HBase 中加盐（Salting）之后的表如何读取：协处理器篇

在《HBase Rowkey 设计指南》文章中，我们介绍了避免数据热点的三种比较常见方法：

- 加盐 - Salting
- 哈希 - Hashing
- 反转 - Reversing

其中在加盐（Salting）的方法里面是这么描述的：给 Rowkey
分配一个随机前缀以使得它和之前排序不同。但是在 Rowkey 前面加了随机前缀，那么我们怎么将这些数据读出来呢？我将分三篇文章来介绍如何读取加盐之后的表，其中每篇文章提供一种方法，主要包括：

- 使用协处理器读取加盐的表
- 使用 Spark 读取加盐的表
- 使用 MapReduce 读取加盐的表

关于协处理器的入门及实战，请参见这里
。本文使用的各组件版本：hadoop-2.7.7，hbase-2.0.4，jdk1.8.0_201。

测试数据生成

在介绍如何查询数据之前，我们先创建一张名为 iteblog 的 HBase
表，用于测试。为了数据均匀和介绍的方便，这里使用了预分区，并设置了27个分区，如下：

```
hbase(main):002:0> create 'iteblog', 'f', SPLITS => ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
0 row(s) in 2.4880 seconds
```

然后我们使用下面方法生成了1000000条测试数据。RowKey 的形式为 UID +
当前数据生成时间戳；由于 UID 的长度为4，所以1000000条数据会存在大量的 UID
一样的数据，所以我们使用加盐方法将这些数据均匀分散到上述27个 Region
里面（注意，其实第一个 Region 其实没数据）。具体代码如下：

```
package com.iteblog.data;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HConstants;
import org.apache.hadoop.hbase.TableName;
```

```
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.UUID;

public class HBaseDataGenerator {
    private static byte[] FAMILY = "f".getBytes();
    private static byte[] QUALIFIER_UUID = "uuid".getBytes();
    private static byte[] QUALIFIER_AGE = "age".getBytes();

    private static char generateLetter() {
        return (char) (Math.random() * 26 + 'A');
    }

    private static long generateUid(int n) {
        return (long) (Math.random() * 9 * Math.pow(10, n - 1)) + (long) Math.pow(10, n - 1);
    }

    public static void main(String[] args) throws IOException {
        BufferedMutatorParams bmp = new BufferedMutatorParams(TableName.valueOf("iteblog
"));
        bmp.writeBufferSize(1024 * 1024 * 24);

        Configuration conf = HBaseConfiguration.create();
        conf.set(HConstants.ZOOKEEPER_QUORUM, "https://www.iteblog.com:2181");
        Connection connection = ConnectionFactory.createConnection(conf);

        BufferedMutator bufferedMutator = connection.getBufferedMutator(bmp);

        int BATCH_SIZE = 1000;
        int COUNTS = 1000000;
        int count = 0;
        List<Put> putList = new ArrayList<>();

        for (int i = 0; i < COUNTS; i++) {
            String rowKey = generateLetter() + "-"
                    + generateUid(4) + "-"
                    + System.currentTimeMillis();

            Put put = new Put(Bytes.toBytes(rowKey));
            byte[] uuidBytes = UUID.randomUUID().toString().substring(0, 23).getBytes();
            put.addColumn(FAMILY, QUALIFIER_UUID, uuidBytes);
```

```
            put.addColumn(FAMILY, QUALIFIER_AGE, Bytes.toBytes("" + new Random().nextInt(100)
));
            putList.add(put);
            count++;

            if (count % BATCH_SIZE == 0) {
                bufferedMutator.mutate(putList);
                bufferedMutator.flush();
                putList.clear();
                System.out.println(count);
            }
        }

        if (putList.size() > 0) {
            bufferedMutator.mutate(putList);
            bufferedMutator.flush();
            putList.clear();
        }

    }
}
```

运行完上面代码之后，会生成1000000条数据（注意，这里其实不严谨，因为 Rowkey
设计问题，可能会导致重复的 Rowkey
生成，所以实际情况下可能没有1000000条数据。）。我们limit 10条数据看下长成什么样：

```
hbase(main):001:0> scan 'iteblog', {'LIMIT'=>10}
ROW                     COLUMN+CELL
 A-1000-1550572395399     column=f:age, timestamp=1549091990253, value=54
 A-1000-1550572395399     column=f:uuid, timestamp=1549091990253, value=e9b10a9f-1218
-43fd-bd01
 A-1000-1550572413799     column=f:age, timestamp=1549092008575, value=4
 A-1000-1550572413799     column=f:uuid, timestamp=1549092008575, value=181aa91e-5f1d
-454c-959c
 A-1000-1550572414761     column=f:age, timestamp=1549092009531, value=33
 A-1000-1550572414761     column=f:uuid, timestamp=1549092009531, value=19aad8d3-621a
-473c-8f9f
 A-1001-1550572394570     column=f:age, timestamp=1549091989341, value=64
 A-1001-1550572394570     column=f:uuid, timestamp=1549091989341, value=c6712a0d-3793
-46d5-865b
 A-1001-1550572405337     column=f:age, timestamp=1549092000108, value=96
 A-1001-1550572405337     column=f:uuid, timestamp=1549092000108, value=4bf05d10-bb4d
-43e3-9957
```

```
A-1001-1550572419688    column=f:age, timestamp=1549092014458, value=8
A-1001-1550572419688    column=f:uuid, timestamp=1549092014458, value=f04ba835-d8ac-
49a3-8f96
A-1002-1550572424041    column=f:age, timestamp=1549092018816, value=84
A-1002-1550572424041    column=f:uuid, timestamp=1549092018816, value=99d6c989-afb5-
4101-9d95
A-1003-1550572431830    column=f:age, timestamp=1549092026605, value=21
A-1003-1550572431830    column=f:uuid, timestamp=1549092026605, value=8c1ff1b6-b97c-
4059-9b68
A-1004-1550572395399    column=f:age, timestamp=1549091990253, value=2
A-1004-1550572395399    column=f:uuid, timestamp=1549091990253, value=e240aa0f-
c044-452f-89c0
A-1004-1550572403783    column=f:age, timestamp=1549091998555, value=6
A-1004-1550572403783    column=f:uuid, timestamp=1549091998555, value=e8df15c9-02fa-
458e-bd0c
10 row(s)
Took 0.1104 seconds
```

使用协处理器查询加盐之后的表

现在有数据了，我们需要查询所有 UID = 1000 的用户所有历史数据，那么如何查呢？我们知道 UID = 1000 的用户数据是均匀放到上述的27个 Region 里面的，因为经过加盐了，所以这些数据前缀都是类似于 A-，B-，C- 等开头的。其次我们需要知道，每个 Region 其实是有 Start Key 和 End Key 的，这些 Start Key 和 End Key 其实就是我们创建 iteblog 表指定的。如果你看了 《HBase 协处理器入门及实战》 这篇文章，你就知道协处理器的代码其实是在每个 Region 里面执行的；而这些代码在 Region 里面执行的时候是可以拿到当前 Region 的信息，包括了 Start Key 和 End Key，所以其实我们可以将拿到的 Start Key 信息和查询的 UID 进行拼接，这样就可以查询我们要的数据。协处理器处理篇就是基于这样的思想来查询加盐之后的数据的。

定义 proto 文件

为什么需要定义这个请参见 《HBase 协处理器入门及实战》 这篇文章。因为我们查询的时候需要传入查询的参数，比如tableName、 StartKey 、 EndKey 以及是否加盐等标记；同时当查询到结果的时候，我们还需要将数据返回，所以我们定义的 proto 文件如下：

```
option java_package = "com.iteblog.data.coprocessor.generated";
option java_outer_classname = "DataQueryProtos";
option java_generic_services = true;
option java_generate_equals_and_hash = true;
option optimize_for = SPEED;
```

```
message DataQueryRequest {
  optional string tableName = 1;
  optional string startRow = 2;
  optional string endRow = 3;
  optional bool  incluedEnd = 4;
  optional bool  isSalting = 5;
}

message DataQueryResponse {
  message Cell{
    required bytes value = 1;
    required bytes family = 2;
    required bytes qualifier = 3;
    required bytes row = 4;
    required int64 timestamp = 5;
  }

  message Row{
    optional bytes rowKey = 1;
    repeated Cell cellList = 2;
  }

  repeated Row rowList = 1;
}

service QueryDataService{
  rpc queryByStartRowAndEndRow(DataQueryRequest)
    returns (DataQueryResponse);
}
```

然后我们使用 protobuf-maven-plugin 插件将上面的 proto 生成 java 类，具体如何操作参见《在 IDEA 中使用 Maven 编译 proto 文件》。我们将生成的 DataQueryProtos.java 类拷贝到 com.iteblog.data.coprocessor.generated 包里面。

编写协处理器代码

有了请求和返回的类，现在我们需要编写协处理器的处理代码了，结合上面的分析，协处理器的代码实现如下：

```
package com.iteblog.data.coprocessor;

import com.google.protobuf.ByteString;
```

```
import com.google.protobuf.RpcCallback;
import com.google.protobuf.RpcController;
import com.google.protobuf.Service;
import com.iteblog.data.coprocessor.generated.DataQueryProtos.QueryDataService;
import com.iteblog.data.coprocessor.generated.DataQueryProtos.DataQueryRequest;
import com.iteblog.data.coprocessor.generated.DataQueryProtos.DataQueryResponse;
import org.apache.hadoop.hbase.Cell;
import org.apache.hadoop.hbase.CoprocessorEnvironment;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.coprocessor.CoprocessorException;
import org.apache.hadoop.hbase.coprocessor.RegionCoprocessor;
import org.apache.hadoop.hbase.coprocessor.RegionCoprocessorEnvironment;
import org.apache.hadoop.hbase.regionserver.InternalScanner;
import org.apache.hadoop.hbase.shaded.protobuf.ResponseConverter;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SlatTableDataSearch extends QueryDataService implements RegionCoprocessor {
  private RegionCoprocessorEnvironment env;

  public Iterable<Service> getServices() {
    return Collections.singleton(this);
  }

  @Override
  public void queryByStartRowAndEndRow(RpcController controller,
                                       DataQueryRequest request,
                                       RpcCallback<DataQueryResponse> done) {
    DataQueryResponse response = null;

    String startRow = request.getStartRow();
    String endRow = request.getEndRow();
    String regionStartKey = Bytes.toString(this.env.getRegion().getRegionInfo().getStartKey());

    if (request.getIsSalting()) {
      String startSalt = null;
      if (null != regionStartKey && regionStartKey.length() != 0) {
        startSalt = regionStartKey;
      }
      if (null != startSalt && null != startRow) {
```

```
            startRow = startSalt + "-" + startRow;
            endRow = startSalt + "-" + endRow;
        }
    }

    Scan scan = new Scan();
    if (null != startRow) {
        scan.withStartRow(Bytes.toBytes(startRow));
    }

    if (null != endRow) {
        scan.withStopRow(Bytes.toBytes(endRow), request.getIncluedEnd());
    }

    try (InternalScanner scanner = this.env.getRegion().getScanner(scan)) {
        List<Cell> results = new ArrayList<>();

        boolean hasMore;
        DataQueryResponse.Builder responseBuilder = DataQueryResponse.newBuilder();
        do {
            hasMore = scanner.next(results);
            DataQueryResponse.Row.Builder rowBuilder = DataQueryResponse.Row.newBuilder(
);
            if (results.size() > 0) {
                Cell cell = results.get(0);
                rowBuilder.setRowKey(ByteString.copyFrom(cell.getRowArray(), cell.getRowOffset()
, cell.getRowLength()));
                for (Cell kv : results) {
                    buildCell(rowBuilder, kv);
                }
            }

            responseBuilder.addRowList(rowBuilder);
            results.clear();
        } while (hasMore);

        response = responseBuilder.build();

    } catch (IOException e) {
        ResponseConverter.setControllerException(controller, e);
    }
    done.run(response);
  }

  private void buildCell(DataQueryResponse.Row.Builder rowBuilder, Cell kv) {
    DataQueryResponse.Cell.Builder cellBuilder = DataQueryResponse.Cell.newBuilder();
```

```
        cellBuilder.setFamily(ByteString.copyFrom(kv.getFamilyArray(), kv.getFamilyOffset(), kv.get
FamilyLength()));
        cellBuilder.setQualifier(ByteString.copyFrom(kv.getQualifierArray(), kv.getQualifierOffset(),
 kv.getQualifierLength()));
        cellBuilder.setRow(ByteString.copyFrom(kv.getRowArray(), kv.getRowOffset(), kv.getRowLe
ngth()));
        cellBuilder.setValue(ByteString.copyFrom(kv.getValueArray(), kv.getValueOffset(), kv.getVal
ueLength()));
        cellBuilder.setTimestamp(kv.getTimestamp());
        rowBuilder.addCellList(cellBuilder);
    }

    /**
     * Stores a reference to the coprocessor environment provided by the
     * {@link org.apache.hadoop.hbase.regionserver.RegionCoprocessorHost} from the region
where this
     * coprocessor is loaded.  Since this is a coprocessor endpoint, it always expects to be loade
d
     * on a table region, so always expects this to be an instance of
     * {@link RegionCoprocessorEnvironment}.
     *
     * @param env the environment provided by the coprocessor host
     * @throws IOException if the provided environment is not an instance of
     *                     {@code RegionCoprocessorEnvironment}
     */
    @Override
    public void start(CoprocessorEnvironment env) throws IOException {
        if (env instanceof RegionCoprocessorEnvironment) {
            this.env = (RegionCoprocessorEnvironment) env;
        } else {
            throw new CoprocessorException("Must be loaded on a table region!");
        }
    }

    @Override
    public void stop(CoprocessorEnvironment env) {
        // nothing to do
    }
}
```

大家可以看到，这里面的代码框架和 《HBase 协处理器入门及实战》 里面介绍的 HBase 提供的 RowCountEndpoint 示例代码很类似。主要逻辑在 queryByStartRowAndEndRow 函数实现里面。我们通过 DataQueryRequest 拿到客户端查询的表，StartKey 和 EndKey

等数据。通过 this.env.getRegion().getRegionInfo().getStartKey() 可以拿到当前 Region 的 StartKey，然后再和客户端传进来的 StartKey 和 EndKey 进行拼接就可以拿到完整的 Rowkey 前缀。剩下的查询就是正常的 HBase Scan 代码了。

现在我们将 SlatTableDataSearch 类进行编译打包，并部署到 HBase 表里面去，具体如何部署参见 《HBase 协处理器入门及实战》

协处理器客户端代码编写

到这里，我们的协处理器服务器端的代码和部署已经完成了，现在我们需要编写协处理器客户端代码。其实也很简单，如下：

```
package com.iteblog.data;

import com.iteblog.data.coprocessor.generated.DataQueryProtos.QueryDataService;
import com.iteblog.data.coprocessor.generated.DataQueryProtos.DataQueryRequest;
import com.iteblog.data.coprocessor.generated.DataQueryProtos.DataQueryResponse;
import com.iteblog.data.coprocessor.generated.DataQueryProtos.DataQueryResponse.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Connection;
import org.apache.hadoop.hbase.client.ConnectionFactory;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.ipc.CoprocessorRpcUtils.BlockingRpcCallback;
import org.apache.hadoop.hbase.ipc.ServerRpcController;

import java.util.LinkedList;
import java.util.List;
import java.util.Map;

public class DataQuery {
  private static Configuration conf = null;

  static {
    conf = HBaseConfiguration.create();
    conf.set("hbase.zookeeper.quorum", "https://www.iteblog.com:2181");
  }

  static List<Row> queryByStartRowAndStopRow(String tableName,
                                             String startRow, String stopRow,
                                             boolean isIncludeEnd, boolean isSalting) {

    final DataQueryRequest.Builder requestBuilder = DataQueryRequest.newBuilder();
    requestBuilder.setTableName(tableName);
    requestBuilder.setStartRow(startRow);
```

```
    requestBuilder.setEndRow(stopRow);
    requestBuilder.setIncluedEnd(isIncludeEnd);
    requestBuilder.setIsSalting(isSalting);

    try {
        Connection connection = ConnectionFactory.createConnection(conf);
        HTable table = (HTable) connection.getTable(TableName.valueOf(tableName));
        Map<byte[], List<Row>> result = table.coprocessorService(QueryDataService.class,
                null, null, counter -> {
                    ServerRpcController controller = new ServerRpcController();
                    BlockingRpcCallback<DataQueryResponse> call = new BlockingRpcCallback<>();
                    counter.queryByStartRowAndEndRow(controller, requestBuilder.build(), call);
                    DataQueryResponse response = call.get();

                    if (controller.failedOnException()) {
                        throw controller.getFailedOn();
                    }

                    return response.getRowListList();
                });

        List<Row> list = new LinkedList<>();
        for (Map.Entry<byte[], List<Row>> entry : result.entrySet()) {
            if (null != entry.getKey()) {
                list.addAll(entry.getValue());
            }
        }
        return list;
    } catch (Throwable e) {
        e.printStackTrace();
    }
    return null;

}

public static void main(String[] args) {
    List<Row> rows = queryByStartRowAndStopRow("iteblog", "1000", "1001", false, true);
    if (null != rows) {
        System.out.println(rows.size());
        for (DataQueryResponse.Row row : rows) {
            List<DataQueryResponse.Cell> cellListList = row.getCellListList();
            for (DataQueryResponse.Cell cell : cellListList) {
                System.out.println(row.getRowKey().toStringUtf8() + " ₩t " +
                        "column=" + cell.getFamily().toStringUtf8() +
                        ":" + cell.getQualifier().toStringUtf8() + ", " +
                        "timestamp=" + cell.getTimestamp() + ", " +
```

```
                    "value=" + cell.getValue().toStringUtf8());
            }
        }
    }
  }
}
```

我们运行上面的代码，可以得到如下的输出：

```
A-1000-1550572395399   column=f:age, timestamp=1549091990253, value=54
A-1000-1550572395399   column=f:uuid, timestamp=1549091990253, value=e9b10a9f-1218-43fd-bd01
A-1000-1550572413799   column=f:age, timestamp=1549092008575, value=4
A-1000-1550572413799   column=f:uuid, timestamp=1549092008575, value=181aa91e-5f1d-454c-959c
A-1000-1550572414761   column=f:age, timestamp=1549092009531, value=33
A-1000-1550572414761   column=f:uuid, timestamp=1549092009531, value=19aad8d3-621a-473c-8f9f
B-1000-1550572388491   column=f:age, timestamp=1549091983276, value=1
B-1000-1550572388491   column=f:uuid, timestamp=1549091983276, value=cf720efe-2ad2-48d6-81b8
B-1000-1550572392922   column=f:age, timestamp=1549091987701, value=7
B-1000-1550572392922   column=f:uuid, timestamp=1549091987701, value=8a047118-e130-48cb-adfe

hbase(main):020:0> scan 'iteblog', {STARTROW => 'A-1000', ENDROW => 'A-1001'}
ROW                    COLUMN+CELL
 A-1000-1550572395399      column=f:age, timestamp=1549091990253, value=54
 A-1000-1550572395399      column=f:uuid, timestamp=1549091990253, value=e9b10a9f-1218-43fd-bd01
 A-1000-1550572413799      column=f:age, timestamp=1549092008575, value=4
 A-1000-1550572413799      column=f:uuid, timestamp=1549092008575, value=181aa91e-5f1d-454c-959c
 A-1000-1550572414761      column=f:age, timestamp=1549092009531, value=33
 A-1000-1550572414761      column=f:uuid, timestamp=1549092009531, value=19aad8d3-621a-473c-8f9f
3 row(s)
Took 0.0569 seconds
```

可以看到，和我们使用 HBase Shell 输出的一致，而且我们还把所有的 UID = 1000

的数据拿到了。好了，到这里，使用协处理器查询 HBase
加盐之后的表已经算完成了，明天我将介绍使用 Spark 如何查询加盐之后的表。