Flink SQL 功能解密系列 —— 维表 JOIN 与异步优化

引子

流计算中一个常见的需求就是为数据流补齐字段。因为数据采集端采集到的数据往往比较有限，在做数据分析之前，就要先将所需的维度信息补全。比如采集到的交易日志中只记录了商品 id，但是在做业务时需要根据店铺维度或者行业纬度进行聚合，这就需要先将交易日志与商品维表进行关联，补全所需的维度信息。这里所说的维表与数据仓库中的概念类似，是维度属性的集合，比如商品维，地点维，用户维等等。

在流计算中，这是一个典型的 stream-to-table jon 的问题。本文主要讲解在 Flink SQL 中是如何解决这个问题的，用户如何简单上手使用这个功能。

维表 JOIN 语法

由于维表是一张不断变化的表（静态表只是动态表的一种特例）。那如何 JOIN 一张不断变化的表呢？如果用传统的 JOIN 语法`SELECT * FROM T JOIN dim_table on T.id = dim_table.id`来表达维表 JOIN，是不完整的。因为维表是一直在更新变化的，如果用这个语法那么关联上的是哪个时刻的维表呢？我们是不知道的，结果是不确定的。所以 Flink SQL 的维表 JOIN 语法引入了 SQL:2011 Temporal Table 的标准语法，用来声明关联的是维表哪个时刻的快照。维表 JOIN 语法/示例如下。

假设我们有一个 Orders 订单数据流，希望根据产品 ID 补全流上的产品维度信息，所以需要跟 Products 维度表进行关联。Orders 和 Products 的 DDL 声明语句如下：

```
CREATE TABLE Orders (
  orderId VARCHAR,          -- 订单 id
  productId VARCHAR,        -- 产品 id
  units INT,                -- 购买数量
  orderTime TIMESTAMP       -- 下单时间
) with (
  type = 'tt',              -- tt 日志流
  ...
)

CREATE TABLE Products (
  productId VARCHAR,        -- 产品 id
```

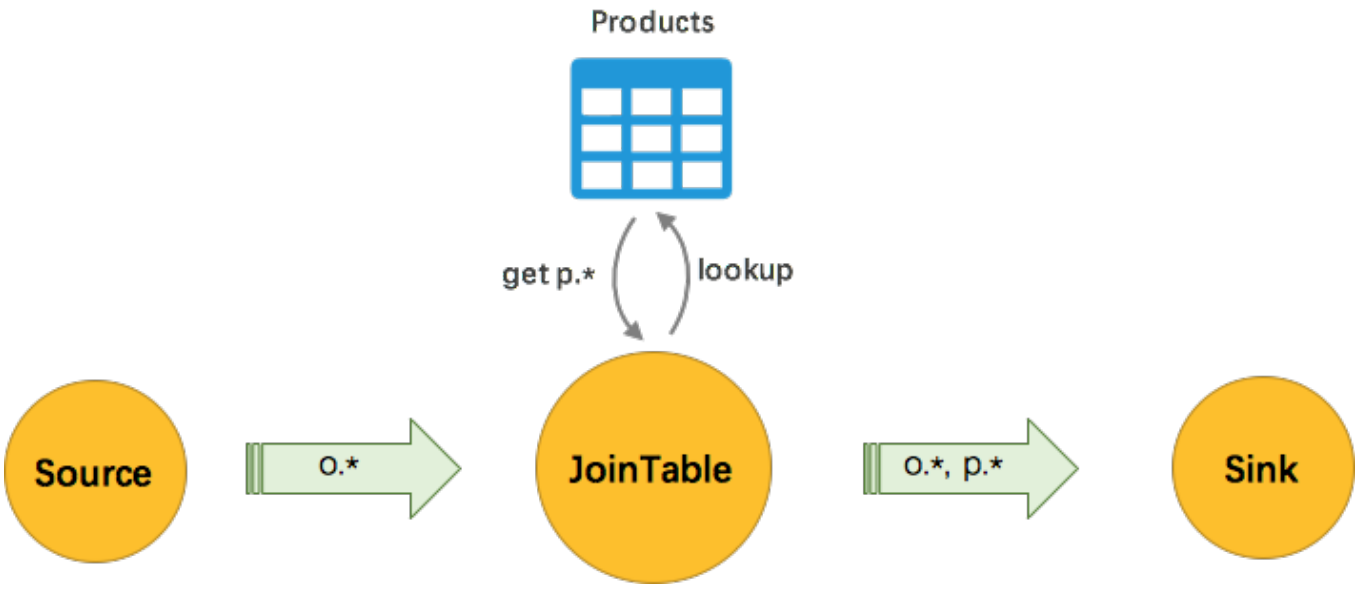
```
  name VARCHAR,               -- 产品名称
  unitPrice DOUBLE            -- 单价
  PERIOD FOR SYSTEM_TIME,     -- 这是一张随系统时间而变化的表，用来声明维表
  PRIMARY KEY (productId)     -- 维表必须声明主键
) with (
  type = 'alihbase',          -- HBase 数据源
  ...
)
```

如上声明了一张来自 TT 的 Orders 订单数据流，和一张存储于 HBase 中的 Products 产品维表。为了补齐订单流的产品信息，需要 JOIN 维表，这里可以分为 JOIN 当前表和 JOIN 历史表。

JOIN 当前维表

```
SELECT *
FROM Orders AS o
[LEFT] JOIN Products FOR SYSTEM_TIME AS OF PROCTIME() AS p
ON o.productId = p.productId
```

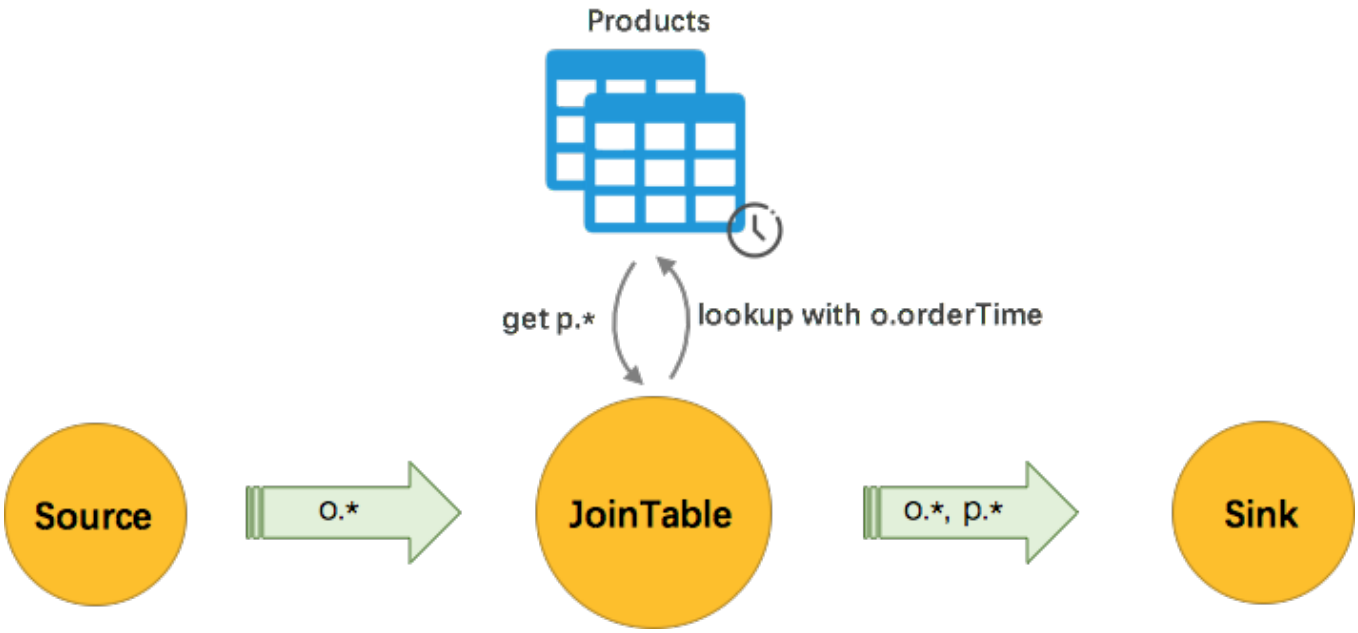
Flink SQL 支持 LEFT JOIN 和 INNER JOIN 的维表关联。如上语法所示的，维表 JOIN 语法与传统的 JOIN 语法并无二异。只是 Products 维表后面需要跟上 `FOR SYSTEM_TIME AS OF PROCTIME()` 的关键字，其含义是每条到达的数据所关联上的是到达时刻的维表快照，也就是说，当数据到达时，我们会根据数据上的 key 去查询远程数据库，拿到匹配的结果后关联输出。这里的 `PROCTIME` 即 processing time。使用 JOIN 当前维表功能需要注意的是，如果维表插入了一条数据能匹配上之前左表的数据时，JOIN的结果流，不会发出更新的数据以弥补之前的未匹配。JOIN行为只发生在处理时间（processing time），即使维表中的数据都被删了，之前JOIN流已经发出的关联上的数据也不会被撤回或改变。



JOIN 历史维表

```
SELECT *
FROM Orders AS o
[LEFT] JOIN Products FOR SYSTEM_TIME AS OF o.orderTime AS p
ON o.productId = p.productId
```

有时候想关联上的维度数据，并不是当前时刻的值，而是某个历史时刻的值。比如，产品的价格一直在发生变化，订单流希望补全的是下单时的价格，而不是当前的价格，那就是 JOIN 历史维表。语法上只需要将上文的 `PROCTIME()` 改成 `o.orderTime` 即可。含义是关联上的是下单时刻的 Products 维表。



Flink 在获取维度数据时，会根据左流的时间去查对应时刻的快照数据。因此 JOIN 历史维表需要外部存储支持多版本存储，如 HBase，或者存储的数据中带有多版本信息。

注：JOIN 历史维表功能目前暂未开放

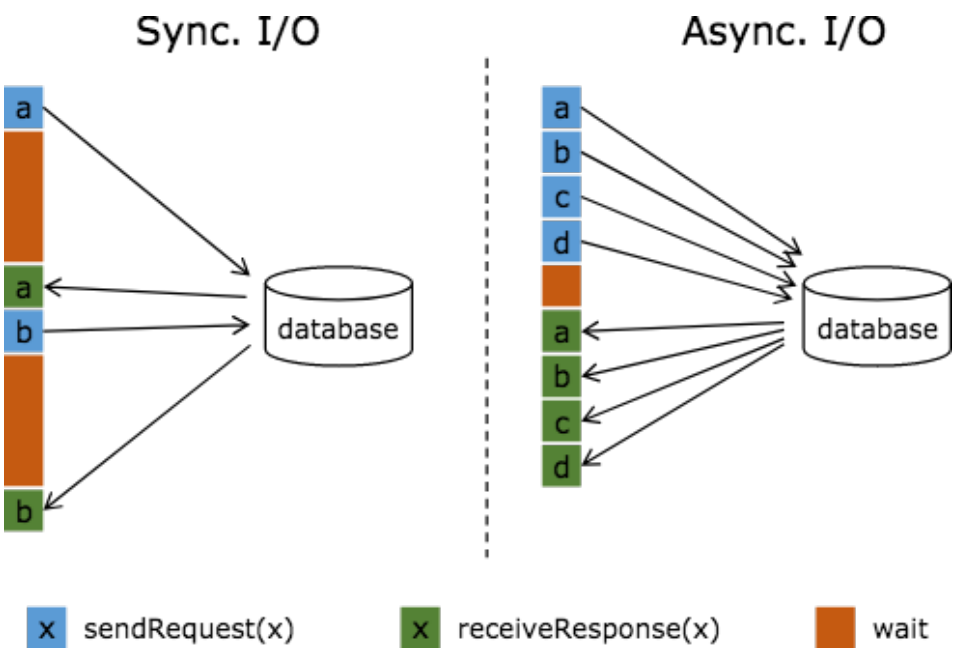
维表优化

在实际使用的过程中，会遇到许多性能问题。为了解决这些性能问题，我们做了大量的优化，性能得到大幅提升，在双11的洪峰下表现特别淡定。

我们的优化主要是为了解决两方面的问题：
1. 提高吞吐。维表的IO请求严重阻塞了数据流的计算处理。
2. 降低维表数据库读压力。如 HBase 也只能承受单机每秒 20 万的读请求。

Async IO

我在 《Flink 原理与实现：Aysnc I/O》 中介绍了 Async IO 的使用场景和实现原理。原始的维表 JOIN 是同步访问的方式，来一条数据，去数据库查询一次，等待返回后输出关联结果。可以发现网络等待时间极大地阻碍了吞吐和延迟。为了解决同步访问的问题，异步模式可以并发地处理多个请求和回复，从而连续的请求之间不需要阻塞等待。



缓存

数据库的维表查询请求，有大量相同 key 的重复请求。如何减少重复请求？本地缓存是常用的方案。Flink SQL 目前提供两种缓存方案：LRU 和 ALL。（详见文档）

LRU

通过 `cache='LRU'`参数可以开启 LRU 缓存优化，Blink 会为每个 JoinTable 节点创建一个 LRU 本地缓存。当每个数据进来的时候，先去缓存中查询，如果存在则直接关联输出，减少了一次 IO 请求。如果不存在，再发起数据库查询请求（异步或同步方式），请求返回的结果会先存入缓存中以备下次查询。

为了防止缓存无限制增长，所以使用的是 LRU 缓存，并且可以通过 `cacheSize` 调整缓存的大小。为了定期更新维表数据，可以通过 `cacheTTLMs` 调整缓存的失效时间。`cacheTTLMs` 是作用于每条缓存数据上的，也就是某条缓存数据在指定 timeout 时间内没有被访问，则会从缓存中移除。

ALL

Async 和 LRU-Cache 能极大提高吞吐率并降低数据库的读压力，但是仍然会有大量的 IO 请求存在，尤其是当 miss key（维表中不存在的 key）很多的时候。如果维表数据不大（通常百万级以内），那么其实可以将整个维表缓存到本地。那么 miss key 永远不会去请求数据库。因为本地缓存就是维表的镜像，缓存中不存在那么远程数据库中也不存在。

ALL cache 可以通过 `cache='ALL'`参数开启，通过`cacheTTLMs`控制缓存的刷新间隔。Flink SQL 会为 JoinTable 节点起一个异步线程去同步缓存。在 Job 刚启动时，会先阻塞主数据流，直到缓存数据加载完毕，保证主数据流流过时缓存就已经 ready。在之后的更新缓存的过程中，不会阻塞主数据流。因为异步更新线程会将维表数据加载到临时缓存中，加载完毕后再与主缓存做原子替换。只有替换操作是加了锁的。

因为几乎没有 IO 操作，所以使用 cache ALL 的维表 JOIN 性能可以非常高。但是由于内存需要能同时容纳下两份维表拷贝，因此需要加大内存的配置。

缓存未命中 key

在使用 LRU 缓存时，如果存在大量的 invalid key ，或者数据库中不存在的 key。由于命中不了缓存，导致缓存的收益较低，仍然会有大量请求打到数据库。因此我们将未命中的 key 也加进了缓存，提高了未命中 key 和 invalid key 情况下的缓存命中率。

Distribute By 提高缓存命中率

默认 JoinTable 节点与上游节点之间的数据是通过 shuffle 传输的。这在缓存大小有限、key总量大、热点不明显的情况下, 缓存的收益可能较低。这种情况下可以将上游节点与 JoinTable 节点的数据传输改成按 key 分区。这样通常可以缩小单个节点的 key 个数，提高缓存的命中率。比如一段时间内 JoinTable 节点总共需要处理100万个key, 节点并发100, 在数据不倾斜时单节点平均只需处理1万个key = 100万/100并发. 如果不做 key 分区, 单节点实际处理的key个数可能远大于1万。使用上也非常简单，在维表的 DDL 参数中加上`partitionedJoin='true'`即可。

最佳实践

在使用维表 JOIN 时，如果维表数据不大，或者 miss key （维表中不存在的 key）非常多，则可以使用 ALL cache，但是可能需要适当调大节点的内存，因为内存需要能同时容纳下两份维表拷贝。如果用不了 ALL cache，则可以使用 Async + LRU 来提高节点的吞吐。

未来工作

1. 使用 SideInput 减少对数据库的全量读取
2. 引入 Partitioned-ALL-cache 支持超大维表
3. 使用批量请求提高单次 IO 的吞吐
4. Multi-Join 优化

流计算中维表 JOIN 是一个非常常见的需求，遇到的挑战也非常多。比如超大维表问题，ALL cache 无法装下整个维表。未来我们打算引入 Partitioned-ALL-cache，也就是上游数据到 JoinTable 节点根据 JOIN key 分区，那么每个节点只需要加载属于该分区key的缓存数据，从而做到了缓存的水平扩展。从而遇到超大维表时可以通过扩并发也能够全量缓存下维表数据。另外，ALL cache 现在每个节点是都会起一个线程去加载全量维表数据，如果有1000个节点，则会全量读数据库1000次。未来打算通过 Side Input功能做到只需要全量读取一次，维表数据会自动分发到各个节点。

另外，Async 极大地提高了吞吐，但是每一次 IO 请求只取了单 key 的数据，效率比较低。未来计划使用 Batch Get 来提高每次 IO 请求的吞吐。

目前在 Table API 上已经支持了 Multi-Join 的优化，能极大提高多维表连续 JOIN 时的性能，减少

网络数据的传输开销。未来会在 SQL 上也支持 Multi-Join 的优化。