Flink SQL 的 9 个示例

作者：贺小令（晓令）

> 本文由阿里巴巴技术专家贺小令分享，社区志愿者郑仲尼整理。文章基于 Flink 1.9 版本，从用户的角度来讲解 Flink 1.9 版本中 SQL 相关原理及部分功能变更，希望对大家有所帮助。主要内容分享以下三个部分:
>
> 1. TableEnvironment 的设计与使用场景
> 2. Catalog 的设计以及 DDL 实践
> 3. Blink Planner 的几点重要改进及优化

TableEnvironment

FLIP-32 中提出，将 Blink 完全开源，合并到 Flink 主分支中。合并后在 Flink 1.9 中会存在两个 Planner：Flink Planner 和 Blink Planner。

在之前的版本中，Flink Table 在整个 Flink 中是一个二等公民。而 Flink SQL 具备的易用性、使用门槛低等特点深受用户好评，越来越被重视，Flink Table 模块也因此被提升为一等公民。而 Blink 在设计之初就考虑到流和批的统一，批只是流的一种特殊形式，所以可以用同一个TableEnvironment来表述流和批。

TableEnvironment 整体设计

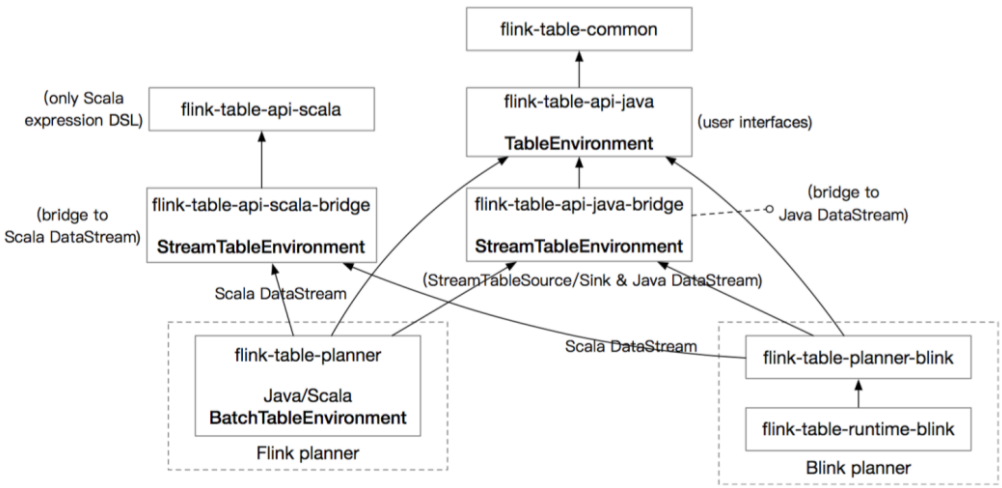

图1 新 Table Environment 整体设计

从图 1 中，可以看出，TableEnvironment 组成部分如下：

- **flink-table-common**：这个包中主要是包含 Flink Planner 和 Blink Planner 一些共用的代码。
- **flink-table-api-java**：这部分是用户编程使用的 API，包含了大部分的 API。
- **flink-table-api-scala**：这里只是非常薄的一层，仅和 Table API 的 Expression 和 DSL 相关。

- **两个 Planner**：flink-table-planner 和 flink-table-planner-blink。
- **两个 Bridge**：flink-table-api-scala-bridge 和 flink-table-api-java-bridge，从图中可以看出，Flink Planner 和 Blink Planner 都会依赖于具体的 JAVA API，也会依赖于具体的 Bridge，通过 Bridge 可以将 API 操作相应的转化为 Scala 的 DataStream、DataSet，或者转化为 JAVA 的 DataStream 或者 Data Set。

新旧 TableEnvironment 对比

在 Flink 1.9 之前，原来的 Flink Table 模块，有 7 个 Environment，使用和维护上相对困难。7 个 Environment 包括：StreamTableEnvironment、BatchTableEnvironment 两类，JAVA 和 Scala 分别 2 个，一共 4 个，加上 3 个父类，一共就是 7 个。

在新的框架之下，社区希望流和批统一，因此对原来的设计进行精简。首先，提供统一的 TableEnvironment，放在 flink-table-api-java 这个包中。然后，在 Bridge 中，提供了两个用于衔接 Scala DataStream 和 Java DataStream 的 StreamTableEnvironment。最后，因为 Flink Planner 中还残存在着 toDataSet() 类似的操作，所以，暂时保留 BatchTableEnvironment。这样，目前一共是 5 个 TableEnvironment。

因为未来 Flink Planner 将会被移除，BatchTableEnvironment 就会被废弃，整个 TableEnvironment 的设计也会更加简洁明了。

新 TableEnvironment 的应用

本节中，将介绍新的应用场景以及相关限制。下图详细列出了新 TableEnvironment 的适用场景：

图2 新 Table Environment 适应场景

第一行，简单起见，在后续将新的 TableEnvironment 称为 UnifyTableEnvironment。在 Blink 中，Batch 被认为是 Stream 的一个特例，因此 Blink 的 Batch 可以使用 UnifyTableEnvironment。

UnifyTableEnvironment 在 1.9 中有一些限制，比如它不能够注册 UDAF 和 UDTF，当前新的 Type System 的类型推导功能还没有完成（Java、Scala 的类型推导还没统一），所以这部分的功能暂时不支持。此外，UnifyTableEnvironment 无法和 DataStream 和 DataSet 互转。

第二行，Stream TableEnvironment 支持转化成 DataStream，也可以注册 UDAF 和 UDTF。如果是 JAVA 写的，就注册到 JAVA 的 StreamTableEnvironment，如果是用 Scala 写的，就注册到 Scala 的 StreamTableEnvironment。

注意，Blink Batch 作业不支持 Stream TableEnvironment ，因为目前 Batch 没法和 DataStream 互转，所以 toDataStream() 这样的语义暂时不支持。从图中也可以看出，目前Blink Batch只能使用 TableEnvironment。

最后一行，BatchTableEvironment 能够使用 toDataSet() 转化为 DataSet。

从上面的图 2 中，可以很清晰的看出各个 TableEnvironment 能够做什么事情，以及他们有哪些限制。

接下来，将使用示例对各种情况进行说明。

示例1：Blink Batch

```
EnvironmentSettings settings = EnvironmentSettings.newInstance().useBlinkPlanner(
TableEnvironment tEnv = TableEnvironment.create(settings);
tEnv…
tEnv.execute(“job name”);
```

从图 2 中可以看出，Blink Batch 只能使用 TableEnvironment（即UnifyTableEnvironment），代码中，首先需要创建一个 EnvironmentSetting，同时指定使用 Blink Planner，并且指定用 Batch 模式。之所以需要指定 Blink Planner，是因为目前 Flink 1.9 中，将 Flink Planner 和 Blink Planner 的 jar 同时放在了 Flink 的 lib 目录下。如果不指定使用的 Planner，整个框架并不知道需要使用哪个 Planner，所以必须显示的指定。当然，如果 lib 下面只有一个 Planner 的 jar，这时不需要显示指定使用哪个 Planner。

另外，还需要注意的是在 UnifyEnvironment 中，用户是无法获取到 ExecutionEnvironment 的，即用户无法在写完作业流程后，使用 executionEnvironment.execute() 方法启动任务。需要显式的使用 tableEnvironment.execute() 方法启动任务，这和之前的作业启动很不相同。

示例 2：Blink Stream

```
EnvironmentSettings settings = EnvironmentSettings.newInstance().useBlinkPlanner(
StreamExecutionEnvironment execEnv = …
StreamTableEnvironment tEnv =  StreamTableEnvironment.create(execEnv, settings);
tEnv…
```

Blink Stream 既可以使用 UnifyTableEnvironment，也可以使用 StreamTableEnvironment，与 Batch 模式基本类似，只是需要将 inBatchMode 换成 inStreamingMode。

示例 3：Flink Batch

```
ExecutionEnvironment execEnv = ...
BatchTableEnvironment tEnv = BatchTableEnvironment.create(execEnv);
tEnv...
```

与之前没有变化，不做过多介绍。

示例 4：Flink Stream

```
EnvironmentSettings settings = EnvironmentSettings.newInstance().useOldPlanner().
TableEnvironment tEnv = TableEnvironment.create(settings);
tEnv…
tEnv.execute(“job name”);
```

Flink Stream 也是同时支持 UnifyEnvironment 和 StreamTableEnvironment，只是在指定 Planner 时，需要指定为 useOldPlanner，也即 Flink Planner。因为未来 Flink Planner 会被移除，因此，特意起了一个 OlderPlanner 的名字，而且只能够使用 inStreamingMode，无法使用 inBatchMode。

Catalog 和 DDL

构建一个新的 Catalog API 主要是 FLIP-30 提出的，之前的 ExternalCatalog 将被废弃，Blink Planner 中已经不支持 ExternalCatalog 了，Flink Planner 还支持 ExternalCatalog。

新 Catalog 设计

下图是新 Catalog 的整体设计：

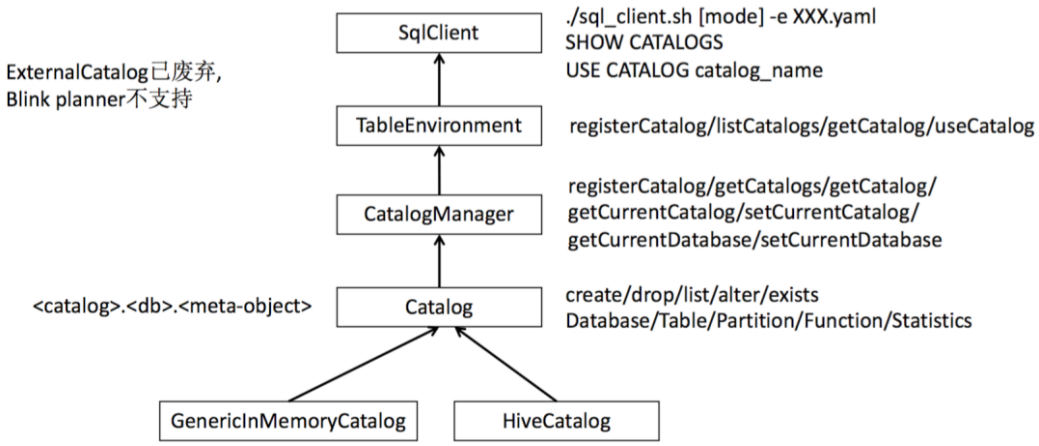

图3 新 Catalog 设计

可以看到，新的 Catalog 有三层结构（..），最顶层是 Catalog 的名字，中间一层是 Database，最底层是各种 MetaObject，如 Table，Partition，Function 等。当前，内置了两个 Catalog 实现：MemoryCatalog 和 HiveCatalog。当然，用户也可以实现自己的 Catalog。

Catalog 能够做什么事情呢？首先，它可以支持 Create，Drop，List，Alter，Exists 等语句，另外它也支持对 Database，Table，Partition，Function，Statistics 等的操作。基本上，常用的 SQL 语法都已经支持。

CatalogManager 正如它名字一样，主要是用来管理 Catalog，且可以同时管理多个 Catalog。也就是说，可以通过在一个相同 SQL 中，跨 Catalog 做查询或者关联操作。例如，支持对 A Hive Catalog 和 B Hive Catalog 做相互关联，这给 Flink 的查询带来了很大的灵活性。

CatalogManager 支持的操作包括：

- 注册 Catalog（registerCatalog）
- 获取所有的 Catalog（getCatalogs）
- 获取特定的 Catalog（getCatalog）
- 获取当前的 Catalog（getCurrentCatalog）
- 设置当前的 Catalog（setCurrentCatalog）
- 获取当前的 Database(getCurrentDatabase)
- 设置当前的 Database(setCurrentDatabase)

Catalog 虽然设计了三层结构，但在使用的时候，并不需要完全指定三层结构的值，可以只写Table Name，这时候，系统会使用 getCurrentCatalog，getCurrentDatabase 获取到默认值，自动补齐三层结构，这种设计简化了对 Catalog 的使用。如果需要切换默认的 Catalog，只需要调用 setCurrentCatalog 就可以了。

在 TableEnvironment 层，提供了操作 Catalog 的方法，例如：

- 注册 Catalog（registerCatalog）
- 列出所有的 Catalog（listCatalogs）
- 获取指定 Catalog（getCatalog）
- 使用某个 Catalog（useCatalog）

在 SQL Client 层，也做了一定的支持，但是功能有一定的限制。用户不能够使用 Create 语句直接创建 Catalog，只能通过在 yarn 文件中，通过定义 Description 的方式去描述 Catalog，然后在启动 SQL Client 的时候，通过传入 -e +file_path 的方式，定义 Catalog。目前 SQL Client 支持列出已定义的 Catalog，使用一个已经存在的 Catalog 等操作。

DDL 设计与使用

有了 Catalog，就可以使用 DDL 来操作 Catalog 的内容，可以使用 TableEnvironment 的 sqlUpdate() 方法执行 DDL 语句，也可以在 SQL Client 执行 DDL 语句。

sqlUpdate() 方法中，支持 Create Table、Create View、Drop Table、Drop View 四个命令。当然，inset into 这样的语句也是支持的。

下面分别对 4 个命令进行说明：

- **Create Table**：可以显示的指定 Catalog Name 或者 DB Name，如果缺省，那就按照用户设定的 Current Catalog 去补齐，然后可以指定字段名称，字段的说明，也可以支持 Partition By 语法。最后是一个 With 参数，用户可以在此处指定使用的 Connector，例如，Kafka，CSV，HBase 等。With 参数需要配置一堆的属性值，可以从各个 Connector 的 Factory 定义中找到。Factory 中会指出有哪些必选属性，哪些可选属性值。

需要注意的是，目前 DDL 中，还不支持计算列和 Watermark 的定义，后续的版本中将会继续完善这部分。

```
Create Table [[catalog_name.]db_name.]table_name(
  a int comment 'column comment',
  b bigint,
  c varchar
)comment 'table comment'
[partitioned by(b)]
With(
    update-mode='append',
    connector.type='kafka',
    ...
)
```

- **Create View**：需要指定 View 的名字，然后紧跟着的是 SQL。View 将会存储在 Catalog 中。

```
CREATE VIEW view_name AS SELECT xxx
```

- **Drop Table&Drop View**：和标准 SQL 语法差不多，支持使用 IF EXISTS 语法，如果未加 IF EXISTS，Drop 一个不存在的表，会抛出异常。

```
DROP TABLE [IF EXISTS] [[catalog_name.]db_name.]table_name
```

- **SQL Client 中执行 DDL**：大部分都只支持查看操作，仅可以使用 Create View 和 Drop View。Catalog，Database，Table，Function 这些只能做查看。用户可以在 SQL Client 中 Use 一个已经存在的 Catalog，修改一些属性，或者做 Description，Explain 这样的一些操作。

```
CREATE VIEW
DROP VIEW
SHOW CATALOGS/DATABASES/TABLES/FUNCTIONS l USE CATALOG xxx
SET xxx=yyy
DESCRIBE table_name
EXPLAIN SELECT xxx
```

DDL 部分，在 Flink 1.9 中其实基本已经成型，只是还有一些特性，在未来需要逐渐的完善。

Blink Planner

本节将主要从 SQL/Table API 如何转化为真正的 Job Graph 的流程开始，让大家对 Blink Planner 有一个比较清晰的认识，希望对大家阅读 Blink 代码，或者使用 Blink 方面有所帮助。然后介绍 Blink Planner 的改进及优化。

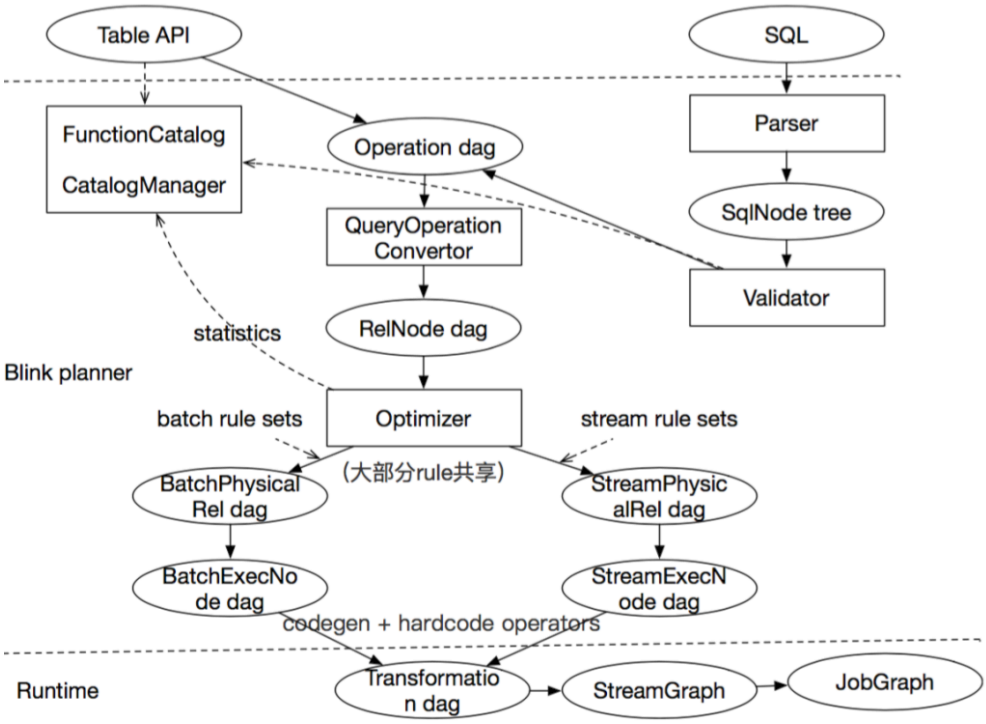

图4 主要流程

从上图可以很清楚的看到，解析的过程涉及到了三层：Table API/SQL，Blink Planner，Runtime，下面将对主要的步骤进行讲解。

- **Table API&SQL 解析验证**：在 Flink 1.9 中，Table API 进行了大量的重构，引入了一套新的 Operation，这套 Operation 主要是用来描述任务的 Logic Tree。

当 SQL 传输进来后，首先会去做 SQL 解析，SQL 解析完成之后，会得到 SqlNode Tree(抽象语法树)，然后会紧接着去做 Validate（验证），验证时会去访问 FunctionManger 和 CatalogManger，FunctionManger 主要是查询用户定义的 UDF，以及检查 UDF 是否合法，CatalogManger 主要是检查这个 Table 或者 Database 是否存在，如果验证都通过，就会生成一个 Operation DAG（有向无环图）。

从这一步可以看出，Table API 和 SQL 在 Flink 中最终都会转化为统一的结构，即 Operation DAG。

- **生成RelNode**：Operation DAG 会被转化为 RelNode(关系表达式) DAG。

优化：优化器会对 RelNode 做各种优化，优化器的输入是各种优化的规则，以及各种统计信息。当前，在 Blink Planner 里面，绝大部分的优化规则，Stream 和 Batch 是共享的。差异在于，对 Batch 而言，它没有 state 的概念，而对于 Stream 而言，它是不支持 sort 的，所以目前 Blink Planner 中，还是运行了两套独立的规则集（Rule Set），然后定义了两套独立的 Physical Rel：BatchPhysical Rel 和 StreamPhysical Rel。优化器优化的结果，就是具体的 Physical Rel DAG。

- **转化**：得到 Physical Rel Dag 后，继续会转化为 ExecNode，通过名字可以看出，ExecNode 已经属于执行层的概念了，但是这个执行层是 Blink 的执行层，在 ExecNode 中，会进行大量的 CodeGen 的操作，还有非 Code 的 Operator 操作，最后，将 ExecNode 转化为 Transformation DAG。
- **生成可执行 Job Graph**：得到 Transformation DAG 后，最终会被转化成 Job Graph，完成 SQL 或者 Table API 的解析。

Blink Planner 改进及优化

Blink Planner 功能方面改进主要包含如下几个方面：

- 更完整的 SQL 语法支持：例如，IN，EXISTS，NOT EXISTS，子查询，完整的 Over 语句，Group Sets 等。而且已经跑通了所有的 TPCH，TPCDS 这两个测试集，性能还非常不错。
- 提供了更丰富，高效的算子。
- 提供了非常完善的 cost 模型，同时能够对接 Catalog 中的统计信息，使 cost 根据统计信息得到更优的执行计划。
- 支持 join reorder。
- shuffle service：对 Batch 而言，Blink Planner 还支持 shuffle service，这对 Batch 作业的稳定性有非常大的帮助，如果遇到 Batch 作业失败，通过 shuffle service 能够很快的进行恢复。

性能方面，主要包括以下部分：

- 分段优化。
- Sub-Plan Reuse。
- 更丰富的优化 Rule：共一百多个 Rule ，并且绝大多数 Rule 是 Stream 和 Batch 共享的。
- 更高效的数据结构 BinaryRow：能够节省序列化和反序列化的操作。
- mini-batch 支持（仅 Stream）：节省 state 的访问的操作。
- 节省多余的 Shuffle 和 Sort（Batch 模式）：两个算子之间，如果已经按 A 做 Shuffle，紧接着他下的下游也是需要按 A Shuffle 的数据，那中间的这一层 Shuffle，就可以省略，这样就可以省很多网络的开销，Sort 的情况也是类似。Sort 和 Shuffle 如果在整个计算里面是占大头，对整个性能是有很大的提升的。

深入性能优化及实践

本节中，将使用具体的示例进行讲解，让你深入理解 Blink Planner 性能优化的设计。

■ 分段优化

示例 5

```
create view MyView as select word, count(1) as freq from SourceTable group by wor
insert into SinkTable2 select count(word) as freq2, freq from MyView group by fre
```

上面的这几个 SQL，转化为 RelNode DAG，大致图形如下：

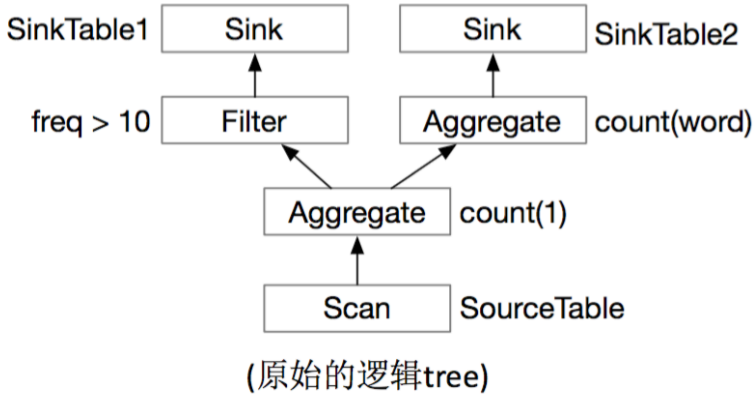

图5 示例5 RelNode DAG

如果是使用 Flink Planner，经过优化层后，会生成如下执行层的 DAG：

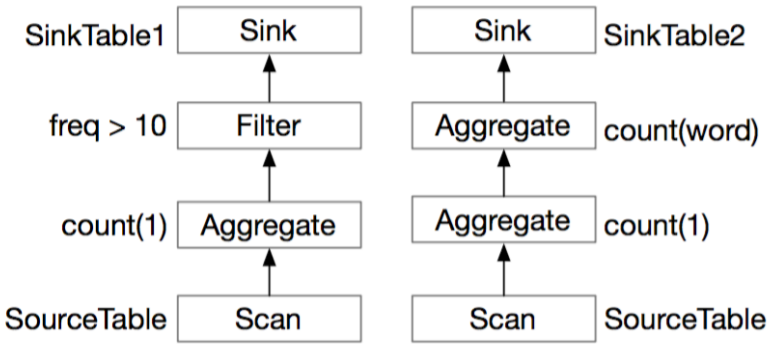

图6 示例 5 old planner DAG

可以看到，old planner 只是简单的从 Sink 出发，反向的遍历到 Source，从而形成两个独立的执行链路，从上图也可以清楚的看到，Scan 和第一层 Aggregate 是有重复计算的。

在 Blink Planner 中，经过优化层之后，会生成如下执行层的 DAG：

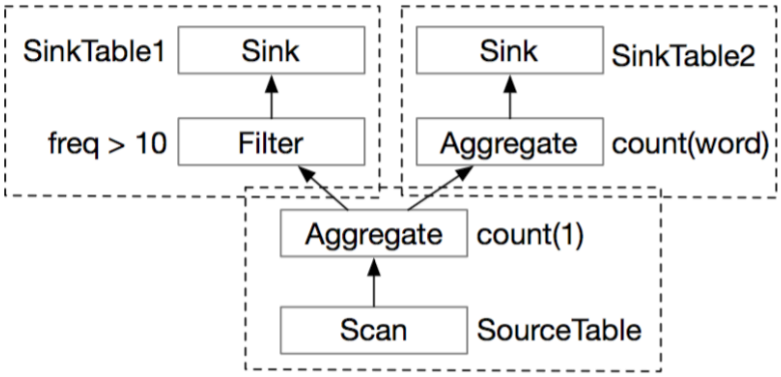

图7 示例 5 Blink Planner DAG

Blink Planner 不是在每次调用 insert into 的时候就开始优化，而是先将所有的 insert into 操作缓存起来，等到执行前才进行优化，这样就可以看到完整的执行图，可以知道哪些部分是重复计算的。Blink Planner 通过寻找可以优化的最大公共子图，找到这些重复计算的部分。经过优化后，Blink Planner 会将最大公共子图的部分当

做一个临时表，供其他部分直接使用。

这样，上面的图可以分为三部分，最大公共子图部分（临时表），临时表与 Filter 和 SinkTable1 优化，临时表与第二个 Aggregate 和 SinkTable 2 优化。

Blink Planner 其实是通过声明的 View 找到最大公共子图的，因此在开发过程中，如果需要复用某段逻辑，就将其定义为 View，这样就可以充分利用 Blink Planner 的分段优化功能，减少重复计算。

当然，当前的优化也不是最完美的，因为提前对图进行了切割，可能会导致一些优化丢失，今后会持续地对这部分算法进行改进。

总结一下，Blink Planner 的分段优化，其实解的是多 Sink 优化问题（DAG 优化），单 Sink 不是分段优化关心的问题，单 Sink 可以在所有节点上优化，不需要分段。

■ Sub-Plan Reuse

示例 6

```
insert into SinkTabl
select freq from (select word, count(1) as freq from SourceTable group by word) t
union all
select count(word) as freq2 from (select word, count(1) as freq from SourceTable
```

这个示例的 SQL 和分段优化的 SQL 其实是类似的，不同的是，没有将结果 Sink 到两个 Table 里面，而是将结果 Union 起来，Sink 到一个结果表里面。

下面看一下转化为 RelNode 的 DAG 图：

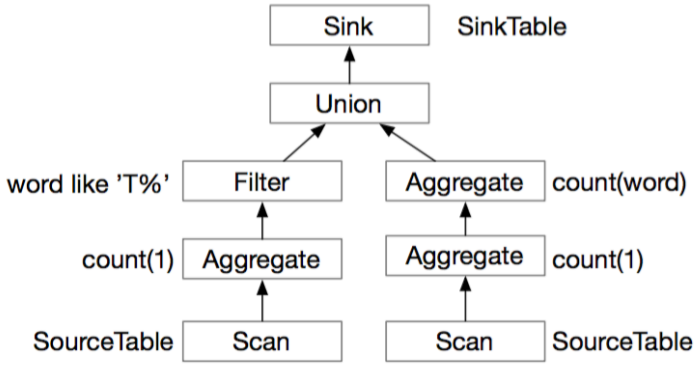

图 8 示例 6 RelNode DAG

从上图可以看出，Scan 和第一层的 Aggregate 也是有重复计算的，Blink Planner 其实也会将其找出来，变成下面的图：

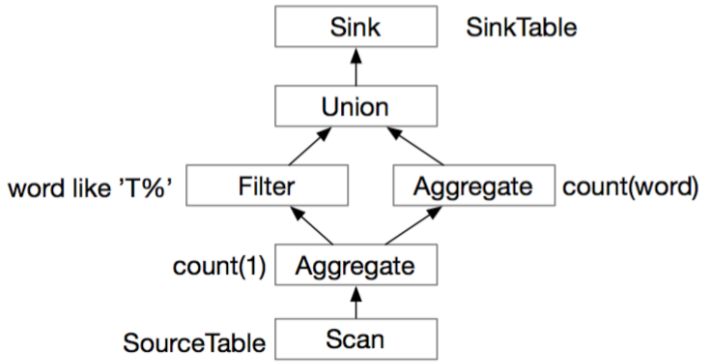

图9 示例 6 Blink Planner DAG

Sub-Plan 优化的启用，有两个相关的配置：

- table.optimizer.reuse-sub-plan-enabled （默认开启）
- table.optimizer.reuse-source-enabled（默认开启）

这两个配置，默认都是开启的，用户可以根据自己的需求进行关闭。这里主要说明一下 table.optimizer.reuse-source-enabled 这个参数。在 Batch 模式下，join 操作可能会导致死锁，具体场景是在执行 hash-join 或者 nested-loop-join 时一定是先读 build 端，然后再读 probe 端，如果启用 reuse-source-enabled，当数据源是同一个 Source 的时候，Source 的数据会同时发送给 build 和 probe 端。这时候，build 端的数据将不会被消费，导致 join 操作无法完成，整个 join 就被卡住了。

为了解决死锁问题，Blink Planner 会先将 probe 端的数据落盘，这样 build 端读数据的操作才会正常，等 build 端的数据全部读完之后，再从磁盘中拉取 probe 端的数据，从而解决死锁问题。但是，落盘会有额外的开销，会多一次写的操作；有时候，读两次 Source 的开销，可能比一次写的操作更快，这时候，可以关闭 reuse-source，性能会更好。

当然，如果读两次 Source 的开销，远大于一次落盘的开销，可以保持 reuse-source 开启。需要说明的是，Stream 模式是不存在死锁问题的，因为 Stream 模式 join 不会有选边的问题。

总结而言，sub-plan reuse 解的问题是优化结果的子图复用问题，它和分段优化类似，但他们是一个互补的过程。

> 注：Hash Join：对于两张待 join 的表 t1, t2。选取其中的一张表按照 join 条件给的列建立hash 表。然后扫描另外一张表，一行一行去建好的 hash 表判断是否有对应相等的行来完成 join 操作，这个操作称之为 probe (探测)。前一张表叫做 build 表，后一张表的叫做 probe 表。

■ Agg 分类优化

Blink 中的 Aggregate 操作是非常丰富的：

- group agg，例如：select count(a) from t group by b
- over agg，例如：select count(a) over (partition by b order by c) from t
- window agg，例如：select count(a) from t group by tumble(ts, interval '10' second), b

- table agg ，例如：tEnv.scan('t').groupBy('a').flatAggregate(flatAggFunc('b' as ('c', 'd')))

下面主要对 Group Agg 优化进行讲解，主要是两类优化。

■ Local/Global Agg 优化

Local/Global Agg 主要是为了减少网络 Shuffle。要运用 Local/Global 的优化，必要条件如下：

- Aggregate 的所有 Agg Function 都是 mergeable 的，每个 Aggregate 需要实现 merge 方法，例如 SUM，COUNT，AVG，这些都是可以分多阶段完成，最终将结果合并；但是求中位数，计算 95% 这种类似的问题，无法拆分为多阶段，因此，无法运用 Local/Global 的优化。
- table.optimizer.agg-phase-strategy 设置为 AUTO 或者 TWO_PHASE。
- Stream 模式下，mini-batch 开启 ；Batch 模式下 AUTO 会根据 cost 模型加上统计数据，选择是否进行 Local/Global 优化。

示例 7

```
select count(*) from t group by color
```

没有优化的情况下，下面的这个 Aggregate 会产生 10 次的 Shuffle 操作。

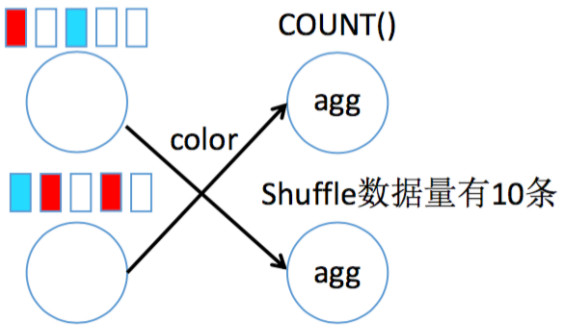

图 10 示例 7 未做优化的 Count 操作

使用 Local/Global 优化后，会转化为下面的操作，会在本地先进行聚合，然后再进行 Shuffle 操作，整个 Shuffle 的数据剩下 6 条。在 Stream 模式下，Blink 其实会以 mini-batch 的维度对结果进行预聚合，然后将结果发送给 Global Agg 进行汇总。

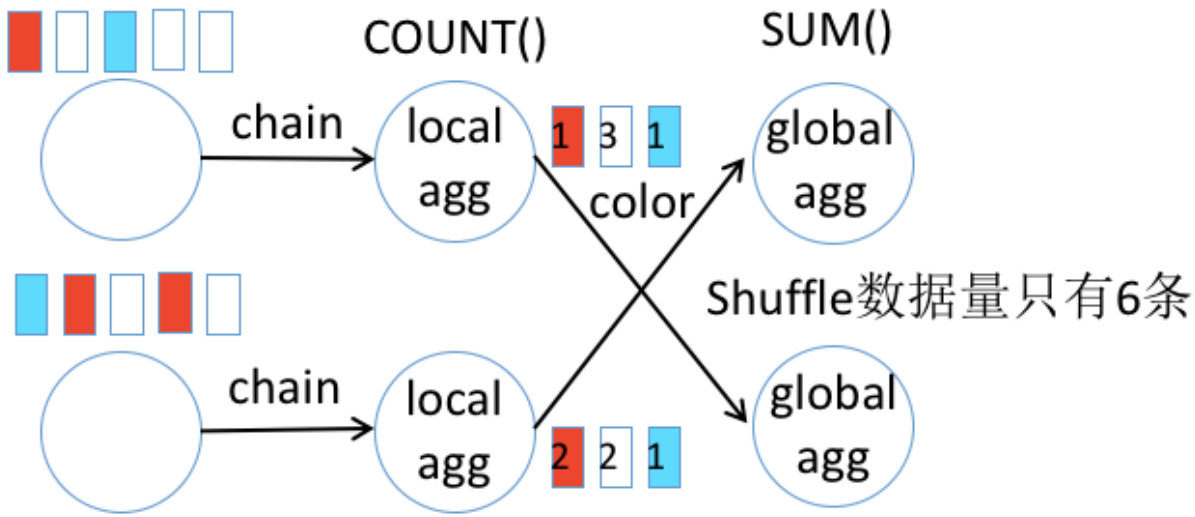

图 11 示例 7 经过 Local/Global 优化的 Count 操作

■ Distinct Agg 优化

Distinct Agg 进行优化，主要是对 SQL 语句进行改写，达到优化的目的。但 Batch 模式和 Stream 模式解决的问题是不同的：

- Batch 模式下的 Distinct Agg，需要先做 Distinct，再做 Agg，逻辑上需要两步才能实现，直接实现 Distinct Agg 开销太大。
- Stream 模式下，主要是解决热点问题，因为 Stream 需要将所有的输入数据放在 State 里面，如果数据有热点，State 操作会很频繁，这将影响性能。

Batch 模式

第一层，求 distinct 的值和非 distinct agg function 的值，第二层求 distinct agg function 的值。

示例 8

```
select color, count(distinct id), count(*) from t group by color
```

手工改写成：

```
select color, count(id), min(cnt) from (
   select color, id, count(*) filter (where $e=2) as cnt from (
      select color, id, 1 as $e from t --for distinct id
      union all
      select color, null as id, 2 as $e from t -- for count(*)
  ) group by color, id, $e
) group by color
```

转化的逻辑过程，如下图所示：

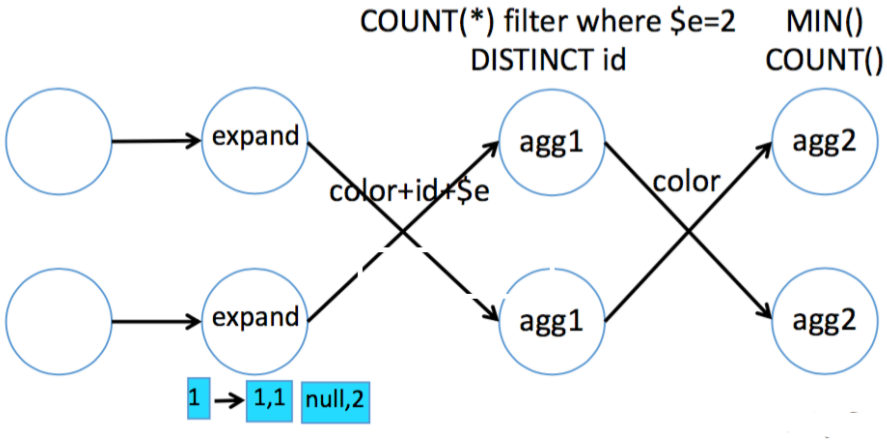

图 12 示例 8 Batch 模式 Distinct 改写逻辑

Stream 模式

Stream 模式的启用有一些必要条件：

- 必须是支持的 agg function：avg/count/min/max/sum/first_value/concat_agg/single_value；
- table.optimizer.distinct-agg.split.enabled（默认关闭）

示例 9

```
select color, count(distinct id), count(*) from t group by color
```

手工改写成：

```
select color, sum(dcnt), sum(cnt) from (
  select color, count(distinct id) as dcnt, count(*) as cnt from t
  group by color, mod(hash_code(id), 1024)
) group by color
```

改写前，逻辑图大概如下：

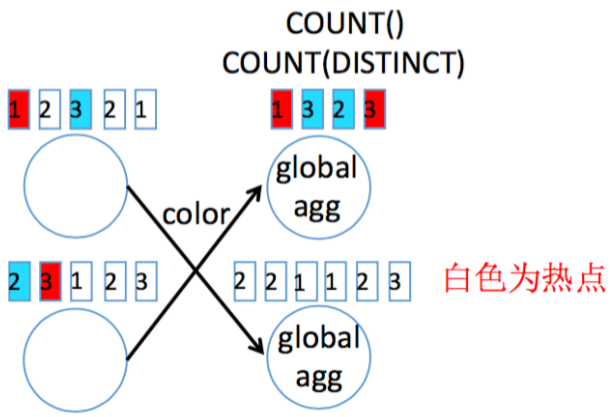

图 13 示例 9 Stream 模式未优化 Distinct

改写后，逻辑图就会变为下面这样，热点数据被打散到多个中间节点上。

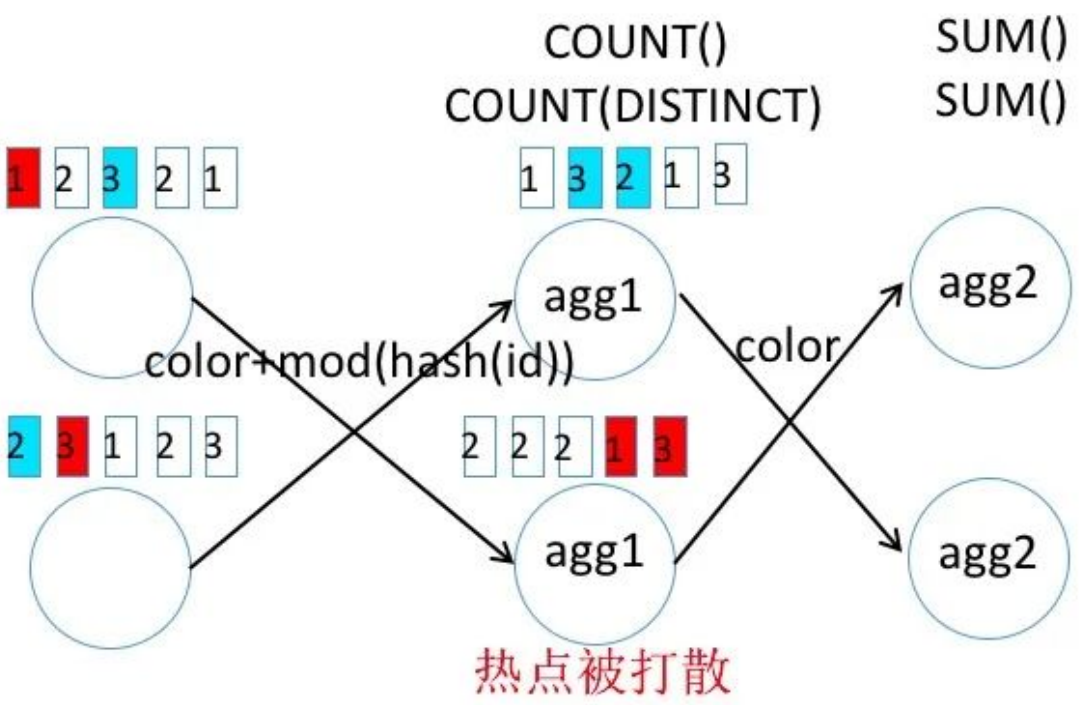

图14 示例 9 Stream 模式优化 Distinct

需要注意的是，示例 5 的 SQL 中 mod(hash_code(id),1024)中的这个 1024 为打散的维度，这个值建议设置大一些，设置太小产生的效果可能不好。

总结

本文首先对新的 TableEnvironment 的整体设计进行了介绍，并且列举了各种模式下TableEnvironment 的选择，然后通过具体的示例，展示了各种模式下代码的写法，以及需要注意的事项。

在新的 Catalog 和 DDL 部分，对 Catalog 的整体设计、DDL 的使用部分也都以实例进行拆分讲解。最后，对 Blink Planner 解析 SQL/Table API 的流程、Blink Planner 的改进以及优化的原理进行了讲解，希望对大家探索和使用 Flink SQL 有所帮助。