深入分析 Flink SQL 工作机制

整理 | 陈婧敏（Flink 社区志愿者）

摘要：本文整理自 Flink Forward 2020 全球在线会议中文精华版，由 Apache Flink PMC 伍翀（云邪）分享，社区志愿者陈婧敏（清樾）整理。旨在帮助大家更好地理解 Flink SQL 引擎的工作原理。文章主要分为以下四部分：

Flink SQL Architecture

How Flink SQL Works?

Flink SQL Optimizations

Summary and Futures

Tips：点击下方链接可查看作者分享的原版视频～
https://ververica.cn/developers/flink-forward-virtual-conference/

Apache Flink 社区在最近的两个版本（1.9 & 1.10 ）中为面向未来的统一流批处理在架构层面做了很多优化，其中一个重大改造是引入了 Blink Planner，开始支持 SQL & Table API 使用不同的 SQL Planner 进行编译（Planner 的插件化）。

本文首先会介绍推动这些优化背后的思考，展示统一的架构如何更好地处理流式和批式查询，其次将深入剖析 Flink SQL 的编译及优化过程，包括：

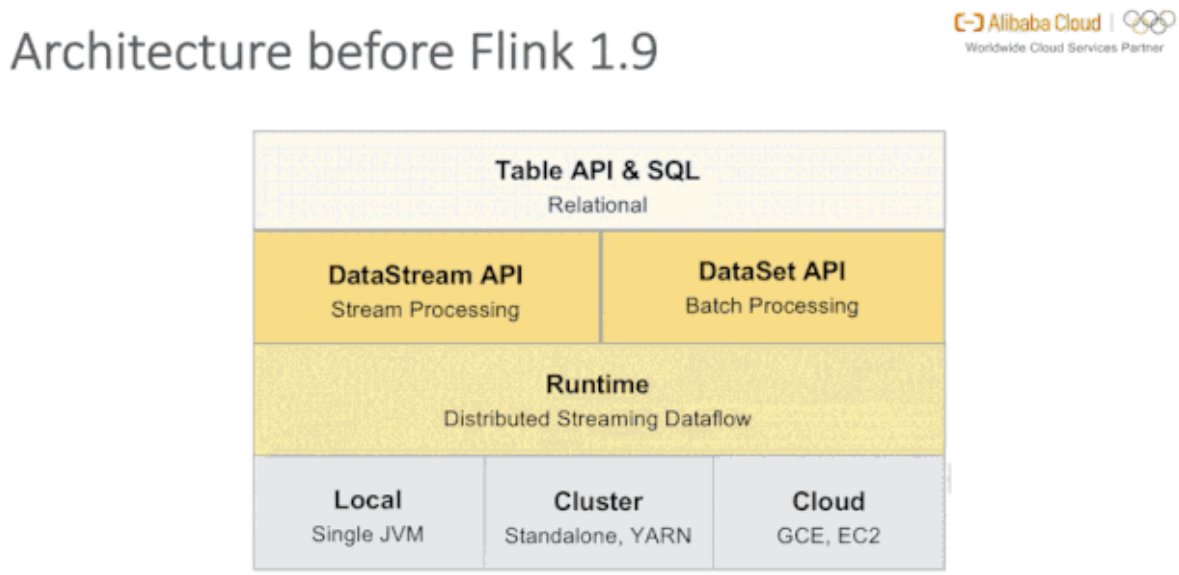
Flink SQL 利用 Apache Calcite 将 SQL 翻译为关系代数表达式，使用表达式折叠（Expression Reduce），下推优化（Predicate / Projection Pushdown ）等优化技术生成物理执行计划（Physical Plan），利用 Codegen 技术生成高效执行代码。

Flink SQL 使用高效的二进制数据存储结构 BinaryRow 加速计算性能；使用 Mini-batch 攒批提高吞吐，降低两层聚合时由 Retraction 引起的数据抖动；聚合场景下数据倾斜处理和 Top-N 排序的优化原理。

Flink SQL 架构 & Blink Planner（1.9+ ）

1.1 Old Planner 的限制

要想了解 Flink SQL 在1.9 版本引入新架构的动机，我们首先看下 1.9 版本之前的架构设计。



从图中可以看出，虽然面向用户的 Table API & SQL 是统一的，但是流式和批式任务在翻译层分别对应了 DataStreamAPI 和 DataSetAPI，在 Runtime 层面也要根据不同的 API 获取执行计划，两层的设计使得整个架构能够复用的模块有限，不易扩展。

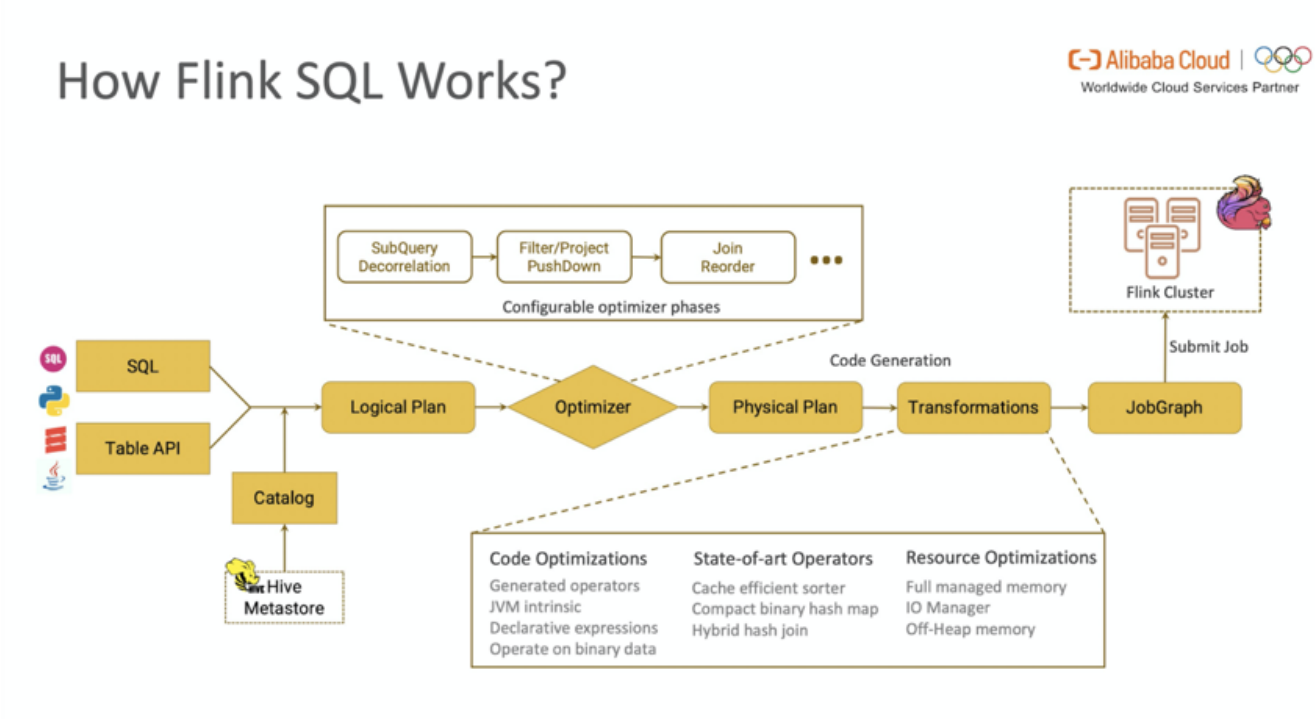
1.2 统一的 Blink Planner

Flink 在设计之初就遵循“批是流的特例”的理念，在架构上做到流批统一是大势所趋。在社区和阿里巴巴的共同努力下，1.9 版本引入了新的 Blink Planner，将批 SQL 处理作为流 SQL 处理的特例，尽量对通用的处理和优化逻辑进行抽象和复用，通过 Flink 内部的 Stream Transformation API 实现流 & 批的统一处理，替代原 Flink Planner 将流 & 批区分处理的方式。

此外，新架构通过灵活的插件化方式兼容老版本 Planner，用户可自行选择。不过在 1.11 版本 Blink Planner 会代替 Old Planner 成为默认的 Planner 来支持流 & 批进一步融合统一（ Old Planner 将在之后逐步退出历史舞台）。

Flink SQL 工作流

Flink SQL 引擎的工作流总结如图所示。



从图中可以看出，一段查询 SQL / 使用TableAPI 编写的程序（以下简称 TableAPI 代码）从输入到编译为可执行的 JobGraph 主要经历如下几个阶段

将 SQL文本 / TableAPI 代码转化为逻辑执行计划（Logical Plan）

Logical Plan 通过优化器优化为物理执行计划（Physical Plan）

通过代码生成技术生成 Transformations 后进一步编译为可执行的 JobGraph 提交运行

本节将重点对 Flink SQL 优化器的常用优化方法和 CodeGen 生成 Transformations 进行介绍。

2.1 Logical Planning

Flink SQL 引擎使用 Apache Calcite SQL Parser 将 SQL 文本解析为词法树，SQL Validator 获取 Catalog 中元数据的信息进行语法分析和验证，转化为关系代数表达式（RelNode），再由 Optimizer 将关系代数表达式转换为初始状态的逻辑执行计划。

备注：TableAPI 代码使用 TableAPI Validator 对接 Catalog 后生成逻辑执行计划。

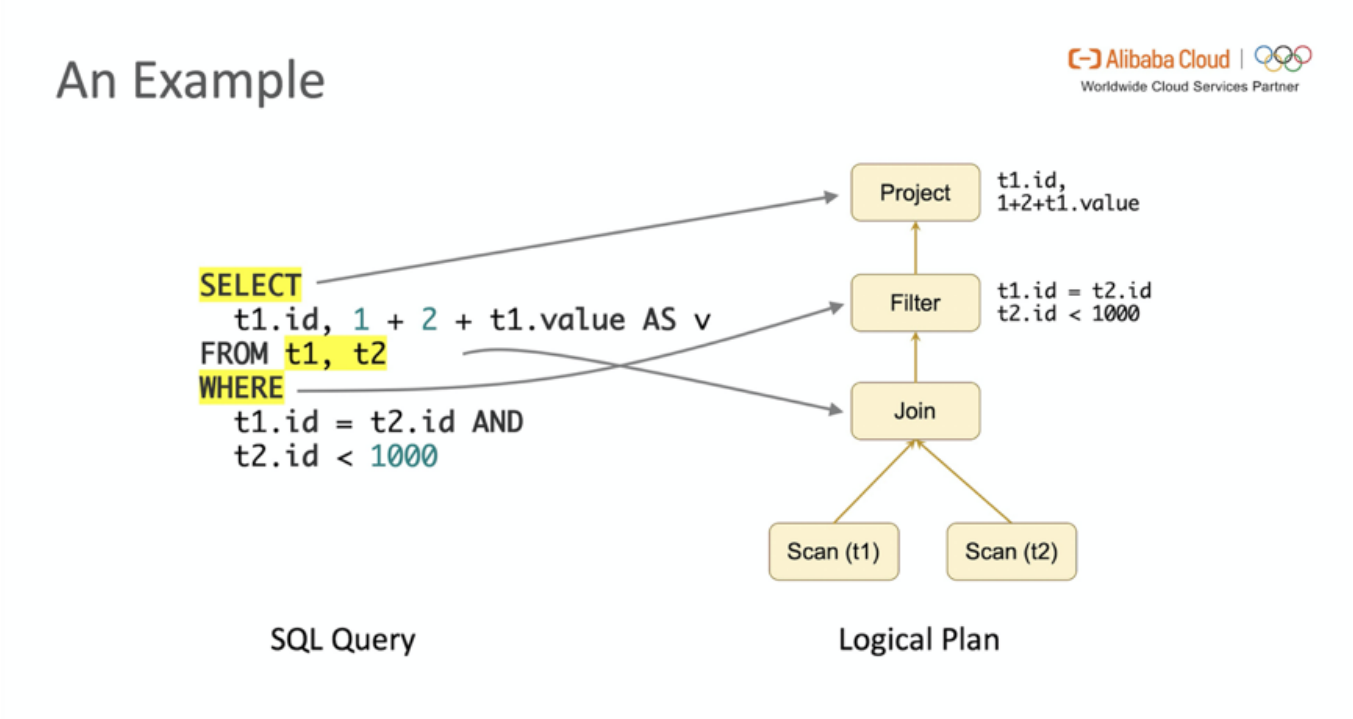
E.g.1 考虑如下表达 JOIN 操作的一段 SQL。

```
SELECT
  t1.id, 1 + 2 + t1.value AS v
FROM t1, t2
WHERE
  t1.id = t2.id AND
  t2.id < 1000
```

经过上述操作后得到了一个树状结构的逻辑执行计划，根节点对应最上层的 Select 语句，叶子节点对应输入表 t1 和 t2 的 TableScan 操作，Join 和 Where 条件过滤 分别对应了 Join 和 Filter 节点。

```
LogicalProject(id=[$0], v=[+(+(1, 2), $1)])
+- LogicalFilter(condition=[AND(=($0, $3), <($3, 1000))])
   +- LogicalJoin(condition=[true], joinType=[inner])
      :- LogicalTableScan(table=[[default_catalog, default, t1]])
      +- LogicalTableScan(table=[[default_catalog, default, t2]])
```

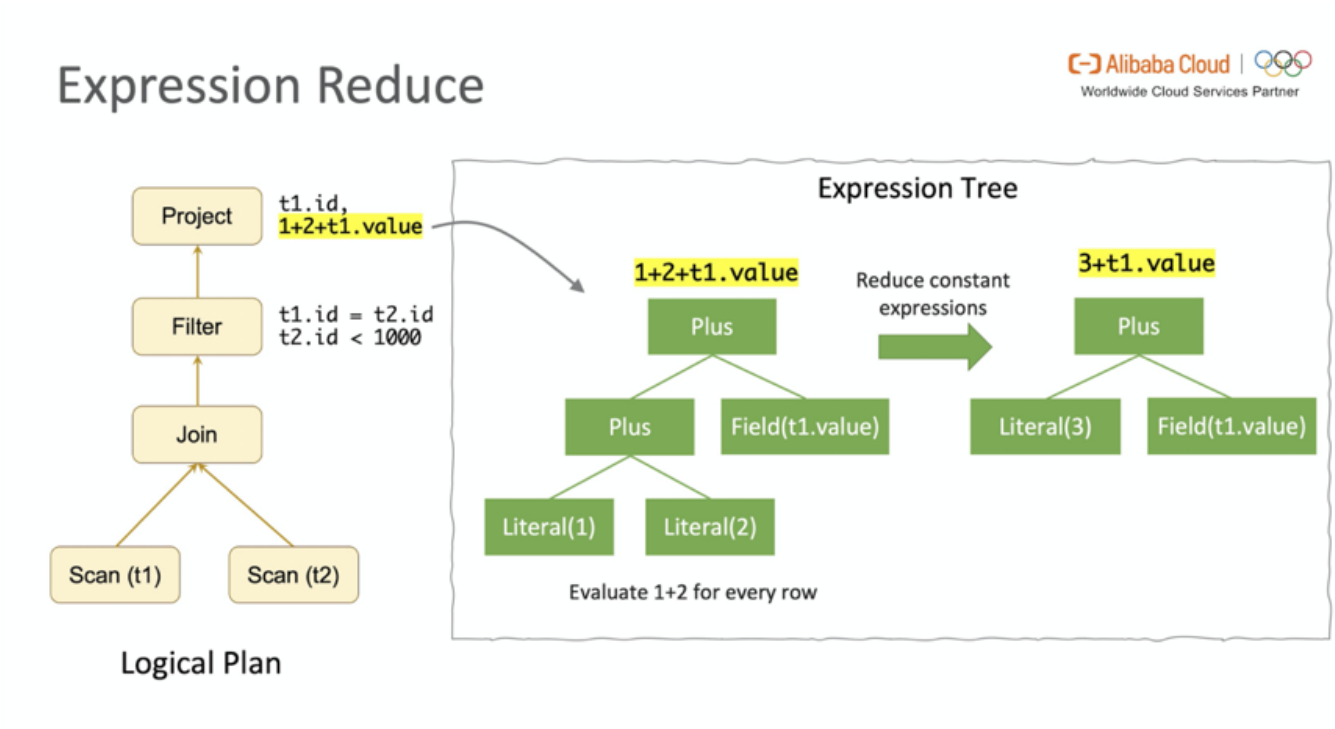
可视化后如图所示，这是优化器开始工作的初始状态。



下面开始介绍 Flink SQL 优化器常见的几种优化方式。

■ 2.1.1 Expression Reduce

表达式（Expression） 是 SQL 中最常见的语法。比如 t1.id 是一个表达式， 1 + 2 + t1.value 也是一个表达式。优化器在优化过程中会递归遍历树上节点，尽可能预计算出每个表达式的值，这个过程就称为表达式折叠。这种转换在逻辑上等价，通过优化后，真正执行时不再需要为每一条记录都计算一遍 1 + 2。



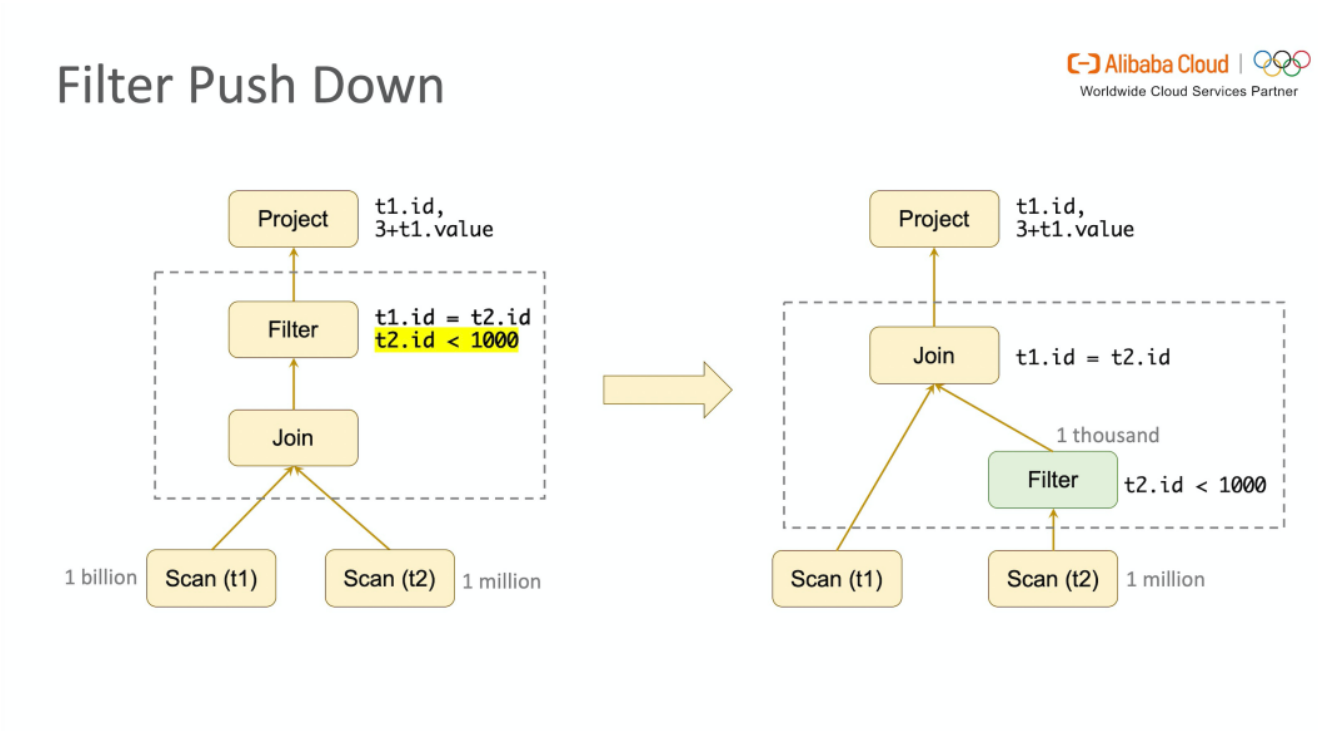
■ 2.1.2 PushDown Optimization

下推优化是指在保持关系代数语义不变的前提下将 SQL 语句中的变换操作尽可能下推到靠近数据源的位置以获得更优的性能，常见的下推优化有谓词下推（Predicate Pushdown），投影下推（Projection Pushdown，有时也译作列裁剪）等。

Predicate Pushdown

回顾 E.g.1，我们发现 WHERE 条件表达式中 t2.id < 1000 这个过滤条件描述的是对表 t2 的约束，跟表 t1 无关，完全可以下推到 JOIN 操作之前完成。假设表 t2 中有一百万行数据，但是满足 id < 1000 的数据只有 1,000 条，则通过谓词下推优化后到达 JOIN 节点的数据量降低了1,000 倍，极大地节省了 I / O 开销，提升了 JOIN 性能。

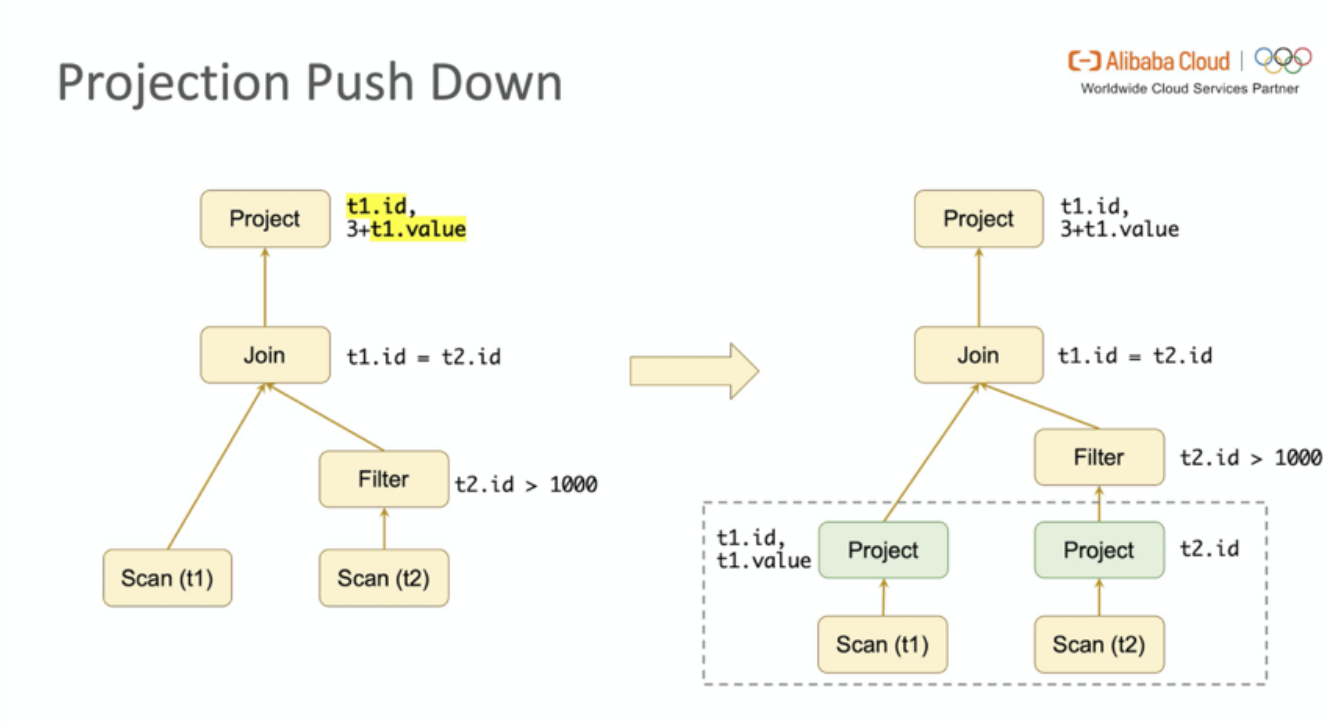
谓词下推（Predicate Pushdown）是优化 SQL 查询的一项基本技术，谓词一词来源于数学，指能推导出一个布尔返回值（TRUE / FALSE）的函数或表达式，通过判断布尔值可以进行数据过滤。谓词下推是指保持关系代数语义不变的前提下将 Filter 尽可能移至靠近数据源的位置（比如读取数据的 SCAN 阶段）来降低查询和传递的数据量（记录数）。



Projection Pushdown

列裁剪是 Projection Pushdown 更直观的描述方式，指在优化过程中去掉没有使用的列来降低 I / O 开销，提升性能。但与谓词下推只移动节点位置不同，投影下推可能会增加节点个数。比如最后计算出的投影组合应该放在 TableScan 操作之上，而 TableScan 节点之上没有 Projection 节点，优化器就会显式地新增 Projection 节点来完成优化。另外如果输入表是基于列式存储的（如 Parquet 或 ORC 等），优化还会继续下推到 Scan 操作中进行。

回顾 E.g.1，我们发现整个查询中只用到了表 t1 的 id 和 value 字段，表 t2 的 id 字段，在 TableScan 节点之上分别增加 Projection 节点去掉多余字段，极大地节省了 I / O 开销。

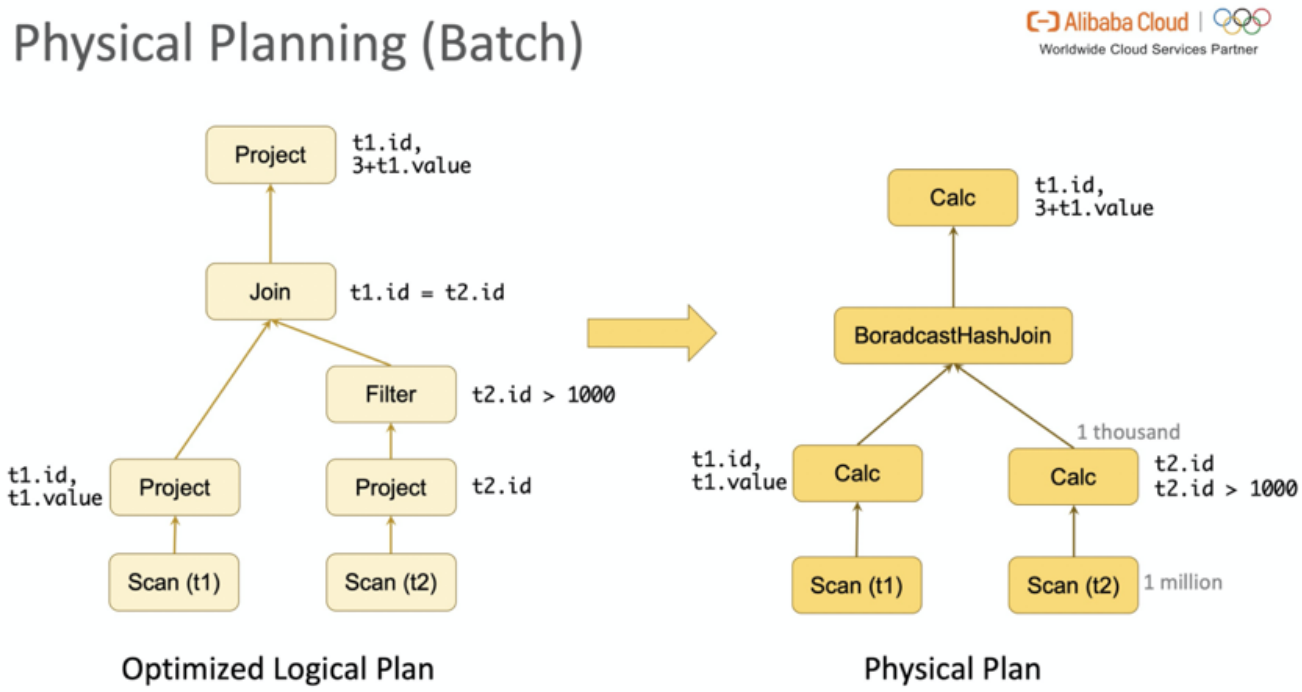


简要总结一下，谓词下推和投影下推分别通过避免处理不必要的记录数和字段数来降低 I / O 开销提升性能。

2.2 Physical Planning on Batch

通过上述一系列操作后，我们得到了优化后的逻辑执行计划。逻辑执行计划描述了执行步骤和每一步需要完成的操作，但没有描述操作的具体实现方式。而物理执行计划会考虑物理实现的特性，生成每一个操作的具体实现方式。比如 Join 是使用 SortMergeJoin、HashJoin 或 BroadcastHashJoin 等。优化器在生成逻辑执行计划时会计算整棵树上每一个节点的 Cost，对于有多种实现方式的节点（比如 Join 节点），优化器会展开所有可能的 Join 方式分别计算。最终整条路径上 Cost 最小的实现方式就被选中成为 Final Physical Plan。

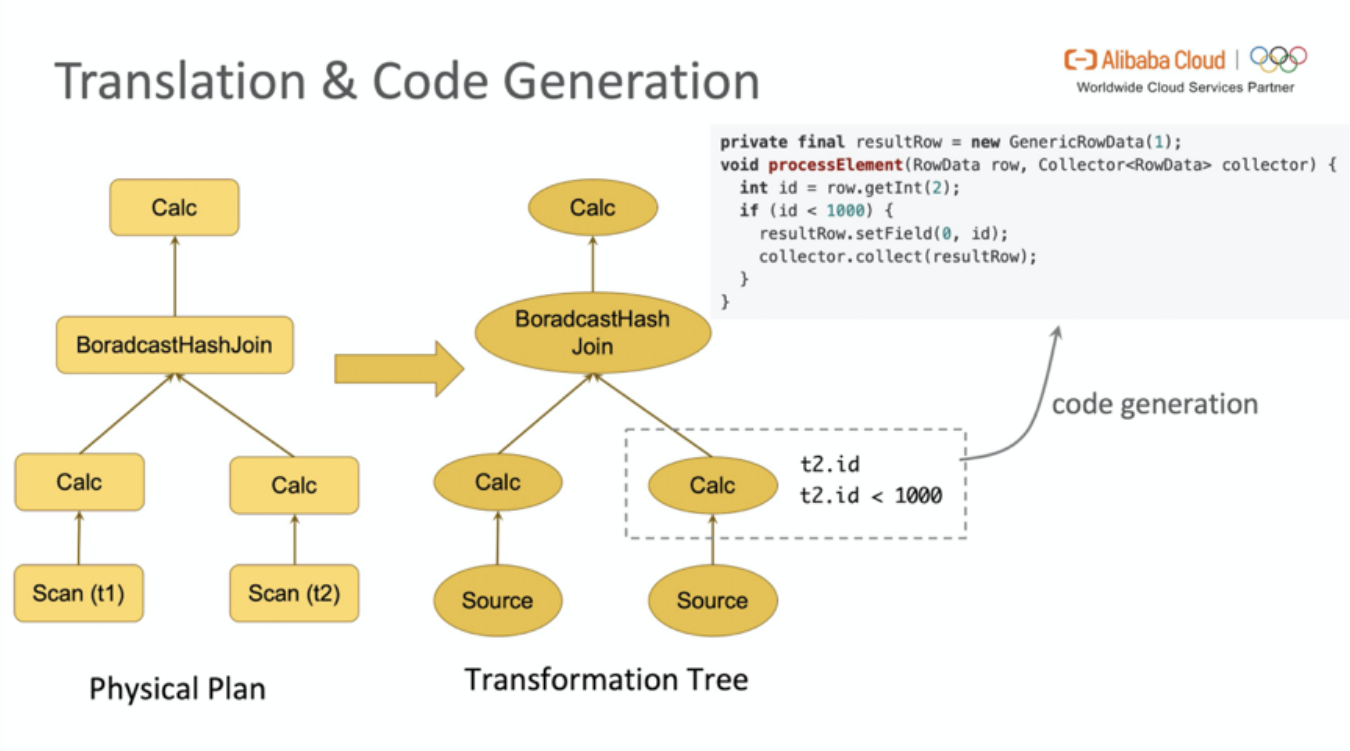
回顾 E.g.1，当它以批模式执行，同时我们可以拿到输入表的 Statistics 信息。在经过前述优化后，表 t2 到达 Join 节点时只有 1,000 条数据，使用 BroadcastJoin 的开销相对最低，则最终的 Physical Plan 如下图所示。



2.3 Translation & Code Generation

代码生成（Code Generation） 在计算机领域是一种广泛使用的技术。在 Physical Plan 到生成 Transformation Tree 过程中就使用了 Code Generation。

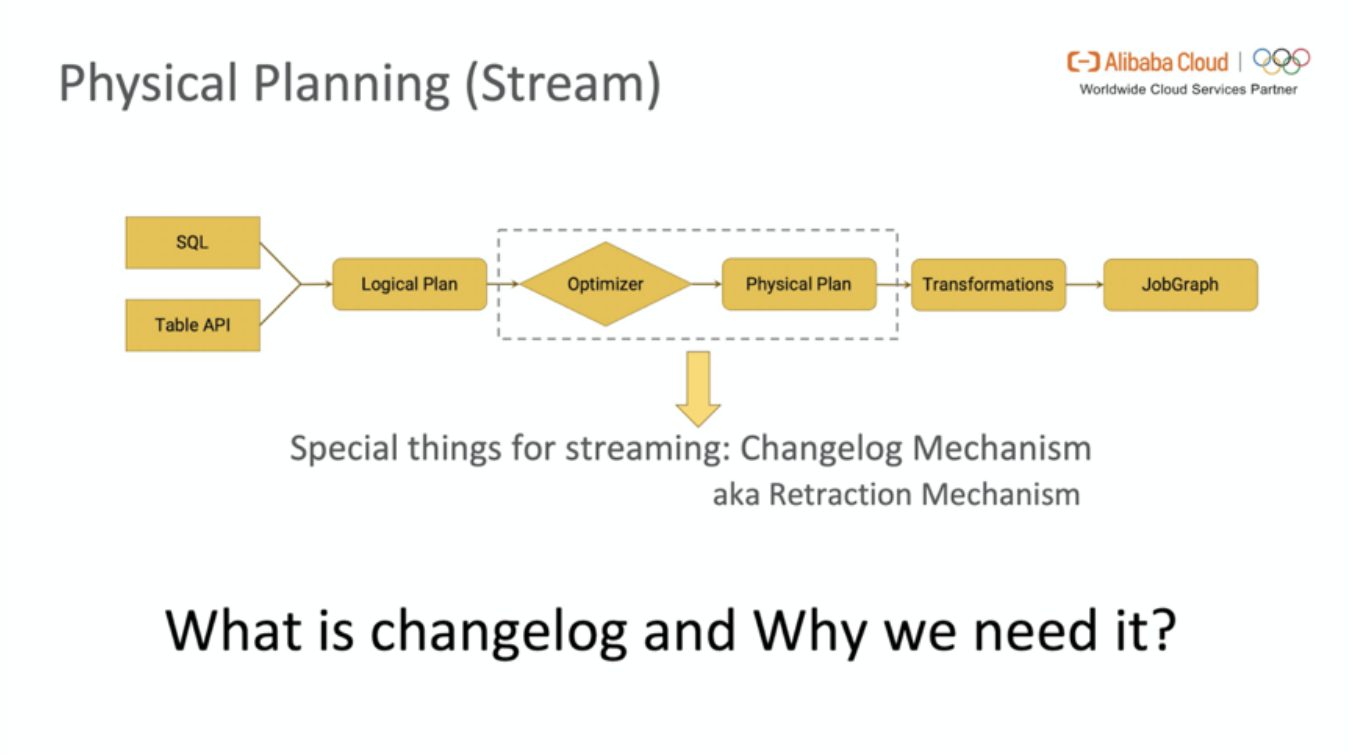
回顾 E.g.1，以 表 t2 之上的 Calc 节点 t2.id < 1000 表达式为例，通过 Code Generation 后生成了描述 Transformation Operator 的一段 Java 代码，将接收到的 Row 中 id < 1000 的 Row 发送到下一个 Operator。



Flink SQL 引擎会将 Physical Plan 通过 Code Generation 翻译为 Transformations，再进一步编译为可执行的 JobGraph。

2.4 Physical Planning on Stream

以上介绍了 Flink SQL 引擎的整体工作流，上述例子是假定以批模式编译的，下面我们来介绍一下以流模式编译时，在生成 Physical Plan 过程中的一个重要机制：Retraction Mechanism （aka. Changelog Mechanism）。



■ 2.4.1 Retraction Mechanism

Retraction 是流式数据处理中撤回过早下发（Early Firing）数据的一种机制，类似于传统数据库的 Update 操作。级联的聚合等复杂 SQL 中如果没有 Retraction 机制，就会导致最终的计算结果与批处理不同，这也是目前业界很多流计算引擎的缺陷。

E.g.2 考虑如下统计词频分布的 SQL。

```
SELECT cnt, COUNT(cnt) as freq
FROM (
  SELECT word, COUNT(*) as cnt
  FROM words
  GROUP BY word)
GROUP BY cnt
```

假设输入数据是：

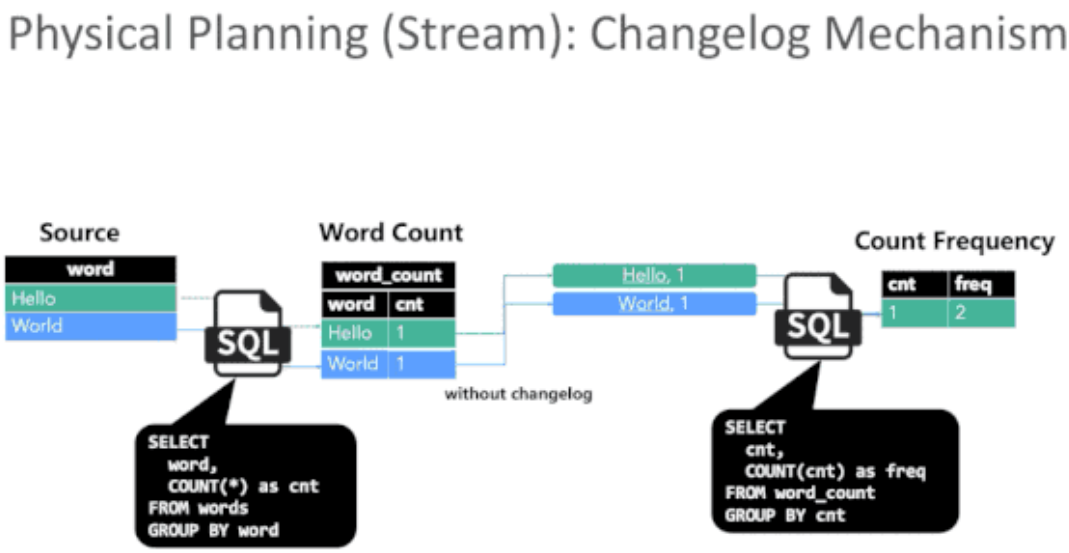
word
Hello

World
Hello

则经过上面的计算后，预期的输出结果应该是：

cnt	freq
1	1
2	1

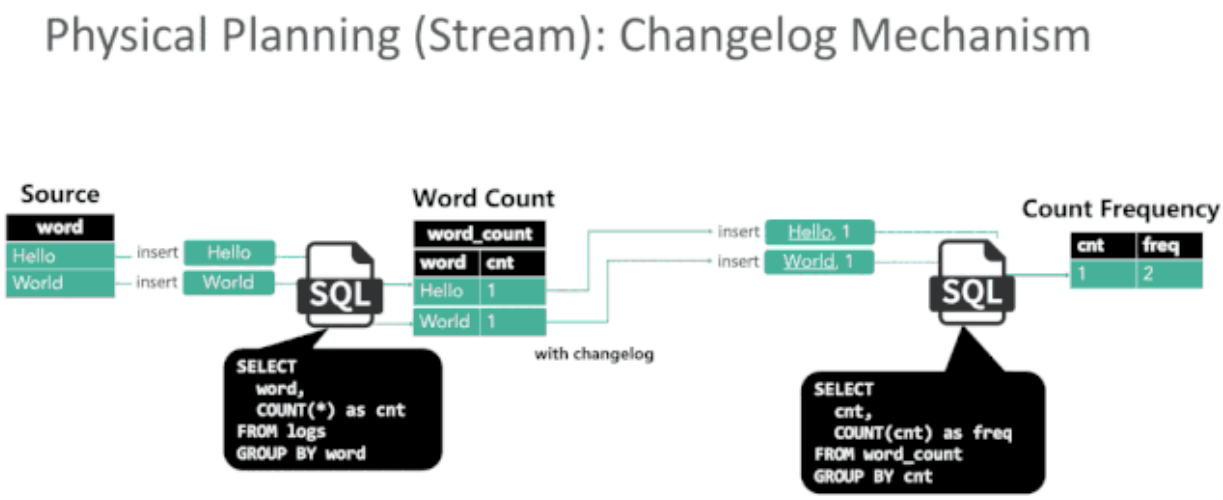
但与批处理不同，流处理的数据是一条条到达的，理论上每一条数据都会触发一次计算，所以在处理了第一个 Hello 和第一个 World 之后，词频为 1 的单词数已经变成了 2，此时再处理第二个 Hello 时，如果不能修正之前的结果，Hello 就会在词频等于 1 和词频等于 2 这两个窗口下被同时统计，显然这个结果是错误的，这就是没有 Retraction 机制带来的问题。



Flink SQL 在流计算领域中的一个重大贡献就是首次提出了这个机制的具体实现方案。Retraction 机制又名 Changelog 机制，因为某种程度上 Flink 将输入的流数据看作是数据库的 Changelog，每条输入数据都可以看作是对数据库的一次变更操作，比如 Insert，Delete 或者 Update。以 MySQL 数据库为例，其Binlog 信息以二进制形式存储，其中 Update_rows_log_event 会对应 2 条标记 Before Image （BI） 和 After Image （AI），分别表示某一行在更新前后的信息。

在 Flink SQL 优化器生成流作业的 Physical Plan 时会判断当前节点是否是更新操作，如果是则会同时发出 2 条消息 update_before 和 update_after 到下游节点，update_before 表示之前“错误”下发的数据，需要被撤回，update_after 表示当前下发的“正确”数据。下游收到后，会在结果上先减去 update_before，再加上 update_after。

回顾 E.g.2，下面的动图演示了加入 Retraction 机制后正确结果的计算过程。



update_before 是一条非常关键的信息，相当于标记出了导致当前结果不正确的那个“元凶”。不过额外操作会带来额外的开销，有些情况下不需要发送 update_before 也可以获得正确的结果，比如下游节点接的是 UpsertSink（MySQL 或者 HBase的情况下，数据库可以按主键用 update_after 消息覆盖结果）。是否发送 update_before 由优化器决定，用户不需要关心。

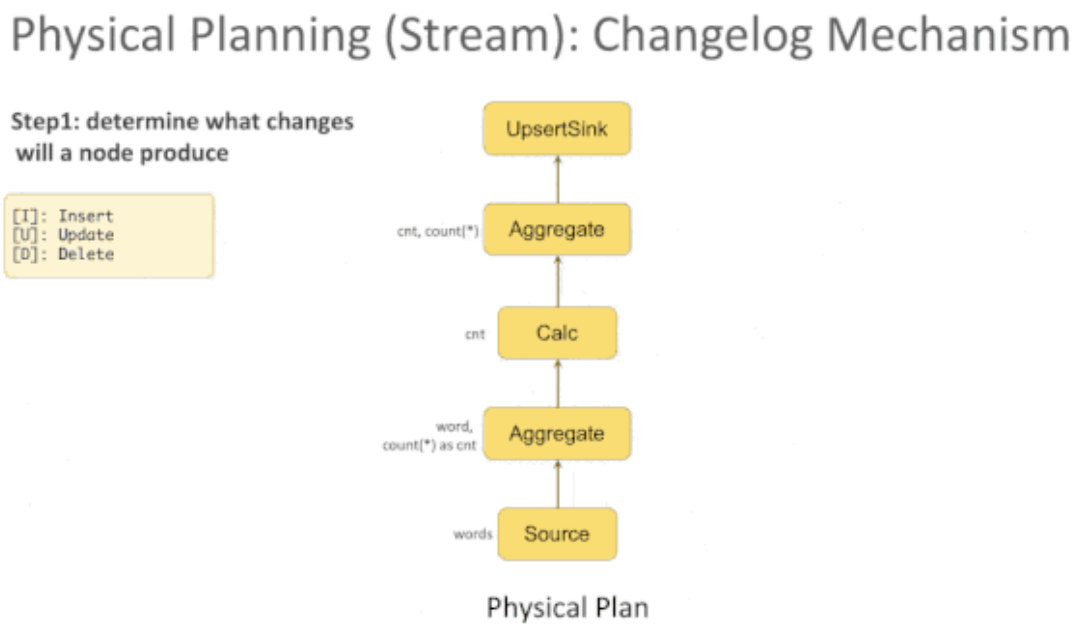
■ 2.4.2 Update_before Decision

前面介绍了 Retraction 机制和 update_before，那优化器是怎样决定是否需要发送 update_before 呢？本节将介绍这一部分的工作。

Step1：确定每个节点对应的 Changelog 变更类型

数据库中最常见的三种操作类型分别是 Insert （记为 [I]），Delete（记为 [D]），Update（记为 [U]）。优化器首先会自底向上检查每个节点，判断它属于哪（几）种类型，分别打上对应标记。

回顾 E.g.2，第一个 Source 节点由于只产生新数据，所以属于 Insert，记为 [I]；第二个节点计算内层的聚合，所以会发出更新的消息，记为 [I，U]；第三个节点裁掉 word 字段，属于简单计算，传递了上游的变更类型，记为 [I，U]；第四个节点是外层的聚合计算，由于它收到了来自上游的 Update 消息，所以额外需要 Delete 操作来保证更新成功，记为 [I，U，D]。



Step2：确定每个节点发送的消息类型

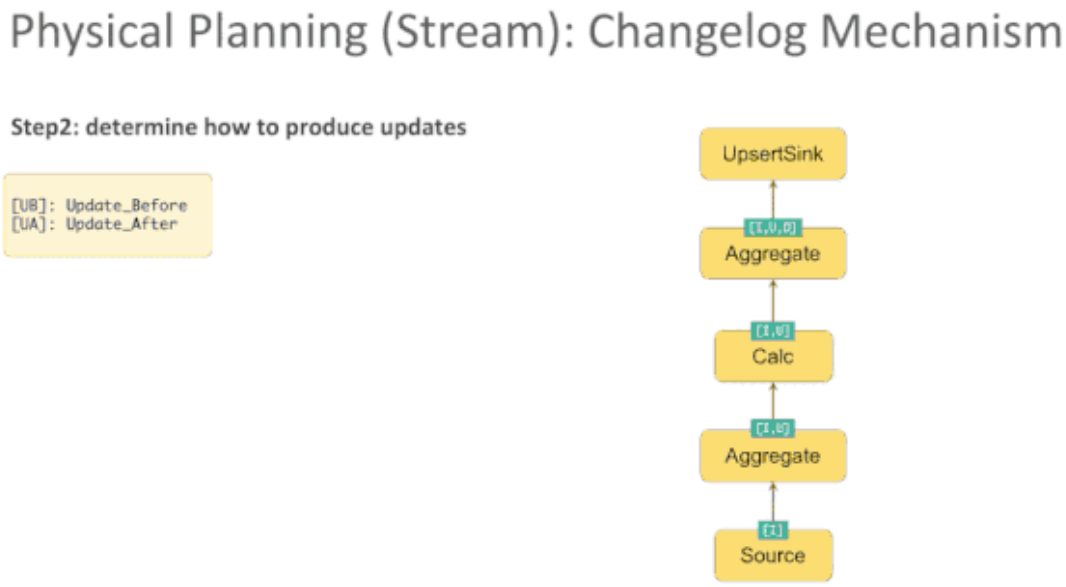
在介绍 Step2 之前，我们先介绍下 Flink 中 Update 消息类型的表示形式。在 Flink 中 Update 由两条 update_before（简称 UB）和 update_after （简称 UA）来表示，其中 UB 消息在某些情况下可以不发送，从而提高性能。

在 Step1 中优化器自底向上推导出了每个节点对应的 Changelog 变更操作，这一步里会先自顶向下推断当前节点需要父节点提供的消息类型，直到遇到第一个不需要父节点提供任何消息类型的节点，再往上回推每个节点最终的实现方式和需要的消息类型。

回顾 E.g.2，由于最上层节点是 UpsertSink 节点，只需要它的父节点提供 [UA] 即可。到了外层聚合的 Aggregate 节点，由于 Aggregate 节点的输入有 Update 操作，所以需要父节点需要提供 [UB，UA]，这样才能正确更新自己的计算状态。

再往下到 Calc 节点，它需要传递 [UB，UA] 的需求给它的父节点，也就是内层的 Aggregate 节点。而到了内层 Aggregation 节点，它的父节点是 Source 节点，不会产生 Update 操作，所以它不需要 Source 节点额外发送任何 [UB / UA]。当优化器遍历到 Source 节点，便开始进行回溯，如果当前节点能满足子节点的 requirement，则将对应的标签更新到节点上，否则便无法生成 plan。首先内层的 Aggregate 能产生 UB，所以能满足子节点的 requirement，所以优化器会给内层的 Aggregate 节点打上 [UB，UA] 的标签，然后向上传递到 Calc 节点，同样打上 [UB，UA] ，再到外层的 Aggregate 节点，由于它的下游只需要接受更新后的消息，所以打上 [UA] 标签，表示它只需要向下游发送 update_after 即可。

这些标签最终会影响算子的物理实现，比如外层的 Aggregate 节点，由于它会接收到来自上游的 [UB]，所以物理实现会使用带 Retract 的 Count，同时它只会向 Sink 发送 update_after。而内层的 Aggregate 节点，由于上游发送过来的数据没有 [UB]，所以可以采用不带 Retract 的 Count 实现，同时由于带有 [UB] 标签，所以需要往下游发送 update_before。

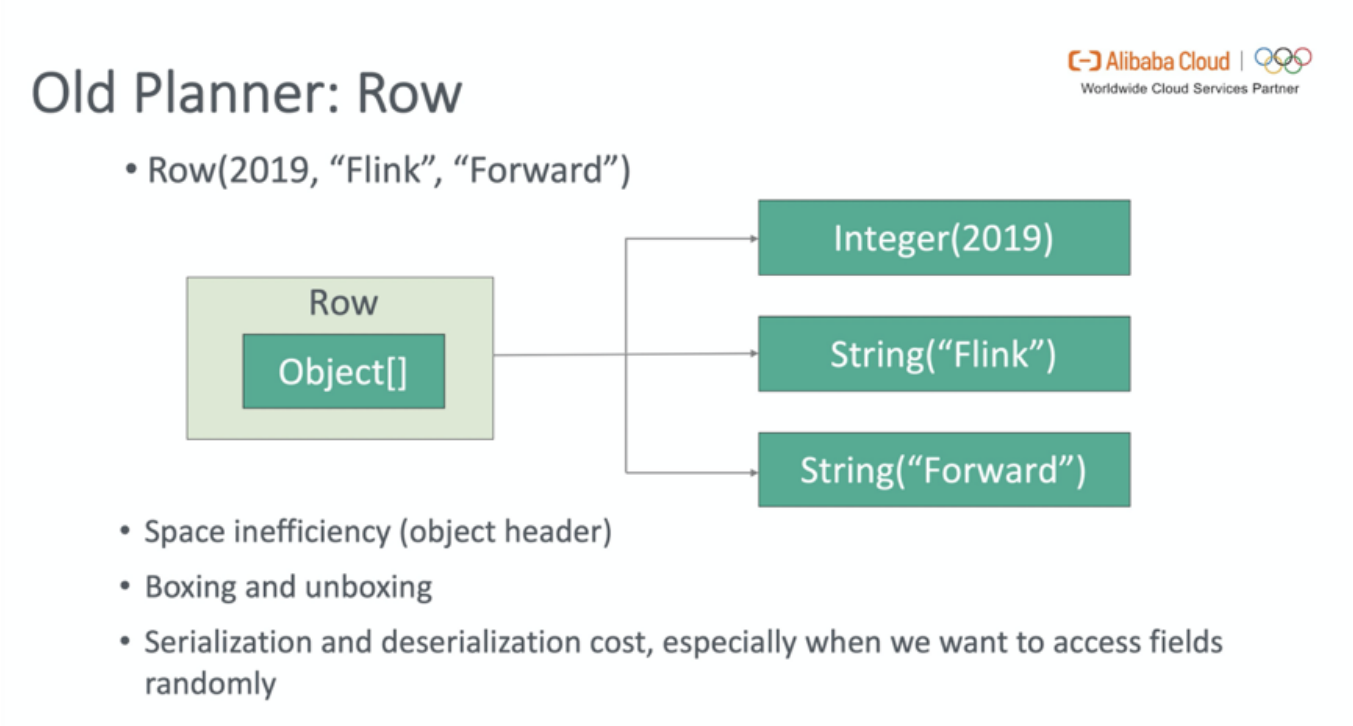


Flink SQL Internal Optimization

前面介绍了 Flink SQL 引擎的工作原理，接下来会简要概括一下 Flink SQL 内部的一些优化，更多资料可以在 Flink Forward Asia 2019 查看。

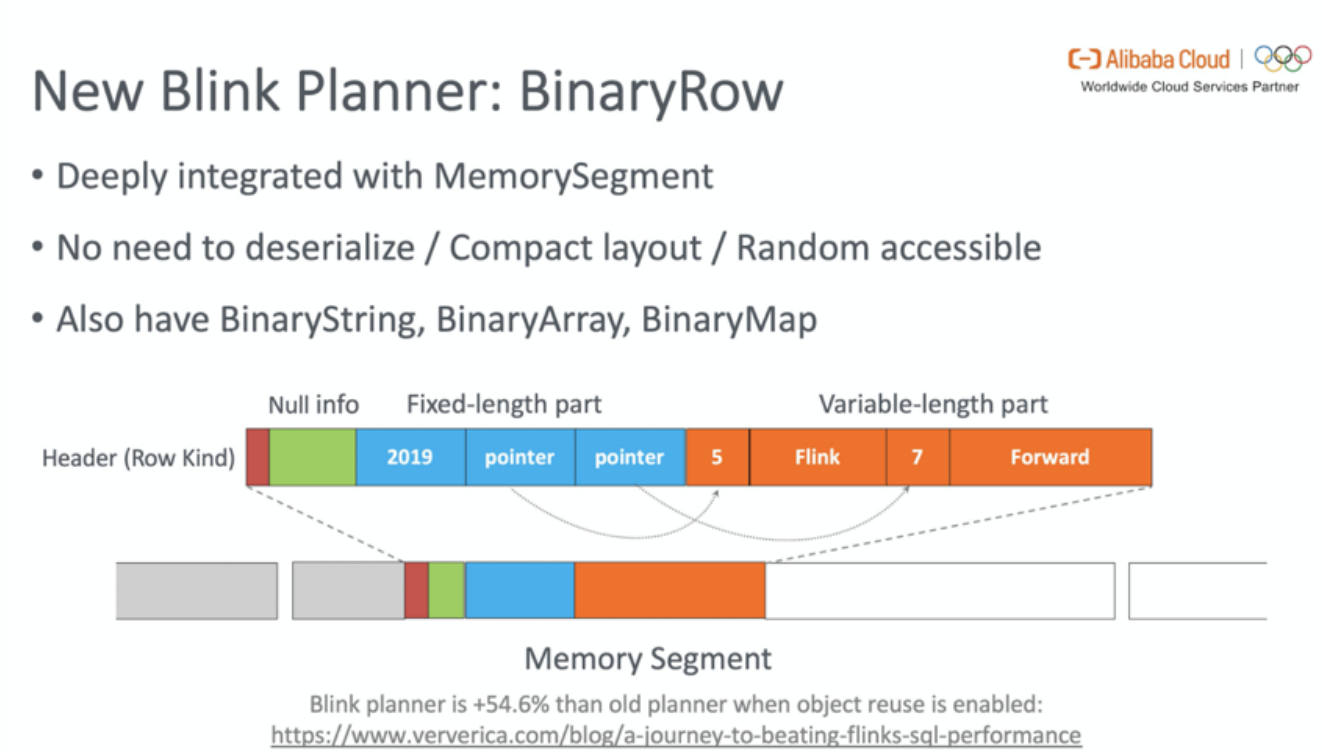
3.1 BinaryRow

在 Flink 1.9+ 前， Flink Runtime 层各算子间传递的数据结构是 Row，其内部实现是 Object[]。这种数据结构的问题在于不但需要额外开销存 Object Metadata，计算过程中还涉及到大量序列化 / 反序列 （特别是只需要处理某几个字段时需要反序列化整个 Row），primitive 类型的拆 / 装箱等，都会带来大量额外的性能开销。



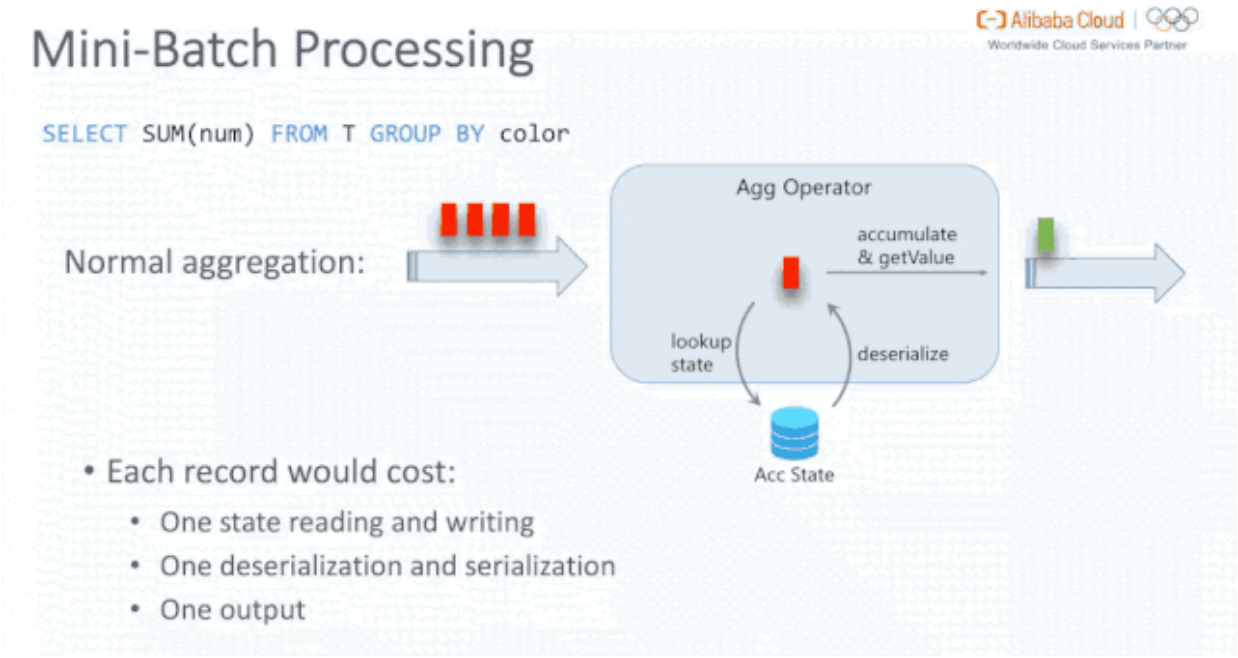
Flink 1.9 开始引入了 Blink Planner，使用二进制数据结构的 BinaryRow 来表示 Record。BinaryRow 作用于默认大小为 32K 的 Memory Segment，直接映射到内存。BinaryRow 内部分为 Header，定长区和变长区。Header 用于存储 Retraction 消息的标识，定长区使用 8 个 bytes 来记录字段的 Nullable 信息及所有 primitive 和可以在 8 个 bytes 内表示的类型。其它类型会按照基于起始位置的 offset 存放在变长区。

BinaryRow 作为 Blink Planner 的基础数据结构，带来的好处是显而易见的：首先存储上更为紧凑，去掉了额外开销；其次在序列化和反序列化上带来的显著性能提升，可根据 offset 只反序列化需要的字段，在开启 Object Reuse 后，序列化可以直接通过内存拷贝完成。



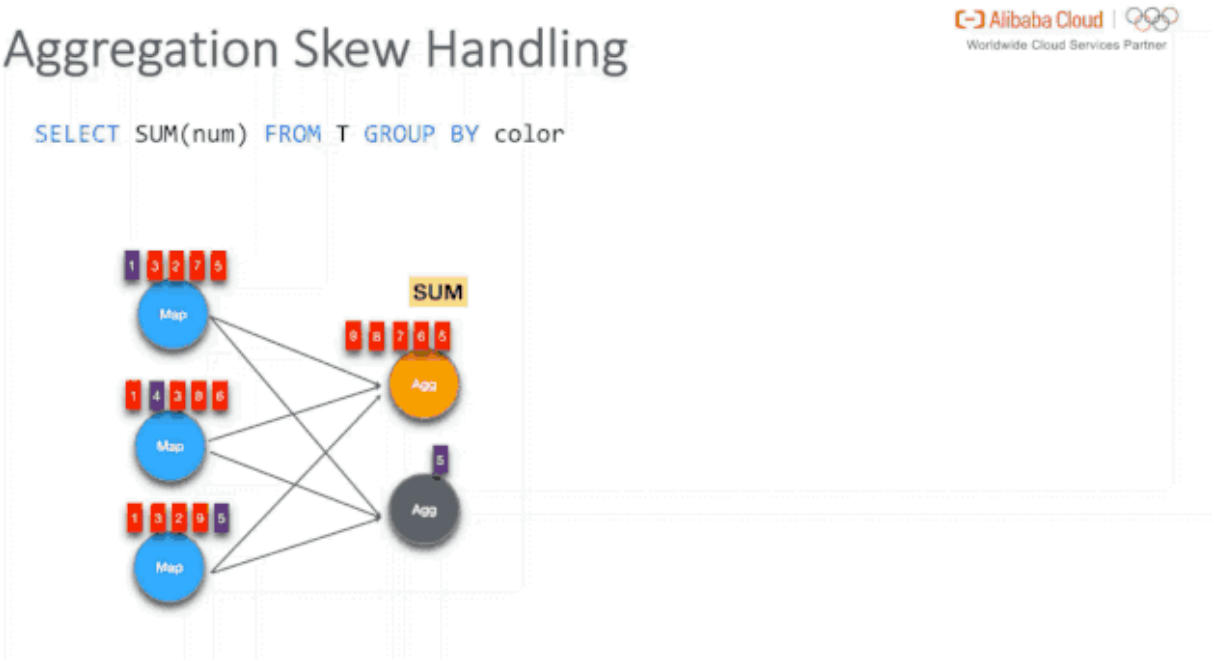
3.2 Mini-batch Processing

Flink 是纯流式处理框架，在理论上每一条新到的数据都会触发一次计算。然而在实现层面，这样做会导致聚合场景下每处理一条数据都需要读写 State 及序列化 / 反序列化。如果能够在内存中 buffer 一定量的数据，预先做一次聚合后再更新 State，则不但会降低操作 State 的开销，还会有效减少发送到下游的数据量，提升 throughput，降低两层聚合时由 Retraction 引起的数据抖动, 这就是 Mini-batch 攒批优化的核心思想。



3.3 Skew Processing

对于数据倾斜的优化，主要分为是否带 DISTINCT 去重语义的两种方式。对于普通聚合的数据倾斜，Flink 引入了 Local-Global 两阶段优化，类似于 MapReduce 增加 Local Combiner 的处理模式。而对于带有去重的聚合，Flink 则会将用户的 SQL 按原有聚合的 key 组合再加上 DISTINCT key 做 Hash 取模后改写为两层聚合来进行打散。



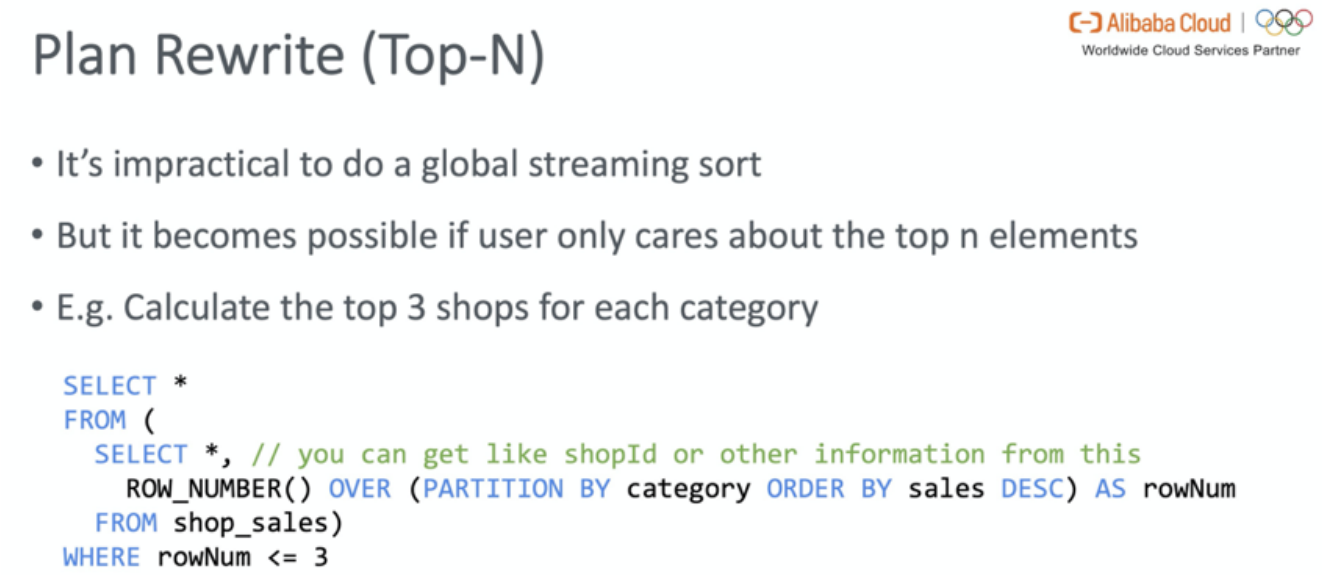
3.4 Top-N Rewrite

全局排序在流式的场景是很难实现的，但如果只需要计算到目前的 Top-N 极值，问题就

变得可解。不过传统数据库求排序的 SQL 语法是通过 ORDER BY 加 LIMIT 限制条数，背后实现的机制也是通过扫描全表排序后再返回 LIMIT 条数的记录。另外如果按照某些字段开窗排序，ORDER BY 也无法满足要求。Flink SQL 借鉴了批场景下开窗求 Top-N 的语法，使用 ROW_NUMBER 语法来做流场景下的 Top-N 排序。

E.g.3 下面这段 SQL 计算了每个类目下销量 Top3 的店铺

```
SELECT*
FROM(
  SELECT *, -- you can get like shopId or other information from this
    ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) AS rowNum
  FROM shop_sales )
WHERE rowNum <= 3
```



在生成 Plan 方面，ROW_NUMBER 语义对应 OverAggregate 窗口节点和一个过滤行数的 Calc 节点，而这个窗口节点在实现层面需要为每一个到达的数据重新将 State 中的历史数据拿出来排序，这显然不是最优解。

我们知道流式场景求解极大 / 小值的最优操作是通过维护一个 size 为 N 的 minHeap / maxHeap。由实现反推出我们需要在优化器上新增一条规则，在遇到 ROW_NUMBER

生成的逻辑节点后，将其优化为一个特殊的 Rank 节点，对应上述的最优实现方式（当然这只是特殊 Rank 对应的其中一种实现）。这便是 Top-N Rewrite 的核心思想。

Summary & Futures

本文内容回顾

1. 简要介绍 Flink 1.9 + 在 SQL & TableAPI 上引入新架构，统一技术栈，朝着流 & 批一体的方向迈进了一大步。

2. 深入介绍 Flink SQL 引擎的内部运行机制，以及在对用户透明的同时，Flink SQL 在优化方面做的许多工作。

未来工作计划

> 在 Flink 1.11+ 后的版本，Blink Planner 将作为默认的 Planner 提供生产级别的支持。
>
> FLIP-95：重构 TableSource & TableSink 的接口设计，面向流批一体化，在 Source 端支持 changelog 消息流，从而支持 FLIP-105 的 CDC 数据源。
>
> FLIP-105：Flink TableAPI & SQL 对 CDC 的支持。

FLIP-115：扩展目前只支持 CSV 的 FileSystem Connector，使其成为流批统一的 Generalized FileSystem Connector。

FLIP-123：对 Hive DDL 和 DML 的兼容，支持用户在 Flink 中运行 Hive DDL。