Rapids团队 - 通用版本实现补充说明

1. git版本信息

该通用版本的提交在 https://code.aliyun.com/191836400/IncrementalSync/commits/master 上(私有仓库，需要有middlewarerace2017 reporter权限访问到)，对应版本号 为 **44f5b54f27074f374ceae9428dd0d3d8d572d3c0** ，可以通过链接 https://code.aliyun.com/191836400/IncrementalSync/commit/44f5b54f27074f374ceae9428dd0d3d8d572d3c0 直接访问到

2. 通用性的说明

2.1 我们版本支持的通用性

我们这个版本在算法上支持任何的数据操作，包括update key/update property/insert key/delete key。**支持当前正式赛和热身赛schema基础上数据集的任意变更**，如果schema修改需要修改下面所说的 `NonDeleteOperation` 和 `RecordScanner` 的两个注意点（我们assume线上版本schema是固定的，这样实现有利于提升性能）。这个通用性的版本在线上的最好成绩为8.979s(对应提交时间:2017-06-26 08:13:14)。这个实现中的主要数据结构和操作如下:

- 两个关键的成员变量，一个记录数据库(含有垃圾，因为不remove,但包含数据库中当前所有信息和垃圾)，一个记录range范围内记录。该实现中的HashMap参考gnu trove hashmap进行修改，使其更memory友好，但是也更为专用。

```
public static YcheHashMap recordMap = new YcheHashMap(24 * 1024 * 1024);
public static THashSet<LogOperation> inRangeRecordSet = new THashSet<>(4 * 1024 * 1024);
```

- 对DeleteOperation 操作

```
@Override
public void act() {
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }
}
```

- 对InsertOperation 操作

```
@Override
public void act(){
    recordMap.put(this); //1
    if (PipelinedComputation.isKeyInRange(relevantKey)) {
        inRangeRecordSet.add(this);
    }
}
```

- 对UpdateOperation 操作

```
@Override
public void act(){
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this);
//2
    insertOperation.mergeAnother(this); //3
};
```

- 对UpdateKeyOperation 操作

```
@Override
public void act() {
    InsertOperation insertOperation = (InsertOperation) recordMap.get(this);
//2
    if (PipelinedComputation.isKeyInRange(this.relevantKey)) {
        inRangeRecordSet.remove(this);
    }

    insertOperation.changePK(this.changedKey); //4
    recordMap.put(insertOperation); //5

    if (PipelinedComputation.isKeyInRange(insertOperation.relevantKey)) {
        inRangeRecordSet.add(insertOperation);
    }
}
```

我们的实现将占主要时间的第一阶段计算流水线进行了分工，mmap reader, mediator, transformer和computation worker四种actor, 对应的模块和actor交互可能可以被提取出来进行实际场景的应用。如果该实现有可能应用到实际场景，需要注意下面`NonDeleteOperation`和`RecordScanner`的两个注意点。

2.2 NonDeleteOperation通用化注意点

由于实际生产环境中，数据库的schema基本是不会变化的，所以我们针对比赛数据对应的单表结构进行了存储的设计。

在NonDeleteOperation中，数据通过index存储了下来。详细信息如下所示，如果schema改变用户可以依据对应table的meta信息来改变存储结构。

```java
byte firstNameIndex = -1;
byte lastNameFirstIndex = -1;
byte lastNameSecondIndex = -1;
byte sexIndex = -1;
short score = -1;
int score2 = -1;
```

我们团队实现了index和真实char 之间的转换，详细代码如下:

通过`String toChineseChar(byte index)`可以把对应index转化成为具体的`char`，通过`byte getIndexOfChineseChar(byte[] data, int offset)`可以把具体的byte[]转化成为index。

```java
public static int[] INTEGER_CHINESE_CHAR = {14989440, 14989441, 14989443, 14989449, 14989450, 14989465, 14989712, 14989721, 14989725, 14989964, 14989972, 14989996, 14990010, 14990230, 14991005, 14991023, 14991242, 15041963, 15041965,
15041970, 15042203, 15042712, 15042714, 15043227, 15043249, 15043969,
15044497, 15044749, 15044763, 15044788, 15045032, 15047579, 15048590,
15049897, 15050163, 15050917, 15052185, 15056301, 15056528, 15106476,
15108240, 15108241, 15111567, 15112334, 15112623, 15112630, 15113614,
15113640, 15113879, 15114163, 15118481, 15118751, 15175307, 15176860,
15176880, 15176882, 15176887, 15178634, 15182731, 15240841, 15249306,
15250869, 15303353, 15303569, 15307441, 15307693, 15308725, 15308974,
15309192, 15309736, 15310233, 15313551, 15313816, 15317902};
private static HashMap<Integer, Byte> indexMap = new HashMap<>();
public static byte[][] BYTES_POINTERS = new byte[INTEGER_CHINESE_CHAR.length][];

static {
    for (byte i = 0; i < INTEGER_CHINESE_CHAR.length; i++) {
        indexMap.put(INTEGER_CHINESE_CHAR[i], i);
        BYTES_POINTERS[i] = InsertOperation.toChineseChar(i).getBytes();
    }
}

public static String toChineseChar(byte index) {
    int intC = INTEGER_CHINESE_CHAR[index];
    byte[] tmpBytes = new byte[3];
    tmpBytes[0] = (byte) (intC >>> 16);
    tmpBytes[1] = (byte) (intC >>> 8);
    tmpBytes[2] = (byte) (intC >>> 0);
    return new String(tmpBytes);
}

private static byte getIndexOfChineseChar(byte[] data, int offset) {
    int intC = toInt(data, offset);
    return indexMap.get(intC);
}
```

现实生活中字符个数是有限的，所以我们才想到了该index的方式，而且关系型数据库中一般会specify对应字段最长长度，扩展代码时候可以类似 lastName进行处理，存多个index，如果index的字符比较多的话，可以直接考虑存储char(2 byte)。要扩展到实际中进行使用的话，需要修改`INTEGER_CHINESE_CHAR`为所有可能的char，或者直接考虑存储char(2 byte)(需要修改`byte getIndexOfChineseChar(byte[] data, int offset)`为直接返回`char`)。

相应地，下面两个函数也要进行相应的修改，针对实际中某个table，`addData(int index, ByteBuffer byteBuffer)`在构建`InsertOperation`时候使用，而`mergeAnother(NonDeleteOperation nonDeleteOperation)`在落实update到对应Record时候使用。

```java
public void addData(int index, ByteBuffer byteBuffer) {
    switch (index) {
        case 0:
            firstNameIndex = getIndexOfChineseChar(byteBuffer.array(), 0);
            break;
        case 1:
            lastNameFirstIndex = getIndexOfChineseChar(byteBuffer.array(), 0);
            if (byteBuffer.limit() == 6)
                lastNameSecondIndex = getIndexOfChineseChar(byteBuffer.array
(), 3);
            break;
        case 2:
            sexIndex = getIndexOfChineseChar(byteBuffer.array(), 0);
            break;
        case 3:
            short result = 0;
            for (int i = 0; i < byteBuffer.limit(); i++)
                result = (short) ((10 * result) + (byteBuffer.get(i) - '0'));
            score = result;
            break;
        case 4:
            int resultInt = 0;
            for (int i = 0; i < byteBuffer.limit(); i++)
                resultInt = ((10 * resultInt) + (byteBuffer.get(i) - '0'));
            score2 = resultInt;
            break;
        default:
            if (Server.logger != null)
                Server.logger.info("add data error");
            System.err.println("add data error");
    }
}

public void mergeAnother(NonDeleteOperation nonDeleteOperation) {
    if (nonDeleteOperation.score != -1) {
        this.score = nonDeleteOperation.score;
        return;
    }
    if (nonDeleteOperation.score2 != -1) {
        this.score2 = nonDeleteOperation.score2;
        return;
    }
    if (nonDeleteOperation.firstNameIndex != -1) {
        this.firstNameIndex = nonDeleteOperation.firstNameIndex;
        return;
    }
    if (nonDeleteOperation.lastNameFirstIndex != -1) {
        this.lastNameFirstIndex = nonDeleteOperation.lastNameFirstIndex;
        this.lastNameSecondIndex = nonDeleteOperation.lastNameSecondIndex;
        return;
    }
    if (nonDeleteOperation.sexIndex != -1) {
        this.sexIndex = nonDeleteOperation.sexIndex;
        }
    }
```

2.3 RecordScanner通用化注意点

当前的RecordScanner利用了当前表中数据字段长度范围和fieldName的特点，在实际使用中需要修改RecordScanner得以适应其他表结构。

例如，下面的一些skip函数需要作相应数据表的修改。

```
private void skipField(int index) {
    switch (index) {
        case 0:
            nextIndex += 4;
            break;
        case 1:
            nextIndex += 4;
            if (mappedByteBuffer.get(nextIndex) != FILED_SPLITTER)
                nextIndex += 3;
            break;
        case 2:
            nextIndex += 4;
            break;
        default:
            nextIndex += 3;
            while (mappedByteBuffer.get(nextIndex) != FILED_SPLITTER) {
                nextIndex++;
            }
    }
}

private void skipHeader() {
    nextIndex += 20;
    while ((mappedByteBuffer.get(nextIndex)) != FILED_SPLITTER) {
        nextIndex++;
    }
    nextIndex += 34;
}

private void skipKey() {
    nextIndex += RecordField.KEY_LEN + 3;
}

private void skipNull() {
    nextIndex += 5;
}

private void skipFieldForInsert(int index) {
    nextIndex += fieldSkipLen[index];
}

private void getNextBytesIntoTmp() {
    nextIndex++;

    tmpBuffer.clear();
    byte myByte;
    while ((myByte = mappedByteBuffer.get(nextIndex)) != FILED_SPLITTER) {
        tmpBuffer.put(myByte);
        nextIndex++;
    }
    tmpBuffer.flip();
}
```

```
private long getNextLong() {
    nextIndex++;

    byte tmpByte;
    long result = 0L;
    while ((tmpByte = mappedByteBuffer.get(nextIndex)) != FILED_SPLITTER) {
        nextIndex++;
        result = (10 * result) + (tmpByte - '0');
    }
    return result;
}

private long getNextLongForUpdate() {
    primaryKeyDigitNum = 0;
    nextIndex++;

    byte tmpByte;
    long result = 0L;
    while ((tmpByte = mappedByteBuffer.get(nextIndex)) != FILED_SPLITTER) {
        nextIndex++;
        primaryKeyDigitNum++;
        result = (10 * result) + (tmpByte - '0');
    }
    return result;
}

private int skipFieldName() {
    // stop at '|'
    if (mappedByteBuffer.get(nextIndex + 1) == 'f') {
        nextIndex += 15;
        return 0;
    } else if (mappedByteBuffer.get(nextIndex + 1) == 'l') {
        nextIndex += 14;
        return 1;
    } else {
        if (mappedByteBuffer.get(nextIndex + 2) == 'e') {
            nextIndex += 8;
            return 2;
        } else if (mappedByteBuffer.get(nextIndex + 6) == ':') {
            nextIndex += 10;
            return 3;
        } else {
            nextIndex += 11;
            return 4;
        }
    }
}
```