Flink SQL 如何实现数据流的 Join?

无论在 OLAP 还是 OLTP 领域，Join 都是业务常会涉及到且优化规则比较复杂的 SQL 语句。对于离线计算而言，经过数据库领域多年的积累，Join 语义以及实现已经十分成熟，然而对于近年来刚兴起的 Streaming SQL 来说 Join 却处于刚起步的状态。

其中最为关键的问题在于 Join 的实现依赖于缓存整个数据集，而 Streaming SQL Join 的对象却是无限的数据流，内存压力和计算效率在长期运行来说都是不可避免的问题。下文将结合 SQL 的发展解析 Flink SQL 是如何解决这些问题并实现两个数据流的 Join。

离线 Batch SQL Join 的实现

传统的离线 Batch SQL （面向有界数据集的 SQL）有三种基础的实现方式，分别是 Nested-loop Join、Sort-Merge Join 和 Hash Join。

- Nested-loop Join 最为简单直接，将两个数据集加载到内存，并用内嵌遍历的方式来逐个比较两个数据集内的元素是否符合 Join 条件。Nested-loop Join 虽然时间效率以及空间效率都是最低的，但胜在比较灵活适用范围广，因此其变体 BNL 常被传统数据库用作为 Join 的默认基础选项。
- Sort-Merge Join 顾名思义，分为两个 Sort 和 Merge 阶段。首先将两个数据集进行分别排序，然后对两个有序数据集分别进行遍历和匹配，类似于归并排序的合并。值得注意的是，Sort-Merge 只适用于 Equi-Join（Join 条件均使用等于作为比较算子）。Sort-Merge Join 要求对两个数据集进行排序，成本很高，通常作为输入本就是有序数据集的情况下的优化方案。
- Hash Join 同样分为两个阶段，首先将一个数据集转换为 Hash Table，然后遍历另外一个数据集元素并与 Hash Table 内的元素进行匹配。第一阶段和第一个数据集分别称为 build 阶段和 build table，第二个阶段和第二个数据集分别称为 probe 阶段和 probe table。Hash Join 效率较高但对空间要求较大，通常是作为 Join 其中一个表为适合放入内存的小表的情况下的优化方案。和 Sort-Merge Join 类似，Hash Join 也只适用于 Equi-Join。

实时 Streaming SQL Join

相对于离线的 Join，实时 Streaming SQL（面向无界数据集的 SQL）无法缓存所有数据，因此 Sort-Merge Join 要求的对数据集进行排序基本是无法做到的，而 Nested-loop Join 和 Hash Join 经过一定的改良则可以满足实时 SQL 的要求。

我们通过例子来看基本的 Nested Join 在实时 Streaming SQL 的基础实现（案例及图来自 Piotr Nowojski 在 Flink Forward San Francisco 的分享[2]）。

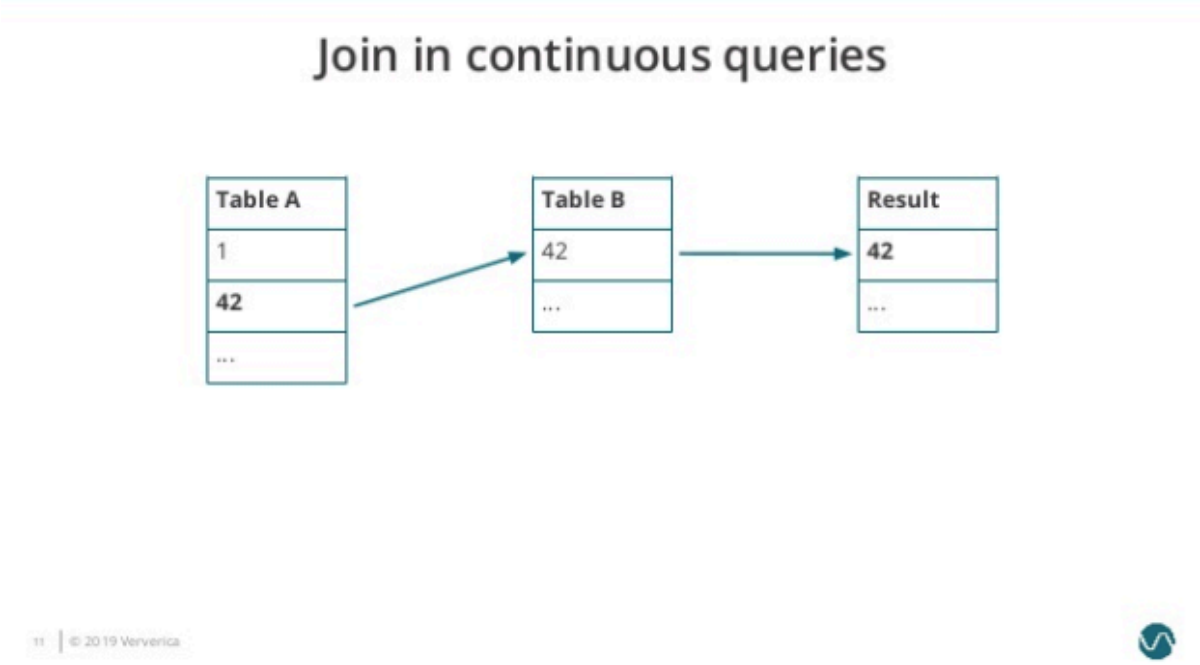

图1. Join-in-continuous-query-1

Table A 有 1、42 两个元素，Table B 有 42 一个元素，所以此时的 Join 结果会输出 42。

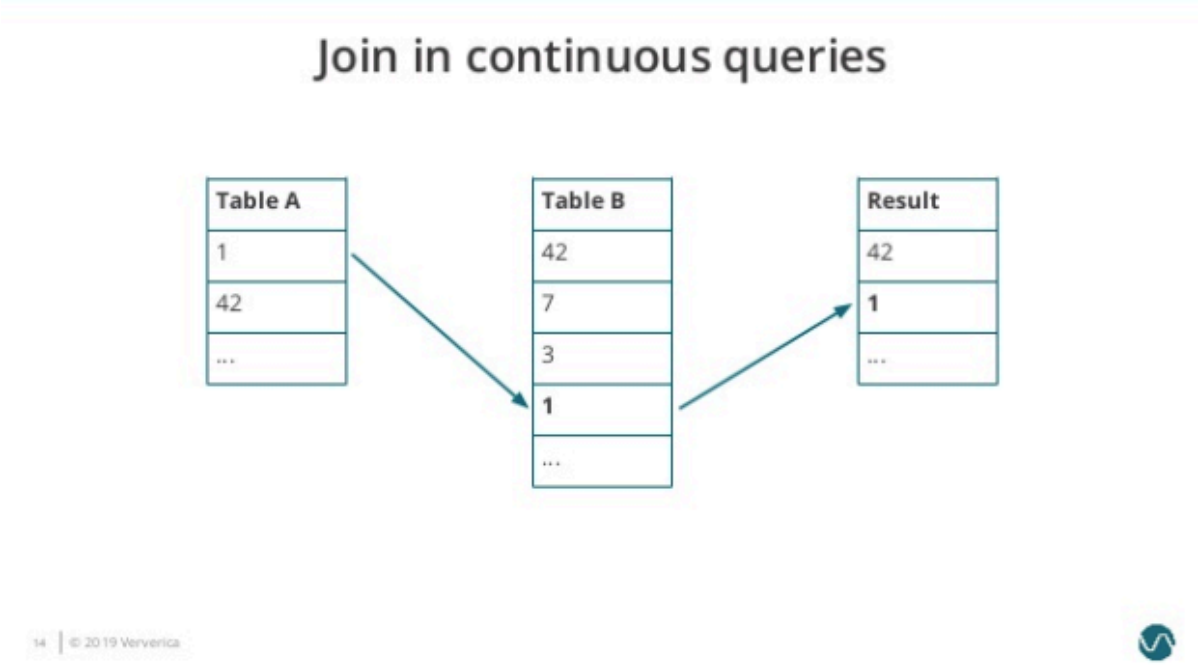

图2. Join-in-continuous-query-2

接着 Table B 依次接受到三个新的元素，分别是 7、3、1。因为 1 匹配到 Table A 的元素，因此结果表再输出一个元素 1。

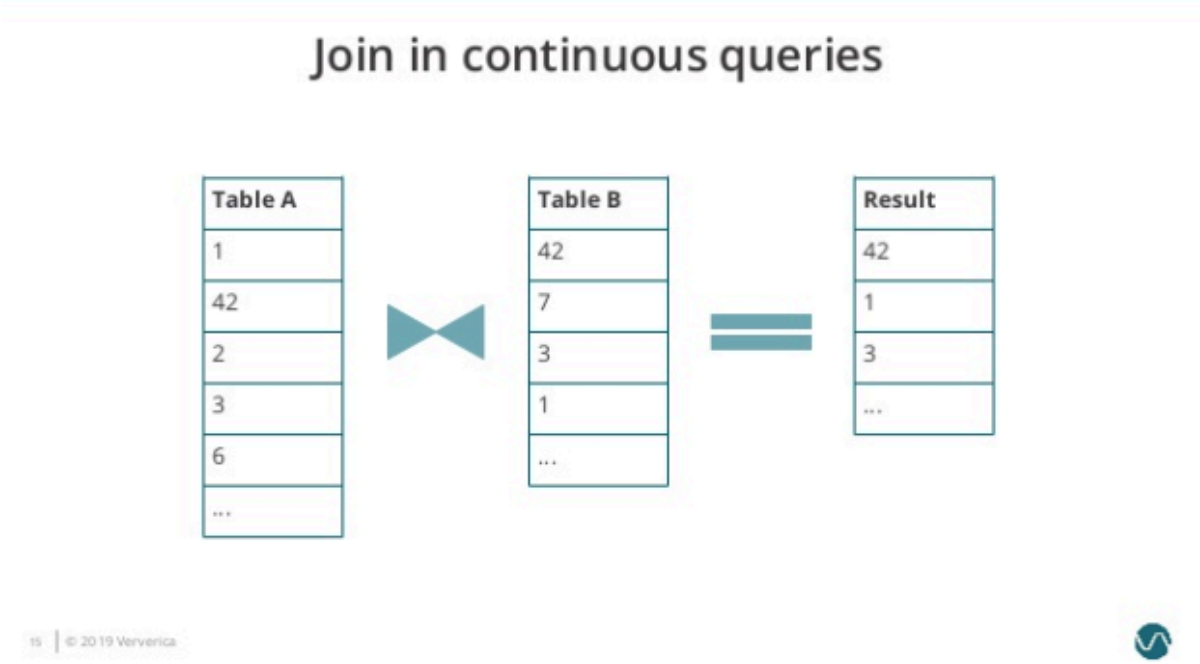

图3. Join-in-continuous-query-3

随后 Table A 出现新的输入 2、3、6，3 匹配到 Table B 的元素，因此再输出 3 到结果表。

可以看到在 Nested-Loop Join 中我们需要保存两个输入表的内容，而随着时间的增长 Table A 和 Table B 需要保存的历史数据无止境地增长，导致很不合理的内存磁盘资源占用，而且单个元素的匹配效率也会越来越低。类似的问题也存在于 Hash Join 中。

那么有没有可能设置一个缓存剔除策略，将不必要的历史数据及时清理呢？答案是肯定的，关键在于缓存剔除策略如何实现，这也是 Flink SQL 提供的三种 Join 的主要区别。

Flink SQL 的 Join

- **Regular Join**

Regular Join 是最为基础的没有缓存剔除策略的 Join。Regular Join 中两个表的输入和更新都会对全局可见，影响之后所有的 Join 结果。举例，在一个如下的 Join 查询里，Orders 表的新纪录会和 Product 表所有历史纪录以及未来的纪录进行匹配。

```
SELECT * FROM Orders
INNER JOIN Product
ON Orders.productId = Product.id
```

因为历史数据不会被清理，所以 Regular Join 允许对输入表进行任意种类的更新操作（insert、update、delete）。然而因为资源问题 Regular Join 通常是不可持续的，一般只用做有界数据流的 Join。

- **Time-Windowed Join**

Time-Windowed Join 利用窗口给两个输入表设定一个 Join 的时间界限，超出时间范围的数据则对 JOIN 不可见并可以被清理掉。值得注意的是，这里涉及到的一个问题是时间的语义，时间可以指计算发生的系统时间（即 Processing Time），也可以指从数据本身的时间字段提取的 Event Time。如果是 Processing Time，Flink 根据系统时间自动划分 Join 的时间窗口并定时清理数据；如果是 Event Time，Flink 分配 Event Time 窗口并依据 Watermark 来清理数据。

以更常用的 Event Time Windowed Join 为例，一个将 Orders 订单表和 Shipments 运输单表依据订单时间和运输时间 Join 的查询如下:

```
SELECT *
FROM
  Orders o,
  Shipments s
WHERE
  o.id = s.orderId AND
  s.shiptime BETWEEN o.ordertime AND o.ordertime + INTERVAL '4' HOUR
```

这个查询会为 Orders 表设置了 o.ordertime > s.shiptime- INTERVAL ‘4’ HOUR 的时间下界（图4）。

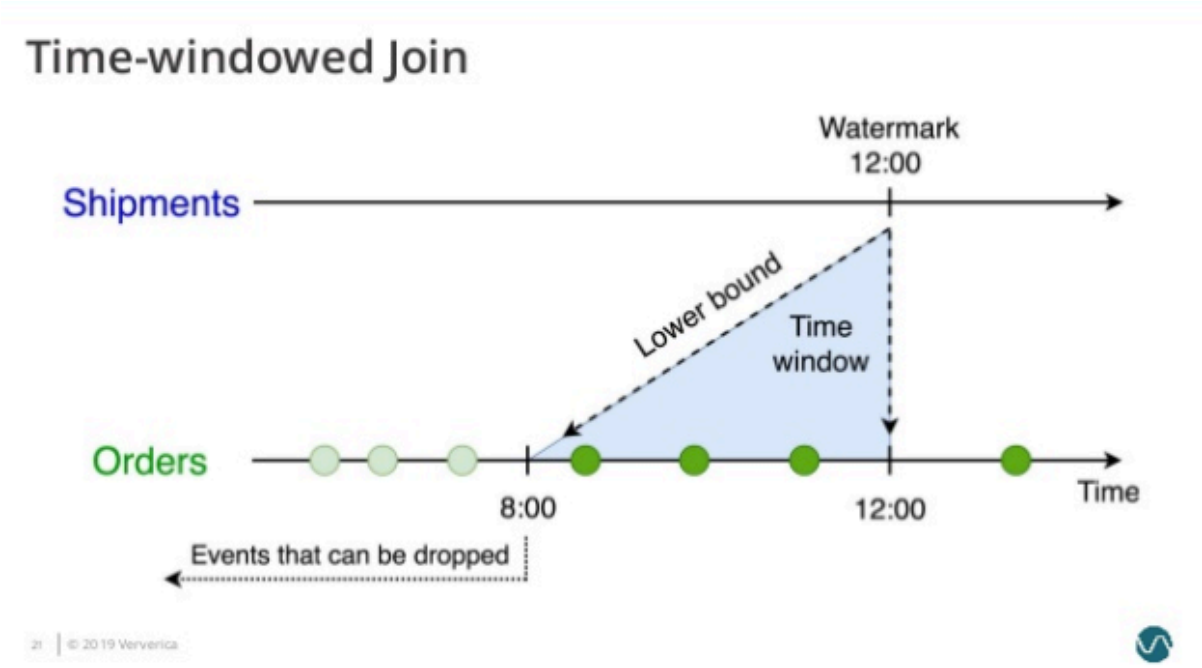

图4. Time-Windowed Join 的时间下界 - Orders 表

并为 Shipmenets 表设置了 s.shiptime >= o.ordertime 的时间下界（图5）。

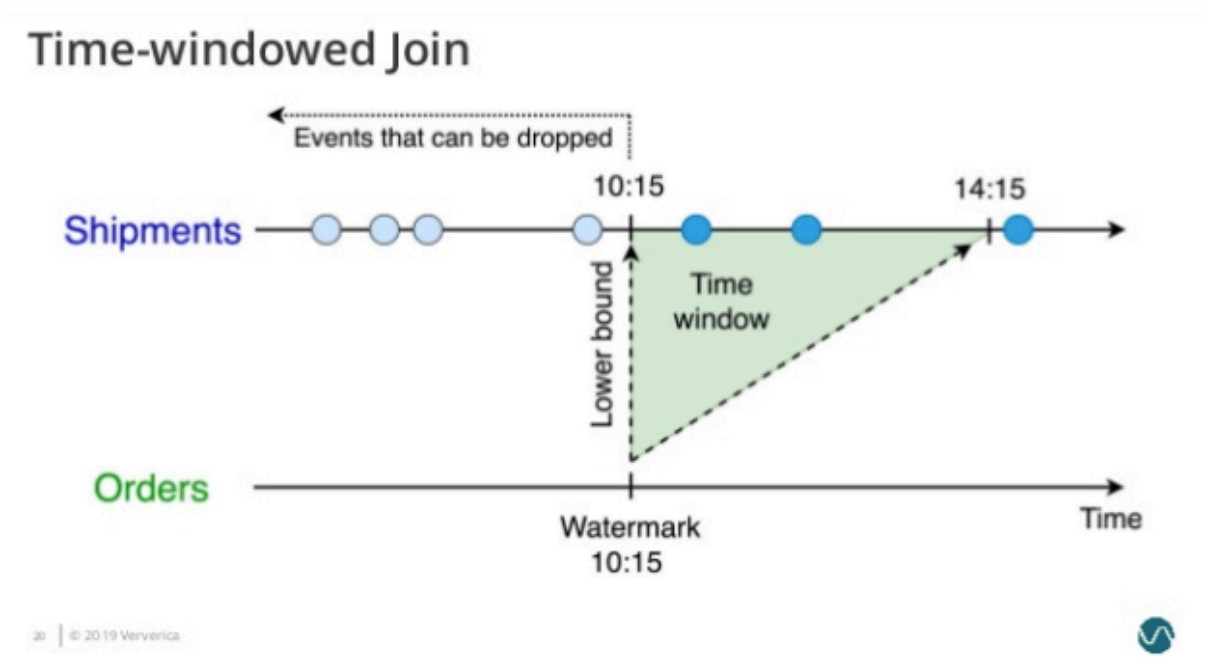

图5. Time-Windowed Join 的时间下界 - Shipment 表

因此两个输入表都只需要缓存在时间下界以上的数据，将空间占用维持在合理的范围。

不过虽然底层实现上没有问题，但如何通过 SQL 语法定义时间仍是难点。尽管在实时计算领域 Event Time、Processing Time、Watermark 这些概念已经成为业界共识，但在 SQL 领域对时间数据类型的支持仍比较弱[4]。因此，定义 Watermark 和时间语义都需要通过编程 API 的方式完成，比如从 DataStream 转换至 Table ，不能单纯靠 SQL 完成。这方面的支持 Flink 社区计划通过拓展 SQL 方言来完成，感兴趣的读者可以通过 FLIP-66[7] 来追踪进度。

- **Temporal Table Join**

虽然 Timed-Windowed Join 解决了资源问题，但也限制了使用场景: Join 两个输入流都必须有时间下界，超过之后则不可访问。这对于很多 Join 维表的业务来说是不适用的，因为很多情况下维表并没有时间界限。针对这个问题，Flink 提供了 Temporal Table Join 来满足用户需求。

Temporal Table Join 类似于 Hash Join，将输入分为 Build Table 和 Probe Table。前者一般是纬度表的 changelog，后者一般是业务数据流，典型情况下后者的数据量应该远大于前者。在 Temporal Table Join 中，Build Table 是一个基于 append-only 数据流的带时间版本的视图，所以又称为 Temporal Table。Temporal Table 要求定义一个主键和用于版本化的字段（通常就是 Event Time 时间字段），以反映记录在不同时间的内容。

比如典型的一个例子是对商业订单金额进行汇率转换。假设有一个 Orders 流记录订单金额，需要和 RatesHistory 汇率流进行 Join。RatesHistory 代表不同货币转为日元的汇率，每当汇率有变化时就会有一条更新记录。两个表在某一时间节点内容如下:

RatesHistory

time	currency	rate
09:00	USD	102
09:00	Euro	114
09:00	Yen	1
10:45	Euro	116
11:15	Euro	119
11:49	USD	99
...	...	...

```
TemporalTableFunction rates =
  ratesHistory
    .createTemporalTableFunction(
        "time",        // <- "versioning" field
        "currency"); // <- primary key

tableEnv.registerFunction("Rates", rates);
```


图6. Temporal Table Join Example]

我们将 RatesHistory 注册为一个名为 Rates 的 Temporal Table，设定主键为 currency，版本字段为 time。

RatesHistory

time	currency	rate
09:00	USD	102
09:00	Euro	114
09:00	Yen	1
10:45	Euro	116
11:15	Euro	119
11:49	USD	99
...	...	...

```
TemporalTableFunction rates =
  ratesHistory
    .createTemporalTableFunction(
        "time",        // <- "versioning" field
        "currency"); // <- primary key

tableEnv.registerFunction("Rates", rates);
```


图7. Temporal Table Registration]

此后给 Rates 指定时间版本，Rates 则会基于 RatesHistory 来计算符合时间版本的汇率转换内容。

图8. Temporal Table Content]

在 Rates 的帮助下，我们可以将业务逻辑用以下的查询来表达:

```
SELECT
  o.amount * r.rate
FROM
  Orders o,
  LATERAL Table(Rates(o.time)) r
WHERE
  o.currency = r.currency
```

值得注意的是，不同于在 Regular Join 和 Time-Windowed Join 中两个表是平等的，任意一个表的新记录都可以与另一表的历史记录进行匹配，在 Temporal Table Join 中，Temoparal Table 的更新对另一表在该时间节点以前的记录是不可见的。这意味着我们只需要保存 Build Side 的记录直到 Watermark 超过记录的版本字段。因为 Probe Side 的输入理论上不会再有早于 Watermark 的记录，这些版本的数据可以安全地被清理掉。

总结

实时领域 Streaming SQL 中的 Join 与离线 Batch SQL 中的 Join 最大不同点在于无法缓存完整数据集，而是要给缓存设定基于时间的清理条件以限制 Join 涉及的数据范围。根据清理策略的不同，Flink SQL 分别提供了 Regular Join、Time-Windowed Join 和 Temporal Table Join 来应对不同业务场景。

另外，尽管在实时计算领域 Join 可以灵活地用底层编程 API 来实现，但在 Streaming SQL 中 Join 的发展仍处于比较初级的阶段，其中关键点在于如何将时间属性合适地融入 SQL 中，这点 ISO SQL 委员会制定的 SQL 标准并没有给出完整的答案。或者从另外一个角度来讲，作为 Streaming SQL 最早的开拓者之一，Flink 社区很适合探索出一套合理的 SQL 语法反过来贡献给 ISO。

参考

- Flux capacitor, huh? Temporal Tables and Joins in Streaming SQL
- How to Join Two Data Streams? - Piotr Nowojski
- Joins in Continuous Queries
- Temporal features in SQL:2011

- Hash join in MySQL 8
- SQL:2011
- FLIP-66: Support Time Attribute in SQL DDL

作者介绍：林小铂，网易游戏高级开发工程师，负责游戏数据中心实时平台的开发及运维工作，目前专注于 Apache Flink 的开发及应用。探究问题本来就是一种乐趣。