> 你想要的改变，没有从现在开始，就永远不会开始

FlinkSql之流式持续查询

> Table API和SQL，本质上还是基于**关系型表的操作方式**；而关系型表、关系代数，以及SQL本身，一般是有界的，更适合批处理的场景。**这就导致在进行流处理的过程中，理解会稍微复杂一些，需要引入一些特殊概念。**

	关系代数（表）/SQL	流处理
处理的数据对象	字段元组的有界集合	字段元组的无限序列
查询（Query）对数据的访问	可以访问到完整的数据输入	无法访问所有数据，必须持续"等待"流式输入
查询终止条件	生成固定大小的结果集后终止	永不停止，根据持续收到的数据不断更新查询结果

可以看到，其实关系代数（主要就是指关系型数据库中的表）和SQL，主要就是针对批处理的，这和流处理有天生的隔阂。

【两者的世界观不同，因此需要引入一些新概念】

动态表（Dynamic Tables）【重点掌握】

> **因为流处理面对的数据，是连续不断的**，这和我们熟悉的关系型数据库中保存的"表"完全不同。所以，如果我们把流数据转换成Table，然后执行类似于table的select操作，**结果就不是一成不变的，而是随着新数据的到来，会不停更新。**

我们可以随着新数据的到来，不停地在之前的基础上更新结果。这样得到的表，**在Flink Table API概念里，就叫做动态表（Dynamic Tables）**。动态表是Flink对流数据的Table API和SQL支持的核心概念。与表示批处理数据的静态表不同，动态表是随时间变化的。动态表可以像静态的批处理表一样进行查询，查询一个动态表会产生持续查询（Continuous Query）。连续查询永远不会终止，并会生成另一个动态表。查询（Query）会不断更新其动态结果表，以反映其动态输入表上的更改。

流式持续查询的过程

流式持续查询的过程为：

- 流被转换为动态表。

- 对动态表计算连续查询，生成新的动态表。
- 生成的动态表被转换回流。

将流转换成表（Table）

> **为了处理带有关系查询的流，必须先将其转换为表。**从概念上讲，流的每个数据记录，都被解释为对结果表的插入（Insert）修改。因为流式持续不断的，而且之前的输出结果无法改变。本质上，我们其实是从一个、只有插入操作的changelog（更新日志）流，来构建一个表。【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

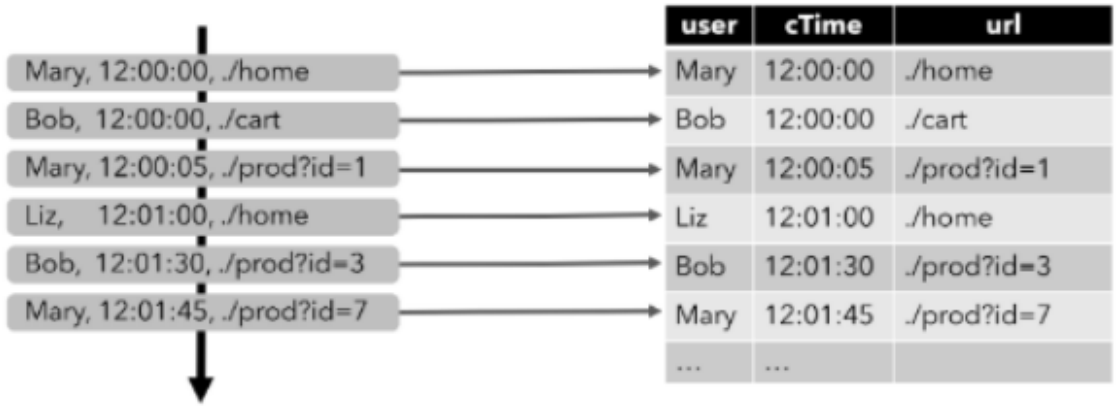
【又要转换成表了哦，但是这里的是动态表！】

为了更好地说明动态表和持续查询的概念，我们来举一个具体的例子。

比如，我们现在的输入数据，就是用户在网站上的访问行为，数据类型（Schema）

```
[
  user:  VARCHAR,   // 用户名
  cTime: TIMESTAMP, // 访问某个URL的时间戳
  url:   VARCHAR    // 用户访问的URL
]
```

下图显示了如何将访问URL事件流，或者叫点击事件流（左侧）转换为表（右侧）

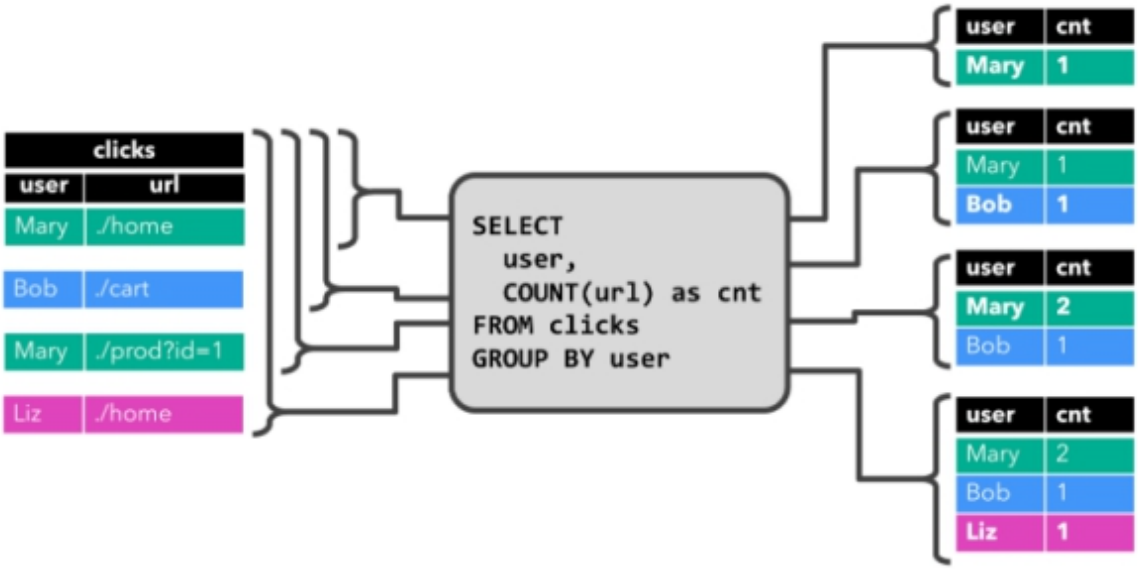


随着插入更多的访问事件流记录，生成的表将不断增长。

持续查询（Continuous Query）

持续查询，**会在动态表上做计算处理，并作为结果生成新的动态表。**与批处理查询不同，连续查询从不终止，并根据输入表上的更新更新其结果表。在任何时间点，连续查询的结果在语义上，等同于在输入表的快照上，以批处理模式执行的同一查询的结果。

> 在下面的示例中，我们展示了对点击事件流中的一个持续查询。这个Query很简单，是一个分组聚合做count统计的查询。它将用户字段上的clicks表分组，并统计访问的url数。图中显示了随着时间的推移，当clicks表被其他行更新时如何计算查询。



将动态表转换成流

> 与常规的数据库表一样，**动态表可以通过插入（Insert）、更新（Update）和删除（Delete）更改，进行持续的修改。**将动态表转换为流或将其写入外部系统时，需要对这些更改进行编码。Flink的Table API和SQL支持三种方式对动态表的更改进行编码：

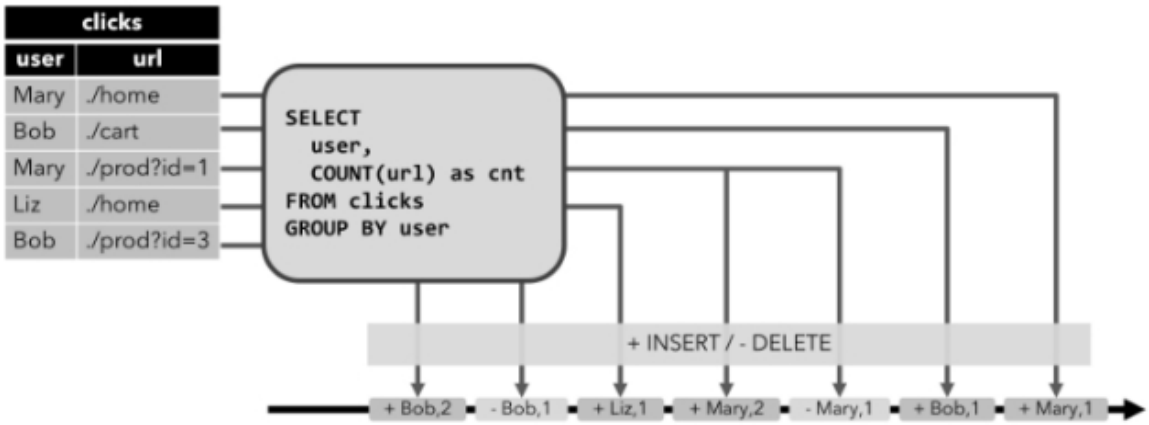
(1)仅追加（Append-only）流

> 仅通过插入（Insert）更改，来修改的动态表，可以直接转换为“仅追加”流。这个流中发出的数据，就是动态表中新增的每一行。

(2)撤回（Retract）流

> Retract流是包含两类消息的流，添加（Add）消息和撤回（Retract）消息。动态表通过将INSERT 编码为add消息、DELETE 编码为retract消息、UPDATE编码为被更改行（前一行）的retract消息和更新后行（新行）的add消息，转换为retract流。

下图显示了将动态表转换为Retract流的过程。

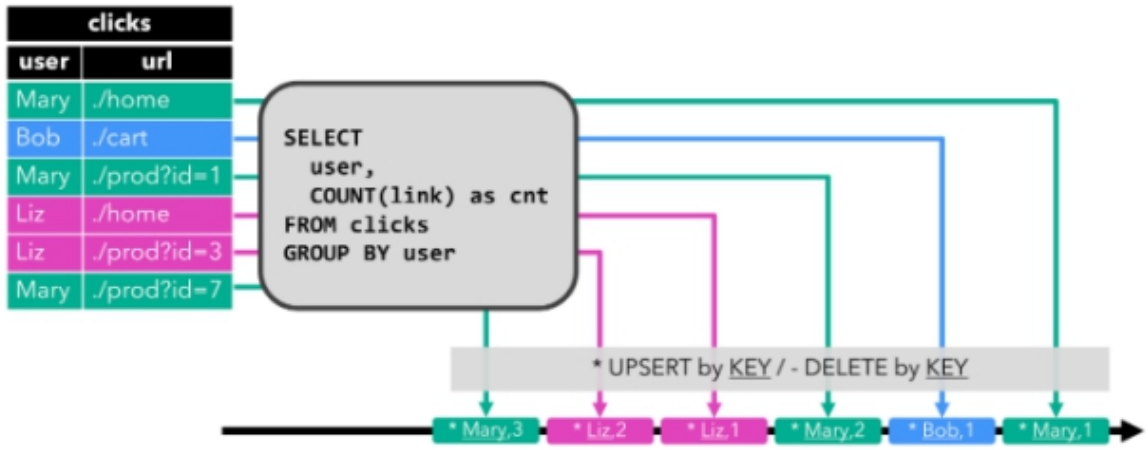


(3)Upsert（更新插入）流

> Upsert流包含两种类型的消息：Upsert消息和delete消息。转换为upsert流的动态表，需要有唯一的键（key）。

通过将INSERT和UPDATE更改编码为upsert消息，将DELETE更改编码为DELETE消息，就可以将具有唯一键（Unique Key）的动态表转换为流。

下图显示了将动态表转换为upsert流的过程。



> 这些概念我们之前都已提到过。需要注意的是，**在代码里将动态表转换为DataStream时，仅支持Append和Retract流。**而向外部系统输出动态表的TableSink接口，则可以有不同的实现，比如之前我们讲到的ES，就可以有Upsert模式。

时间特性

> 基于时间的操作（比如Table API和SQL中窗口操作），需要定义相关的**时间语义和时间数据来源**的信息。所以，Table可以提供一个逻辑上的时间字段，用于在表处理程序中，指示时间和访问相应的时间戳。时间属性，**可以是每个表schema的一部分。**一旦定义了时间属性，它就可以作为一个字段引用，并且可以在基于时间的操作中使用。时间属性的行为类似于常规时间戳，可以访问，并且进行计算。

【和前面的事件语义一样】

处理时间（Processing Time）

> 处理时间语义下，**允许表处理程序根据机器的本地时间生成结果**.它是时间的最简单概念。它既不需要提取时间戳，也不需要生成watermark。定义处理时间属性有三种方法：在DataStream转化时直接指定；在定义Table Schema时指定；在创建表的DDL中指定。

(1)DataStream转化成Table时指定

由DataStream转换成表时，可以在后面指定字段名来定义Schema。在定义Schema期间，可以使用.proctime，定义处理时间字段。注意，这个proctime属性只能通过附加逻辑字段，来扩展物理schema。因此，只能在schema定义的末尾定义它。

代码如下

```
// 定义好 DataStream
val inputStream: DataStream[String] = env.readTextFile("\\sensor.txt")
val dataStream: DataStream[SensorReading] = inputStream
  .map(data => {
    val dataArray = data.split(",")
    SensorReading(dataArray(0), dataArray(1).toLong, dataArray(2).toDouble)
  })

// 将 DataStream转换为 Table，并指定时间字段
val sensorTable = tableEnv.fromDataStream(dataStream, 'id, 'temperature,
'timestamp, 'pt.proctime)
```

(2)定义Table Schema时指定

这种方法其实也很简单，只要在定义Schema的时候，加上一个新的字段，并指定成proctime就可以了。

代码如下

```
tableEnv.connect(
  new FileSystem().path("..\\sensor.txt"))
  .withFormat(new Csv())
  .withSchema(new Schema()
    .field("id", DataTypes.STRING())
    .field("timestamp", DataTypes.BIGINT())
    .field("temperature", DataTypes.DOUBLE())
    .field("pt", DataTypes.TIMESTAMP(3))
      .proctime()    // 指定 pt字段为处理时间
  ) // 定义表结构
  .createTemporaryTable("inputTable") // 创建临时表
```

(3)创建表的DDL中指定

在创建表的DDL中，增加一个字段并指定成proctime，也可以指定当前的时间字段。

代码如下

```
val sinkDDL: String =
  """
    |create table dataTable (
    |  id varchar(20) not null,
    |  ts bigint,
    |  temperature double,
    |  pt AS PROCTIME()
    |) with (
    |  'connector.type' = 'filesystem',
    |  'connector.path' = 'file:///D:\\..\\sensor.txt',
    |  'format.type' = 'csv'
    |)
  """.stripMargin

tableEnv.sqlUpdate(sinkDDL) // 执行 DDL
```

注意：运行这段DDL，必须使用Blink Planner。

事件时间（Event Time）

> 事件时间语义，允许**表处理程序根据每个记录中包含的时间生成结果**。这样即使在有乱序事件或者延迟事件时，也可以获得正确的结果。为了处理无序事件，并区分流中的准时和迟到事件；Flink需要从事件数据中，提取时间戳，并用来推进事件时间的进展（watermark）。

(1)DataStream转化成Table时指定

在DataStream转换成Table，schema的定义期间，使用.rowtime可以定义事件时间属性。注意，必须在转换的数据流中分配时间戳和watermark。

> 在将数据流转换为表时，有两种定义时间属性的方法。根据指定的.rowtime字段名是否存在于数据流的架构中，timestamp字段可以：

- 作为新字段追加到schema
- 替换现有字段

在这两种情况下，定义的事件时间戳字段，都将保存DataStream中事件时间戳的值。

代码如下

```
val inputStream: DataStream[String] = env.readTextFile("\\sensor.txt")
val dataStream: DataStream[SensorReading] = inputStream
    .map(data => {
        val dataArray = data.split(",")
        SensorReading(dataArray(0), dataArray(1).toLong, dataArray(2).toDouble)
      })
    .assignAscendingTimestamps(_.timestamp * 1000L)

// 将 DataStream转换为 Table，并指定时间字段
val sensorTable = tableEnv.fromDataStream(dataStream, 'id, 'timestamp.rowtime,
'temperature)
// 或者，直接追加字段
val sensorTable2 = tableEnv.fromDataStream(dataStream, 'id, 'temperature,
'timestamp, 'rt.rowtime)
```

(2)定义Table Schema时指定

这种方法只要在定义Schema的时候，将事件时间字段，并指定成rowtime就可以了。

代码如下

```
tableEnv.connect(
  new FileSystem().path("sensor.txt"))
  .withFormat(new Csv())
  .withSchema(new Schema()
    .field("id", DataTypes.STRING())
    .field("timestamp", DataTypes.BIGINT())
      .rowtime(
        new Rowtime()
          .timestampsFromField("timestamp")    // 从字段中提取时间戳
          .watermarksPeriodicBounded(1000)    // watermark延迟1秒
      )
    .field("temperature", DataTypes.DOUBLE())
  ) // 定义表结构
  .createTemporaryTable("inputTable") // 创建临时表
```

(3)创建表的DDL中指定

事件时间属性，是使用CREATE TABLE DDL中的WARDMARK语句定义的。watermark语句，定义现有事件时间字段上的watermark生成表达式，该表达式将事件时间字段标记为事件时间属性。

代码如下

```
val sinkDDL: String =
"""
|create table dataTable (
|  id varchar(20) not null,
```

```
|  ts bigint,
|  temperature double,
|  rt AS TO_TIMESTAMP( FROM_UNIXTIME(ts) ),
|  watermark for rt as rt - interval '1' second
|) with (
|  'connector.type' = 'filesystem',
|  'connector.path' = 'file:///D:\\..\\sensor.txt',
|  'format.type' = 'csv'
|)
""".stripMargin
tableEnv.sqlUpdate(sinkDDL) // 执行 DDL
```

> 这里**FROM_UNIXTIME**是系统内置的时间函数，用来将一个整数（秒数）转换成“YYYY-MM-DD hh:mm:ss”格式（默认，也可以作为第二个String参数传入）的日期时间字符串（date time string）；然后再用**TO_TIMESTAMP**将其转换成Timestamp。