一行配置作业性能提升53%！Flink SQL 性能之旅

作者｜Nico Kruber
翻译｜毛家琦
校对｜伍翀

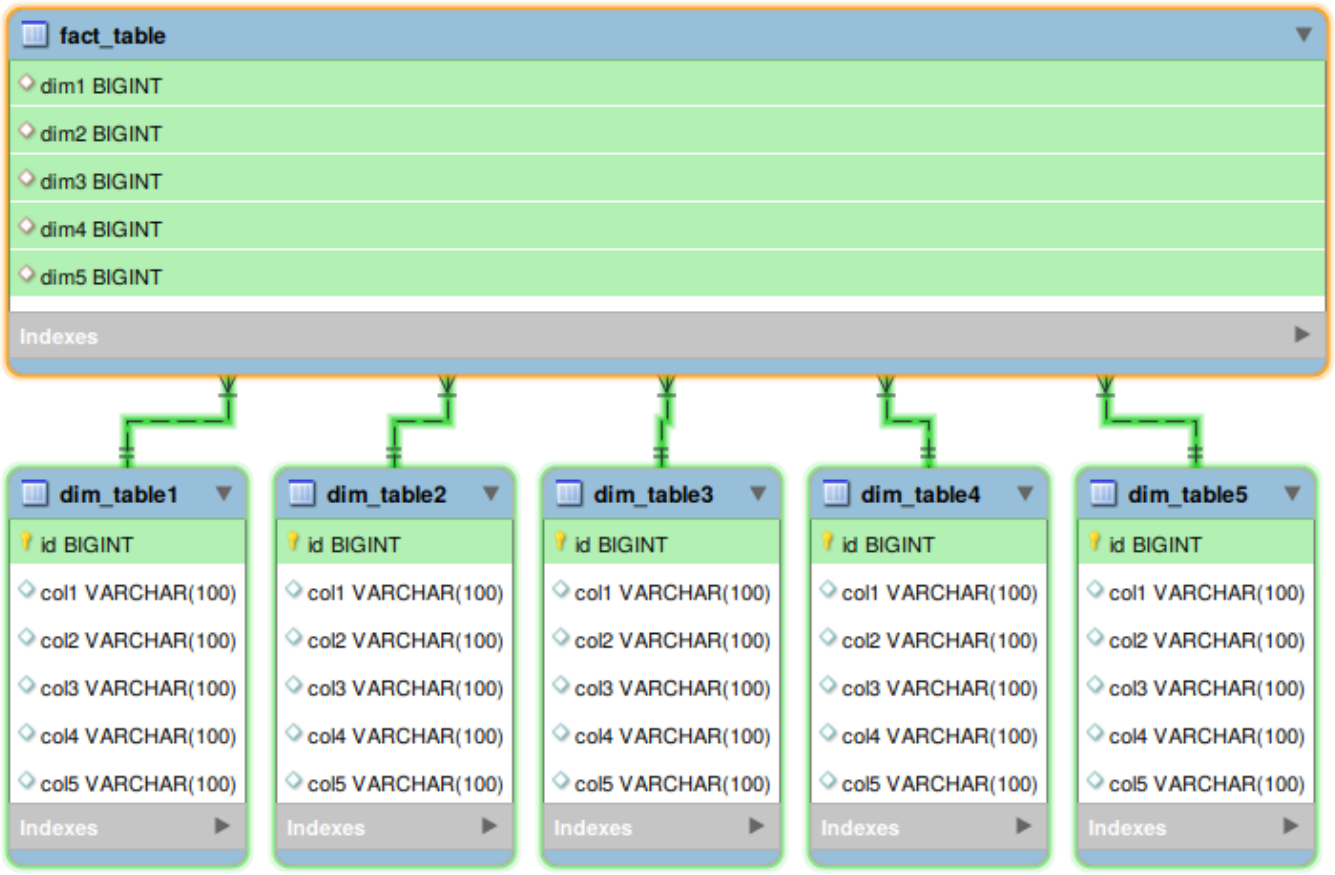
最近，我们用 SQL 查询做了一些实验，这个查询关联了一些维表的丰富原始记录。同时，我们也考虑如果使用 DataStream API 实现相同的任务，是否能够从现有机器中激发出更多的性能。在本文中，我们想带你一起看看这是否有可能发生，以及如何实现？我们还会为不同于 PoC 代码的作业提供进一步的提示，并对未来的工作进行展望。

我们在 Azure Kubernetes 设置中执行了 10 个标准的 v3 实例（每个实例 2 个CPU）、Ververica Platform 2.0、Flink 1.10 以及每个实例单核 8 并发的实验，这些作为这次试验的背景[1]。我们将给出在稳定状态（即 15 分钟长基准的最后 5 分钟）期间合并的所有源的平均吞吐量，即 numRecordsOutPerSecond。如下示例的源代码可以从链接位置检索：

https://github.com/verververica/lab-sql-vs-datastream

SQL 查询

首先，让我们看看我们试图超越的查询。下面概述的查询来源于一个流式 SQL 作业中获得的灵感。它执行来自 fact_table 的输入流与几个维表的连接，维表定义如下（维表中的数据为每个最多 100 个字符的随机字符串）：



这个作业的重要部分是，所有输入表都可以被更改；它们作为流被使用。在这种情况下，我们需要确定关联的是哪个版本的行。这就是 Flink 的 temporal table Join 的使用场景：在执行 join 时，fact_table 中的每一行都应该与来自相匹配维度表的最新行进行连接和合并。如果您进一步研究 temporal joins（通过 LATERAL TABLE 语句和围绕表 dim_TABLE 1，...，dim_TABLE 5 的包装时态表函数），会发现 Flink 还支持 event-time 连接，它可能会派上用场。然而，我们为了示例简单，在这里就不使用 event-time 了。

```
SELECT
    D1.col1 AS A,
    D1.col2 AS B,
    D1.col3 AS C,
    D1.col4 AS D,
    D1.col5 AS E,
    D2.col1 AS F,
    D2.col2 AS G,
    ...
    D5.col4 AS X,
    D5.col5 AS Y
FROM
    fact_table,
    LATERAL TABLE (dimension_table1(f_proctime)) AS D1,
    LATERAL TABLE (dimension_table2(f_proctime)) AS D2,
    LATERAL TABLE (dimension_table3(f_proctime)) AS D3,
    LATERAL TABLE (dimension_table4(f_proctime)) AS D4,
    LATERAL TABLE (dimension_table5(f_proctime)) AS D5
WHERE
    fact_table.dim1     = D1.id
    AND fact_table.dim2 = D2.id
```

```
    AND fact_table.dim3 = D3.id
    AND fact_table.dim4 = D4.id
    AND fact_table.dim5 = D5.id
```

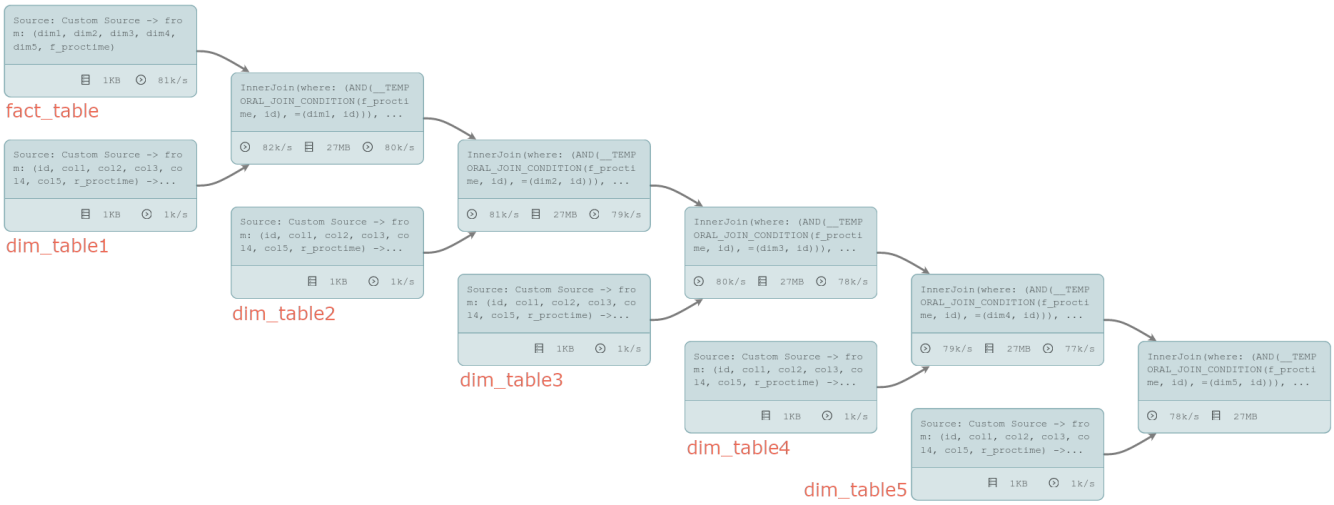
LateralTableJoin.java 里的 DataStream 代码为每个输入表创建了一个流式输入源，并将输出转换为一个 append 的 DataStream，然后导入了 DiscardingSink。在 Flink 1.10 中设置此 SQL 作业有两种方法：使用旧的 Flink Planner 或使用新的 Blink Planner。让我们看看这两者有什么不同。

旧 Flink Planner

旧 Planner 当前（Flink 1.10）是默认使用的，或者可以通过以下：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironme
  .create(env, EnvironmentSettings.newInstance().useOldPlanner().build());
```

它将 SQL 查询转换为以下作业图，作业图中的每个节点包含了一些 chain operators，例如将 DataStream source 转成 table、join 后选出列子集：



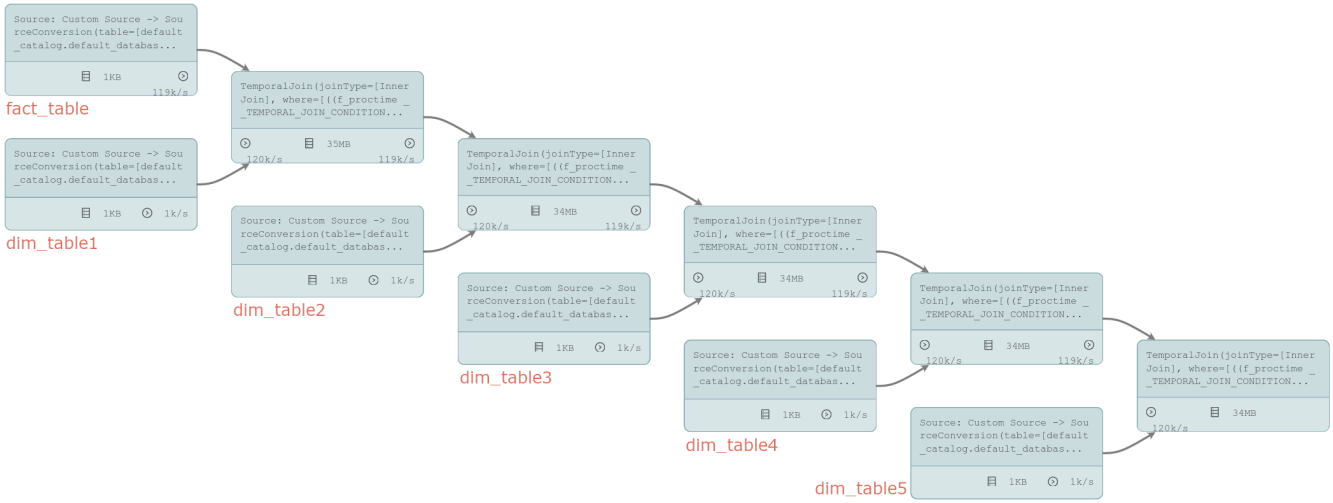
开箱即用，它提供 84279 个事件/秒的稳定吞吐量。

新 Blink Planner

Flink 的新 Blink Planner 实现了一些增强功能，例如改进的功能集，并且在性能表现上，尽可能地多使用二进制类型，以避免序列化/反序列化开销。它可以在 StreamTableEnvironment 的初始化期间启用：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironme

StreamTableEnvironment tEnv = StreamTableEnvironment

  .create(env, EnvironmentSettings.newInstance().useBlinkPlanner().build())
```

创建的作业图看起来与旧的 Planner 没有太大的不同点，并且具有类似的概念算子：



在这种情况下，只需启用 Blink Planner，吞吐量就会略微提高到 89048 个事件/秒（+5%）。

对象重用

这两个 Planner 创建的任务实际上都是由两个链式算子组成的：源附加了某种表转换/字段选择，在（temporal）join 之后附加了字段选择。如果你还记得 Flink 是如何处理算子之间的数据对象交换的[2]，你会注意到，链接算子之间的数据传输会经过序列化/反序列化/复制阶段，以防止在下一个算子中修改对象时意外将其存储在一个算子中。这种行为不仅会影响批处理程序，还会影响流处理作业，并且可以通过启用对象重用来更改，这通常是非常危险的，但是在 Flink 的 Table/SQL API 中这是非常安全的：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironme
env.getConfig () 。enableObjectReuse () ;
```

从下面的数字可以看出，启用此开关会略微提高旧 Planner 的性能（+5%），这也可能是正常的结果波动。但这里遇到的瓶颈可能是其他原因导致的。更重要的是，启用对象重用可以显著提高 Blink Planner 的吞吐量（+53%）。通常，使用的 SQL 函数/算子越多，启用对象重用产生的影响就越大。

	Without Object Reuse		With Object Reuse
Old Planner	84,279	→	88,426 (+5%)
Blink Planner	89,048	→	136,710 (+53%)

看到这些数字，您可能会想，为什么使用新的 Blink planner 启用对象重用比使用旧的 planner 更能提高性能。其原因在 Flink 的堆栈深处，而且可能与我们运行的查询有关联，因为我们在这个查询中大量使用字符串：

如果没有对象重用，在旧 Planner 中，同一任务的两个算子之间的数据交换最终将通过 StringSerializer#copy(String)。在 Blink planner 中，它最终将调用 BinaryString#copy（）。如果查看实现代码，可以发现 StringSerializer#copy（String）可以依赖于 Java 字符串的不变性，因此可以高效的使用和传递引用。而 BinaryString#copy（）需要复制底层 MemorySegment 的字节。通过启用对象重用来避免复制，可以有效提升速度。然后删除 StringSerializer-copy（String）调用可能只会轻微地减少开销。

接下来，除了寻找 Table API 执行引擎和优化器的调优参数[3]之外，没有太多当前可以优化的点了。然而，对于给定的工作，这些调整开关似乎都不能保证有进一步的改进。如果尝试使用 profiler 进行进一步的调查，您将实际看到字符串序列化和反序列化以及 Table API 的二进制数据类型处理对总体性能的影响是最大的，其次是状态访问。

对于未来，有想法引入新的 source 和 sink 接口，可以直接处理二进制的数据类型。此外，用户定义的函数目前正沿着 FLIP-65 方向进行修改，以便它们也可以直接处理二进制数据。这可以使得 UDFs 与内置函数一样强大。这两种增强都将进一步减少堆栈中的序列化开销。

与 DataStream API 同行

我的第一个天真的想法是，我可以很轻易地通过使用 DataStream API 来击败这个吞吐量。我可以将相同的 joins 实现在一起，无需 SQL 的任何转换层。这自然涉及到更多的编码，于是我从使用 Java 开始。

第一次尝试

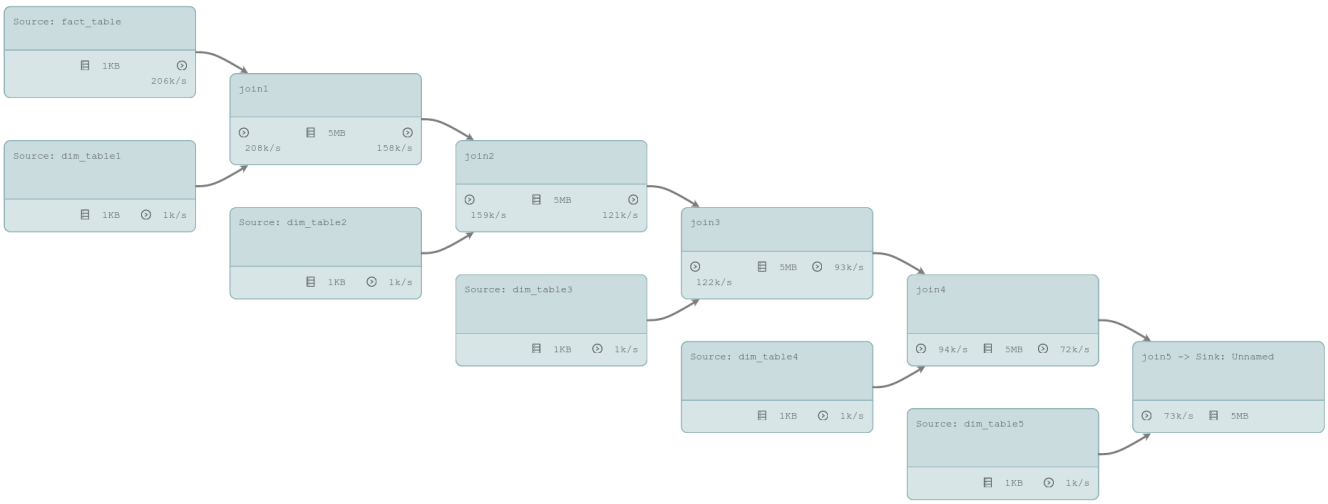
我的第一个草图大致围绕这些代码行展开，FactTable 和 dimensiontable 是与上面的 SQL 作业相同的源函数。你可以在 LateralStreamJoin1 找到它：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironme

DataStream factTableSource = env
    .addSource(new FactTable(factTableRate, maxId))
    .name("fact_table");
DataStream dimTable1Source = env
    .addSource(new DimensionTable(dimTableRate, maxId))
    .name("dim_table1");
// ...

DataStream joinedStream =
    factTableSource
        .keyBy(x -> x.dim1)
        .connect(dimTable1Source.keyBy(x -> x.id))
        .process(
            new AbstractFactDimTableJoin<fact, fact1="">() {
              @Override
              Fact1 join(Fact value, Dimension dim) {
                return new Fact1(value, dim);
              }
            })
        .name("join1").uid("join1")
        .keyBy(x -> x.dim2)
        .connect(dimTable2Source.keyBy(x -> x.id))
// ...

joinedStream.addSink(new DiscardingSink<>());
env.execute();</fact,>
```

通过合适的 join key 进行 keyBy 后，fact 表与每个维度一个接一个地联接在一起。在这段代码中，helper 类 AbstractFactDimTableJoin 实际上按照处理时间进行关联：它跟踪 processElement2 中每个键的最新维度数据对象，并且在 processElement1 中丰富的每个事实事件，如果有最新的状态对象出现就会提取它们并填充到实时事件中。这里的 PoC 将忽略缺少维度数据的事件。

```
abstract class AbstractFactDimTableJoin<in1, out=""> extends CoProcessFunction<in
  protected transient ValueState dimState;

  @Override
  public void processElement1(IN1 value, Context ctx, Collector out) throws Excep
    Dimension dim = dimState.value();
    if (dim == null) {
      return;
    }
    out.collect(join(value, dim));
  }

  abstract OUT join(IN1 value, Dimension dim);

  @Override
  public void processElement2(Dimension value, Context ctx, Collector out) throws
    dimState.update(value);
  }

  @Override
  public void open(Configuration parameters) throws Exception {
    super.open(parameters);
    ValueStateDescriptor dimStateDesc =
        new ValueStateDescriptor<>("dimstate", Dimension.class);
    this.dimState = getRuntimeContext().getState(dimStateDesc);
  }
}
```

总之，这创建了下面的作业图，它与上面的 SQL 作业非常相似，但不需要 Table conversion，也不需要选择相应的键—它内置在 join 函数中，该函数使用 Fact 衍生类来代表每次字段的扩展：Fact1、Fact2、...、Fact4、DenormalizedFact。

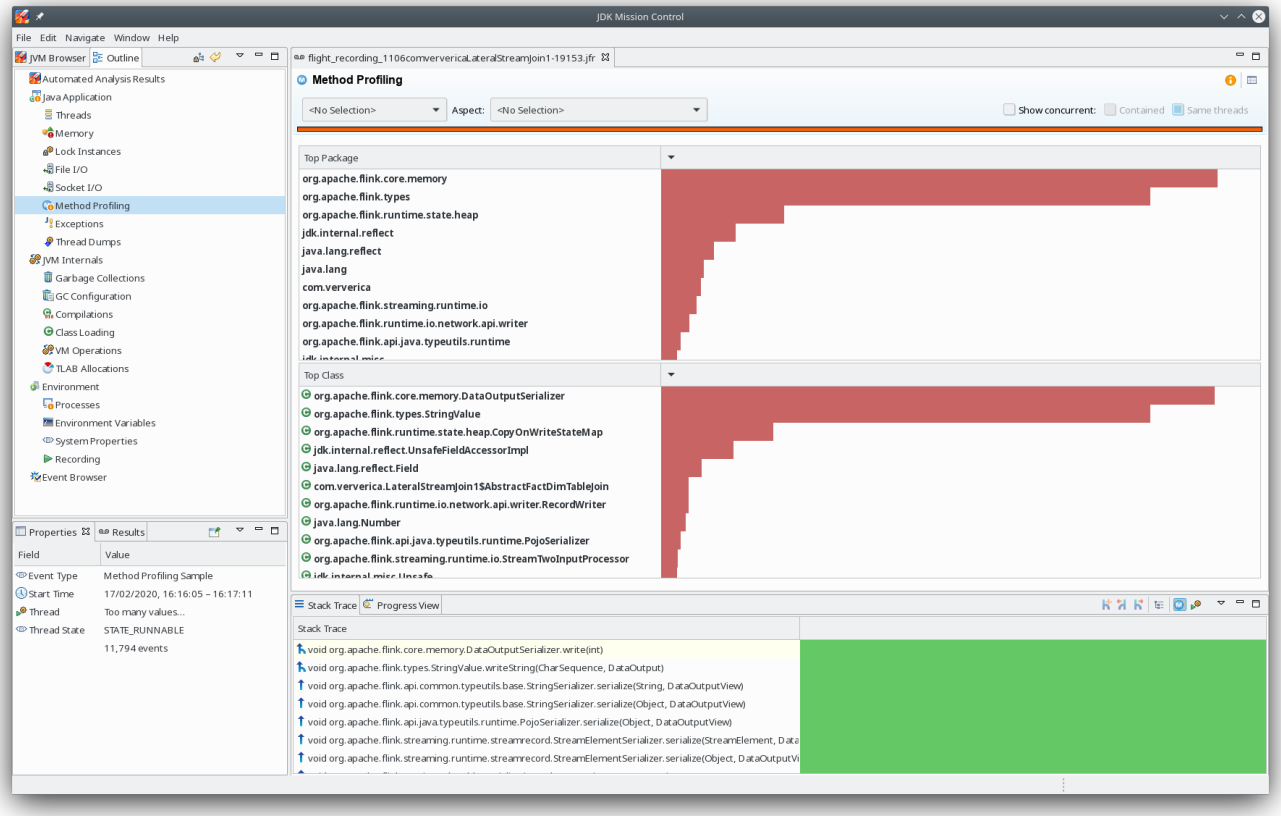


到目前为止，不管有没有对象重用，我们都能比 Flink 的旧 planner 强 17%。然而，新的 Blink planner 在启用对象重用之后，每秒就能够挤出更多的事件。实际上，对象重用不应该对我们的数据流作业产生任何影响，因为没有链式算子；关于下面 4% 的数字差异似乎来自基准测试期间的正常波动。

	Without Object Reuse	With Object Reuse
SQL Join, Old Planner	84,279	88,426
SQL Join, Blink Planner	89,048	136,710
DataStream Join 1	99,025	103,102

DataStream 作业的性能分析

实际上，我希望有更大的差异显示，所以让我们来看下 profiler，看看 CPU 时间花在哪里。您可以使用任何您喜欢的 profiler；我将显示来自 JMC 7 的结果（https://jdk.java.net/jmc/），并使用 Java 11 运行 LateralStreamJoin1 来获得这些结果。如你所见，来自 DataOutputSerializer、StringValue 和反射访问的序列化开销超过了实际的业务逻辑开销，例如 CopyOnWriteStateMap，它正在从 on-heap 状态或 com.ververica.LateralStreamJoin1 中检索匹配的维度数据。



这结果并不意外，因为（反）序列化通常总体上具有很高的成本，并且所呈现的作业本身没有（很多）计算量—fact 流中的每个事件实际上只有一个状态访问。另一方面，如果我们从上面回忆起作业图，那么在 join1、join2、join3、join4 和 join5 中的 dim_table1 中的数据将（反）序列化，这是实际使用它的唯一/第一个位置。这与其他维度表类似并且是我们可以避免的。

减少序列化开销

为了避免一次又一次地对同一数据进行连续（反）序列化，我们可以在源代码处序列化一次，然后直接传递它，直到它真正被需要为止（例如在 join5 任务中）。使用我们的 source 生成器，最快的方法是添加执行此转换的 Map 函数：

```
class MapDimensionToBinary extends RichMapFunction<dimension, binarydimension="">

  private transient TypeSerializer dimSerializer;
  private transient DataOutputSerializer serializationBuffer;

  @Override
  public BinaryDimension map(Dimension value) throws Exception {
    return BinaryDimension.fromDimension(value, dimSerializer, serializationBuffe
  }

  @Override
  public void open(Configuration parameters) throws Exception {
    serializationBuffer = new DataOutputSerializer(100);
    dimSerializer = TypeInformation.of(Dimension.class).createSerializer(getRunti
  }
}
```

BinaryDimension 是一个简单 POJO，一个 long 表示维度键，一个字节数组表示序列化数据。这是我们接下来要传递的对象。实际上，如果您查看 LateralStreamJoin2 的代码，您将看到我们现在改用 Tuple2，...，Tuple5（为了简单起见），这是为了进一步（微小的）性能提升，因为 Tuple 序列化比 POJO 序列化快一点。因此，作业的主体只轻微地改动了：包括使用了这些额外的 Map 映射，对参与的类型轻微修改了下，以及最后一级 join 之后的反序列化。

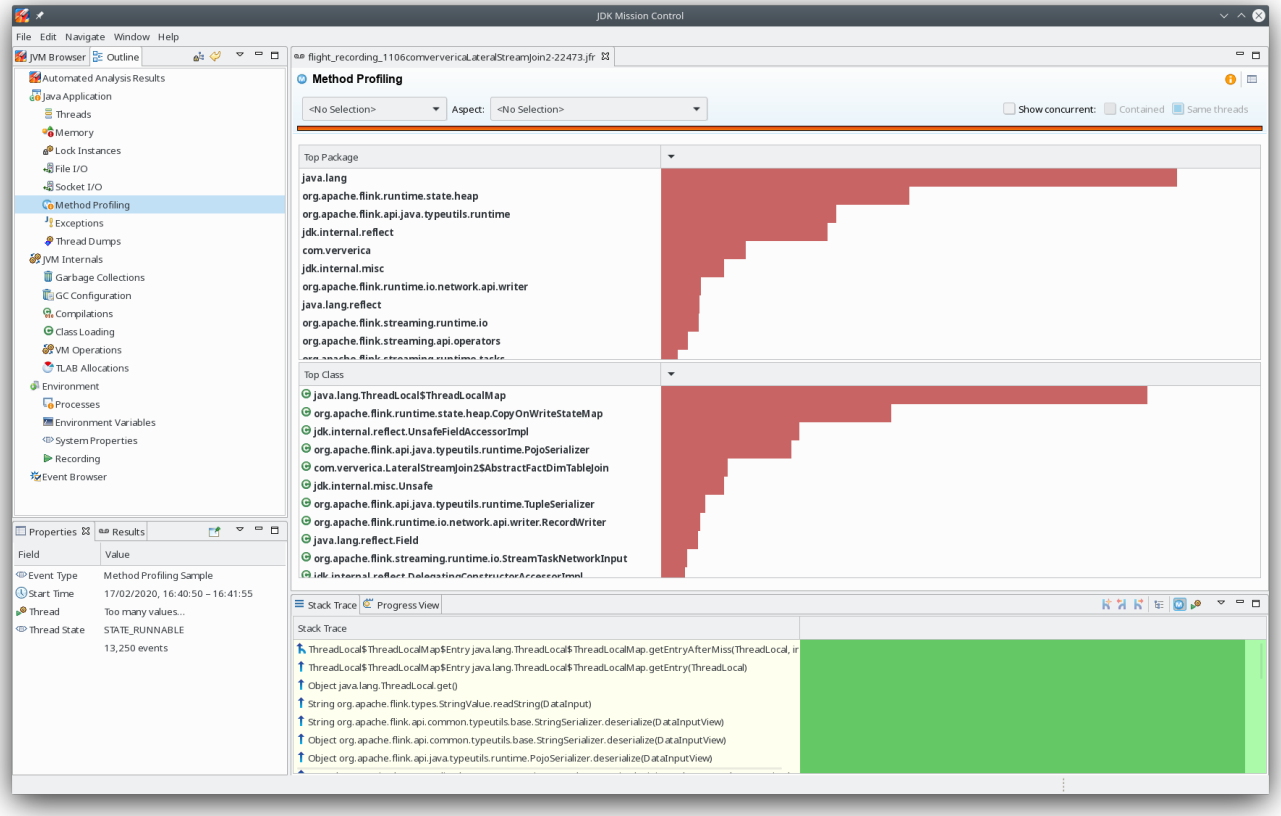
```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironme

DataStream factTableSource = env
    .addSource(new FactTable(factTableRate, maxId))
    .name("fact_table");
DataStream dimTable1Source = env
    .addSource(new DimensionTable(dimTableRate, maxId))
    .name("dim_table1");
// ...

DataStream joinedStream =
    factTableSource
        .keyBy(x -> x.dim1)
        .connect(dimTable1Source.map(new MapDimensionToBinary()).keyBy(x -> x.id)
        .process(
            new AbstractFactDimTableJoin<fact, tuple2<fact,="" byte[]="">>() {
              @Override
              Tuple2<fact, byte[]=""> join(Fact value, BinaryDimension dim) {
                return Tuple2.of(value, dim.data);
              }
```

```
            })
        .name("join1").uid("join1")
// ...
        .keyBy(x -> x.f0.dim5)
        .connect(dimTable5Source.map(new MapDimensionToBinary()).keyBy(x -> x.id)
        .process(
            new AbstractFactDimTableJoin<tuple5<fact, byte[],="" byte[]="">, Deno
              private TypeSerializer dimSerializer;

              @Override
              DenormalizedFact join(Tuple5<fact, byte[],="" byte[]=""> value, Bin
                  throws IOException {
                DataInputDeserializer deserializerBuffer = new DataInputDeseriali
                deserializerBuffer.setBuffer(value.f1);
                Dimension dim1 = dimSerializer.deserialize(deserializerBuffer);
                // ...
                return new DenormalizedFact(value.f0, dim1, dim2, dim3, dim4, dim
              }

              @Override
              public void open(Configuration parameters) throws Exception {
                dimSerializer =
                    TypeInformation.of(Dimension.class)
                        .createSerializer(getRuntimeContext().getExecutionConfig(
              }
            })
        .name("join5").uid("join5");

joinedStream.addSink(new DiscardingSink<>());
env.execute();</fact,></tuple5<fact,></fact,></fact,>
```

尽管技术上不需要此代码，但我们将联接表的反序列化保留到 DenormalizedFact 对象。删除它在比较中是不公平的，甚至在任何实际工作中，您都将继续使用此数据或重新格式化它以获得正确的输出。

使用 profiler 对运行该修改版本的作业做性能分析，可以发现一些不同点：

运行此作业修改版本的探查器快照与其他有很多不同，虽然顶级 ThreadLocal 仍然来自（反）序列化（从 StringValue.readString（）调用），但相当多的 CPU 时间变成了实际业务逻辑（例如 CopyOnWriteStateMap）。



这额外的效果也反映在我们的基准数值上。未启用对象重用，优化后的 DataStream 作业现在大约比 Blink Planner 的 SQL join 快 70%。启用对象重用后，降低了新的 map 算子以及最后一阶段（写入 sink）的开销，并获得了 13% 的提升。而 Blink Planner，开启对象重用后可以获得 53% 的显著提升。

最后，我们优化后的 DataStream 作业比 SQL 的最佳成绩要快 28%。

	Without Object Reuse	With Object Reuse
SQL Join, Old Planner	84,279	88,426
SQL Join, Blink Planner	89,048	136,710
DataStream Join 1	99,025	103,102
DataStream Join 2	154,402	174,766

结论

与 DataStream API 相比，运行此实验展示了 Blink planner 出色的性能，这给我留下了非常深刻的印象。此外，对 Table API 的进一步改进实际上也可以减少差距，特别是在需要用 DataStream API 桥接 Table API 的地方。使用 DataStream API 进一步提高作业的性能，相较于 Blink planner 来说需要更多对序列化栈的思考，以及对特定作业的微调。

本文所提出的技术基本上依赖于尽可能长时间保持序列化形式的数据（当通过网络移交时），并且只在特定时候时反序列化它。这可能可以封装在一些好用的工具方法或基类下，以提高最终代码的可读性和可用性。然而，相

比于 SQL 的简单，当您查看优化所需的代码量时，还必须考虑可维护性和开发成本，从而做出权衡的选择。更重要的是，如果 Blink planner 在将来引入一个专门的维度关联，那么将在底层作用应用这种优化，并且不会用户对 SQL 查询本身进行任何更改。

参考链接：

[1] https://www.ververica.com/blog/getting-started-with-ververica-platform-on-microsoft-azure-part-2
[2] https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/batch/#operating-on-data-objects-in-functions
[3] https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/config.html