する値を持ってきた場合、データを書き込むリージョンが常に1つになるため、そのリージョンをもつRegionServerに負荷が集中してしまう。
 - 行キーに単調増加する値を利用する場合の解決策としては、ハッシュや十分分散された値を行キーの先頭に追加する方法がある
 - 例
 - User IDを行キーの先頭に追加する
 - キーをハッシュ値に変換したものを行キーの先頭に追加する
 - RegionServer数のモジュロを取ったものを行キーの先頭に追加する
- 単調増加でなくても、特定の行キーに書き込みが集中してしまうような行キー設計を行うと、暇なリージョンと忙しいリージョンができてしまい、特定のRegionServerに負荷が集中することになる。
 - 各リージョンに均等に負荷がかかるよう行キーを設計する必要がある。
- 行キーでソートした時に、書き込み先が散らばるような行キーにすることがポイント



【参考】事前にリージョンを分割する

- HBaseでは、リージョン分割の仕組み上、最初はリージョンが1つであるため、どうしても処理が偏ってしまう。
- そのまま、運用していれば、自動的にリージョンが分割され、負荷が分散されることで、性能は改善される。
- ただ、このサービス開始直後の性能を担保する必要がある場合は、事前に、ある程度リージョンを作っておくことで、開始直後から負荷を分散させることができる。
 - 1コマ目に説明した、HBase shell にて、splitコマンドを利用して手動でスプリットすることも可能



行キー設計 -読み込みコストを削減- (1/5)

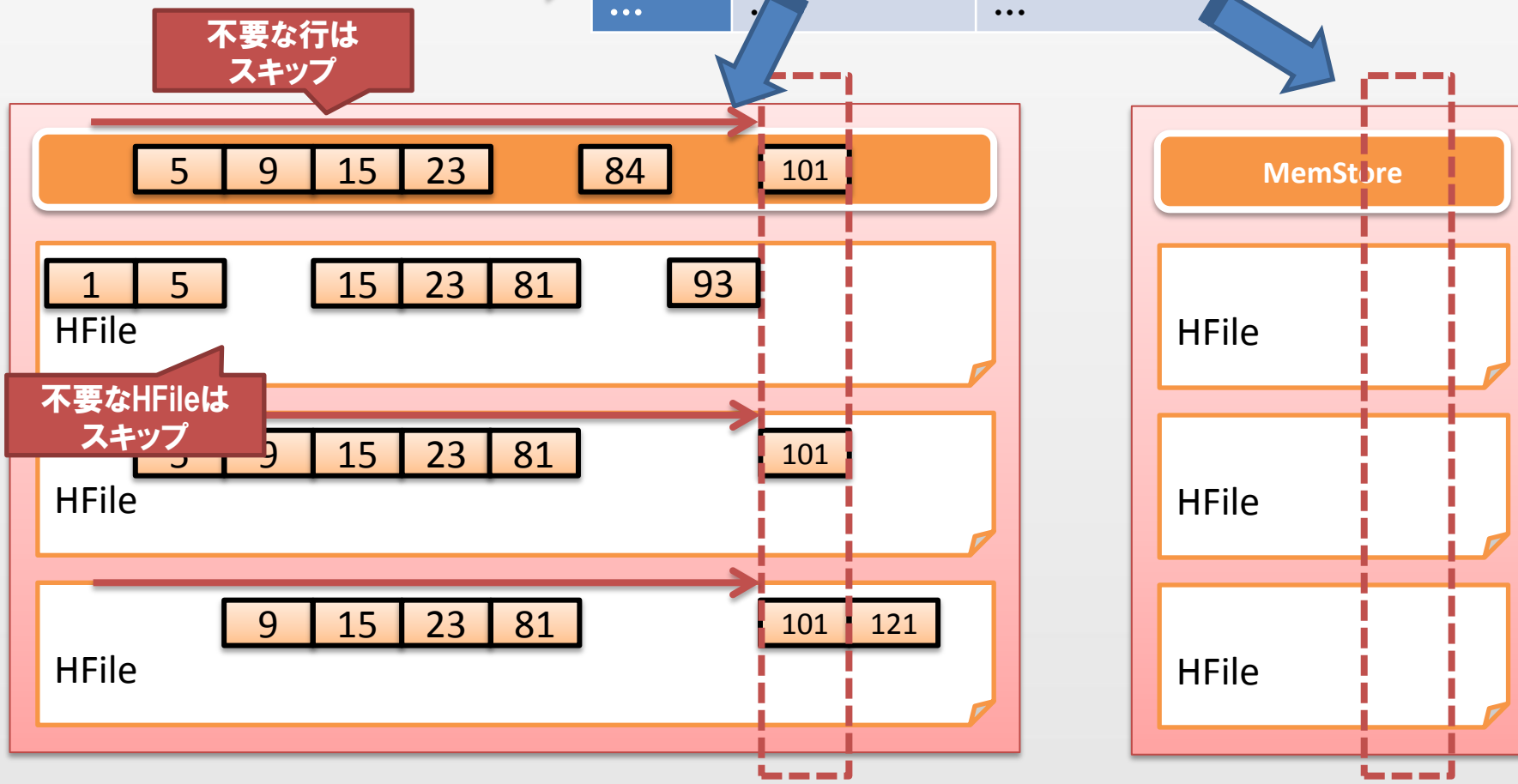
- 読み込み処理では、すべてのHFileとMemStoreから該当のデータを読み込み、データの再構成が行われる。
- ただし、検索条件による読み込みの効率化はできる

行キー設計 -読み込みコストを削減- (2/5)

■ 検索条件による読み込みの効率化

■ 行キーを指定した場合

ツイートID	ツイート	User ID
...
101	101つ目のつぶやき	51
...






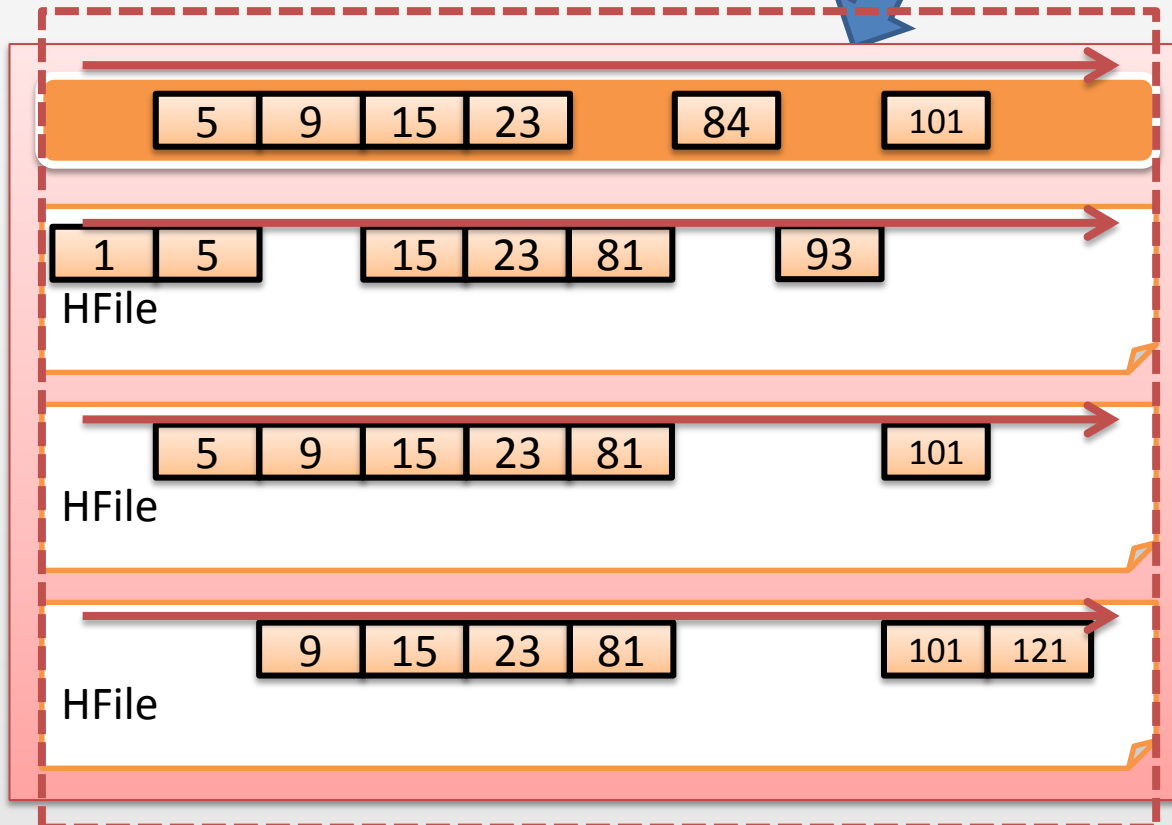

行キー設計 -読み込みコストを削減- (3/5)

■ 検索条件による読み込みの効率化

■ 列ファミリを指定した場合



ツイートID	ツイート	User ID
...
101	101つ目のつぶやき	51
...



不要なストアは
スキップ

MemStore

HFile

HFile

HFile



行キー設計 -読み込みコストを削減- (4/5)

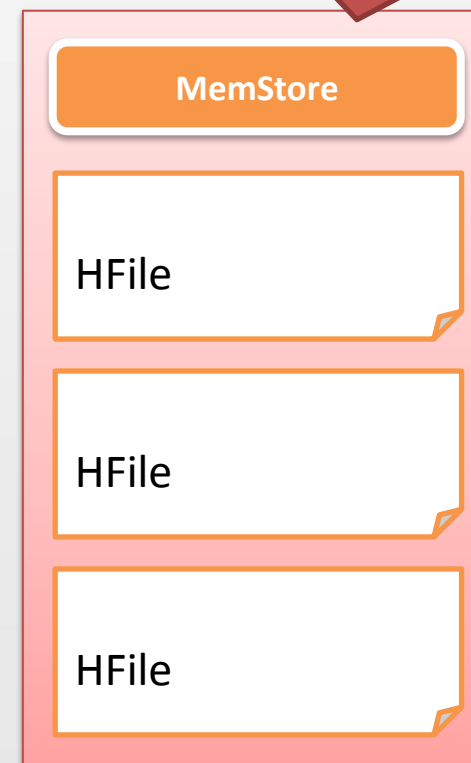
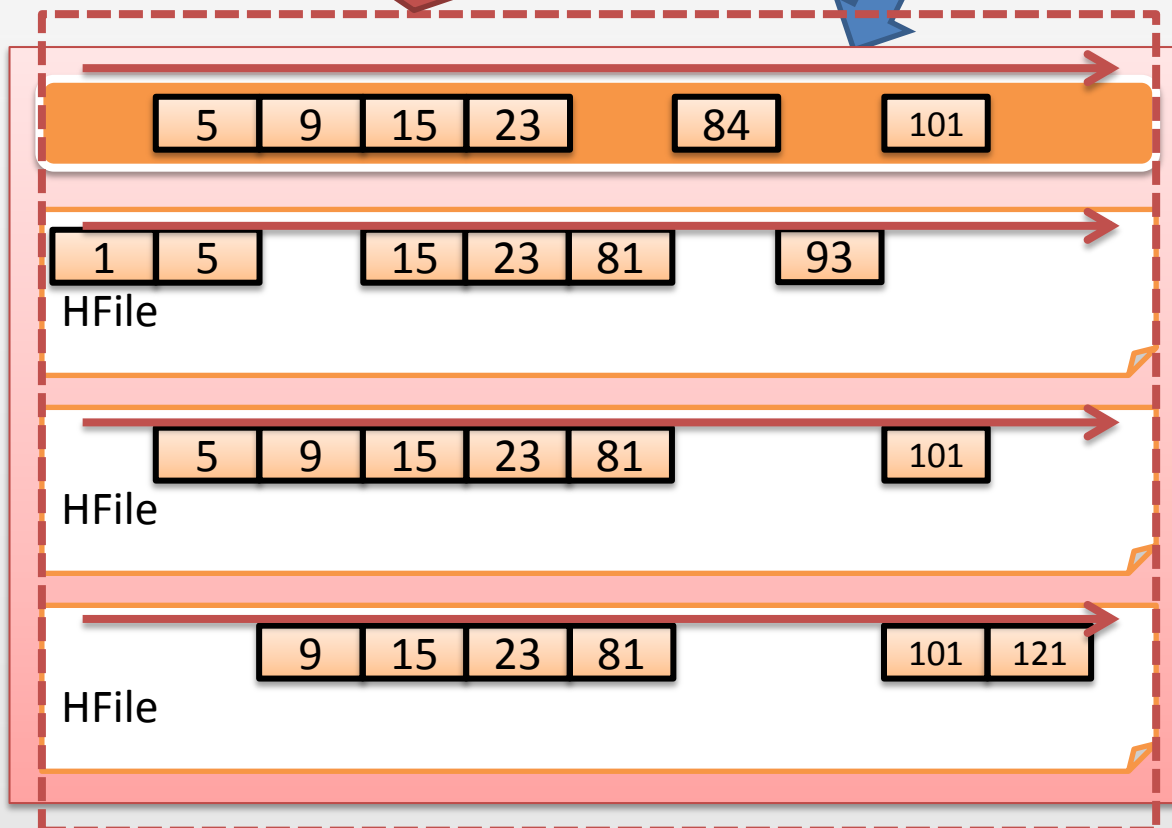
■ 検索条件による読み込みの効率化

■ 修飾子やセルの値を条件にする場合

すべてのHFileを読み込んで、
列を再構築後、比較する

ツイートID	ツイート	User ID
...
101	101つ目のつぶやき	51
...

不要なストアは
スキップ





行キー設計 -読み込みコストを削減- (5/5)

■ 検索条件による読み込みの効率化

- HFile内で行キーはソート済みであるため、行キーの検索は性能が良い
 - また、検索条件に行キーを指定すると、不要なHFileの読み込みもスキップできる
 - 対して、列ファミリの修飾子やセル値を条件とした場合、行の再構築が必要であるため、HFileのスキップは行われない。
-
- そのため、検索条件になりうる値については、行キーに含める方が読み込み効率が良い。



【参考】効率的な読み込みのテクニック

- また、読み込み時の効率化として、以下のテクニックもある
 - 合わせて取得されることが多い行同士を隣接させることで、ブロックキャッシュによるヒット率を高めることができる
 - 例えば、UserIDを行キーにすることは、同一ユーザの情報をまとめて取得する場合に有効
 - 同様に、単一行内で同時に使うデータは同じ列ファミリ(の別の列)として格納するとよい
- 行キーを複合キーで作る場合は、荒い順につなげると範囲検索がしやすい
 - 行キーは、バイト配列で格納されており、バイト単位で比較される。そのため、行キーでのソートは、以下のようになる

行キー	列ファミリ
100	100の値
100_aaa	100のaaa
200	
200_bbb	

- 行キーの範囲指定($100 \leq \text{行キー} < 200$)により、100で始まる行が効率的に取得できる



列ファミリ設計のポイント（1/3）

■ 列ファミリ設計のポイント

■ 列ファミリ数はなるべく抑えるよう設計する

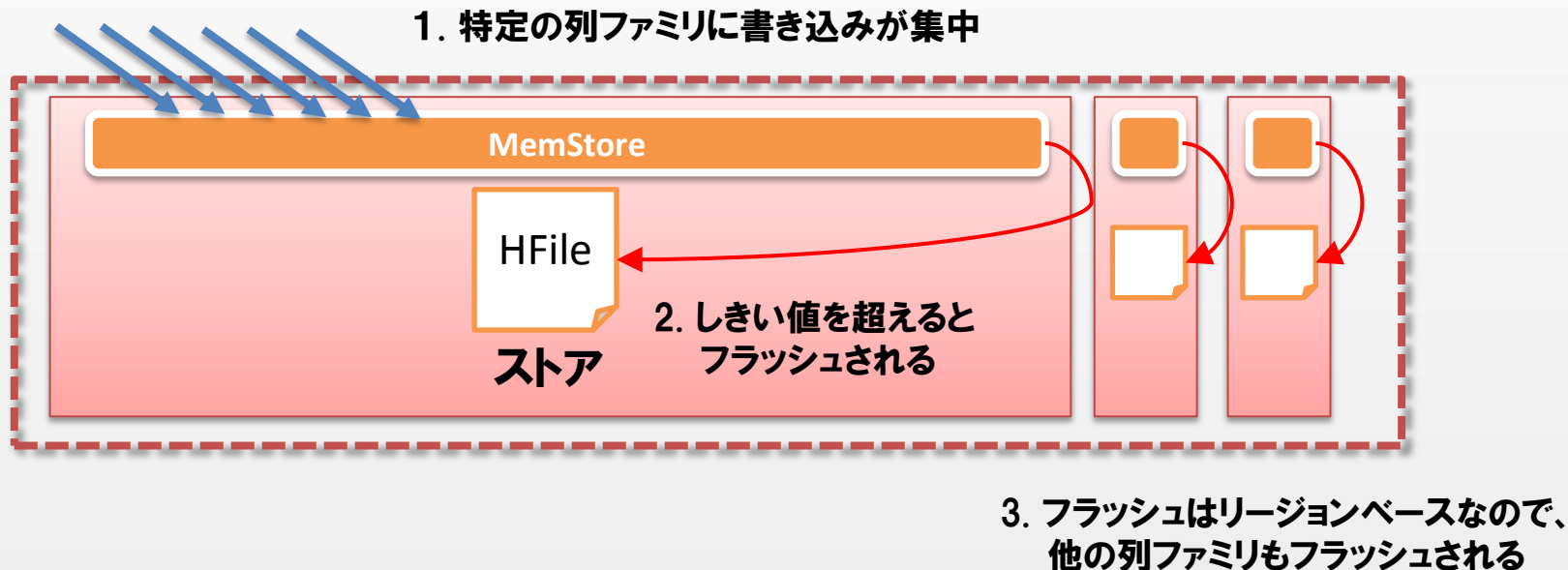
- 「多くの行、少ない列」
- フラッシュ/コンパクションがリージョンベース
- 柔軟なテーブル構成

■ ただし、扱い方・アクセス方法の異なる値は列ファミリを分ける

- ストアファイルの肥大化
- ブロックキャッシュが効きづらく、パフォーマンスが劣化

列ファミリ設計のポイント（2/3）

■ 極端に使用頻度が違う列ファミリが同一テーブルにあった場合



- コンパクションも同様に1つの列ファミリのHFile数によって実行される。
- 列ファミリ数が多いと、フラッシュとコンパクション処理によって、不必要にI/O負荷が高まる可能性がある。
- また、スプリットもリージョン内の最大のHFileで閾値確認し実行されるため、データが少ない列ファミリは、過度にたくさんのリージョンに散らばってしまう。



列ファミリー設計のポイント（3/3）

- 列ファミリー数はなるべく抑えるよう設計するのがポイント
 - HBaseでは、列ファミリー内の修飾子の追加は、HBaseの停止なしで実行することが可能
 - ただし、扱い方・アクセス方法の異なる値を1つの列ファミリーにしてしまうと、HFileファイルが肥大化し、読み込み時のキャッシュも効きづらくなってしまうため、パフォーマンスが劣化することがある。



【参考】その他のポイント

■ 列ファミリー名や修飾子の長さによる影響

- 前回に確認したように、HBaseでは、以下のようなKeyValueペアでデータを保存している。
 - Key: 行キー+ 列ファミリー名:修飾子+バージョン(タイムスタンプ)+ 操作名+ *value*のbyte配列サイズ
 - Value: セルの値
- 列ファミリー名や修飾子は、全てのペアに含まれているため、できるだけ短くすることで、データサイズを抑えることが出来る。



2. Twitterのツイートログ



Twitterのツイートログ

- Twitterには、Twitter APIが用意されており、タイムラインに流れるログをXML形式やJSON形式で取得することができます。
- 今回の演習では、擬似的に作成したツイートログ (JSON形式) を用いて演習を行います
 - フォーマットは、実際のログと合わせているため、応用は可能です
 - ただし、しばしばログのフォーマットは変更されることがあります



ツイート ログ

■ ツイートログ

- 以下のようなログが取得できる。(JSONフォーマットで1ツイートの例)

```
{"contributors": null, "coordinates": null, "created_at": "Wed Nov 30 18:08:57
+0000 2011", "entities": {"hashtags": [{"indices": [17, 27], "text": "jobs"}]},
"urls": [], "user_mentions": []}, "favorited": false, "geo": null, "id": 10000000,
"id_str": "10000000", "in_reply_to_screen_name": null, "in_reply_to_status_id":
null, "in_reply_to_status_id_str": null, "in_reply_to_user_id": null,
"in_reply_to_user_id_str": null, "place": null, "retweet_count": 732, "retweeted":
null, "source": null, "text": "hoge hoge #jobs", "truncated": false, "user":
{"contributors_enabled": false, "created_at": "Sat Apr 02 10:35:09 +0000 2011",
"default_profile": false, "default_profile_image": false, "description": "profile",
"favourites_count": 0, "follow_request_sent": null, "followers_count": 671,
"following": null, "friends_count": 0, "geo_enabled": false, "id": 523255,
"id_str": "523255", "is_translator": false, "lang": "ja", "listed_count": 0,
"location": "東京", "name": "hoge hoge", "notifications": null,
"profile_background_color": "000000", "profile_background_image_url":
"http://hoge hoge.com/images/bg.gif", "profile_background_image_url_https":
"https://hoge hoge.com/images/bg.gif", "profile_background_tile": false,
"profile_image_url": "http://hoge hoge.com/images/pfimage.gif",
"profile_image_url_https": "http://hoge hoge.com/images/pfimage.gif",
"profile_link_color": "000000", "profile_sidebar_border_color": "000000",
"profile_sidebar_fill_color": "000000", "profile_text_color": "000000",
"profile_use_background_image": false, "protected": false, "screen_name":
"hoge hoge", "show_all_inline_media": false, "statuses_count": 0, "time_zone":
"Tokyo", "url": null, "utc_offset": 32400, "verified": false}}
```



ツイート ログ（抜粋）

■ ツイートログ(整形済み、抜粋)

```
{
  "created_at": "Wed Nov 30 18:08:57 +0000 2011"
  "entities": {
    "hashtags": [ { "indices": [ 28, 37 ], "text": "testHash" } ],
    "urls": [],
  },
  "id": 100000000,
  "id_str": "100000000",
  "in_reply_to_screen_name": "ReplyUserSan",
  "in_reply_to_status_id": null,
  "in_reply_to_status_id_str": null,
  "in_reply_to_user_id": "123456789",
  "in_reply_to_user_id_str": "123456789",
  "retweet_count": 0,
  "retweeted": false,
  "text": "@ReplyUserSan Test Message #testHash",
  "user": {
    "created_at": "Mon Jul 11 02:45:55 +0000 2011",
    "favourites_count": 2,
    "followers_count": 218,
    "id": 523255,
    "id_str": "523255",
    "location": "東京",
    "name": "tweetUser",
    "screen_name": "tweeter",
  }
}
```



ツイートログ（抜粋）

■ ツイートログ（整形済み、抜粋）

```
{  
  "created_at": "Wed Nov 30 18:08:57 +0000 2011"  
  "entities": {  
    "hashtags": [ { "indices": [ 28, 37 ], "text": "testHash" } ],  
    "urls": [],  
  },  
  "id": 10000000,  
  "id_str": "10000000",  
  "in_reply_to_screen_name": "ReplyUserSan",  
  "in_reply_to_status_id": null,  
  "in_reply_to_status_id_str": null,  
  "in_reply_to_user_id": "123456789",  
  "in_reply_to_user_id_str": "123456789",  
  "retweet_count": 0,  
  "retweeted": false,  
  "text": "@ReplyUserSan Test Message #testHash",  
  "user": {  
    "created_at": "Mon Jul 11 02:45:55 +0000 2011",  
    "favourites_count": 2,  
    "followers_count": 218,  
    "id": 523255,  
    "id_str": "523255",  
    "location": "東京",  
    "name": "tweetUser",  
    "screen_name": "tweeter",  
  }  
}
```

ツイート時間

HashTag

ツイート ID

reply情報

ツイート本文

ツイートしたuser情報

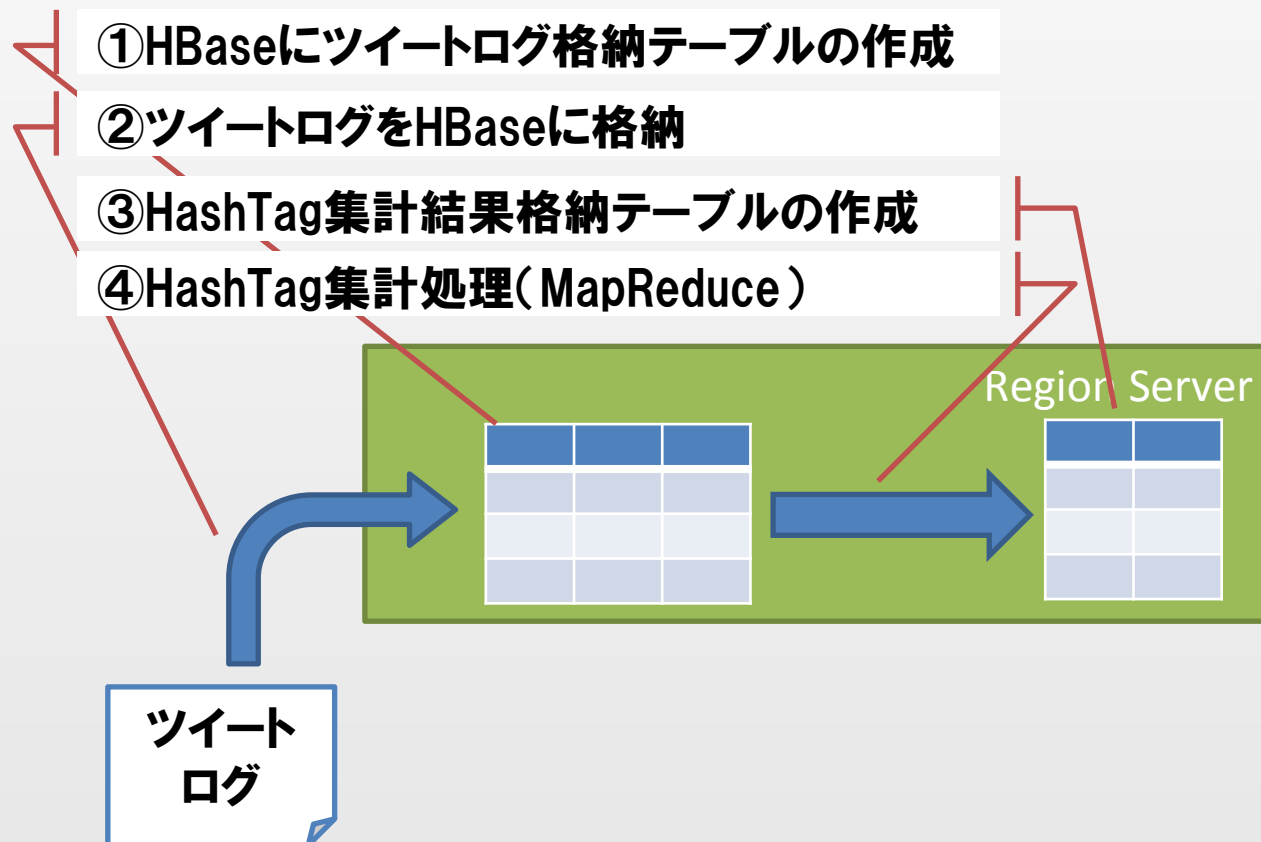


ツイート ログを使ったアプリ演習

■ 演習内容

- ツイートログを解析し、HashTagの利用数をカウントする。
(特につぶやかれているHashTagから傾向を分析する)

■ 演習項目



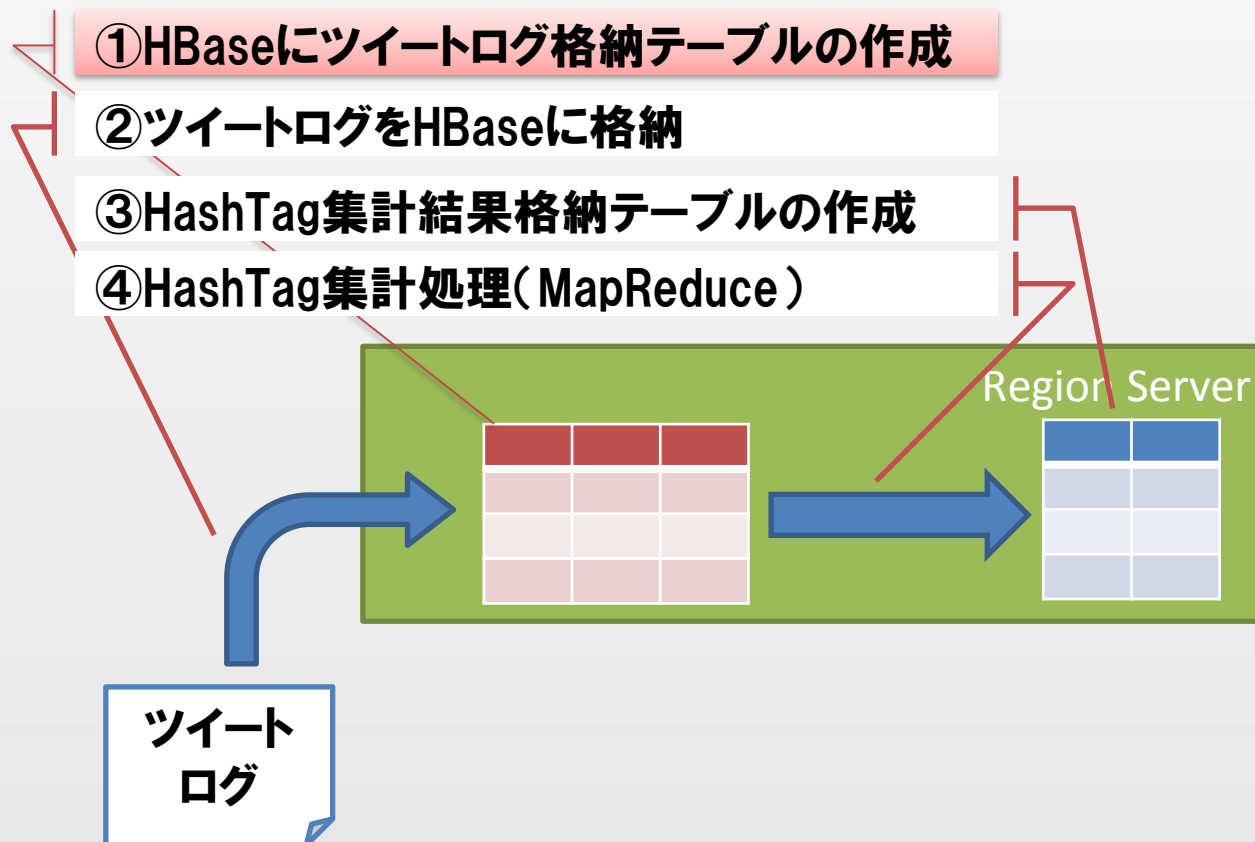


ツイート ログを使ったアプリ演習

■ 演習内容

- ツイートログを解析し、HashTagの利用数をカウントする。
(特につぶやかれているHashTagから傾向を分析する)

■ 演習項目





演習①HBaseにツイートログ格納テーブルの作成

■ 演習①HBaseにツイートログ格納テーブル設計

- ①-1: 講義で話したテーブル設計のポイントを踏まえ、以下の4項目を格納するテーブルを設計する。
 - 格納対象: ツイート本文、HashTag
 - 設計ポイント
 - 行キーの選び方
 - 列ファミリの選び方
- ①-2: HBase Shellを利用し、①-1で設計したテーブルを作成する。
 - HBase Shellコマンド
 - create (第13回 HBaseの基本操作を参照)

注意: テーブル名は「TwitterTable」とする



演習①HBaseにツイートログ格納テーブルの作成【回答例】

■ 回答例

TwitterTable

行キー (UserID+ツイートID)	tweet	
	text	hashtag

```
hbase(main):001:0> create 'TwitterTable', 'tweet'
```

■ 行キー設計

- 単調増加するツイートIDの先頭に、userIDをつけることで、特定のリージョンに処理が集中しないように設計

■ 列ファミリ設計

- 列ファミリ数を最小限に抑えるよう、1つ(tweet)と設計

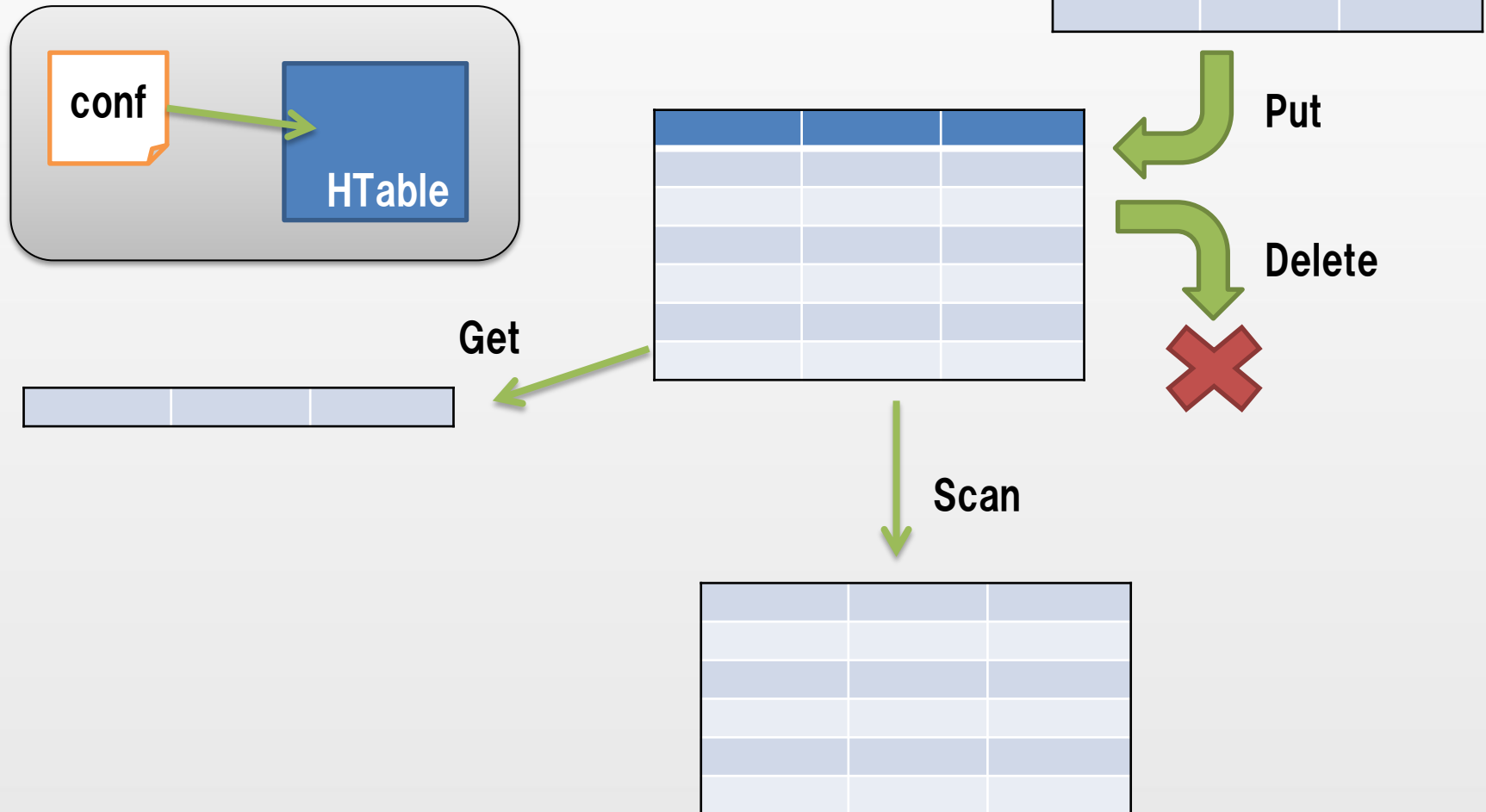
- ※列ファミリ名、修飾子名は、演習における可読性を考慮し、文字数の節約は実施していない



3. HBase の Java API

HBase の Java API 概要

- HTableインスタンスを作成し、テーブル操作を行う。





Configurationクラス

■ HBaseConfigurationクラス

- Hadoop MapReduceのConfigurationクラスと同様、hbase-site.xmlなどで設定されたプロパティ値を保持するConfigurationクラス。
- HBaseクラスタへ接続するには、HBaseConfigurationクラスのロードが必要。

```
// Configurationインスタンス作成  
Configuration conf = HBaseConfiguration.create();
```



HTableクラス

■ HTableクラス

- テーブルデータにアクセスするには、HTableインスタンスを作成し、HTableクラスのメソッドを呼び出す。
- HTableはスレッドセーフでは無いので、マルチスレッドで利用する場合は、スレッドごとに個別インスタンスを利用する必要がある。

```
// Configurationインスタンス作成
Configuration conf = HBaseConfiguration.create();

// HTableインスタンス作成
try {
    HTable table = new HTable(conf, "<tableName>");
} catch (IOException e) {
    e.printStackTrace();
}
```



Putクラス

■ Putクラス

- テーブルに行の追加するのに利用する。
- HBaseでは、各データはByte配列で保持する。
 - HBaseは、文字列をByte配列に変換するメソッド(`Bytes.toBytes()`)を提供している。
- 修飾子がない場合は、"`HConstants.EMPTY_BYTE_ARRAY`"を指定する。
 - `HConstants.EMPTY_BYTE_ARRAY = new byte [0]`

```
// putインスタンス作成
Put p = new Put(Bytes.toBytes("<rowKey>"));

// 追加する行データを格納
p.add(Bytes.toBytes("<family>"),
      Bytes.toBytes("<qualifier>"),
      Bytes.toBytes("<value>"));

// テーブルに追加
table.put(p);
```



Putクラス

■ Putクラス

- タイムスタンプを指定して、行を追加することもできる。

```
// put インスタンス作成時に指定する
```

```
Put p = new Put(Bytes.toByteArray("<rowKey>"), <timestamp>);
```

```
// 行データ格納時に指定する
```

```
p.add(Bytes.toByteArray("<family>"),  
      Bytes.toByteArray("<qualifier>"),  
      <timestamp>,  
      Bytes.toByteArray("<value>"));
```



Putクラス

■ Putクラス

- 同一行キーに対しては、複数回のputを1つにまとめる。
 - 異なる列ファミリーを設定する場合など
 - トランザクションは、putインスタンス1つに対応するので、意識的にまとめたほうが良い
- putをまとめることで、HLogへの書き込みもまとめられるため、別々にputするよりも性能向上もみこめる。

```
// put インスタンス作成  
Put p = new Put(Bytes.toBytes("<rowKey>"));
```

```
// 追加する複数の行データを格納  
p.add(Bytes.toBytes("<family1>"),  
      Bytes.toBytes("<qualifier>"),  
      Bytes.toBytes("<value>"));  
p.add(Bytes.toBytes("<family2>"),  
      Bytes.toBytes("<qualifier>"),  
      Bytes.toBytes("<value>"));
```

```
// テーブルに追加  
table.put(p);
```



【参考】性能向上のテクニック

- データ保全性と引き換えに性能をとることもできる
- HLog (WAL) への書き込み制御
 - Put単位でWALへの書き込みを制御できる
 - WALへの書き込みをなくすことで、性能向上が見込める
- 自動フラッシュ制御
 - 通常、putは一度に1つずつリージョンサーバに送られるが、自動フラッシュを無効にすることで、put処理は書き込みバッファに入れられ、まとめて送信される。
(書き込みバッファがいっぱい、もしくは、HTableインスタンスがcloseの時)

```
// WALへの書き込みをoffにする  
p.setWriteToWAL(false);
```

```
// 自動フラッシュの無効  
table.setAutoFlush(false);
```



Deleteクラス

■ Deleteクラス

- テーブルから行を削除するのに利用する。

```
// deleteインスタンス作成  
Delete d = new Delete(Bytes.toBytes("<rowKey>"));  
  
// テーブルから削除  
table.delete(d);
```

- セルや列ファミリを指定して削除することも可能。

```
// deleteインスタンス作成  
Delete d = new Delete(Bytes.toBytes("<rowKey>"));  
  
// 列ファミリを指定  
d.deleteFamily(Bytes.toBytes("<family>"));  
  
// セルを指定  
d.deleteColumns(Bytes.toBytes("<family>"),  
                Bytes.toBytes("<qualifier>"));
```


■ Getクラス

- 単一の行キーの行を取得するために利用する。
- 修飾子がない場合は、"HConstants.EMPTY_BYTE_ARRAY"を指定する。
- 取得結果は、Resultクラスで返される
 - result.size () やresult.isEmpty () により、取得行が検証できる
 - 一致する行キーが見つからない場合は、サイズが0になる

```
Get q = new Get (Bytes.toBytes ("<rowKey>")) ;
```

Result `r = table.get(g);`

[illegible]



Resultクラス

■ Resultクラス

- **特定列ファミリのデータを取得するには、getFamilyMap () を利用する。**

- Map<qualifier,value> を返す

```
// getインスタンス作成
Get g = new Get(Bytes.toBytes("<rowKey>"));

// 指定行キーの行を取得
Result r = table.get(g);

// 取得した行からMap形式で列ファミリをすべて取得
NavigableMap<byte[],byte[]> map =
    r.getFamilyMap(Bytes.toBytes("<family>"));

// 列ファミリの全データを順次取り出す
for(Entry<byte[], byte[]> entry : map.entrySet() ) {
    String qualifier = Bytes.toString(entry.getKey());
    String value = Bytes.toString(entry.getValue());
    ...
}
```



Resultクラス

■ Resultクラス

■ 全ファミリのデータ取得するには、getMap () を利用する。

■ Map<family, Map<qualifier, Map<timestamp,value>>> を返す

```
// 取得した行からMap形式で全ファミリのデータを取得
NavigableMap<byte[], NavigableMap<byte[], NavigableMap<Long, byte[]>>> map
                                                                    = r.getMap();

// 全ファミリのデータを順次取り出す
for(Entry<byte[], NavigableMap<byte[], NavigableMap<Long, byte[]>>> entry :
                                                                    map.entrySet() ) {

    String family = Bytes.toString(entry.getKey());
    NavigableMap<byte[], NavigableMap<Long, byte[]>> famMap = entry.getValue();
    for(Entry<byte[], NavigableMap<Long, byte[]>> famEntry : famMap.entrySet() ){

        String qualifier = Bytes.toString(famEntry.getKey());
        NavigableMap<Long, byte[]> quaMap = famEntry.getValue();
        for(Entry<Long, byte[]> quaEntry : quaMap.entrySet() ){
            long timestamp = quaEntry.getKey();
            String value = Bytes.toString(quaEntry.getValue());

            ...
        }
    }
}
```



Resultクラス

■ Resultクラス

- 全ファミリのデータから**最新のみ**取得するには、`getNoVersionMap()` を利用する。

- `Map<family, Map<qualifier, value>>` を返す

```
// 取得した行から列ファミリをすべて取得
```

```
NavigableMap<byte[], NavigableMap<byte[], byte[]>> map =  
                                                    r.getNoVersionMap();
```

```
// 全ファミリのデータを順次取り出す
```

```
for(Entry<byte[], NavigableMap<byte[], byte[]>> entry :  
                                           map.entrySet() ) {  
    String family = Bytes.toString(entry.getKey());  
    NavigableMap<byte[], byte[]> famMap = entry.getValue();  
    for(Entry<byte[], byte[]> famEntry : famMap.entrySet() ) {  
        String qualifier = Bytes.toString(famEntry.getKey());  
        String value = Bytes.toString(famEntry.getValue());  
        ...  
    }  
}
```



KeyValueクラス

■ KeyValueクラス

- Resultインスタンスから、Mapではなく、**KeyValue**でデータを取り出すことも出来る。

```
// getインスタンス作成
Get g = new Get(Bytes.toBytes("<rowKey>"));

// 指定行キーの行を取得
Result r = table.get(g);

// ResultのデータをKeyValue配列で返す
KeyValue[] kv = r.raw();

// 行キーでソートしたデータをKeyValue配列で返す
KeyValue[] kv = r.sorted();

// 行キーでソートしたデータをKeyValueのリストで返す
List<KeyValue> list = r.list();
```



KeyValueクラス

■ KeyValueクラス

■ KeValueから各要素を取得する

```
byte[] row = kv.getRow();  
byte[] family = kv.getFamily();  
byte[] qualifier = kv.getQualifier();  
byte[] value = kv.getValue();  
long timestamp = kv.getTimestamp();
```



Scanクラス

■ Scanクラス

■ テーブル内の全ての行をスキャンする。

■ Resultオブジェクトを含むイテレータであるResultScannerを返す

```
// scanインスタンス作成
Scan s = new Scan();

// 全行を取得
ResultScanner rs = table.getScanner(s);

// 各行を順次取り出す
for (Result r : rs ) {
    ...
}
```

■ Scanクラスを使わずにResultScannerを取得することも可能

```
// 特定列ファミリーの全行を取得
ResultScanner rs = table.getScanner("<family>");
// 特定列ファミリ、修飾子の全行を取得
ResultScanner rs = table.getScanner("<family>", "<qualifier>");
```



Scanクラス

■ Scanクラス

- scanクラスでは、さまざまな検索条件を指定することが可能

```
// scanインスタンス作成
Scan s = new Scan();

// 列ファミリを指定
s.addFamily(Bytes.toBytes("<family>"));

// 列ファミリ、修飾子を指定
s.addColumn(Bytes.toBytes("<family>"), Bytes.toBytes("<qualifier>"));

// 取得バージョン数を指定(デフォルトは1)
s.setMaxVersions(<num>);

// 取得する時間範囲を指定(minTimestamp ≤ 時刻 < maxTimestamp)
s.setTimeRange(<minTimestamp>, <maxTimestamp>)
```




Scanクラス

■ Scanクラス

- scanクラスでは、さまざまな検索条件を指定することが可能

```
// scanインスタンス作成
Scan s = new Scan();

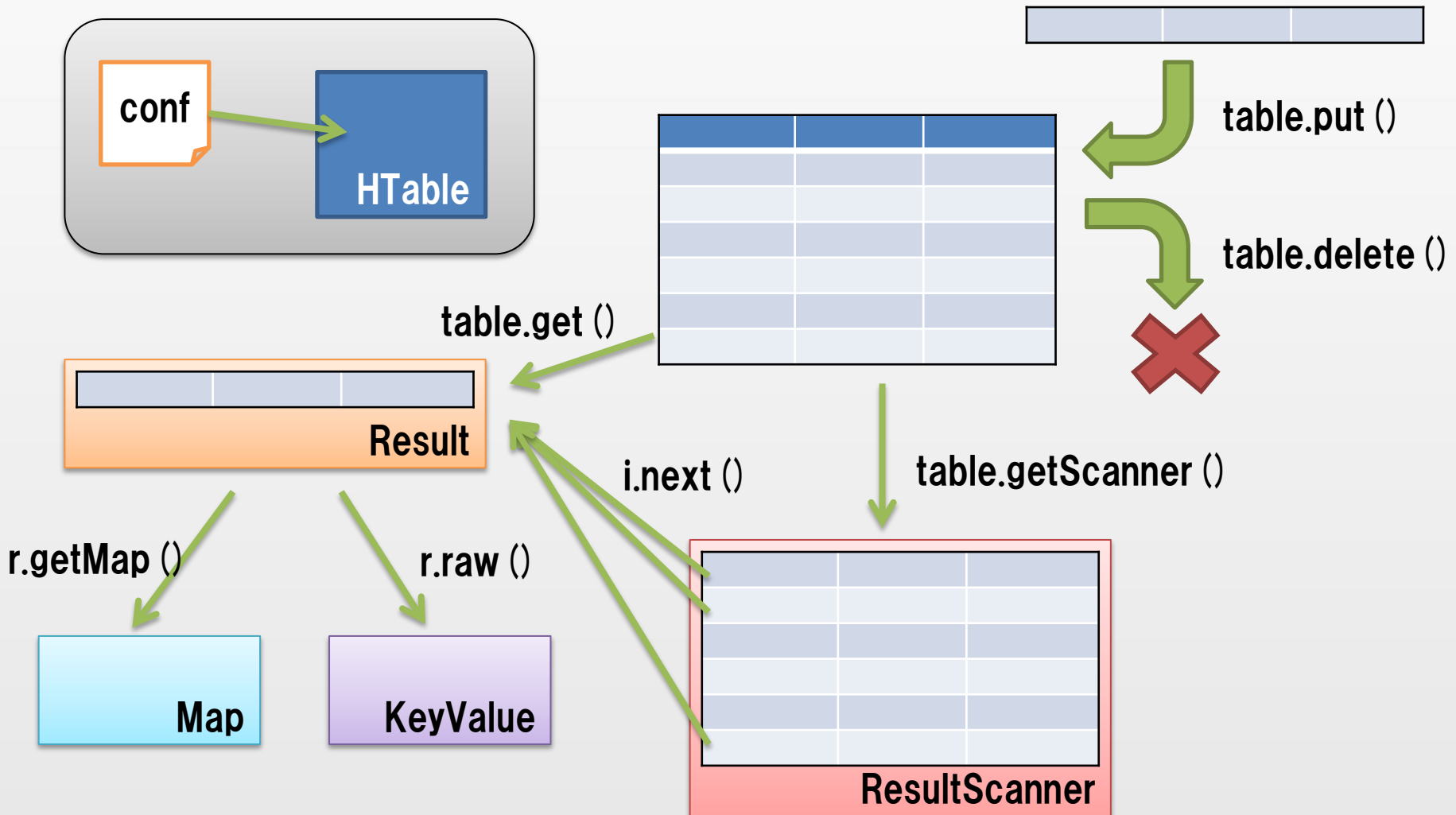
// 以降の行を取得(startRowKey ≤ 行キー)
s.setStartRow(Bytes.toBytes("<startRowKey>"));

// 以前の行を取得(行キー < stopRowKey)
s.setStopRow(Bytes.toBytes("<stopRowKey>"));

// StartRowとStopRowを共に指定することで範囲指定も可能
// (startRowKey ≤ 行キー < stopRowKey)

// Scanインスタンス作成時に指定することもできる
Scan s = new Scan(Bytes.toBytes("<startRowKey>"),
Bytes.toBytes("<stopRowKey>"));
```

HBaseアクセスまとめ



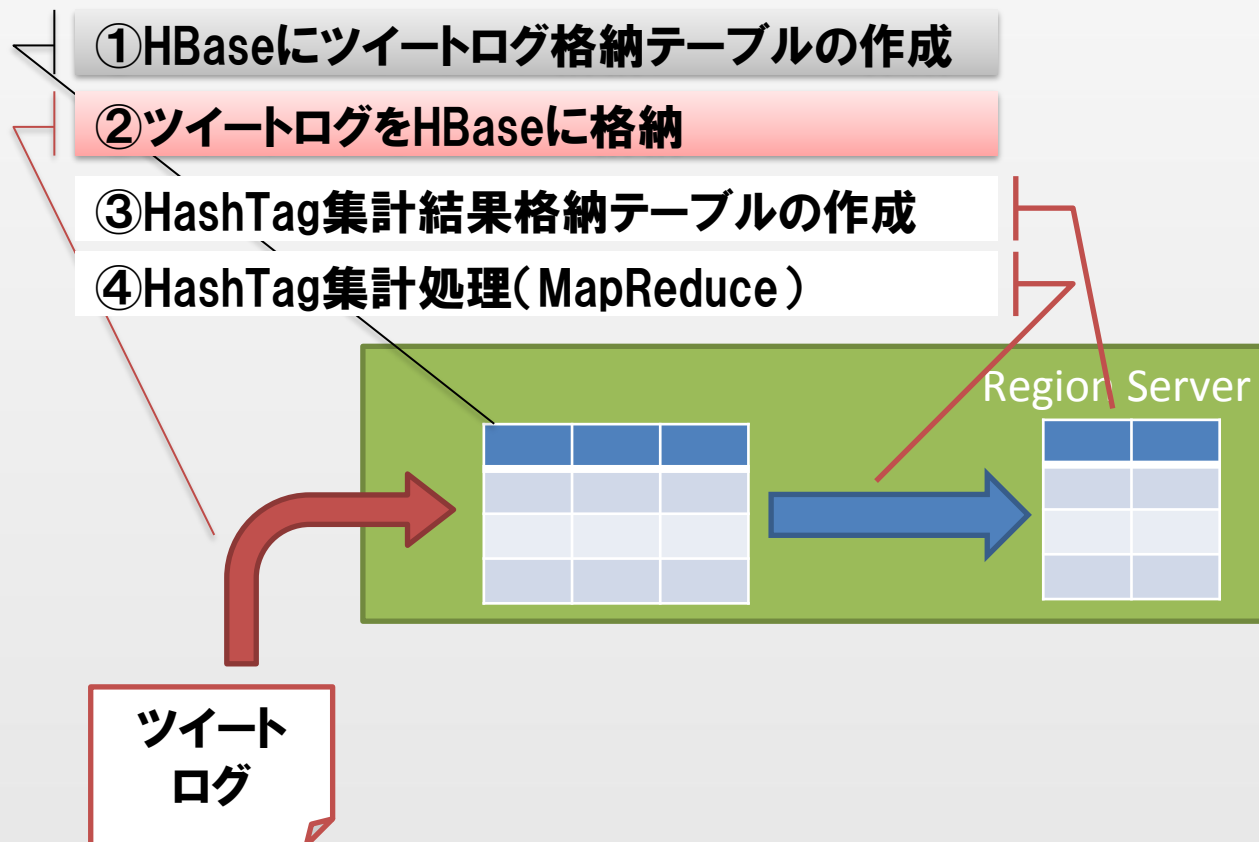


ツイート ログを使ったアプリ演習

■ 演習内容

- ツイートログを解析し、HashTagの利用数をカウントする。
(特につぶやかれているHashTagから傾向を分析する)

■ 演習項目





演習②ツイートログをHBaseに格納

■ 演習②ツイートログをHBase格納する。

- ②-1 クライアント端末に格納されたツイート ログを演習①で作成したテーブルにインポートするjavaアプリを作成する。
- ②-2 1で作成したアプリを実行し、ツイートログをHBaseにインポートする。
- ②-3 インポート結果を確認する。



演習②ツイートログをHBaseに格納（実習環境）

■ 演習環境

■ ツイートログ格納場所

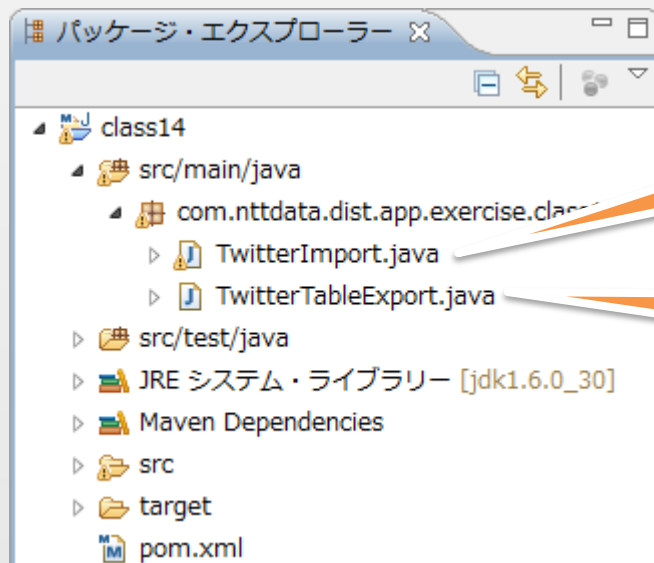
```
(hdclient01)$ /root/hadoop_exercise/14/data/tweets.log  
(hdclient01)$ /root/hadoop_exercise/14/data/tweets_mini.log
```

大きいサイズ

■ ソース格納場所

■ EclipseのClass14フォルダに必要な資材を格納

デバッグ用
ミニサイズ



演習で作成するソース
TwitterImport.java
（未完成）

HBaseの中身確認用
TwitterTableExport.java
（完成済）

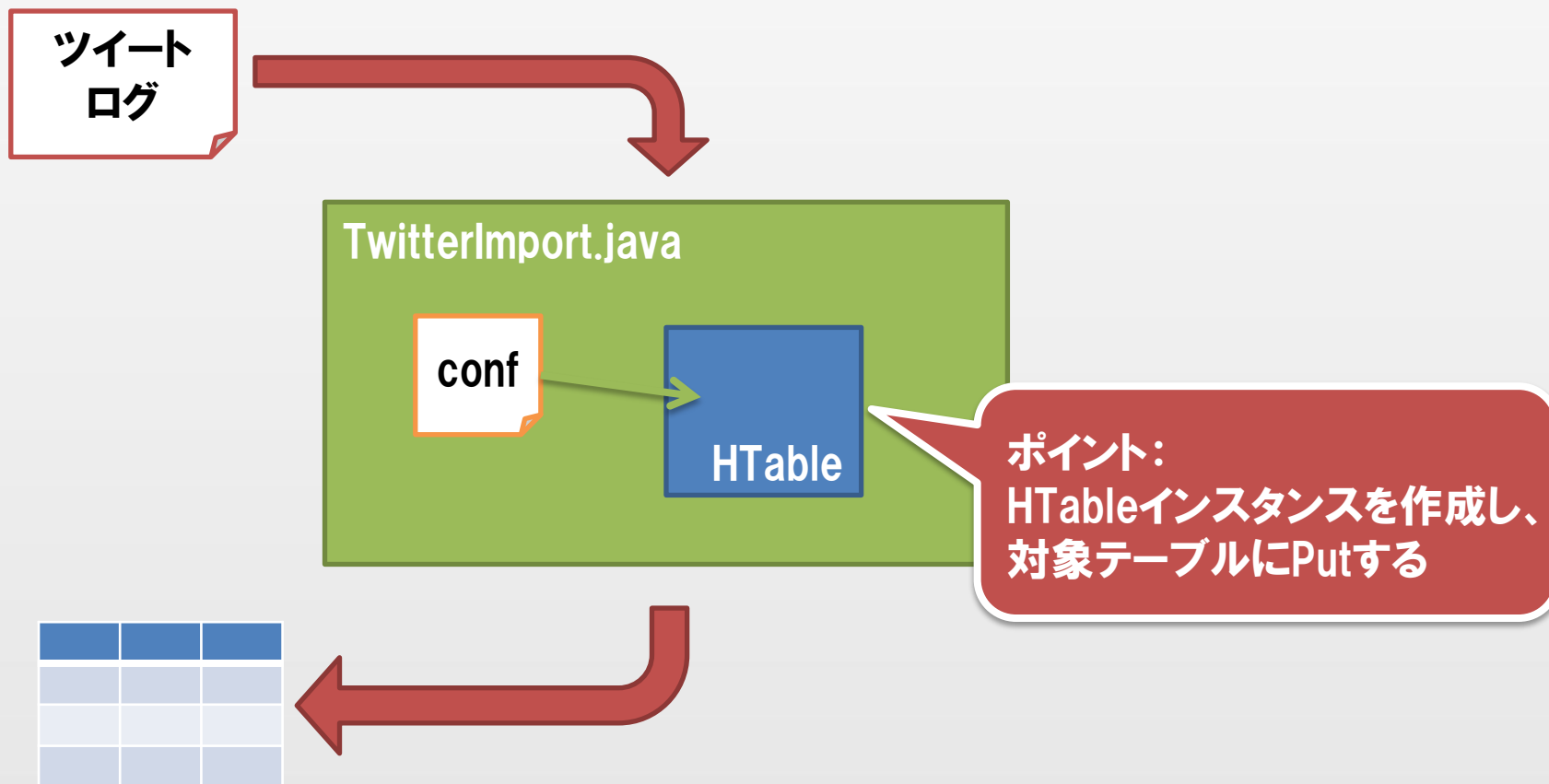


演習②ツイートログをHBaseに格納（アプリ概要）

■ ②-1 ツイートログをHBase格納するjavaアプリの作成

■ 作成ファイル

■ TwitterImport.java





演習②ツイートログをHBaseに格納（インポート仕様）

■ ツイートログのインポート仕様

```
{
  "created_at": "Tue Nov 08 02:31:44 +0000 2011",
  "entities": {
    "hashtags": [ { "indices": [ 28, 37 ], "text": "testHash1" },
                  { "indices": [ 38, 47 ], "text": "testHash2" } ]
  },
  "id": 133733411750809600,
  "text": "@ReplyUserSan Test Message #testHash1 #testHash2",
  "user": {
    "id": 333160681
  }
}
```

ツイートTable

UserID+ツイート ID	ツイート			timestamp
	text	hashtag1	hashtag2	
333160681- 133733411750809600	@ReplyUserSan Test Message #testHash1 #testHash2	testHash1	testHash2	1326937337043

UserIDとツイートIDを“-”でつ
なぎ行キーとする

hashtagが複数ある場合に備え、
修飾子に数字をつける

ツイートされた日時(ミリ秒)
を、行のtimestampとする



演習補足:JSON データフォーマット

■ JSONのデータ型は以下のとおり

JSONデータ型	例	対応するJavaデータ型
数値	123	Number
文字列	"abc"	String
真偽	true	boolean
配列	["abc","def","ghi"]	List
オブジェクト	{ name: "abc", age: 33 }	Map



演習補足：JSONのパーズ方法

■ JSONのparseには、json-simpleというライブラリを利用

■ <http://code.google.com/p/json-simple/>

```
{
  "name": "sasaki",
  "user" : { "age" : 72 , "id" : 1234 }
  "hoge": [ 28, 12 ,900 ]
}
```

←JSONフォーマットのファイル

```
JSONParser jsonParser = new JSONParser();

line = JSON形式のファイルの中身(String);
JSONObject jsonObj = (JSONObject) jsonParser.parse(line);

String text = (String) jsonObj.get("name"); // "sasaki"が取得できる

Map user = (Map) jsonObj.get("user");
Number age = (Number) user.get("age"); // 72が取得できる

List hoge = (List) jsonObj.get("hoge");
for (Object list : hoge) {
    Number num = (Number) list; // 28,12,900の取得
}
```



実行手順

■ 実行方法

■ 事前準備

- 演習①で作成した"TwitterTable"を利用する。
- 必要に応じて、作り直す(drop/create)か、truncateしておく。

■ 実行時にjson-simpleライブラリをクラスパスに追加する

■ 引数でツイートログを指定する

```
$ HADOOP_CLASSPATH="/root/hadoop_exercise/14/lib/json-simple-1.1.jar" ¥  
hadoop jar ~/workspace/Class14/target/hbaseImport-0.1.jar ¥  
com.example.dpap.class14.TwitterImport ¥  
~/hadoop_exercise/14/data/tweets.log
```

■ 【補足】TwitterTableに格納されているデータを標準出力に表示する

■ com.example.dpap.class14.TwitterTableExport

```
$ hadoop jar ~/workspace/Class14/target/hbaseImport-0.1.jar ¥  
com.example.dpap.class14.TwitterTableExport 2 2> /dev/null
```

表示させる閾値
(この数字の行数のみ表示)

不要なメッセージは、
/dev/nullに捨てる



回答解説

■ 別紙で説明



まとめ

本講義で学んだ内容

■ HBaseスキーマ設計

- HBaseの特徴から見るスキーマ設計のポイント
- 行キー設計のポイント
- 列ファミリ設計のポイント

■ ツイートログを利用したスキーマ設計演習

- javaでHBaseにアクセスする方法
- JSON形式のツイートログ
- ツイートログ格納テーブル設計
- HBaseへのインポート演習