基于 Flink 的典型 ETL 场景实现

> **简介：**本文将从数仓诞生的背景、数仓架构、离线与实时数仓的对比着手，综述数仓发展演进，然后分享基于 Flink 实现典型 ETL 场景的几个方案。

1.实时数仓的相关概述

1.1 实时数仓产生背景

我们先来回顾一下数据仓库的概念。

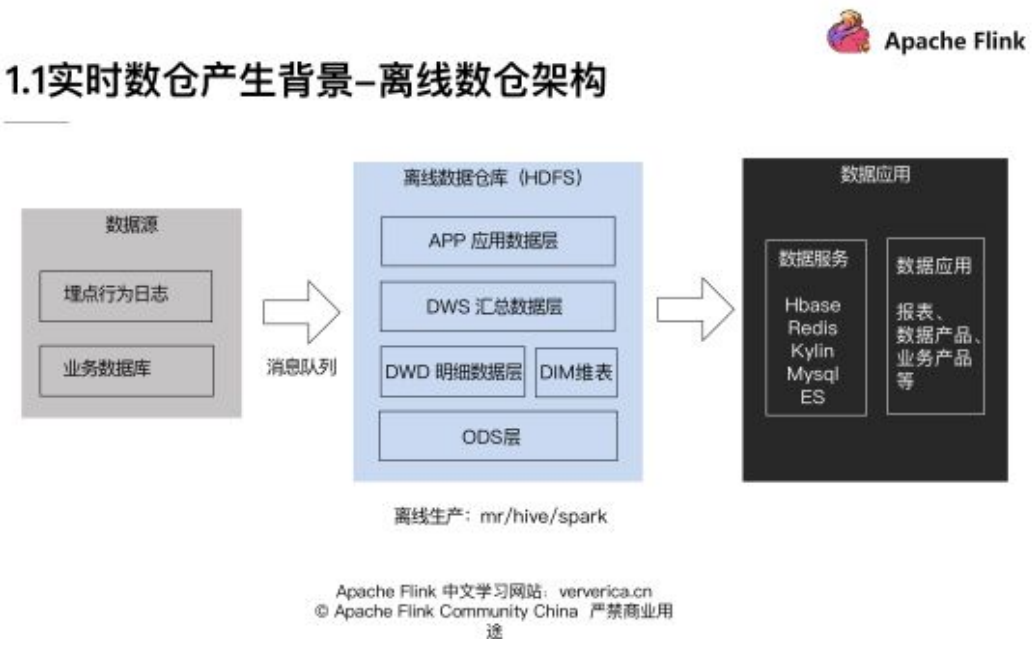
数据仓库的概念是于90年代由 Bill Inmon 提出，当时的背景是传统的 OLTP 数据库无法很好的支持长周期分析决策场景，所以数据仓库概念的4个核心点，我们要结合着 OLTP 数据库当时的状态来对比理解。

面向主题的：数据仓库的数据组织方式与 OLTP 面向事务处理不同。因为数据仓库是面向分析决策的，所以数据经常按分析场景或者是分析对象等主题形式来组织。
集成的：对于数据仓库来说，经常需要去集合多个分散的、异构的数据源，做一些数据清洗等 ETL 处理，整合成一块数据仓库，OLTP 则不需要做类似的集成操作。
相对稳定的：OLTP 数据库一般都是面向业务的，它主要的作用是把当前的业务状态精准的反映出来，所以 OLTP 数据库需要支持大量的增、删、改的操作。但是对于数据仓库来说，只要是入仓存下来的数据，一般使用场景都是查询，因此数据是相对稳定的。

反映历史变化：数据仓库是反映历史变化的数据集合，可以理解成它会将历史的一些数据的快照存下来。而对于 OLTP 数据库来说，只要反映当时的最新的状态就可以了。

以上这4个点是数据仓库的一个核心的定义。我们也可以看出对于实时数据仓库来说，传统数据仓库也就是离线数据仓库中的一些定义会被弱化，比如说在反映历史变化这一点。介绍完数据仓库的基本概念，简单说下数据仓库建模这块会用到一些经典的建模方法，主要有范式建模、维度建模和 Data Vault。在互联网大数据场景下，用的最多的是维度建模方法。

然后先看一下离线数仓的经典架构。如下图：



这个数仓架构主要是偏向互联网大数据的场景方案，由上图可以看出有三个核心环节。

第一个环节是数据源部分，一般互联网公司的数据源主要有两类：

第1类是通过在客户端埋点上报，收集用户的行为日志，以及一些后端日志的日志类型数据源。对于埋点行为日志来说，一般会经过一个这样的流程，首先数据会上报到 Nginx 然后经过 Flume 收集，然后存储到 Kafka 这样的消息队列，然后再由实时或者离线的一些拉取的任务，拉取到我们的离线数据仓库 HDFS。
第2类数据源是业务数据库，而对于业务数据库的话，一般会经过 Canal 收集它的 binlog，然后也是收集到消息队列中，最终再由 Camus 拉取到 HDFS。

这两部分数据源最终都会落地到 HDFS 中的 ODS 层，也叫贴源数据层，这层数据和原始数据源是保持一致的。

第二个环节是离线数据仓库，是图中蓝色的框展示的部分。可以看到它是一个分层的结构，其中的模型设计是依据维度建模思路。

最底层是 ODS 层，这一层将数据保持无信息损失的存放在 HDFS，基本保持原始的日志数据不

变。
在 ODS 层之上，一般会进行统一的数据清洗、归一，就得到了 DWD 明细数据层。这一层也包含统一的维度数据。
然后基于 DWD 明细数据层，我们会按照一些分析场景、分析实体等去组织我们的数据，组织成一些分主题的汇总数据层 DWS。
在 DWS 之上，我们会面向应用场景去做一些更贴近应用的 APP 应用数据层，这些数据应该是高度汇总的，并且能够直接导入到我们的应用服务去使用。

在中间的离线数据仓库的生产环节，一般都是采用一些离线生产的架构引擎，比如说 MapReduce、Hive、Spark 等等，数据一般是存在 HDFS 上。

经过前两个环节后，我们的一些应用层的数据会存储到数据服务里，比如说 HBase 、Redis、Kylin 这样的一些 KV 的存储。并且会针对存在这些数据存储上的一些数据，封装对应的服务接口，对外提供服务。在最外层我们会去产出一些面向业务的报表、面向分析的数据产品，以及会支持线上的一些业务产品等等。这一层的话，称之为更贴近业务端的数据应用部分。

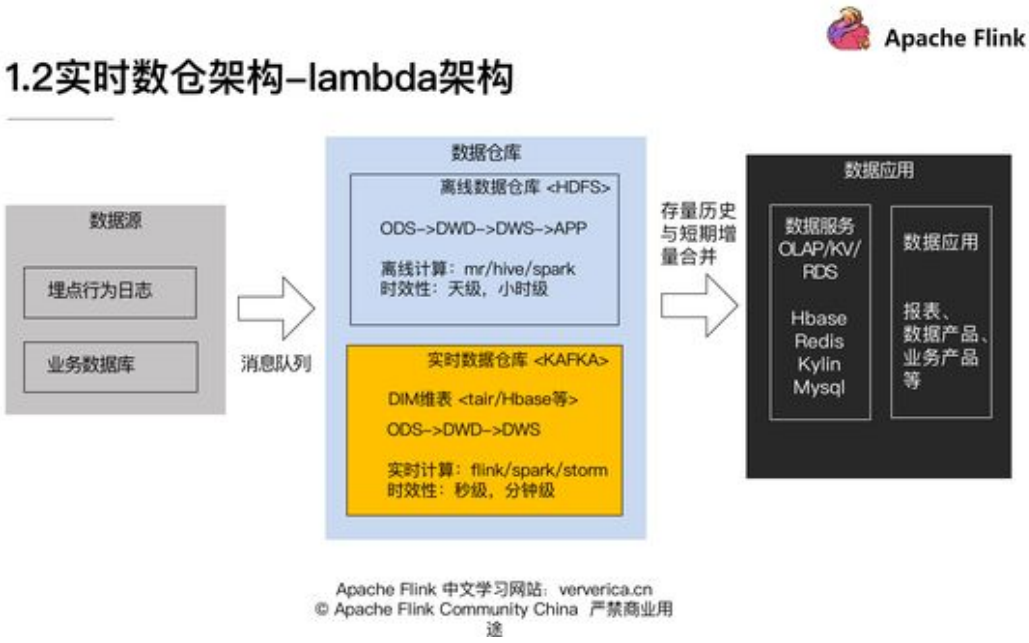
以上是一个基本的离线数仓经典架构的介绍。

大家都了解到现在随着移动设备的普及，我们逐渐的由制造业时代过渡到了互联网时代。在制造业的时代，传统的数仓，主要是为了去支持以前的一些传统行业的企业的业务决策者、管理者，去做一些业务决策。那个时代的业务决策周期是比较长的，同时当时的数据量较小，Oracle、DB2 这一类数据库就已经足够存了。

但随着分布式计算技术的发展、智能化技术发展、以及整体算力的提升、互联网的发展等等因素，我们现在在互联网上收集的数据量，已经呈指数级的增长。并且业务不再只依赖人做决策，做决策的主体很大部分已转变为计算机算法，比如一些智能推荐场景等等。所以这个时候决策的周期，就由原来的天级要求提升到秒级，决策时间是非常短的。在场景上的话，也会面对更多的需要实时数据处理的场景，例如实时的个性化推荐、广告的场景、甚至一些传统企业已经开始实时监控加工的产品是否有质量问题，以及金融行业重度依赖的反作弊等等。因此在这样的一个背景下，实时数仓就必须被提出来了。

1.2 实时数仓架构

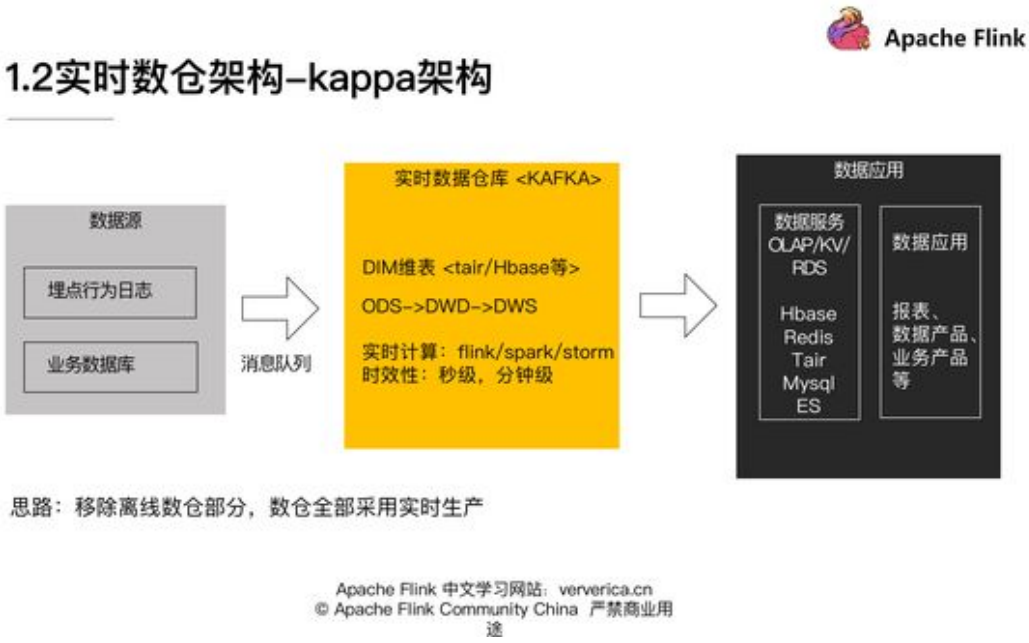
首先跟大家介绍一下实时数仓经典架构 - Lambda 架构：



这个架构是 Storm 的作者提出来的，其实 Lambda 架构的主要思路是在原来离线数仓架构的基础上叠加上实时数仓的部分，然后将离线的存量数据与我们 t+0 的实时的数据做一个 merge，就可以产生数据状态实时更新的结果。

和上述1.1离线数据仓库架构图比较可以明显的看到，实时数仓增加的部分是上图黄色的这块区域。我们一般会把实时数仓数据放在 Kafka 这样的消息队列上，也会有维度建模的一些分层，但是在汇总数据的部分，我们不会将 APP 层的一些数据放在实时数仓，而是更多的会移到数据服务侧去做一些计算。
然后在实时计算的部分，我们经常会使用 Flink、Spark-streaming 和 Storm 这样的计算引擎，时效性上，由原来的天级、小时级可以提升到秒级、分钟级。

大家也可以看到这个架构图中，中间数据仓库环节有两个部分，一个是离线的数据仓库，一个是实时的数据仓库。我们必须要运维两套（实时计算和离线计算）引擎，并且在代码层面，我们也需要去实现实时和离线的业务代码。然后在合并的时候，我们需要保证实施和离线的数据一致性，所以但凡我们的代码做变更，我们也需要去做大量的这种实时离线数据的对比和校验。其实这对于不管是资源还是运维成本来说都是比较高的。这是 Lamda 架构上比较明显和突出的一个问题。因此就产生了 Kappa 结构。

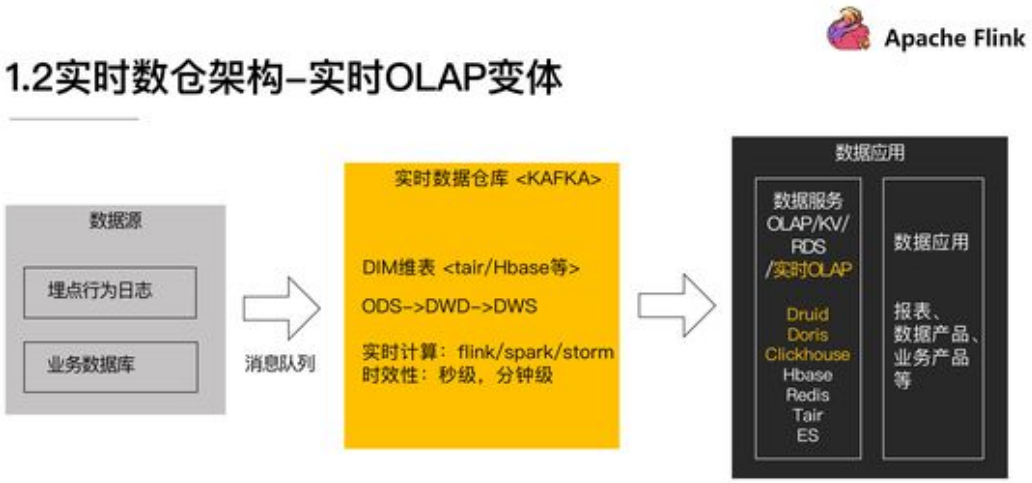


Kappa 架构的一个主要的思路就是在数仓部分移除了离线数仓，数仓的生产全部采用实时数仓。从上图可以看到刚才中间的部分，离线数仓模块已经没有了。

关于 Kappa 架构，熟悉实时数仓生产的同学，可能会有一个疑问。因为我们经常会面临业务变更，所以很多业务逻辑是需要去迭代的。之前产出的一些数据，如果口径变更了，就需要重算，甚至重刷历史数据。对于实时数仓来说，怎么去解决数据重算问题?

Kappa 架构在这一块的思路是：首先要准备好一个能够存储历史数据的消息队列，比如 Kafka，并且这个消息对列是可以支持你从某个历史的节点重新开始消费的。 接着需要新起一个任务，从原来比较早的一个时间节点去消费 Kafka 上的数据，然后当这个新的任务运行的进度已经能够和现在的正在跑的任务齐平的时候，你就可以把现在任务的下游切换到新的任务上面，旧的任务就可以停掉，并且原来产出的结果表也可以被删掉。

随着我们现在实时 OLAP 技术的一些提升，有一个新的实时架构被提了出来，这里暂且称为实时 OLAP 变体。



这个思路是把大量的聚合、分析、计算由实时 OLAP 引擎来承担。在实时数仓计算的部分，我们不需要做的特别重，尤其是聚合相关的一些逻辑，然后这样就可以保障我们在数据应用层能灵活的面对各种业务分析的需求变更，整个架构更加灵活。

最后我们来整体对比一下，实时数仓的这几种架构：

这是整体三个关于实时数仓架构的一个对比：

从计算引擎角度：Lamda 架构它需要去维护批流两套计算引擎，Kappa 架构和实时 OLAP 变体只需要维护流计算引擎就好了。
开发成本：对 Lamda 架构来说，因为它需要维护实时离线两套代码，所以它的开发成本会高一些。 Kappa 架构和实时 OLAP 变体只用维护一套代码就可以了。

分析灵活性：实时 OLAP 变体是相对最灵活的。
在实时 OLAP 引擎依赖上：实时 OLAP 变体是强依赖实时 OLAP 变体引擎的能力的，前两者则不强依赖。
计算资源：Lamda 架构需要批流两套计算资源，Kappa 架构只需要流计算资源，实时 OLAP 变体需要额外的 OLAP 资源。
逻辑变更重算：Lamda 架构是通过批处理来重算的，Kappa 架构需要按照前面介绍的方式去重新消费消息队列重算，实时 OLAP 变体也需要重新消费消息队列，并且这个数据还要重新导入到 OLAP 引擎里，去做计算。

1.3 传统数仓 vs 实时数仓

然后我们来看一下传统数仓和实时数仓整体的差异。

首先从**时效性**来看：离线数仓是支持小时级和天级的，实时数仓到秒级分钟级，所以实时数仓时效性是非常高的。
在**数据存储方式**来看：离线数仓它需要存在HDFS和RDS上面，实时数仓一般是存在消息队列，还有一些kv存储，像维度数据的话会更多的存在kv存储上。
在**生产加工过程**方面，离线数仓需要依赖离线计算引擎以及离线的调度。 但对于实时数仓来说，主要是依赖实时计算引擎。

2.基于 Flink 实现典型的 ETL 场景

这里我们主要介绍两大实时 ETL 场景：维表 join 和双流 join。

维表 join
预加载维表

热存储关联
广播维表
Temporal table function join

双流 join
离线join vs. 实时join
Regular join
Interval join
Window join

2.1 维表 join

2.1.1 预加载维表

方案1：

将维表全量预加载到内存里去做关联，具体的实现方式就是我们定义一个类，去实现 RichFlatMapFunction，然后在 open 函数中读取维度数据库，再将数据全量的加载到内存，然后在 probe 流上使用算子 ，运行时与内存维度数据做关联。

这个方案的优点就是实现起来比较简单，缺点也比较明显，因为我们要把每个维度数据都加载到内存里面，所以它只支持少量的维度数据。同时如果我们要去更新维表的话，还需要重启作业，所以它在维度数据的更新方面代价是有点高的，而且会造成一段时间的延迟。对于预加载维表来说，它适用的场景就是小维表，变更频率诉求不是很高，且对于变更的及时性的要求也比较低的这种场景。

接下来我们看一个简单的代码的示例：

在这段代码截取的是关键的一个片段。这里定义了一个 DimFlatMapFunction 来实现 RichFlatMapFunction。其中有一个 Map 类型的 dim，其实就是为了之后在读取 DB 的维度数据以后，可以用于存放我们的维度数据，然后在 open 函数里面我们需要去连接我们的 DB，进而获取 DB 里的数据。然后在下面代码可以看到我们的场景是从一个商品表里面去取出商品的 ID、商品的名字。然后我们在获取到 DB 里面的维度数据以后会把它存放到 dim 里面。

接下来在 flatMap 函数里面我们就会使用到 dim，我们在获取了 probe 流的数据以后，我们会去 dim 里面比较。是否含有同样的商品 ID 的数据，如果有的话就把相关的商品名称 append 到数据元组，然后做一个输出。这就是一个基本的流程。

其实这是一个基本最初版的方案实现。但这个方案也有一个改进的方式，就是在 open 函数里面，可以新建一个线程，定时的去加载维表。这样就不需要人工的去重启 job 来让维度数据做更新，可以实现一个周期性的维度数据的更新。

方案2：

通过 Distributed cash 的机制去分发本地的维度文件到task manager后再加载到内存做关联。实现方式可以分为三步：

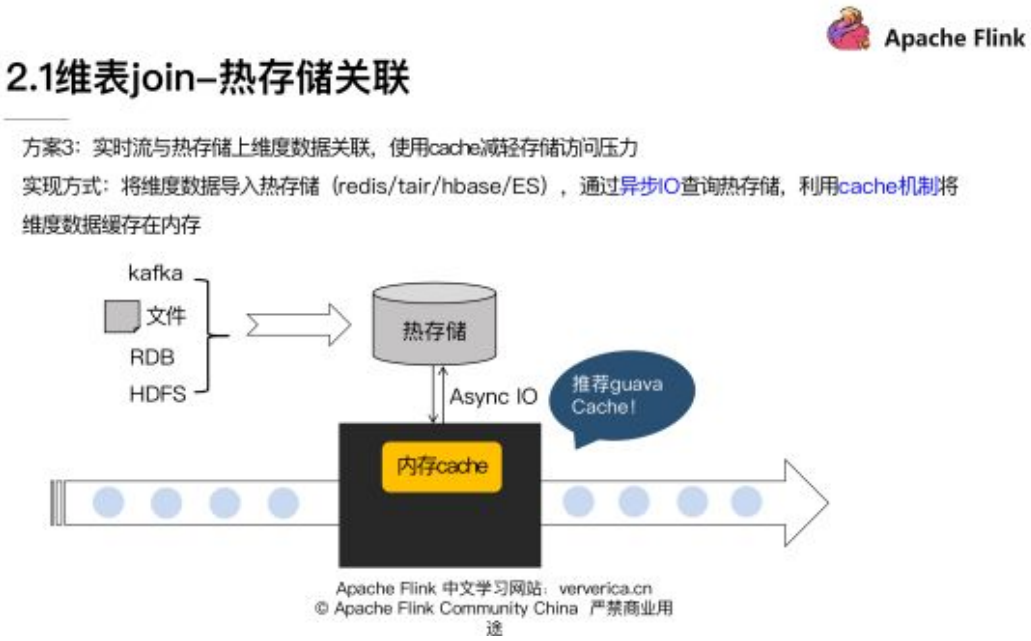
第1步是通过 env.registerCached 注册文件。
第2步是实现 RichFunction，在 open 函数里面通过 RuntimeContext 来获取 cache 文件。
第3步是解析和使用这部分文件数据。

这种方式的一个优点是你不需要去准备或者依赖外部数据库，缺点就是因为数据也是要加载到内存中，所以支持的维表数据量也是比较小。而且如果这个维度数据需要做更新，也需要重启作业。 因此在正规的生产过程中不太建议使用这个方案，因为其实从数仓角度，希望所有的数据都能够通过 schema 化方式来管理。把数据放在文件里面去做这样一个操作，不利于我们做整体数

据的管理和规范化。所以这个方式的话，大家在做一些小的 demo 的时候，或者一些测试的时候可以去使用。

那么它适用的场景就是维度数据是文件形式的、数据量比较小、并且更新的频率也比较低的一些场景，比如说我们读一个静态的码表、配置文件等等。

2.1.2 热存储关联



维表 join 里第二类大的实现思路是热存储关联。具体是我们把维度数据导入到像 Redis、Tair、HBase 这样的一些热存储中，然后通过异步 IO 去查询，并且叠加使用 Cache 机制，还可以加一些淘汰的机制，最后将维度数据缓存在内存里，来减轻整体对热存储的访问压力。

如上图展示的这样的一个流程。在 Cache 这块的话，比较推荐谷歌的 Guava Cache，它封装了一些关于 Cache 的一些异步的交互，还有 Cache 淘汰的一些机制，用起来是比较方便的。

刚才的实验方案里面有两个重要点，一个就是我们需要用异步 IO 方式去访问存储，这里也跟大家一起再回顾一下同步 IO 与异步 IO 的区别：

对于同步 IO 来说，发出一个请求以后，必须等待请求返回以后才能继续去发新的 request。所以整体吞吐是比较小的。由于实时数据处理对于延迟特别关注，这种同步 IO 的方式，在很多场景是不太能够接受的。
异步 IO 就是可以并行发出多个请求，整个吞吐是比较高的，延迟会相对低很多。如果使用异步 IO 的话，它对于外部存储的吞吐量上升以后，会使得外部存储有比较大的压力，有时也会成为我们整个数据处理上延迟的瓶颈。所以引入 Cache 机制是希望通过 Cache 来去减少我们对外部存储的访问量。

刚才提到的 Cuava Cache，它的使用是非常简单的，下图是一个定义 Cache 样例：

可以看到它的使用接口非常简单，大家可以去尝试使用。对于热存储关联方案来说，它的优点就是维度数据因为不用全量加载在内存里，所以就不受限于内存大小，维度数据量可以更多。在美团点评的流量场景里面，我们的维度数据可以支持到 10 亿量级。另一方面该方案的缺点也是比较明显的，我们需要依赖热存储资源，而且维度的更新反馈到结果是有一定延迟的。 因为我们首先需要把数据导入到热存储，然后同时在 Cache 过期的时间上也会有损失。

总体来说这个方法适用的场景是维度数据量比较大，又能够接受维度更新有一定延迟的情况。

2.1.3 广播维表

第三个大的思路是广播维表，主要是利用 broadcast State 将维度数据流广播到下游 task 做 join。

实现方式：

将维度数据发送到 Kafka 作为广播原始流 S1
定义状态描述符 MapStateDescriptor。调用 S1.broadcast()，获得 broadCastStream S2
调用非广播流 S3.connect(S2),得到 BroadcastConnectedStream S4
在 KeyedBroadcastProcessFunction/BroadcastProcessFunction 实现关联处理逻辑，并作为参数调用 S4.process()

这个方案，它的优点是维度的变更可以及时的更新到结果。然后缺点就是数据还是需要保存在内存中，因为它是存在 state 里的，所以支持维表数据量仍然不是很大。适用的场景就是我们需要时时的去感知维度的变更，且维度数据又可以转化为实时流。

下面是一个小的 demo：

我们这里面用到的广播流 pageStream，它其实是定义了一个页面 ID 和页面的名称。对于非广播流 probeStream，它是一个 json 格式的 string，里面包含了设备 ID、页面的 ID、还有时间戳，我们可以理解成用户在设备上做 PV 访问的行为记录。

整个实现来看，就是遵循上述4个步骤：

第1步骤是要定义广播的状态描述符。
第2步骤我们这里去生成 broadCastStream。
第3步骤的话我们就需要去把两个 stream 做 connect。
第4步最主要的一个环节就是需要实现 BroadcastProcessFunction。第1个参数是我们的 probeStream，第2个参数是广播流的数据，第3个参数就是我们的要输出的数据，可以看到主要的数据处理逻辑是在processElement里面。

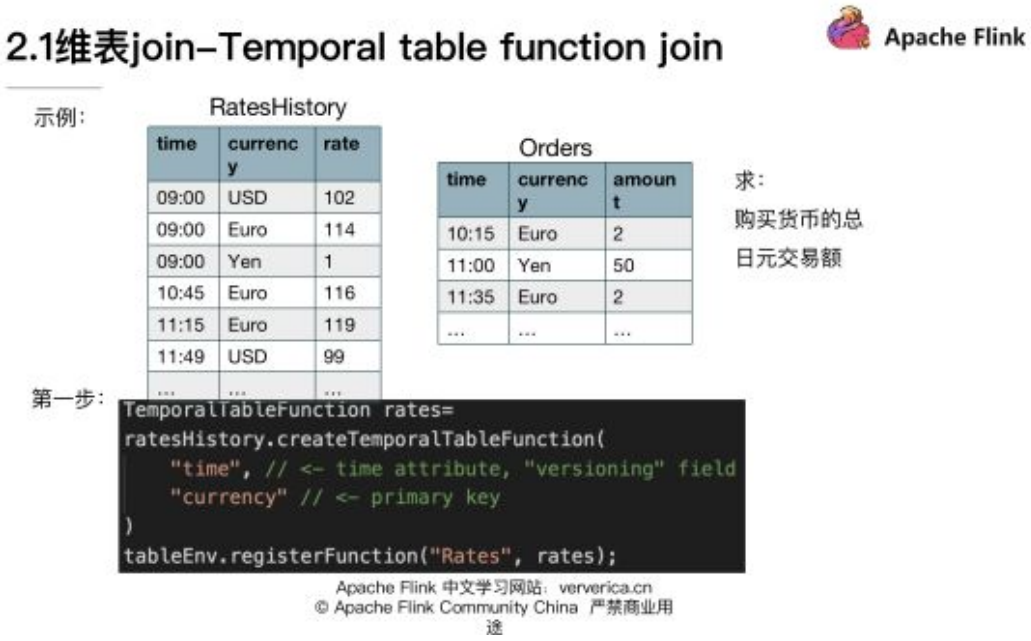
在数据处理过程中，我们首先通过 context 来获取我们的 broadcastStateDesc，然后解析 probe 流的数据，最终获取到对应的一个 pageid。接着就在我们刚才拿到了 state 里面去查询是否有同样的 pageid，如果能够找到对应的 pageid 话，就把对应的 pagename 添加到我们整个 json stream 去做输出。

2.1.4 Temporal table function join

介绍完了上面的方法以后，还有一种比较重要的方法是用 Temporal table function join。首先说明一下什么是 Temporal table？它其实是一个概念：就是能够返回持续变化表的某一时刻数据内容的视图，持续变化表也就是 changingtable，可以是一个实时的 changelog 的数据，也可以是放在外部存储上的一个物化的维表。

它的实现是通过 UDTF 去做 probe 流和 Temporal table 的 join，称之 Temporal table function join。这种 join 的方式，它适用的场景是维度数据为 changelog 流的形式，而且我们有需要按时间版本去关联的诉求。

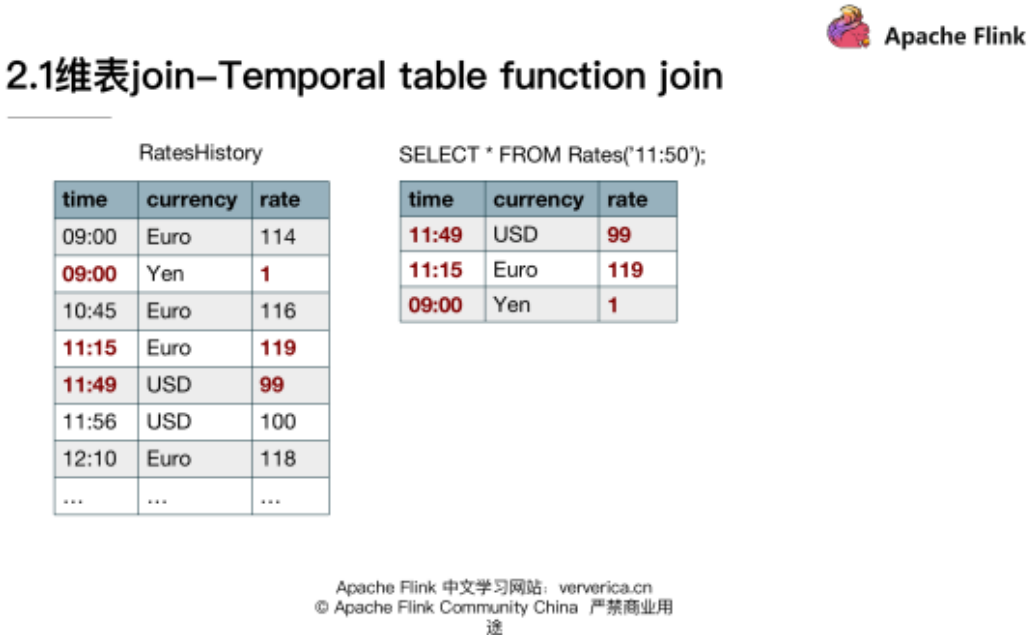
首先来看一个例子，这里使用的是官网关于汇率和货币交易的一个例子。对于我们的维度数据来说，也就是刚刚提到的 changelog stream，它是 RateHistory。它反映的是不同的货币相对于日元来说，不同时刻的汇率。



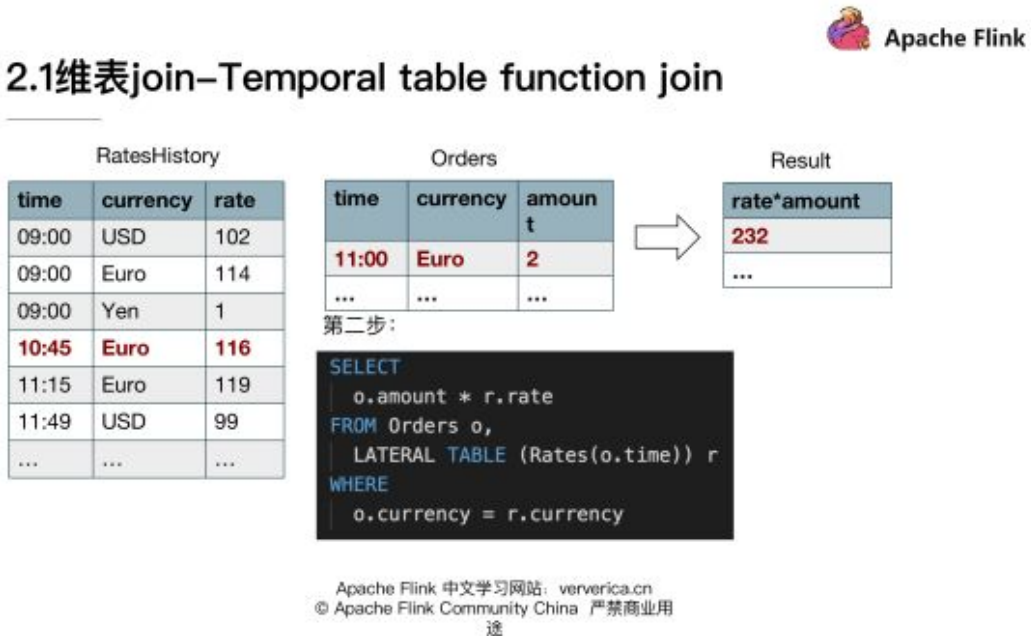
第1个字段是时间，第2个字段是 currency 货币。第3个字段是相对日元的汇率，然后在我们的 probe table 来看的话，它定义的是购买不同货币的订单的情况。比如说在 10:15 购买了两欧元，该表记录的是货币交易的一个情况。在这个例子里面，我们要求的是购买货币的总的日元交易额，如何通 Temporal table function join 来去实现我们这个目标呢?

第1步首先我们要在 changelog 流上面去定义 TemporalTableFunction，这里面有两个关键的参数是必要的。第1个参数就是能够帮我们去识别版本信息的一个 time attribute，第2个参数是需要去做关联的组件，这里的话我们选择的是 currency。
接着的话我们在 tableEnv 里面去注册 TemporalTableFunction 的名字。

然后我们来看一下我们注册的 TemporalTableFunction，它能够起到什么样的效果。



比如说如果我们使用 rates 函数，去获取11:50的状态。可以看到对于美元来说，它在11：50的状态其实落在11:49~11:56这个区间的，所以选取的是99。然后对于欧元来说，11:50的时刻是落在11:15和12:10之间的，所以我们会选取119这样的一条数据。它其实实现的是我们在一刚开始定义的 TemporalTable 的概念，能够获取到 changelog 某一时刻有效数据。定义好 TemporalTableFunction 以后，我们就要需要使用这个 Function，具体实现业务逻辑。

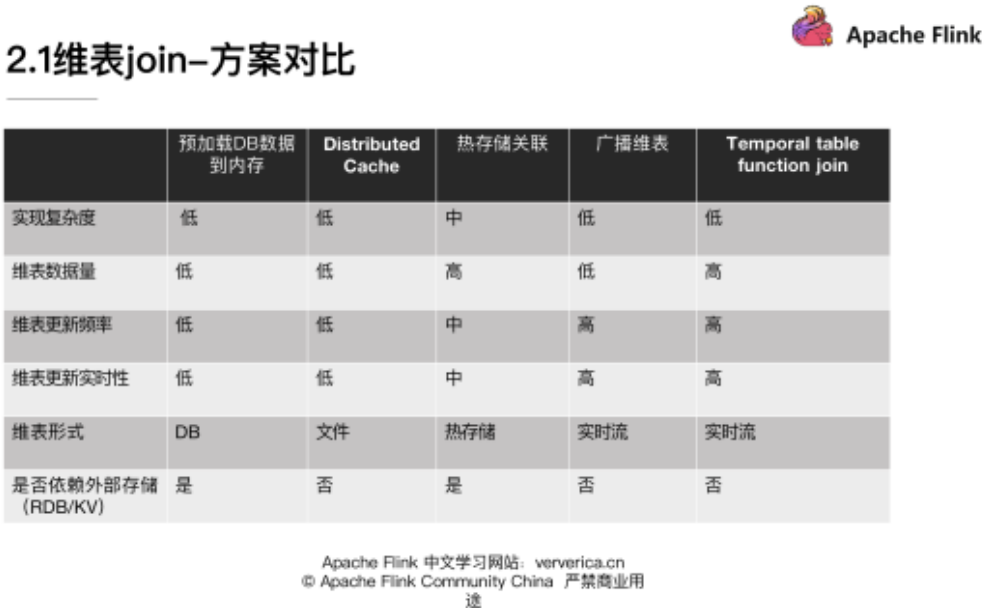


大家注意这里需要去指定我们具体需要用到的 join key。比如说因为两个流都是在一直持续更新的，对于我们的 order table 里面 11:00 的这一条记录来说，关联到的就是欧元在 10:45 这一条状态，然后它是 116，所以最后的结果就是 232。

刚才介绍的就是 Temporal table function join 的用法。

2.1.5 维表 join 的对比

然后来整体回顾一下在维表 join 这块，各个维度 join 的一些差异，便于我们更好的去理解各个方法适用的场景。



在实现复杂度上面的：除了热存储关联稍微复杂一些，其它的实现方式基本上复杂度是比较低的。
在维表数据量上：热存储关联和 Temporal table function join 两种方式可以支持比较多的数据量。其它的方式因为都要把维表加载到内存，所以就受限内存的大小。
在维表更新频率上面：因为预加载 DB 数据到内存和 Distributed Cache 在重新更新维表数据的时候都需要重启，所以它们不适合维表需要经常变更的场景。而对于广播维表和 Temporal table function join 来说，可以实时的更新维表数据并反映到结果，所以它们可以支持维表频繁更新的场景。
对维表更新实时性来说：在广播维表和 Temporal table function join，它们可以达到比较快的实时更新的效果。热存储关联在大部分场景也是可以满足业务需求的。
在维表形式上面：可以看到第1种方式主要是支持访问 DB 存储少量数据的形式，Distributed Cache 支持文件的形式，热存储关联需要访问 HBase 和 Tair 等等这种热存储。广播维表和 Temporal table function join 都需要维度数据能转化成实时流的形式。
在外部存储上面：第1种方式和热存储关联都是需要依赖外部存储的。

在维表 join 这一块，我们就先介绍这几个基本方法。可能有的同学还有一些其他方案，之后可以反馈交流，这里主要提了一些比较常用的方案，但并不限于这些方案。

2.2 双流 join

首先我们来回顾一下，批处理是怎么去处理两个表 join的？一般批处理引擎实现的时候，会采用两个思路。

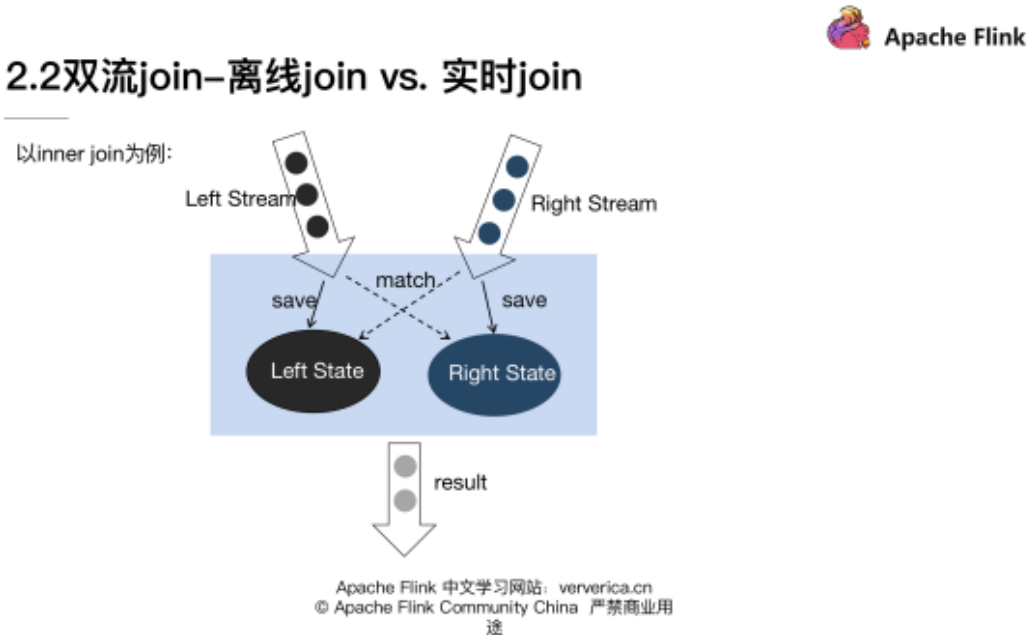
一个是基于排序的 Sort-Merge join。另外一个是转化为 Hash table 加载到内存里做 Hash join。这两个思路对于双流 join 的场景是否还同样适用？在双流 join 场景里面要处理的对象不再是这种批数据、有限的数据，而是是无穷数据集，对于无穷数据集来说，我们没有办法排序以后再做处理，同样也没有办法把无穷数据集全部转成 Cache 加载到内存去做处理。 所以这两种方式基本是不能够适用的。同时在双流 join 场景里面，我们的 join 对象是两个流，数据也是不断在进入的，所以我们 join 的结果也是需要持续更新的。

那么我们应该有什么样的方案去解决双流 join 的实现问题？Flink 的一个基本的思路是将两个流的数据持续性的存到 state 中，然后使用。因为需要不断的去更新 join 的结果，之前的数据理论上如果没有任何附加条件的话是不能丢弃的。但是从实现上来说 state 又不能永久的保存所有的数据，所以需要通过一些方式将 join 的这种全局范围局部化，就是说把一个无限的数据流，尽可能给它拆分切分成一段一段的有线数据集去做 join。

其实基本就是这样一个大的思路，接下来去看一下具体的实现方式。

2.2.1 离线 join vs. 实时 join

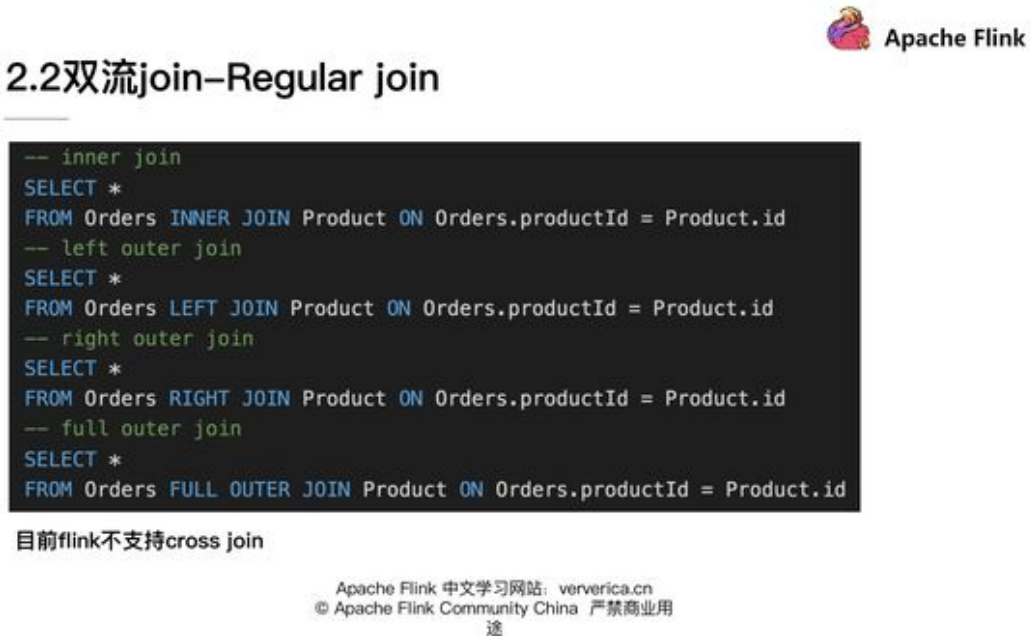
接下来我们以 inner join 为例看一下，一个简单的实现的思路：



左流是黑色标出来的这一条，右流是蓝色标出来的，这条两流需要做 inner join。首先左流和右流在元素进入以后，需要把相关的元素存储到对应的 state 上面。除了存储到 state 上面以外，左流的数据元素到来以后需要去和右边的 Right State 去做比较看能不能匹配到。同样右边的流元素到了以后，也需要和左边的 Left State 去做比较看是否能够 match，能够 match 的话就会作为 inner join 的结果输出。这个图是比较粗的展示出来一个 inner join 的大概细节。也是让大家大概的体会双流 join 的实现思路。

2.2.2 Regular join

我们首先来看一下第1类双流 join 的方式，Regular join。这种 join 方式需要去保留两个流的状态，持续性地保留并且不会去做清除。两边的数据对于对方的流都是所有可见的，所以数据就需要持续性的存在state里面，那么 state 又不能存的过大，因此这个场景的只适合有界数据流。它的语法可以看一下，比较像离线批处理的 SQL：



在上图页面里面是现在 Flink 支持 Regular join 的一些写法，可以看到和我们普通的 SQL 基本是一致的。

2.2.3 Interval join

在双流 join 里面 Flink支持的第2类 join 就是 Interval join 也叫区间 join。它是什么意思呢？就是加入了一个时间窗口的限定，要求在两个流做 join 的时候，其中一个流必须落在另一个流的时间戳的一定时间范围内，并且它们的 join key 相同才能够完成 join。加入了时间窗口的限定，就使得我们可以对超出时间范围的数据做一个清理，这样的话就不需要去保留全量的 State。

Interval join 是同时支持 processing time 和 even time去定义时间的。如果使用的是 processing time，Flink 内部会使用系统时间去划分窗口，并且去做相关的 state 清理。如果使用 even time 就会利用 Watermark 的机制去划分窗口，并且做 State 清理。

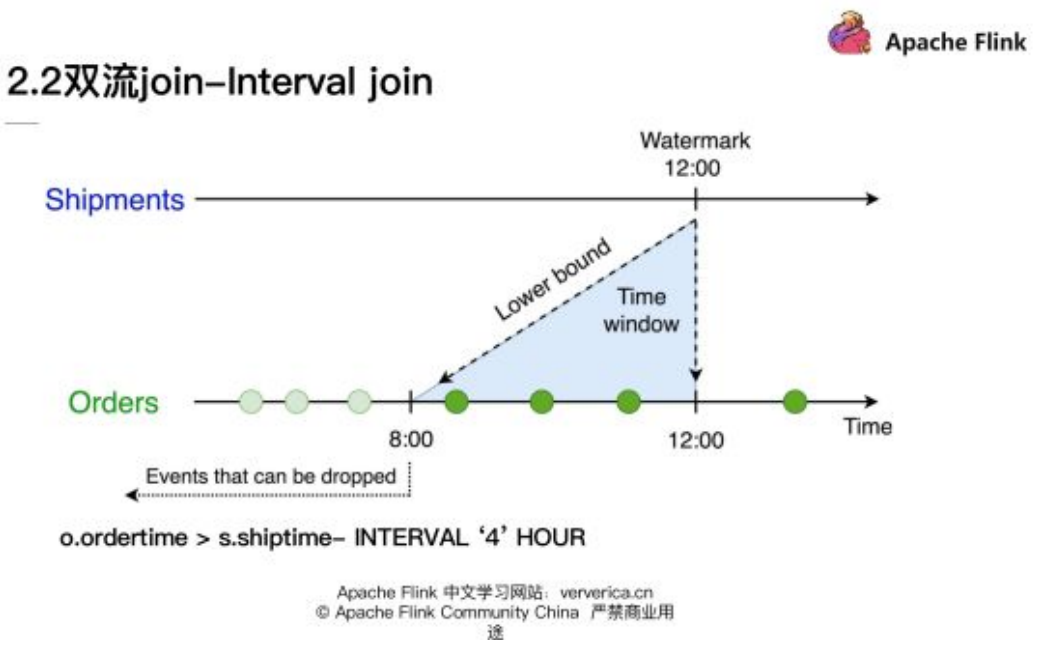
下面我们来看一些示例：

支持processing time 和 event time

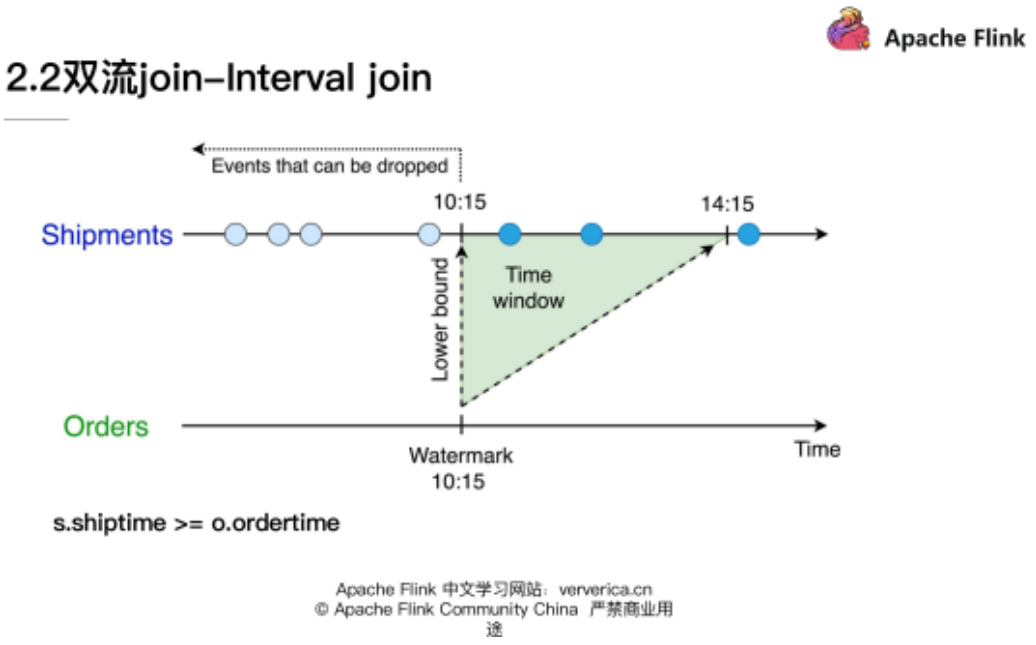
```
SELECT
  *
FROM Orders o,
  Shipments s
WHERE
  o.id = s.orderId
  AND s.shiptime BETWEEN o.ordertime
  AND o.ordertime + INTERVAL '4' HOUR
```

上图这个示例用的数据是两张表：一个是订单表，另外一个是配送表。这里定义的时间限定是配送的时间必须在下单后的4个小时内。

Flink 的作者之前有一个内容非常直观的分享，这里就直接复用了他这部分的一个示例：



我们可以看到对于 Interval join 来说：它定义一个时间的下限，就可以使得我们对于在时间下限之外的数据做清理。比如在刚才的 SQL 里面，其实我们就限定了 join 条件是 ordertime 必须要大于 shiptime 减去4个小时。 对于 Shipments 流来说，如果接收到12:00 点的 Watermark，就意味着对于 Orders 流的数据小于 8:00 点之前的数据时间戳就可以去做丢弃，不再保留在 state 里面了。



同时对于 shiptime 来说，其实它也设定了一个时间的下限，就是它必须要大于 ordertime。对于 Orders 流来说如果接收到了一个10:15点的 Watermark， 那么 Shipments 的 state 10:15 之前的数据就可以抛弃掉。 所以 Interval join 使得我们可以对于一部分历史的 state 去做清理。

2.2.4 Window join

最后来说一下双流 join 的第3种 Window join：它的概念是将两个流中有相同 key 和处在相同 window 里的元素去做 join。它的执行的逻辑比较像 Inner join，必须同时满足 join key 相同，而且在同一个 window 里元素才能够在最终结果中输出。具体使用的方式是这样的：

使用方式：

```
stream.join(otherStream)
    .where(<KeySelector>)
    .equalTo(<KeySelector>)
    .window(<WindowAssigner>) //WindowAssigner定义window划分方式
    .apply(<JoinFunction>) //JoinFunction中实现匹配元素的处理逻辑
```

符合条件的元素进入JoinFunction或FlatJoinFunction

目前 Window join 只支持 Datastream 的 API，所以这里使用方式也是 Datastream 的一个形式。可以看到我们首先把两流去做 join，然后在 where 和 equalTo 里面去定义 join key 的条件，然后在 window 中需要去指定 window 划分的方式 WindowAssigner，最后要去定义 JoinFunction 或者是 FlatJoinFunction，来实现我们匹配元素的具体处理逻辑。

因为 window 其实划分为三类，所以我们的 Window join 这里也会分为三类：

第1类 Tumbling Window join：它是按照时间区间去做划分的 window。

可以看到这个图里面是两个流（绿色的流年和黄色的流）。在这个例子里我们定义的是一个两毫秒的窗口，每一个圈是我们每个流上一个单个元素，上面的时间戳代表元素对应的时间，所以我们可以看到它是按照两毫秒的间隔去做划分的，window 和 window 之间是不会重叠的。 对于第1个窗口我们可以看到绿色的流有两个元素符合，然后黄色流也有两个元素符合，它们会以 pair 的方式组合，最后输入到 JoinFunction 或者是 FlatJoinFunction 里面去做具体的处理。

第2类 window 是 Sliding Window Join：这里用的是 Sliding Window。

sliding window 是首先定义一个窗口大小，然后再定义一个滑动时间窗的大小。如果滑动时间窗的大小小于定义的窗口大小，窗口和窗口之间会存在重叠的情况。就像这个图里显示出来的，红色

的窗口和黄色窗口是有重叠的，其中绿色流的0元素同时处于红色的窗口和黄色窗口，说明一个元素是可以同时处于两个窗口的。然后在具体的 Sliding Window Join 的时候，可以看到对于红色的窗口来说有两个元素，绿色0和黄色的0，它们两个元素是符合 window join 条件的，于是它们会组成一个0,0的 pair。 然后对于黄色的窗口符合条件的是绿色的0与黄色0和1两位数，它们会去组合成0,1、0,0和1,0两个pair，最后会进入到我们定义的 JoinFunction 里面去做处理。

第3类是 SessionWindow join：这里面用到的 window 是 session window。

session window 是定义一个时间间隔，如果一个流在这个时间间隔内没有元素到达的话，那么它就会重新开一个新的窗口。在上图里面我们可以看到窗口和窗口之间是不会重叠的。我们这里定义的Gap是1，对于第1个窗口来说，可以看到有绿色的0元素和黄色的1、2元素都是在同一个窗口内，所以它会组成在1 ,0和2,0这样的一个pair。剩余的也是类似，符合条件的pair都会进入到最后 JoinFunction 里面去做处理。

整体我们可以回顾一下，这一节主要是介绍了维表 join 和双流 join 两大类场景的 Flink ETL 实现方法。在维表 join 上主要介绍了预加载维表、热存储关联、广播维表、Temporal table function join 这4种方式。然后在双流 join 上我们介绍了 Regular join、Interval join 和 Window join。

作者：买蓉 · 美团点评高级技术专家

原文链接