Flink DataStream 关联维表实战

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。

上篇博客提到 Flink SQL 如何 Join 两个数据流，有读者反馈说如果不打算用 SQL 或者想自己实现底层操作，那么如何基于 DataStream API 来关联维表呢？实际上由于 Flink DataStream API 的灵活性，实现这个需求的方式是非常多样的，但是大部分用户很难在设计架构时就考虑得很全面，可能会走不少弯路。

针对于此，笔者根据工作经验以及社区资源整理了用 DataStream 实现 Join 维表的常见方式，并给每种的方式优劣和适用场景给出一点可作为参考的个人观点。

衡量指标

总体来讲，关联维表有三个基础的方式：实时数据库查找关联（Per-Record Reference Data Lookup）、预加载维表关联（Pre-Loading of Reference Data）和维表变更日志关联（Reference Data Change Stream），而根据实现上的优化可以衍生出多种关联方式，且这些优化还可以灵活组合产生不同效果（不过为了简单性这里不讨论同时应用多种优化的实现方式）。对于不同的关联方式，我们可以从以下 7 个关键指标来衡量（每个指标的得分将以 1-5 五档来表示）：

1. 实现简单性: 设计是否足够简单，易于迭代和维护。
2. 吞吐量: 性能是否足够好。
3. 维表数据的实时性: 维度表的更新是否可以立刻对作业可见。

4. 数据库的负载: 是否对外部数据库造成较大的负载（负载越低分越高）。
5. 内存资源占用: 是否需要大量内存来缓存维表数据（内存占用越少分越高）。
6. 可拓展性: 在更大规模的数据下会不会出现瓶颈。
7. 结果确定性: 在数据延迟或者数据重放情况下，是否可以得到一致的结果。

和大多数架构设计一样，这三类关联方式不存在绝对的好坏，更多的是针对业务场景在各指标上的权衡取舍，因此这里的得分也仅仅是针对通用场景来说。

实时数据库查找关联

实时数据库查找关联是在 DataStream API 用户函数中直接访问数据库来进行关联的方式。这种方式通常开发量最小，但一般会给数据库带来很大的压力，而且因为关联是基于 Processing Time 的，如果数据有延迟或者重放，会得到和原来不一致的数据。

同步数据库查找关联

同步实时数据库查找关联是最为简单的关联方式，只需要在一个 Map 或者 FlatMap 函数中访问数据库，处理好关联逻辑后，将结果数据输出。

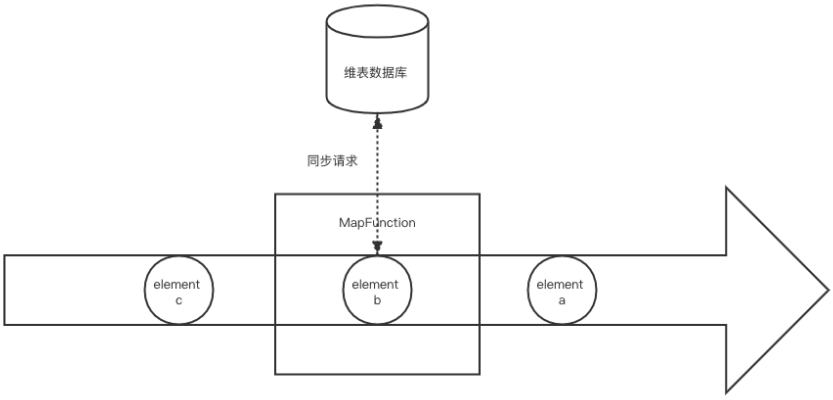

图1.同步数据库查找关联架构

这种方式的主要优点在于实现简单、不需要额外内存且维表的更新延迟很低，然而缺点也很明显:

1. 因为每条数据都需要请求一次数据库，给数据库造成的压力很大；
2. 访问数据库是同步调用，导致 subtak 线程会被阻塞，影响吞吐量；
3. 关联是基于 Processing Time 的，结果并不具有确定性；
4. 瓶颈在数据库端，但实时计算的流量通常远大于普通数据库的设计流量，因此可拓展性比较低。

图2.同步数据库查找关联关键指标

从应用场景来说，同步数据库查找关联可以用于流量比较低的作业，但通常不是最好的选择。

异步数据库查找关联

异步数据库查找关联是通过 AsyncIO[2]来访问外部数据库的方式。利用数据库提供的异步客户端，AsyncIO 可以并发地处理多个请求，很大程度上减少了对 subtask 线程的阻塞。

因为数据库请求响应时长是不确定的，可能导致后输入的数据反而先完成计算，所以 AsyncIO 提供有序和无序两种输出模式，前者会按请求返回顺序输出数据，后者则会缓存提前完成计算的数据，并按输入顺序逐个输出结果。

图3.异步数据库查找关联架构

比起同步数据库查找关联，异步数据库查找关联稍微复杂一点，但是大部分的逻辑都由 Flink AsyncIO API 封装，因此总体来看还是比较

简单。然而，有序输出模式下的 AsyncIO 会需要缓存数据，且这些数据会被写入 checkpoint，因此在内容资源方面的得分会低一点。另一方面，同步数据库查找关联的吞吐量问题得到解决，但仍不可避免地有数据库负载高和结果不确定两个问题。

图4.异步数据库查找关联关键指标

从应用场景来说，异步数据库查找关联比较适合流量低的实时计算。

带缓存的数据库查找关联

为了解决上述两种关联方式对数据库造成太大压力的问题，可以引入一层缓存来减少直接对数据库的请求。缓存并一般不需要通过 checkpoint 机制持久化，因此简单地用一个 WeakHashMap 或者 Guava Cache 就可以实现。

图5.带缓存的数据库查找关联架构

虽然在冷启动的时候仍会给数据库造成一定压力，但后续取决于缓存命中率，数据库的压力将得到一定程度的缓解。然而使用缓存带来的问题是维表的更新并不能及时反应到关联操作上，当然这也和缓存剔除的策略有关，需要根据维度表更新频率和业务对过时维表数据的容忍程度来设计。

图6.带缓存的数据库查找关联关键指标

总而言之，带缓存的数据库查找关联适合于流量比较低，且对维表数据实时性要求不太高或维表更新比较少的业务场景。

预加载维表关联

相比起实时数据库查找在运行期间为每条数据访问一次数据库，预加载维表关联是在作业启动时就将维表读到内存中，而在后续运行期间，每条数据都会和内存中的维表进行关联，而不会直接触发对数据的访问。与带缓存的实时数据库查找关联相比，区别是后者如果不命中缓存还可以 fallback 到数据库访问，而前者如果不名中则会关联不到数据。

启动预加载维表

启动预加载维表是最为简单的一种方式，即在作业初始化的时候，比如用户函数的 open() 方法，直接从数据库将维表拷贝到内存中。维表并不需要用 State 来保存，因为无论是手动重启或者是 Flink 的错误重试机制导致的重启，open() 方法都会被执行，从而得到最新的维表数据。

图7.启动预加载维表架构

启动预加载维表对数据库的压力只持续很短时间，但因为是拷贝整个维表所以压力是很大的，而换来的优势是在运行期间不需要再访问数据库，可以提高效率，有点类似离线计算。相对地，问题在于运行期间维表数据不能更新，且对 TaskManager 内存的要求比较高。

图8.启动预加载维表关键指标

启动预加载维表适合于维表比较小、变更实时性要求不高的场景，比如根据 ip 库解析国家地区，如果 ip 库有新版本，重启作业即可。

启动预加载分区维表

对于维表比较大的情况，可以启动预加载维表基础之上增加分区功能。简单来说就是将数据流按字段进行分区，然后每个 Subtask 只需要加在对应分区范围的维表数据。值得注意的是，这里的分区方式并不是用 keyby 这种通用的 hash 分区，而是需要根据业务数据定制化分区策略，然后调用 DataStream#partitionCustom。比如按照 userId 等区间划分，0-999 划分到 subtask 1，1000-1999 划分到

subtask 2，以此类推。而在 open() 方法中，我们再根据 subtask 的 id 和总并行度来计算应该加载的维表数据范围。

图9.启动预加载分区维表架构

通过这种分区方式，维表的大小上限理论上可以线性拓展，解决了维表大小受限于单个 TaskManager 内存的问题（现在是取决于所有 TaskManager 的内存总量），但同时给带来设计和维护分区策略的复杂性。

图10.启动预加载分区维表关键指标

总而言之，启动预加载分区维表适合维表比较大而变更实时性要求不高的场景，比如用户点击数据关联用户所在地。

启动预加载维表并定时刷新

除了维表大小的限制，启动预加载维表的另一个主要问题在于维度数据的更新，我们可以通过引入定时刷新机制的办法来缓解这个问题。定时刷新可以通过 Flink ProcessFucntion 提供的 Timer 或者直接在 open() 初始化一个线程（池）来做这件事。不过 Timer 要求 KeyedStream，而上述的 DataStream#partitionCustom 并不会返回一个 KeyedStream，因此两者并不兼容。而如果使用额外线程定时刷新的办法则不受这个限制。

图11.启动预加载维表并定时刷新架构

比起基础的启动预加载维表 ，这种方式在于引入比较小复杂性的情况下大大缓解了的维度表更新问题，但也给维表数据库带来更多压力，因为每次 reload 的时候都是一次请求高峰。

图12.启动预加载维表并定时刷新关键指标

启动预加载维表和定时刷新的组合适合维表变更实时性要求不是特别高的场景。取决于定时刷新的频率和数据库的性能，这种方式可以满足大部分关联维表的业务。

启动预加载维表 + 实时数据库查找

启动预加载维表还可以和实时数据库查找混合使用，即将预加载的维表作为缓存给实时关联时使用，若未名中则 fallback 到数据库查找。

图13.启动预加载维表结合实时数据库查找架构

这种方式实际是带缓存的数据库查找关联的衍生，不同之处在于相比冷启动时未命中缓存导致的多次实时数据库访问，该方式直接批量拉取整个维表效率更高，但也有可能拉取到不会访问到的多余数据。下面雷达图中显示的是用异步数据库查找，如果是同步数据库查找吞吐量上会低一些。

图14.启动预加载维表结合实时数据库查找关键指标

这种方式和带缓存的实时数据库查找关联基本相同，适合流量比较低，且对维表数据实时性要求不太高或维表更新比较少的业务场景。

维表变更日志关联

不同于上述两者将维表作为静态表关联的方式，维表变更日志关联将维表以 changelog 数据流的方式表示，从而将维表关联转变为两个数据流的 join。这里的 changelog 数据流类似于 MySQL 的 binlog，通常需要维表数据库端以 push 的方式将日志写到 Kafka 等消息队列中。Changelog 数据流称为 build 数据流，另外待关联的主要数据流成为 probe 数据流。

维表变更日志关联的好处在于可以获取某个 key 数据变化的时间，从而使得我们能在关联中使用 Event Time（当然也可以使用 Processing Time）。

Processing Time 维表变更日志关联

如果基于 Processing Time 做关联，我们可以利用 keyby 将两个数据流中关联字段值相同的数据划分到 KeyedCoProcessFunction 的同一个分区，然后用 ValueState 或者 MapState 将维表数据保存下来。在普通数据流的一条记录进到函数时，到 State 中查找有无符合条件的 join 对象，若有则关联输出结果，若无则根据 join 的类型决定是直接丢弃还是与空值关联。这里要注意的是，State 的大小要尽量控制好。首先是只保存每个 key 最新的维度数据值，其次是要给 State 设置好 TTL，让 Flink 可以自动清理。

图15.Processing Time 维表变更日志关联架构

基于 Processing Time 的维表变更日志关联优点是不需要直接请求数据库，不会对数据库造成压力；缺点是比较复杂，相当于使用 changelog 在 Flink 应用端重新构建一个维表，会占用一定的 CPU 和比较多的内存和磁盘资源。值得注意的是，我们可以利用 Flink 提供的 RocksDB StateBackend，将大部分的维表数据存在磁盘而不是

内存中，所以并不会占用很高的内存。不过基于 Processing Time 的这种关联对两个数据流的延迟要求比较高，否则如果其中一个数据流出现 lag 时，关联得到的结果可能并不是我们想要的，比如可能会关联到未来时间点的维表数据。

图16.Processing Time 维表变更日志关联关键指标

基于 Processing Time 的维表变更日志关联比较适用于不便直接访问数据的场景（比如维表数据库是业务线上数据库，出于安全和负载的原因不能直接访问），或者对维表的变更实时性要求比较高的场景（但因为数据准确性的关系，一般用下文的 Event Time 关联会更好）。

Event Time 维表变更日志关联

基于 Event Time 的维表关联实际上和基于 Processing Time 的十分相似，不同之处在于我们将维表 changelog 的多个时间版本都记录下来，然后每当一条记录进来，我们会找到对应时间版本的维表数据来和它关联，而不是总用最新版本，因此延迟数据的关联准确性大大提高。不过因为目前 State 并没有提供 Event Time 的 TTL，因此我们需要自己设计和实现 State 的清理策略，比如直接设置一个 Event Time Timer（但要注意 Timer 不能太多导致性能问题），再比如对于单个 key 只保存最近的 10 个版本，当有更新版本的维表数据到达时，要清理掉最老版本的数据。

图17.Event Time 维表变更日志关联架构

基于 Event Time 的维表变更日志关联相对基于 Processing Time 的方式来说是一个改进，虽然多个维表版本导致空间资源要求更大，但确保准确性对于大多数场景来说都是十分重要的。相比 Processing Time 对两个数据的延迟都有要求，Event Time 要求 build 数据流的延迟低，否则可能一条数据到达时关联不到对应维表数据或者关联了一个过时版本的维表数据，

图18.Event Time 维表变更日志关联关键指标

基于 Event Time 的维表变更日志关联比较适合于维表变更比较多且对变更实时性要求较高的场景 同时也适合于不便直接访问数据库的场景。

Temporal Table Join

Temporal Table Join 是 Flink SQL/Table API 的原生支持，它对两个数据流的输入都进行了缓存，因此比起上述的基于 Event Time 的维表变更日志关联，它可以容忍任意数据流的延迟，数据准确性更好。Temporal Table Join 在 SQL/Table API 使用时是十分简单的，但如果想在 DataStream API 中使用，则需要自己实现对应的逻辑。

总体思路是使用一个 CoProcessFunction，将 build 数据流以时间版本为 key 保存在 MapState 中（与基于 Event Time 的维表变更日志关联相同），再将 probe 数据流和输出结果也用 State 缓存起来（同样以 Event Time 为 key），一直等到 Watermark 提升到它们对应的 Event Time，才把结果输出和将两个数据流的输入清理掉。

这个 Watermark 触发很自然地是用 Event Time Timer 来实现，但要注意不要为每条数据都设置一遍 Timer，因为一旦 Watermark 提升会触发很多个 Timer 导致性能急剧下降。比较好的实践是为每个 key 只注册一个 Timer。实现上可以记录当前未处理的最早一个 Event Time，并用来注册 Timer。当前 Watermark。每当 Watermark 触发 Timer 时，我们检查处理掉未处理的最早 Event Time 到当前 Event Time 的所有数据，并将未处理的最早 Event Time 更新为当前时间。

图19.Temporal Table Join 架构

Temporal Table Join 的好处在于对于两边数据流的延迟的容忍度较大，但作为代价会引入一定的输出结果的延迟，这也是基于 Watermark 机制的计算的常见问题，或者说，妥协。另外因为吞吐量较大的 probe 数据流也需要缓存，Flink 应用对空间资源的需求会大很多。最好，要注意的是如果维表变更太慢，导致 Watermark 提升太慢，会导致 probe 数据流被大量缓存，所以最好要确保 build 数据流尽量实时，同时给 Source 设置一个比较短的 idle timeout。

图20.Temporal Table Join 关键指标

Temporal Table Join 这种方式最为复杂，但数据准确性最好，适合一些对数据准确性要求高且可以容忍一定延迟（一般分钟级别）的关键业务。

衡量指标

用 Flink DataStream API 实现关联维表的方式十分丰富，可以直接访问数据库查找（实时数据库查找关联），可以启动时就将全量维表读到内存（预加载维表关联），也可以通过维表的 changelog 在 Flink 应用端实时构建一个新的维表（维表变更日志关联）。我们可以从实现简单性、吞吐量、维表数据的实时性、数据库的负载、内存

资源占用、可拓展性和结果确定性这 7 个维度来衡量一个具体实现方式，并根据业务需求来选择最合适的实现。

参考

[1] WEBINAR: 99 Ways to Enrich Streaming Data with Apache Flink

[2] Asynchronous I/O for External Data Access