Flink Streaming SQL实践：十分钟解决实时计算需求

内容简介：Apache Flink（下称Flink）毫无疑问是目前流式计算领域中最流行的计算框架之一，之家也选择基于Flink进行改造和开发，构建了实时计算平台，帮助大家解决实时计算的需求和问题。目前之家Flink实时计算平台支持User Programming和Streaming SQL两种方式的任务提交。其中Streaming SQL以其学习成本低，开发速度快，运行稳定性良好等诸多好处，成为了我们目前极力推荐的作业编写与运行模式。而笔者希望通过这篇文章的介绍，能帮助大家快速学习到流式计算/SQL的一些知识，并分享

总篇86篇 2020年 第10篇

Apache Flink（下称Flink）毫无疑问是目前流式计算领域中最流行的计算框架之一，之家也选择基于Flink进行改造和开发，构建了实时计算平台，帮助大家解决实时计算的需求和问题。目前之家Flink实时计算平台支持User Programming和Streaming SQL两种方式的任务提交。其中Streaming SQL以其学习成本低，开发速度快，运行稳定性良好等诸多好处，成为了我们目前极力推荐的作业编写与运行模式。而笔者希望通过这篇文章的介绍，能帮助大家快速学习到流式计算/SQL的一些知识，并分享之家实时计算平台利用Streaming SQL解决实时计算需求上的经验和案例。

本文大体上有以下三个部分:

1. 介绍Steaming SQL相关的一些理论知识
2. 介绍之家在Flink Streaming SQL的实践与优化
3. 介绍利用之家Flink Streaming SQL快速解决实时需求的经验与案例

（对理论不是很感冒的朋友可以直接跳过1或者1、2部分)

（本文提及的流处理模型默认为Dataflow及其衍生的模型）

1. Streaming SQL Related

Streaming SQL，顾名思义，使用SQL的方式进行流式计算。或者我们这样理解: 利用关系代数模型来完成对流式计算模型的描述 。离线计算使用SQL其实很好理解，而且业界已经有很多成熟框架和方案。而Streaming SQL相对来说就没那么轻松了。为了帮助大家理解更加清楚地理解相关概念， 快速从SQL/批处理的思路迁移到Streaming SQL/流处理上去 ，我们不妨提出几个问题:

- 流处理，批处理， `Table` 和 `Stream` 的联系是怎样的?
- Streaming SQL与SQL的关系是怎么样，Streaming SQL的执行边界在哪?
- Flink对于Streaming SQL实现支持情况如何?

1.1 `Table` 和 `Stream` 的联系

考虑一下传统关系型 数据库 的WAL技术，数据在真正写入磁盘之前，会先进行预写日志，预写日志是 `Append-Only` 的， 然后串行消费日志的内容来完成对数据的变更。对照一下对流的定义，我们不难发现将预写日志看成流并没有什么模型上的不匹配。从而我们不难发现， 通过消费处理日志数据流完成了对表的构建 。反之， 数据库表产生的Change Log天然是流 。

基于上述分析我们可以得出一个流和表的关系：

- 表是“静态”的数据

  这里并非是指表的数据是一直不变的，而是指对于每个 $T = t$ ，表有且仅有一个与之对应的数据快照(snapshot)。表一直在持续累计数据
- 流是“动态”的数据

  相比表是在描述一个时间点的数据，数据流表示数据随着时间的演化情况。流标识了数据变化的行为
- 流 => 表: 流在时间维度上对 `data change` 进行“积分”得到表
- 表 => 流: 表在时间维度上对 `data change` 进行“微分”得到流
- 流 => 表与表 => 流在某种程度上互为逆运算

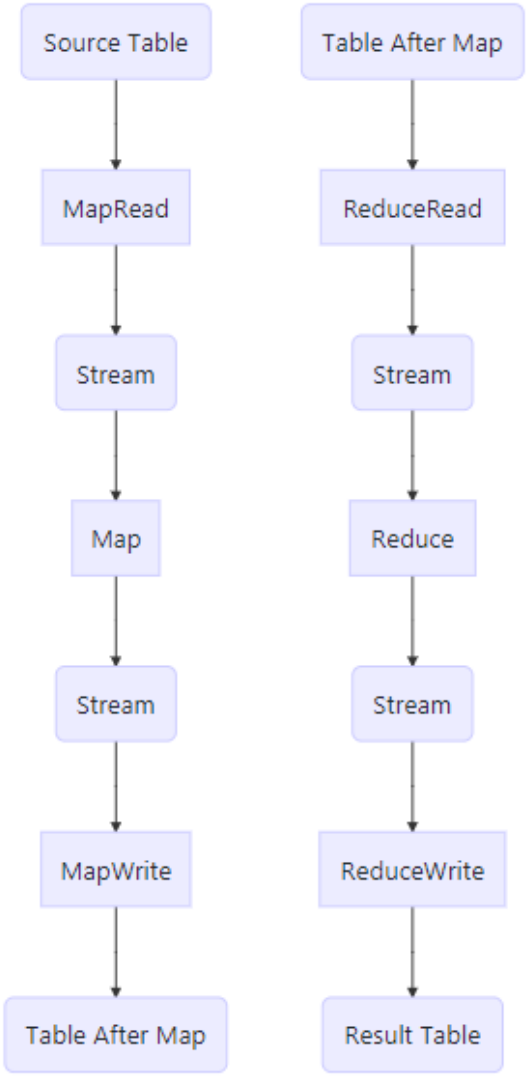
1.2 流批计算、流和表的关系

通过前文的分析我们可以得知流表的关联和与转化关系， 如何将 批处理 融合进这个模型中呢 ？

对于流处理，我们会发现绕不开时间这个维度。想必大家都对 `watermark` 这个概念有一定了解。 `waterwark` 本质上提供了一种数据完整性的承诺:对于任意给定 `watermark` $W = t$ ，所有 $W' \leq t$ 的数据均已到达， 由此可知， `watermark` 衡量一个流或者一个流计算在时间维度上推进程度 。理想状态下， `watermark` 应该能完美标识数据的完整性。（当然实际计算中很多场景下想利用 `watermark` 达成完全的承诺是十分困难的，所以才有了基于启发式 `waterwark` 在准确性和延迟上的trade off手段）。我们思考一下设 T 为 `EventTime` ，对于每个 $T = t$ ，表有且仅有一个与之对应的数据快照(snapshot)，结合上面的分析可知，这个快照应该和对应的流waterwark $W = t$ 时连续累积的结果是一致的，那么批处理计算 t 时刻的这个表数据快照，和相同计算逻辑的流处理计算在 $W = t$ 时的结果也应该是完全一致的。

下面我们看一个来自《Streaming Systems》中对MapReduce的拆解。

1.2.1 对 MapReduce拆 解



分析

简单解释一下上图， 输入的数据为表数据 ， 左边是Map阶段，右边是Reduce阶段:

1. MapRead会将表的数据以迭代的形式逐一读入，这个过程我们完全可以把它看成一个从表到流的过程。
2. Map会消费这个流并产出一条新的流。
3. MapWrite消费Map产出的流，并进行聚合，落盘成表，这个过程恰好和从流到表一致
4. Reduce大致同理

1.2.2 对Spark拆解

我们再来看看Spark是否也能用流表统一性去解释，对Spark熟悉的同学一定很熟悉RDD，RDD模型对比MR的最大好处之一就是可以进行算子融合，避免了不必要的落盘。数据在每个分区以 迭代器 的形式进行处理，仍可以用上文对MR的拆解思路去拆解RDD。

1.3 流式关系代数

众所周知，SQL是利用关系代数来对 关系(等价于表，下文不进行区分) 进行运算，且关系代数具有闭包性（closure property），关系代数经过运算的产出也一定是 关系 ，而在前面我们论证了流表的统一性，那将对表的计算迁移到对流的计算就是流式关系代数的基础所在。

流式关系代数是关系代数的扩展 ，大体是逻辑与思路与关系代数是一致的，但有两点需要注意:

流式关系代数的闭包性

关系代数本身是针对表的，可事实上我们需要在运算中支持流。即除了传统的关系算子以外，还需要 显式地增加执行流表转换动作的算子 ，这在一定程度上牺牲闭包性。

1.3.1 时间属性（time-varying）

考虑一下我在前面的分析， 流与表的转换是在时间维度上进行的 ，传统的关系运算作用在静态的数据集上，准确的说作用在 对应时间点的快照 。随着时间的推移，我们可以得到数据快照在时间维度上的序列， 序列中的每个快照与唯一时间段（数据没变化的时间都合并到相同时间段中）对应 ，流式关系代数等价于将关系代数分别作用于序列中的每个快照。

引用《Streaming Systems》的观点:

- （传统）关系代数所有的算子在流式关系代数中均是合法的
- （传统）关系代数算子内的闭包性在流式关系代数是完整的

(我们目前只讨论语义层面，暂时不考虑实现上的问题)

Streaming SQL与 SQL的对比：

对比	关系代数	流式关系代数
作用主体	表	流
执行有无终点	有，且执行有确定结果	无，随时间变化， *会不断产生结果的更新值*
时间属性	无	有
计算窗口支持	无	有

1.3.2 Streaming SQL的局限性与执行边界

Streaming SQL可以解决所有的流式计算吗？其实并不能，大体有以下两点原因：

1. SQL的局限性

   SQL虽然具有诸多优点，但是其实隐含了关系型模型的约束，相比一些更通用的语言/模型，表达能力和使用范围是受限的

2. 流表转换的隐式约束

   由上文的分析可以知道，流式关系代数中流与表是能互相转化的，且必须存在对应的转化算子，而我们接受这一结论时，其实包含了一个隐式约束： 在流式关系代数中 `Stream` 中的数据满足 `Table` (关系)中的 `Record` 的定义 ，通俗的说， `Stream` 中的数据必须能是 `Table` 的 `Row` ，可以用相同的Schema定义去描述，满足这个流表互转的约束才能满足组关系代数计算的条件。

   考虑到一个实际的流计算要外部输入 `Source` 和外部输出 `Sink`

   - 一定存在输入函数 $P_{source}(X, S_{source})$ 使得对于每一条输入的数据都有 $n(n \in N)$ 条满足关系运算条件的输出，其中 S_{source} 是 Source 端内部状态累积。
   - 一定存在输出函数 $Q_{sink}(R, S_{sink})$,使得经过关系代数运算的输出的每一个元素都被转换写入 Sink 中，其中 S_{sink} 是 Sink 端内部状态累积。

   让我们总结一下St reaming SQL的边界，满足下面三个条件才可以利用Streaming SQL的方式处理问题：

- 对于 `Source` ，须能将输入的数据流转换为满足关系运算的数据流
- 对于 `Sink` ，须能处理带有关系运算约束的输出
- 对于 `Transformation` ，须能保证关系代数可以完全表达执行逻辑

1.4 流表转换与流的操作语义

对于传统意义上的表来说，存在DML/DDL/DCL等一系列的操作。为了简化，我们不妨先只考虑DML（INSERT/DELETE/UPDATE）操作。前面讨论了流表联系与转换，那么对于流来说，也应该能够支持以上操作语义才可以。

一个现实世界的例子：Mysql Binary Log

我们先从现实存在的例子入手，Mysql Binary Log（Binlog）是 Mysql 的Change Log。我们假设目前为ROW模式，其中DML相关的事件：

```
1. WRITE_ROWS事件   -> INSERT
2. UPDATE_ROWS事件 -> UPDATE
3. DELETE_ROWS事件 -> DELETE
```

3种事件对应了对Binlog中数据的三种不同操作，不难发现，对于 每次 变更操作，都会有对应的change log 写入。我们再仔细推敲一下UPDATE事件的结构，其中包含了 `before` ,表示发起更新之前的值和 `after` 表示发起更新后的值。 一个UPDATE操作完全可以被拆分成一个对旧值的DELETE和对新值的INSERT 。

Stream的操作类型

上文的例子我们可以看到，流中数据都携带有操作类型这一属性。事实上，流确实隐含着操作类型这一概念，流的操作类型可以分为两种：

- INSERT/Accumulate 代表新增/聚集

- DELETE/Retract 代表删除/撤销

对于UPDATE更新操作，我们可以通过先DELETE旧值后INSERT新值来完成。

操作类型这一属性虽然游离于数据内容之外，但是却十分重要，通过操作类型我们才能正确在流地表达数据的真实操作。结合前文流表转换的分析，表向流转换时，会在数据中隐式地包含操作类型的属性。流在向表转换时，也需要正确地应用操作类型处理数据。

1.5 Streaming SQL@Flink

Flink基于流式关系代数实现了自己的Streaming SQL， Flink的经典执行模式就是 `Source` -> `Transformation` -> `Sink` ,其实Flink的 Streaming SQL本质上就是依据这三个阶段将SQL进行一系列的解析优化转化，最后变成Flink的一系列Operator并执行

1.5.1 窗口语义

Flink 针对窗口增加了对应的关系算子：

- `TUMBLE` -> 固定窗口
- `HOP` -> 滑动窗口
- `SESSION` -> 会话窗口

1.5.2 流批转换算子

- Flink提供 `fromDataStream` 等从流到表的关系算子
- Flink提供 `toAppendStream` 、 `toRetractStream` 等从表到流的关系算子

1.5.3 输入与输出

对照1.3.2的内容

- Flink提供了 `TableSource` 完成对外部输入的转换
- Flink提供了 `TableSink` 完成对外部输出的转换

1.5.4 聚合操作

聚合操作对应 `group by` ，根据需求选择对应的聚合函数比如 `sum` ， `count` 或者是user-defined-agg-function来完成，支持window聚合和non-window聚合。

1.5.5 Stream类型

Flink包含三作类型的流：

对比	INSERT	DELETE	必须定义KEY
Upsert Stream	√	√	√
Retract Stream	√	√	x
Append Stream	√	x	x

Append Stream

Append Stream 只存在 `INSERT` 一种操作类型，所有数据都是插入

Retract Stream

Retract Stream 包含 `INSERT` 和 `DELETE` 两种操作类型，正如Retract的字面意思，对于 `INSERT` 的数据，可以通过 `DELETE` 来完成撤回

Upsert Stream

Upsert Stream 包含 `INSERT` 和 `DELETE` 两种操作类型，为了正确表达 `Update` 的语义，Upsert Stream 必须要有 `KEY` 。

1.5.6 Stream Join

Flink支持多种 `Stream Join` （详情可以参考官网），但是在实际使用的时候，不得不考虑状态缓存的大小限制和准确性之间的权衡，其次Join存在修改放大问题，实际使用需要进行考量。

2. 之家在Flink Streaming SQL改造优化的实践之路

之家Flink实时平台经过大半年的迭代进化，目前已经在易用性，稳定性，功能全面性已经取得了一定阶段性的成果，截止写稿目前已经有超过600个任务，当前有效任务300+，其中SQL任务的比例超过70%，日均计算量超过9000亿条。我们在SQL上做了很多比较多的优化和实现，包括但不限于：

- Source Sink封装及优化
- 大量的built-in函数支持
- 维表JOIN支持
- 实用 工具 优化策略
- 拥抱1.9+
- 原生支持加载Retract/Upsert Stream

2.1 Source Sink封装及优化

Flink从读取Source的数据进行消费，计算写入Sink中，Source和Sink往往对应的外部系统的抽象，往往也是调优的重点。实时小组结合公司的业务需要，封装了常见的Source和Sink：Kafka/Mysql(TiDB)/Redis/Elasticsearch/Http/etc. 用户在平台上编写建表语句并进行表的“创建”（ 这

里强调一下，在平台建表事实上并没有建出表的实体，只是保存下了建表语句，保存了表的关系描述，并进行解析提取信息，只有真正执行job加载建表语句转换成operator的时候才真正意义的“创建”），以Kafka Source和Mysql Sink为例。

2.1.1 Kafka Source

Kafka Source Connecter官方提供了比较完整的实现，我们决定直接基于官方的 `KafkaConsumerBase` 构建Kafka的Table Source。首先现在KafkaConsumer中嵌入各类监控指标，将监控指标直接发送到基于公司监控平台上，其次是对各类数据格式的支持，根据收集到的需求，我们目前支持了JSON/SPLIT/Protobuf/Raw/BINLOG等格式，依据用户指定的格式进行解析，从中获取字段值。

以deserializationType = JSON为例:

JSON格式

```
{
  "action":"click",
  "dt":"2020-01-20",
  "device_id":"edbdf_ce424452_2"
}
```

在平台上建表语句

```
CREATE TABLE `kafka_source_demo`
(
`action` `text`,
`dt` `text`,
`device_id` `text`
)
WITH (
type` = 'kafka',
`servers` = '***',
`deserializationType` = 'JSON',
`topicName` = 'test'
)
```

2.1.2 Mysql(TiDB) Sink

我们选择自己去编写构建MysqlTableSink。监控覆盖自不必说，MysqlSinkFunc提供了很多优化项供用户进行调整:

优化项	解释
batchSize	刷写批次大小
flushInterval	刷写时间间隔
isAsync	是否使用异步写

isFlushOnCheckpoint	是否在checkpoint是进行数据刷写
isKeyByBeforeSink	是否在写入数据之前进行KeyBy分区（大幅减小写冲突）
oneSink	强制sink并行度=1 （ 应对特殊场景的串行化要求）
isUpdateOnDuplicateKey	使用UpdateOnDuplicateKey进行部分更新模式否则使用Replace into全量更新模式
mergeByKey	写之前按Key Merge，减少写入量
ignoreDelete	忽略Delete事件
ignoreException	忽略异常

用户可以根据自己的需求组合参数，达成最佳写入效果，此外平台MysqlSink提供了自动创建目标库Sink表，自动数据类型转换(当写入类型和目标库类型冲突时，尝试类型转换)。

2.2 大量的built-in函数支持

在平台发展过程中，不断收集反馈用户需求，分析并编写了大量SQL函数，截止到现在平台对外提供了100+ SQL内置函数(UDF/UDTF/UDAF)，增强了SQL功能和表达能力，从而支撑业务开展。函数涉及的场景比较多，包括但不限于 JSON 解析支持/时间转换系列/类型转换系列/统计指标特化系列函数/时间窗口模拟系列函数/字符串操作系列函数/数据类型转换类，下面以统计指标特化系列函数和窗口模拟系列函数举例。

2.2.1 统计指标特化系列函数

2.2.1.1 携带初始值的统计函数

有些指标计算希望从一个初始值开始，基于这个场景，我们提供了可以从外部加载初始值的聚合函数，例如 `count_with_external_redis()` ，用户可以通过显式传参加载来获取 redis 的地址和构成key的逻辑，从而在第一次初始化时获取一个默认值，之后从checkpoint恢复可以选择不从外部读取。

2.2.1.2 hll_distinct_count

计算去重统计是一个很常见的需求，对于数据量大但是精度要求相对低的场景，可以使用平台提供的基于HyperLogLog的去重统计函数，内存空间占用骤减。

```
hll_distinct_count( key:Any)
```

2.2.1.3 性能测试Tp系列函数

Tp这里的Tp是指Top percentile,常用于性能测试和分析，平台支持估算常见的Tp，如Tp99，Tp999等。

```
udf_tp99(v:double)
```

2.2.1.4 单调增序列的统计函数

这类统计函数相比普通的统计函数，参数列表中多了一个Long类型的参数，看下面例子：

```
/**
 *
 *
 * 单调递增式count
 *
 *  given compare number:Long is based on an sequence
 *  so if the value of compare number becomes higher
 *  this count shall  be zero which means starting a new count
 *  but if given an new compare number when is smaller than before,the count won't c
hange.
 */
@NotThreadSafe
public class JavaCountWithNumberCompareAcc {

    private long count = 0l;
    @VisibleForTesting
    public long currLong = 0;

    public void incOrClear(long compare) {
        if (compare == currLong) {
            inc();
        } else if (compare > currLong) {
            currLong = compare;
            clearCount();
            inc();
        }
    }
    // 下略
    }
```

对于数据乱序并且希望一直输出最新term（最新逻辑时间）的统计值时，使用此类统计函数。

2.2.2 时间窗口模拟系列函数

窗口计算是流式计算特有一直计算方式，但是窗口计算是“昂贵”的，我们可以使用 常规聚合 的方式去对窗口计算进行模拟。（具体使用会在案例策略中展开）

2.2.2.1 滚动窗口模拟函数

滚动窗口模拟是 `ScalarFunction` ，函数接受时间，和窗口大小，计算出该条数据对应的窗口归属：

```
tumble_window_group(ts:Long,windowSize:Long): Long
```

2.2.2.2 滑动窗口模拟函数

滑动窗口是`TableFunction`，函数接受时间，窗口大小，滑动大小，计算出该条数据对应的多个窗口归属：

```
slide_window_group(ts:Long,windowSize:Long,slideSize): List[Long]
```

2.3 维表支持

维表支持算是老生常谈了，我们之前是基于1.7.2版本进行开发的，思路简单来说分为2部分：

1. 用户在平台上创建维表时语句带有特殊标识，系统会识别并加以保存管理。
2. 用户提交带有维表Join的SQL任务的时候，平台会识别到并将维表Join改写为FlatMap

2.4 实用工具优化策略

2.4.1 实用工具

2.4.1.1 SQL交互式开发工作台

平台提供SQL在线编辑和运行配置在线编辑能力，用户可以使用平台提供页面编写SQL，配置调整。不仅如此 用户在编写SQL任务的过程中，总是会希望调试SQL，查看结果，平台更提供了 查看即时执行结果 的功能，SQL开发效率大幅上升。

2.4.1.2 任务生命全托管服务

平台用户只需要关心业务逻辑本身，平台负责作业部署，运行及整个生命周期的维护，checkpoint自动管理，作业状态可视化等功能，还提供丰富的监控、报警和日志服务，满足用户各类开发调试运维分析需求。

2.4.1.3 数据可视化系统整合

平台目前有大量任务直接与公司的数据可视化系统 AutoBI 对接，利用Streaming SQL快速完成实时计算，结合 AutoBI 快速可视化报表能力，迅速解决可视化报表的需求。

2.4.2 优化策略

2.4.2.1 SQL预解析

用户在平台编写SQL后，会由 实时计算平台的编写SQL解析组件 而非Flink原生SQL去进行解析，校验和优化后转为基于Table API的Flink代码进行执行。这样平台侧可以在SQL解析阶段获得极大的自由度，更加方便平台去做分析校验拓展优化预加载等动作，很多优化策略都是基于此展开的。

2.4.2.2 Submit Client二次开发

Flink的任务是通过Client向集群提交的，我们选择重新封装和改造Client，理由与2.4.2.1类似，保证平台侧在任务提交阶段也有足够大的自由度，方便平台从中添加逻辑，包括但不限于：

- REST接口式提交
- YARN/K8S提交支持，并对上层行为统一
- JAR包管理与指定SDK的加载
- 任务提交信息记录持久化
- 内置函数注册
- 负载调节避免大量集中提交

2.4.2.3 Source自动复用

当用户编写的SQL中出现同一个Source被多次使用的时候，平台会自动优化执行计划，提前将Source进行一次materialize，避免反复加载数据源造成不必要的开销。

2.4.2.4 临时创建视图语法支持

支持编辑SQL时创建临时视图的语法，增强SQL的表现力。

2.4.2.5 常规聚合代替窗口聚合

利用平台提供的时间处理函数和窗口模拟函数，可以将昂贵的窗口聚合转换为常规聚合，性能和时效性都有大幅改善，被广泛使用。

2.4.2.6 字段字符集扩充

Flink Table默认字段的字符集是 `JavaIdentifier` ，而用户在实际使用有特殊字符的需求。例如创建表时字段名称为 `@timestamp` ，原生的Flink暂时不支持。我们的做法是修改 `ExpressionParserImp` 的部分代码逻辑从而扩大 `FieldRefrence` 合法字符范围，使得含有特殊字符的字段名称也能被顺利创建、使用和计算。

2.4.2.7 micro-batch

众所周知，经过良好预热的系统，其吞吐和延迟通常是成反比的。所以对于时延要求满足的时候，可以通过配置开启micro-batch在 `sink` 之前进行合并（如果可能合并的话）和微批化，减少 `sink` 写入量和写入动作。

2.5 拥抱1.9+

Flink 1.9的发布无疑是带来很多好处，1.9版本在SQL方面进行了大量的改造和优化，不论是功能还是性能，优化和新语法支持，都有着长足的丰富和提升。我们也决定拥抱1.9+：

- 测试验证并在生产上开始使用并逐渐从老系统切量，目前切量5%左右，随着系统稳定会逐渐增加
- 基于1.9+版本进行改造和开发，基于1.9+进行组件的开发，享受新特性红利

2.6 原生加载Retract Stream

在阐述这项工作之前，我们先复习前面提到的一个概念， Stream的操作类型 ，带有操作类型的Stream在转换成表的时候，需要针对操作类型进行处理才能得到正确的表数据。

考虑下面这样一个场景：

- 场景：订单数据是用Mysql存储的，订单表超过100张，且表结构一致，存放对应全量的Binlog的Kafka，topic有N个
- 组件：Flink Streaming SQL
- 需求：希望计算订单的总金额大小，要求快速和准确的得到结果

Binlog在Kafka中的消息结构：

```
{
    "ts":0,
    "binlogFileName":"xxx",
    "binlogOffset":4,
    "op":"UPDATE",
    "tableName":"demo",
    "dbName":"demo",
    "beforeRows":{
        "value":"50"
    },
    "afterRows":{
        "value":"100"
    }
}
```

根据需求我们发现我们只需要关注三个字段的值：beforeRows.value、afterRows.value和op， 现在我们得到一张Kafka流表：

```
CREATE TABLE `kafka_source_demo_normal_1`
(
`before_value` `text`,
`after_value` `text`,
`op`:`text`
//其他字段省略
)
WITH (
type` = 'kafka',
`servers` = '***',
`deserializationType` = 'BINLOG',
`topicName` = 'test'
)
```

对应的SQL（省略union部分）：

```
select sum(
case `op`
when "INSERT" then `text_to_long`(`after_value`)
when "UPDATE" then  `text_to_long`(`after_value`)- `text_to_long`(`before_value`)
```

```
when "DELETE" then 0-`text_to_long`(`before_value`)
else 0 end
) from `kafka_source_demo_normal_1`;
```

由于Flink原生的加载数据的方式，我们没办法原生的创建一个Retract Stream 或是 Upsert Stream，只能以Append Stream的形式加载进来，所以在SQL逻辑中不得不显式地根据对应的操作语义做不同的操作，而Flink本身提供了Retract机制，我们却不能利用起来。

我们意识到这个问题，决定在Flink Table中实现原生加载Retract Stream的功能：第一版思路比较清晰：在执行计划部分增加一系列 `AppendToStream` 的关系算子和对应的 `OptRule` ，可以将Append Stream根据一定规则转换成Retract Stream 但是考虑到由于增加了新的算子，可能会对优化规则产生负面作用，导致一些Rule匹配不上。第二版的核心思路是模仿Flink SQL处理加载time attribute的方式，避免引入新的算子。

支持原生加载Retract Stream后,一条INSERT对应一条INSERT的数据，一条DELETE对应一条RETRACT的数据，一条UPDATE对应对应一条RETRACT old记录的数据和一条INSERT new纪录的数据。

建表语句：

```
CREATE TABLE `kafka_source_demo_ret_1`
(
`value` `text`,
`op`:`text`
//其他字段省略
)
WITH (
type` = 'kafka',
`servers` = '***',
`deserializationType` = 'BINLOG_RET',
`topicName` = 'test'
)
```

对应的SQL（省略union部分）：

```
select sum(text_to_long(value)) from kafka_source_demo_ret_1;
```

支持原生Retract Stream后，我们可以发现SQL和在数据库执行的SQL并没有什么不同，这也印证了前面的流批一体思路，为关系型数据库的Flink实时数仓提供了技术基础。

3. 之家Flink Streaming SQL快速解决需求案例

这篇文章的标题叫 10分钟解决实时计算需求 ，乍一看以为是噱头，但是如果能熟练理解和使用Flink Streaming SQL，10分钟解决实时计算需求也并非不可能。

在案例分析之前，我们先建立思路：

回归Flink的经典执行模式就是 `Source -> Transformation -> Sink`，我们实际解决问题的时候也是按照以上3个阶段分析：

1. 编写Source建表语句，完成Source阶段的数据映射载入
2. 编写计算逻辑SQL语句，完成Transformation阶段计算
3. 编写Sink建表语句，完成Sink阶段映射输出

3.1 经典案例（均基于实际案例简化）

3.1.1 实时统计指标计算

需求：计算每小时对应用户行为的pv和uv，日志在Kafka中，结果写入redis 结果key中

日志格式：

```
{
  "action":"click",
  "dt":"2020112011",
  "device_id":"edbdf_ce424452_2"
}
```

Source

```
CREATE TABLE `kafka_source_demo`
(
`action` `text`,
`dt` `text`,
`device_id` `text`
)
WITH (
`type` = 'kafka',
`servers` = '***',
`deserializationType` = 'JSON',
`topicName` = 'test'
)
```

Sink

```
CREATE TABLE `redis_sink_demo`
(
`action` `text`,
`dt` `text`,
`pv` `long`,
`uv` `long`
)
WITH (
`type` = 'redis',
`server` = '***',
 `valueNames`= 'pv,uv',
```

```
 `keyType`= 'string',
 `keyTemplate` = 'demo_key_${action}_${dt}_' //key的格式模板
)
```

Transformation

```
insert into `redis_sink_demo`
 (select
 `dt`,
 `action`,`count`(1) as `pv`,
 `hll_distinct_count`(`device_id`) as `uv`
 from `kafka_source_demo`
 group by `action`,`dt`
 )
```

3.1.2 实时ETL

需求：将Kafka中的物料数据更新到TiDB表中 部分字段 上，要求延迟小于5s

日志格式：

```
{
  "biz_type":"1",
  "biz_id":"20",
  "property":"test"
  //其他省略
}
```

Source

```
CREATE TABLE `kafka_source_demo_2`
(
`biz_type` `text`,
`biz_id` `text`,
`property` `text`
)
WITH (
`type` = 'kafka',
`servers` = '***',
`deserializationType` = 'JSON',
`topicName` = 'test2'
)
```

Sink

```
CREATE TABLE `ti_sink_demo` (
 `biz_type` `text`,
  `biz_id` `text`,
  `property` `text`
) WITH (
  `type` = 'mysql',
  `url` = 'xxxxx',
```

```
  `mysqlTableName` = 'test',
  `username` = 'xx',
  `password` = 'xxxxx',
  `mysqlDatabaseName` = 'xx',
  `sinkKeyFieldNames` = 'biz_id,biz_type',
  `batchSize` = 200,
  `flushInterval` = 3000,
  `needMerge` = false,
  `ignoreDelete` = true,
  `specificMysqlSinkExecutionLogicClassName` = 'duplicateKeyUpdate'
  `isKeyByBeforeSink` = true //关键，大幅减小数据库写入时锁争用
)
```

Transformation

```
insert into `ti_sink_demo`
 (select * from `kafka_source_demo_2`)
```

3.1.3 实时在线人数窗口计算

需求：客户端每30s上报用户在线信息，信息被存放在Kafka中，计算的在线人数结果写入redis。

日志格式：

```
{
  "room_id":"1",
  "ts":11110000,
  "device_id":"x14xvrt-fgd11zx-fggf"
  //其他省略
}
```

Source

```
CREATE TABLE `kafka_source_demo_3`
(
`room_id` `text`,
`ts` `long`,
`device_id` `text`
)
WITH (
`type` = 'kafka',
`servers` = '***',
`deserializationType` = 'JSON',
`topicName` = 'test3'
)
```

Sink

```
CREATE TABLE `redis_sink_demo_3`
(
`room_id` `text`,
`window_ts` `long`,
```

```
`uv` `long`
)
WITH (
`type` = 'redis',
`server` = '***',
 `valueNames`= 'uv',
 `noValueName` = true,
 `keyType`= 'string',
 `keyTemplate` = 'demo_key2_${room_id}_${window_ts}'
)
```

Transformation

对于本例，朴素思路是使用大小为30s滚动窗口进行计算，但是这样有两个问题：

1. 数据上报的 `ts` 有一定的漂移，单纯基于 `ts` 使用滚动窗口计算的结果会出现很严重的漂移，即上一个 `ts` 计算结果和下一个 `ts` 的计算结果相差很大
2. 基于窗口的计算性能和内存使用均有限制

解决方案是使用滑动窗口模拟函数结合常规聚合，窗口大小1min，滑动大小为30s。

```
insert into `redis_sink_demo_3`
 (
 select
 `room_id`,
 count(distinct `device_id`),
 `ts`
 from `kafka_source_demo_2`,LATERAL TABLE(`slide_window_group`(`timestr`, 60000, 300
00)) AS `T` (`ts`)
 group by `room_id`, `ts`
 )
```

总结

本文先从理论层面帮助大家对Streaming SQL有一个简单认识，其次介绍了之家的一些实践改造经验和历程，最后通过几个简单实例帮助大家对使用Flink Streaming SQL有一个清晰的了解，如何使用SQL去对实时计算问题进行拆解分析，助力大家快速解决实时计算需求。此外，之家也会持续结合实际需求和场景，对Flink实时计算平台不断进行升级改造，在纵深和横向不断扩展，完善平台，赋能用户。