# Flink Table API和Flink SQL的全面解析

## 整体概述

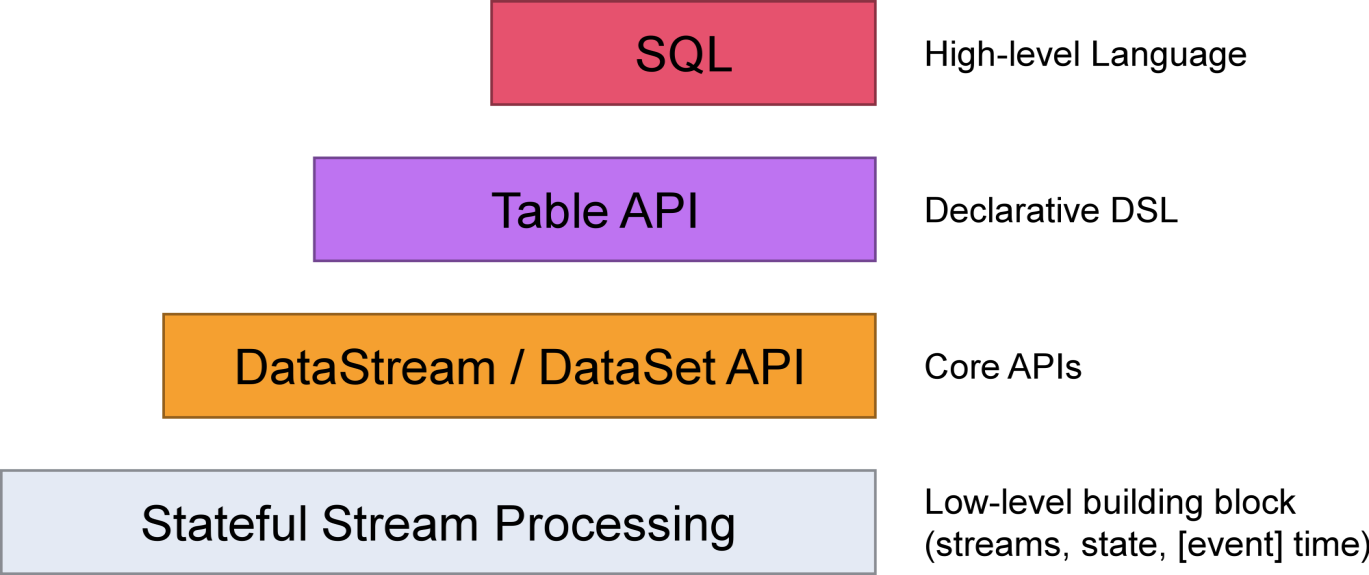
### 什么是 Table API 和 Flink SQL

Apache Flink是批流统一的处理框架，具有两个关系API-**Table API和SQL**-用于统一流和批处理的上层API。

Table API是Java，Scala和Python的语言集成查询API，它允许以非常直观的方式组合来自关系运算符（例如选择，过滤和联接）的查询。

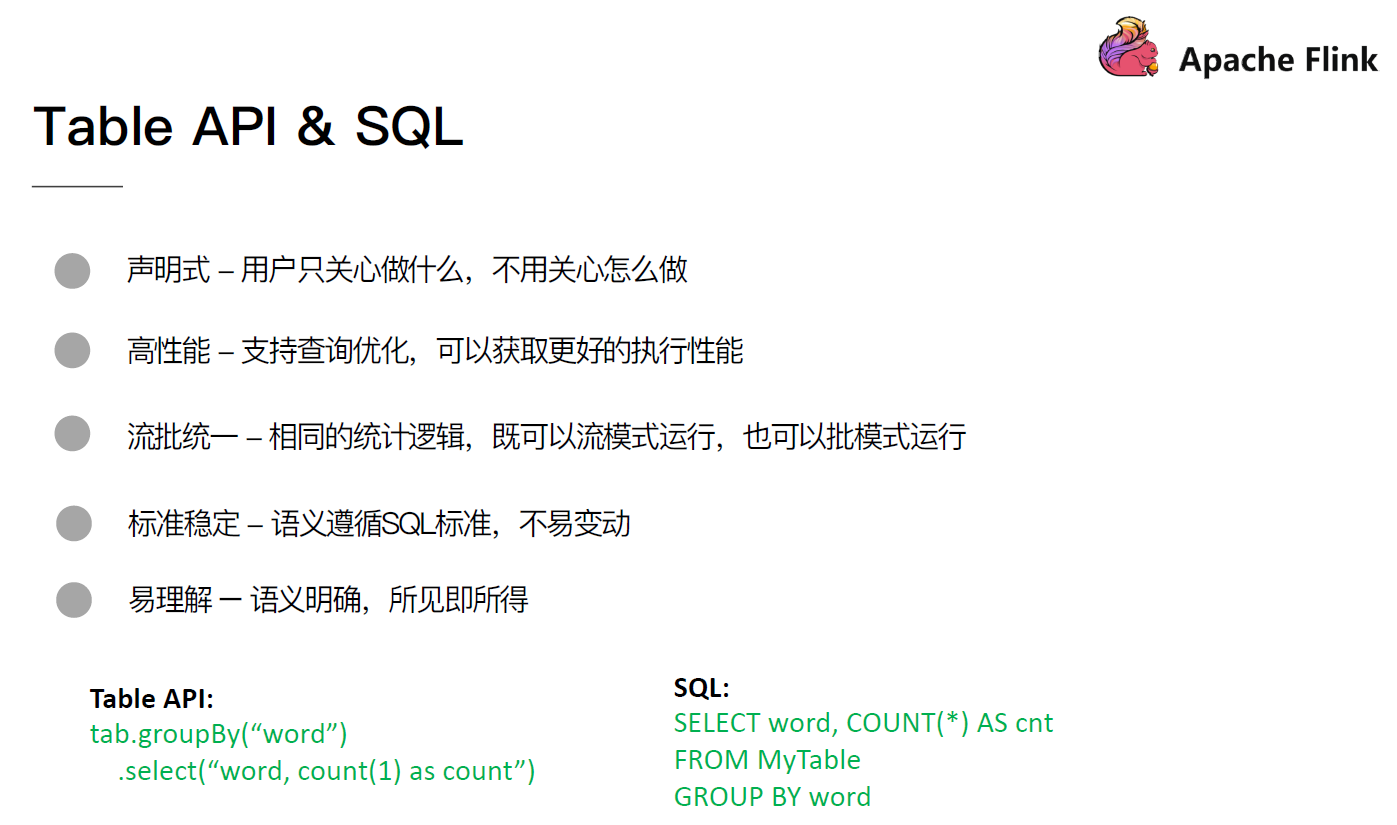
Flink的SQL就是可以在代码中写sql，实现一些查询操作，支持基于实现SQL标准的[Apache Calcite](https://calcite.apache.org/)（Apache开源SQL解析工具）。

无论输入是连续的（流式）还是有界的（批处理），在两个接口中指定的查询都具有相同的语义，并指定相同的结果。



官网介绍：<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/overview/>

### 为什么需要Table API & SQL



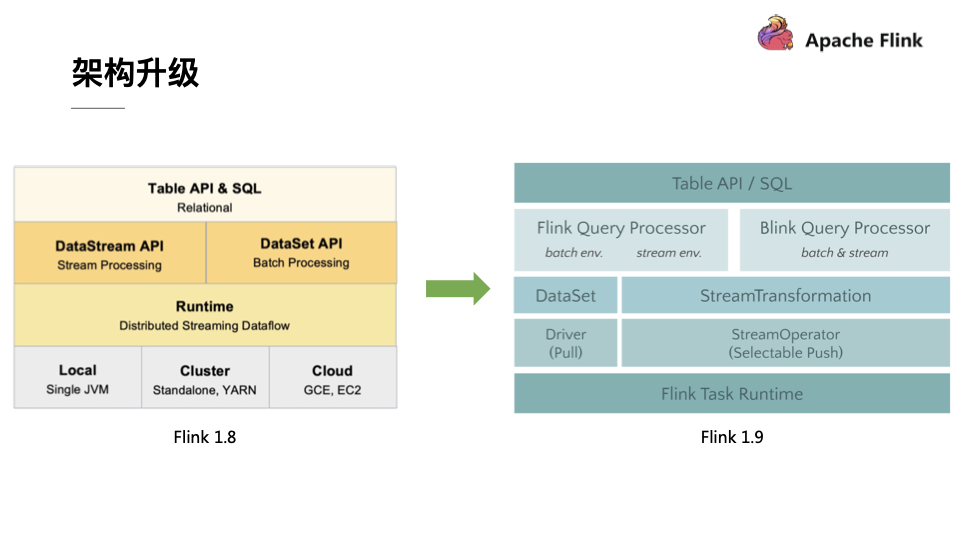
* 声明式：属于设定式语言，用户只要表达清楚需求即可，不需要了解底层执行；
* 高性能：可优化，内置多种查询优化器，这些查询优化器可为 SQL 翻译出最优执行计划；
* 简单易学：易于理解，不同行业和领域的人都懂，学习成本较低；
* 标准稳定：语义遵循SQL标准，非常稳定，在数据库 30 多年的历史中，SQL 本身变化较少；
* 流批统一：可以做到API层面上流与批的统一，相同的SQL逻辑，既可流模式运行，也可批模式运行，Flink底层Runtime本身就是一个流与批统一的引擎

### Table API& SQL发展历程

* 架构升级

自 2015 年开始，阿里巴巴开始调研开源流计算引擎，最终决定基于 Flink 打造新一代计算引擎，针对 Flink 存在的不足进行优化和改进，并且在 2019 年初将最终代码开源，也就是**Blink**。Blink 在原来的 Flink 基础上最显著的一个贡献就是 Flink SQL 的实现。随着版本的不断更新，API 也出现了很多不兼容的地方。

在 Flink 1.9 中，Table 模块迎来了核心架构的升级，引入了阿里巴巴Blink团队贡献的诸多功能



在Flink 1.9 之前，Flink API 层 一直分为DataStream API 和 DataSet API，Table API & SQL 位于 DataStream API 和 DataSet API 之上。可以看出流处理和批处理有各自独立的api (流处理DataStream，批处理DataSet)。而且有不同的执行计划解析过程，codegen过程也完全不一样，完全没有流批一体的概念，面向用户不太友好。

在Flink1.9之后新的架构中，有两个查询处理器：**Flink Query Processor**，也称作**Old Planner和Blink Query Processor**，也称作**Blink Planner**。为了兼容老版本Table及SQL模块，插件化实现了Planner，Flink原有的Flink Planner不变，后期版本会被移除。新增加了Blink Planner，新的代码及特性会在Blink planner模块上实现。批或者流都是通过解析为Stream Transformation来实现的，不像Flink Planner，批是基于Dataset，流是基于DataStream。

* 查询处理器的选择

查询处理器是 Planner 的具体实现，通过parser、optimizer、codegen(代码生成技术)等流程将 Table API & SQL作业转换成 Flink Runtime 可识别的 Transformation DAG，最终由 Flink Runtime 进行作业的调度和执行。

**Flink Query Processor查询处理器**针对流计算和批处理作业有不同的分支处理，流计算作业底层的 API 是 DataStream API， 批处理作业底层的 API 是 DataSet API

**Blink Query Processor查询处理器**则实现流批作业接口的统一，底层的 API 都是Transformation，这就意味着我们和Dataset完全没有关系了

**Flink1.11之后Blink Query Processor查询处理器已经是默认的了**

### 两种planner（old & blink）的区别



* Blink 将批处理作业视作流处理的一种特例。严格来说，Table 和 DataSet 之间不支持相互转换，并且批处理作业也不会转换成 DataSet 程序而是转换成 DataStream 程序，流处理作业也一样。
* Blink 计划器不支持 BatchTableSource，而是使用有界的 StreamTableSource 来替代。
* 旧计划器和 Blink 计划器中 FilterableTableSource 的实现是不兼容的。旧计划器会将 PlannerExpression 下推至 FilterableTableSource，而 Blink 计划器则是将 Expression 下推。
* 基于字符串的键值配置选项仅在 Blink 计划器中使用。（详情参见 [配置](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/) ）
* PlannerConfig 在两种计划器中的实现（CalciteConfig）是不同的。
* Blink 计划器会将多sink（multiple-sinks）优化成一张有向无环图（DAG），TableEnvironment 和 StreamTableEnvironment 都支持该特性。旧计划器总是将每个sink都优化成一个新的有向无环图，且所有图相互独立。
* 旧计划器目前不支持 catalog 统计数据，而 Blink 支持。

<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/common/>

## 概念和通用API

Table API 和 SQL 集成在同一套 API 中。这套 API 的核心概念是Table，用作查询的输入和输出。本文介绍了 Table API 和 SQL 查询程序的通用结构、如何注册 Table 、如何查询 Table 以及如何输出 Table 。

### 需要引入的依赖

参考官网：<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/overview/>

|  |
| --- |
| <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-table-api-scala-bridge\_2.11</artifactId>  <version>${flink.version}</version>  <scope>provided</scope> </dependency> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-table-api-java-bridge\_2.11</artifactId>  <version>${flink.version}</version>  <scope>provided</scope> </dependency> <!-- flink执行计划,这是1.9版本之前的--> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-table-planner\_2.11</artifactId>  <version>${flink.version}</version> </dependency> <!-- blink执行计划,1.11+默认的--> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-table-planner-blink\_2.11</artifactId>  <version>${flink.version}</version>  <scope>provided</scope> </dependency> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-table-common</artifactId>  <version>${flink.version}</version>  <scope>provided</scope> </dependency> |

* 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，

Flink Planner 和 Blink Planner 都会依赖于具体的 JavaAPI，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

### 基本程序结构

所有用于批处理和流处理的 Table API 和 SQL 程序结构，与流式处理的程序结构类似；也可以近似地认为有这么几步：首先创建执行环境，然后定义source、transform和sink。

具体操作流程如下：

|  |
| --- |
| // 创建表的执行环境 TableEnvironment tableEnv = ...;  // 创建一张表，用来读取数据 tableEnv.executeSql("CREATE TEMPORARY TABLE table1 ... WITH ( 'connector' = ... )");  //注册一张表，用来输出计算结果数据 tableEnv.executeSql("CREATE TEMPORARY TABLE outputTable ... WITH ( 'connector' = ... )");  //通过Table API查询算子，得到一张结果表 Table table2 = tableEnv.from("table1").select(...);  //通过SQL查询语句，得到一张结果表 Table table3 = tableEnv.sqlQuery("SELECT ... FROM table1 ... ");  //将结果表写入到输出表中 TableResult tableResult = table2.executeInsert("outputTable"); tableResult... |

### 创建 TableEnvironment

TableEnvironment 是 Table API 和 SQL 的核心概念。它负责:

* 注册catalog
* 在内部 catalog 中注册表
* 执行 SQL 查询
* 注册用户自定义函数
* 将 DataStream转换为表
* 保存对 ExecutionEnvironment 或 StreamExecutionEnvironment 的引用

在创建TableEnv的时候，可以多传入一个EnvironmentSettings或者TableConfig参数，可以用来配置 TableEnvironment的一些特性。

比如：

* 配置老版本的流式查询（Flink-Streaming-Query）：

|  |
| --- |
| // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*// FLINK STREAMING QUERY// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment**;**  **import** org.apache.flink.table.api.EnvironmentSettings**;**  **import** org.apache.flink.table.api.bridge.java.StreamTableEnvironment**;**  EnvironmentSettings fsSettings **=** EnvironmentSettings**.**newInstance**()**  **.**useOldPlanner**()** // 使用老版本planner  **.**inStreamingMode**()** // 流处理模式  **.**build**();**  StreamExecutionEnvironment fsEnv **=** StreamExecutionEnvironment**.**getExecutionEnvironment**();**  StreamTableEnvironment fsTableEnv **=** StreamTableEnvironment**.**create**(**fsEnv**,** fsSettings**);**  // or TableEnvironment fsTableEnv = TableEnvironment.create(fsSettings); |

* 基于老版本的批处理环境（Flink-Batch-Query）：

|  |
| --- |
| // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*// FLINK BATCH QUERY// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  **import** org.apache.flink.api.java.ExecutionEnvironment**;**  **import** org.apache.flink.table.api.bridge.java.BatchTableEnvironment**;**  ExecutionEnvironment fbEnv **=** ExecutionEnvironment**.**getExecutionEnvironment**();**  BatchTableEnvironment fbTableEnv **=** BatchTableEnvironment**.**create**(**fbEnv**);** |

* 基于blink版本的流处理环境（Blink-Streaming-Query）：

|  |
| --- |
| // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*// BLINK STREAMING QUERY// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment**;**  **import** org.apache.flink.table.api.EnvironmentSettings**;**  **import** org.apache.flink.table.api.bridge.java.StreamTableEnvironment**;**  StreamExecutionEnvironment bsEnv **=** StreamExecutionEnvironment**.**getExecutionEnvironment**();**  EnvironmentSettings bsSettings **=** EnvironmentSettings**.**newInstance**()**  **.**useBlinkPlanner**()**// 使用新版本planner  **.**inStreamingMode**()**// 流处理模式  **.**build**();**  StreamTableEnvironment bsTableEnv **=** StreamTableEnvironment**.**create**(**bsEnv**,** bsSettings**);**  // or TableEnvironment bsTableEnv = TableEnvironment.create(bsSettings); |

* 基于blink版本的批处理环境（Blink-Batch-Query）：

|  |
| --- |
| // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*// BLINK BATCH QUERY// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  **import** org.apache.flink.table.api.EnvironmentSettings**;**  **import** org.apache.flink.table.api.TableEnvironment**;**  EnvironmentSettings bbSettings **=** EnvironmentSettings**.**newInstance**()**  **.**useBlinkPlanner**()**// 使用新版本planner  **.**inBatchMode**()**// 批处理模式  **.**build**();**  TableEnvironment bbTableEnv **=** TableEnvironment**.**create**(**bbSettings**);** |

### 在Catalog中注册表

#### 表（Table）的概念

TableEnvironment 维护着一个由标识符（identifier）创建的表 catalog 的映射。标识符由三个部分组成：**Catalog名、数据库（database）名和对象名（表名）**。如果 catalog 或者数据库没有指明，就会使用当前默认值。

Table **可以是虚拟的（视图 VIEWS）也可以是常规的（表 TABLES）**。视图 VIEWS可以从已经存在的Table中创建，一般是 Table API 或者 SQL 的查询结果。 表TABLES描述的是外部数据，例如文件、数据库表或者消息队列。

#### 临时表（Temporary Table）和永久表（Permanent Table）

**表可以是临时的**，并与单个 Flink 会话（session）的生命周期相关

**表也可以是永久的**，并且在多个 Flink 会话和群集（cluster）中可见

永久表需要 [catalog](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/catalogs/)（例如 Hive Metastore）以维护表的元数据。一旦永久表被创建，它将对任何连接到 catalog 的 Flink 会话可见且持续存在，直至被明确删除。

另一方面，**临时表通常保存于内存中并且仅在创建它们的 Flink 会话持续期间存在。**这些表对于其它会话是不可见的。它们不与任何 catalog 或者数据库绑定但可以在一个命名空间（namespace）中创建。即使它们对应的数据库被删除，临时表也不会被删除。

#### 创建表

##### 虚拟表

在 SQL 的术语中，Table API 的对象对应于视图（虚拟表）。它封装了一个逻辑查询计划。它可以通过以下方法在 catalog 中创建：

|  |
| --- |
| // get a TableEnvironment TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section  // table is the result of a simple projection query Table projTable = tableEnv.from("X").select(...);  // register the Table projTable as table "projectedTable" tableEnv.createTemporaryView("projectedTable", projTable); |

* 注意事项

|  |
| --- |
| 从传统数据库系统的角度来看，Table 对象与 VIEW 视图非常像。也就是，定义了 Table 的查询是没有被优化的， 而且会被内嵌到另一个引用了这个注册了的 Table的查询中。如果多个查询都引用了同一个注册了的Table，那么它会被内嵌每个查询中并被执行多次， 也就是说注册了的Table的结果不会被共享（注：Blink 计划器的TableEnvironment会优化成只执行一次） |

##### Connector Tables

另外一个方式去创建 TABLE 是通过 [connector](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/overview/) 声明。Connector 描述了存储表数据的外部系统。存储系统例如 Apache Kafka 或者常规的文件系统都可以通过这种方式来声明。

|  |
| --- |
| tableEnvironment  .connect(...)  .withFormat(...)  .withSchema(...)  .inAppendMode()  .createTemporaryTable("MyTable") |

##### 扩展表标识符

表总是通过三元标识符注册，包括 catalog 名、数据库名和表名。

用户可以指定一个 catalog 和数据库作为 “当前catalog” 和"当前数据库"。有了这些，那么刚刚提到的三元标识符的前两个部分就可以被省略了。如果前两部分的标识符没有指定， 那么会使用当前的 catalog 和当前数据库。用户也可以通过 Table API 或 SQL 切换当前的 catalog 和当前的数据库。

标识符遵循 SQL 标准，因此使用时需要用反引号（`）进行转义。

|  |
| --- |
| TableEnvironment tEnv = ...; tEnv.useCatalog("custom\_catalog"); tEnv.useDatabase("custom\_database"); Table table = ...; //在custom\_catalog”的目录中注册名称“exampleView”的视图 //在“custom\_database”的数据库中 tableEnv.createTemporaryView("exampleView", table); //在“custom\_catalog”的目录中注册名称“exampleView”的视图 //在“other\_database”的数据库中 tableEnv.createTemporaryView("other\_database.exampleView", table); //在“custom\_catalog”的目录中注册名称“example.view”的视图 //在“custom\_database”的数据库中 tableEnv.createTemporaryView("`example.View`", table); //在“other\_catalog”的目录中注册名称“exampleView”的视图 //在“other\_database”的数据库中 tableEnv.createTemporaryView("other\_catalog.other\_database.exampleView", table); |

### 查询表

利用外部系统的连接器connector，我们可以读写数据，并在环境的Catalog中注册表。接下来就可以对表做查询转换了。

Flink给我们提供了两种查询方式：Table API和 SQL。

#### Table API的调用

Table API基于代表一张“表”的Table类，并提供一整套操作处理的方法API。这些方法会返回一个新的Table对象，这个对象就表示对输入表应用转换操作的结果。有些关系型转换操作，可以由多个方法调用组成，构成链式调用结构。例如table.select(…).filter(…)，其中select（…）表示选择表中指定的字段，filter(…)表示筛选条件。

**注意：**

**通过导入可以启用Java Table API org.apache.flink.table.api.java.\*。下面的示例演示如何构造Java Table API程序以及如何将表达式指定为字符串。对于Expression DSL，还必须导入静态org.apache.flink.table.api.Expressions.\***

|  |
| --- |
| import static org.apache.flink.table.api.Expressions.\*; |

代码中的实现如下：

|  |
| --- |
| // 3. 表的查询转换 Table orderTable = tableEnv.from("inputTable");  // 3.1 简单查询转换 Table resultTable = orderTable  .select($("id"),$("timestamp"),$("category"),$("areaName"),$("money"))  .filter($("areaName").isEqual("北京")); |

也可以加上聚合操作，比如我们统计每个城市订单数据出现的个数，做个count统计：

|  |
| --- |
| // 3. 表的查询转换 Table orderTable = tableEnv.from("inputTable");  // 3.2 聚合转换 Table aggResultSqlTable = orderTable  .groupBy($("areaName"))  .select($("areaName"), $("id").count().as("cnt")); |

#### SQL查询

Flink的SQL集成，基于的是[ApacheCalcite](https://calcite.apache.org/)，它实现了SQL标准。在Flink中，用常规字符串来定义SQL查询语句。SQL 查询的结果，是一个新的 Table。

* 下面的示例演示了如何指定查询并将结果作为 Table 对象返回

|  |
| --- |
| // 3. 表的查询转换 Table orderTable = tableEnv.from("inputTable");  // 3.1 简单查询转换 Table resultTable2 = tableEnv.sqlQuery("select id,`timestamp`,category,areaName,money from inputTable where areaName='北京'"); |

也可以加上聚合操作，比如我们统计每个城市订单数据出现的个数，做个count统计：

|  |
| --- |
| // 3. 表的查询转换 Table orderTable = tableEnv.from("inputTable");  // 3.2 聚合转换 Table aggResultSqlTable2 = tableEnv.sqlQuery("select areaName, count(1) as cnt from inputTable group by areaName"); |

* 如下的示例展示了如何指定一个更新查询，将查询的结果插入到已注册的表中

|  |
| --- |
| // get a TableEnvironment TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section // register "Orders" table // register "RevenueFrance" output table // compute revenue for all customers from France and emit to "RevenueFrance" tableEnv.executeSql(  "INSERT INTO RevenueFrance " +  "SELECT cID, cName, SUM(revenue) AS revSum " +  "FROM Orders " +  "WHERE cCountry = 'FRANCE' " +  "GROUP BY cID, cName" ); |

### 将DataStream 转换成表

Flink允许我们把Table和DataStream做转换：可以基于一个DataStream，先流式地读取数据源，然后map成样例类，再把它转成Table。Table的列字段（column fields），就是样例类里的字段，这样就不用再麻烦地定义schema了。

#### 代码表达

代码中实现非常简单，直接用tableEnv.fromDataStream()就可以了。默认转换后的 Table schema 和 DataStream 中的字段定义一一对应，也可以单独指定出来。

这就允许我们更换字段的顺序、重命名，或者只选取某些字段出来，相当于做了一次map操作（或者Table API的 select操作）。

代码具体如下：

|  |
| --- |
| // 从文件读取数据  String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();  DataStreamSource<String> inputStream = env.readTextFile(filePath);  // map成样例类类型  SingleOutputStreamOperator<OrderInfo> dataStream = inputStream.map(new MapFunction<String, OrderInfo>() {  @Override  public OrderInfo map(String data) throws Exception {  String[] dataArray = data.split(",");  return new OrderInfo(dataArray[0], dataArray[1], Double.parseDouble(dataArray[2]), dataArray[3]);  }  });  // 2. 基于tableEnv，将流转换成表  Table dataTable = tableEnv.fromDataStream(dataStream); |

#### 数据类型与Table schema的对应

在上面的例子中，DataStream 中的数据类型，与表的 Schema 之间的对应关系，是按照样例类中的字段名来对应的（name-based mapping），所以还可以用as做重命名。

另外一种对应方式是，直接按照字段的位置来对应（position-based mapping），对应的过程中，就可以直接指定新的字段名了。

* 基于名称的对应：

|  |
| --- |
| Table dataTable = tableEnv.fromDataStream(dataStream,$("id").as("ts"), $("timestamp").as("myId"), $("money"), $("category")); |

* 基于位置的对应：

|  |
| --- |
| Table dataTable3 = tableEnv.fromDataStream(dataStream,$("id"), $("timestamp"), $("money"), $("category")); |

Flink的DataStream和 DataSet API支持多种类型。

组合类型，比如元组（内置Scala和Java元组）、POJO、Scala case类和Flink的Row类型等，允许具有多个字段的嵌套数据结构，这些字段可以在Table的表达式中访问。其他类型，则被视为原子类型。

元组类型和原子类型，一般用位置对应会好一些；如果非要用名称对应，也是可以的：

元组类型，默认的名称是 “\_1”, “\_2”；而原子类型，默认名称是 ”f0”。

#### 使用案例

|  |
| --- |
| /\*\*  \* 将DataStream 转换成表  \*/ public class DataStreamToTable {  public static void main(String[] args) throws Exception {  // 0. 创建流执行环境，读取数据并转换成样例类类型  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();   env.getConfig().setRestartStrategy(RestartStrategies.noRestart());  env.setParallelism(1);   // 从文件读取数据  String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();  DataStreamSource<String> inputStream = env.readTextFile(filePath);   // map成样例类类型  SingleOutputStreamOperator<OrderInfo> dataStream = inputStream.map(new MapFunction<String, OrderInfo>() {  @Override  public OrderInfo map(String data) throws Exception {  String[] dataArray = data.split(",");  return new OrderInfo(dataArray[0], dataArray[1], dataArray[2], dataArray[3], Double.parseDouble(dataArray[4]));  }  });   // 1. 基于env创建表环境  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, bsSettings);   // 2. 基于tableEnv，将流转换成表  Table dataTable = tableEnv.fromDataStream(dataStream);   // 3. 转换操作，得到提取结果  // 3.1 调用table api，做转换操作  Table resultTable = dataTable  .select($("id"),$("timestamp"),$("category"),$("areaName"),$("money"))  .filter($("areaName").isEqual("北京"));   // 3.2 写sql实现转换  //tableEnv.registerTable("dataTable", dataTable)  tableEnv.createTemporaryView("inputTable", dataTable);  Table resultSqlTable = tableEnv.sqlQuery("select id,`timestamp`,category,areaName,money from inputTable where areaName='北京'");   // 4. 把表转换成流，打印输出  tableEnv.toAppendStream(resultTable, Row.class).print("result");  tableEnv.toAppendStream(resultSqlTable, Row.class).print("result");   env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class OrderInfo {  private String id;  private String timestamp;  private String category;  private String areaName;  private Double money;  } } |

### 创建临时视图

创建临时视图的第一种方式，就是直接从DataStream转换而来。同样，可以直接对应字段转换；也可以在转换的时候，指定相应的字段。

|  |
| --- |
| tableEnv.createTemporaryView("orderView", dataStream); tableEnv.createTemporaryView("orderView",dataStream, $("id"), $("timestamp"), $("money"), $("category")); |

另外，当然还可以基于Table创建视图：

|  |
| --- |
| tableEnv.createTemporaryView("orderView", orderTable); |

View和Table的Schema完全相同。事实上，在Table API中，可以认为View和Table是等价的。

### 输出表

Table 通过写入 TableSink 输出。TableSink 是一个通用接口，用于支持多种文件格式（如 CSV、Apache Parquet、Apache Avro）、存储系统（如 JDBC、Apache HBase、Apache Cassandra、Elasticsearch）或消息队列系统（如 Apache Kafka、RabbitMQ）。

批处理 Table 只能写入 **BatchTableSink**

流处理 Table 需要指定写入 **AppendStreamTableSink，RetractStreamTableSink**或者**UpsertStreamTableSink**。

具体实现，输出表最直接的方法，就是通过 Table.insertInto() 方法将一个 Table 写入注册过的 TableSink 中。

#### 输出到文件

|  |
| --- |
| /\*\*  \* 将数据写入到文件中  \*/ public class FsSinkTest {  public static void main(String[] args) throws Exception {  // 0. 创建流执行环境，读取数据并转换成样例类类型  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();   env.getConfig().setRestartStrategy(RestartStrategies.noRestart());  env.setParallelism(1);   // 从文件读取数据  String inputFilePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();  DataStreamSource<String> inputStream = env.readTextFile(inputFilePath);   // map成样例类类型  SingleOutputStreamOperator<DataStreamToTable.OrderInfo> dataStream = inputStream.map(new MapFunction<String, DataStreamToTable.OrderInfo>() {  @Override  public DataStreamToTable.OrderInfo map(String data) throws Exception {  String[] dataArray = data.split(",");  return new DataStreamToTable.OrderInfo(dataArray[0], dataArray[1], dataArray[2], dataArray[3], Double.parseDouble(dataArray[4]));  }  });   // 1. 基于env创建表环境  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, bsSettings);   // 2. 基于tableEnv，将流转换成表  Table dataTable = tableEnv.fromDataStream(dataStream);   // 3. 转换操作，得到提取结果  // 3.1 调用table api，做转换操作  Table resultTable = dataTable  .select($("id"), $("timestamp"), $("category"), $("areaName"), $("money"))  .filter($("areaName").isEqual("北京"));   // 4. 将结果表输出到文件中  tableEnv.connect(new FileSystem().path("F:\\test\\flinksqlbase\\src\\main\\resources\\output.txt"))  .withFormat(new Csv())  .withSchema(new Schema()  .field("id", DataTypes.STRING())  .field("timestamp", DataTypes.STRING())  .field("category", DataTypes.STRING())  .field("areaName", DataTypes.STRING())  .field("money", DataTypes.DOUBLE())  )  .createTemporaryTable("outputTable");   // 5. 输出到文件  resultTable.executeInsert("outputTable");   env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class OrderInfo {  private String id;  private String timestamp;  private String category;  private String areaName;  private Double money;  } } |

#### 更新模式（Update Mode）

在流处理过程中，表的处理并不像传统定义的那样简单。

对于流式查询（Streaming Queries），需要声明如何在（动态）表和外部连接器之间执行转换。与外部系统交换的消息类型，由更新模式（update mode）指定。

Flink Table API中的更新模式有以下三种：

##### 追加模式（Append Mode）

在追加模式下，表（动态表）和外部连接器只交换插入（Insert）消息。

##### 撤回模式（Retract Mode）

在撤回模式下，表和外部连接器交换的是：添加（Add）和撤回（Retract）消息。

* 插入（Insert）会被编码为添加消息；
* 删除（Delete）则编码为撤回消息；
* 更新（Update）则会编码为，已更新行（上一行）的撤回消息，和更新行（新行）的添加消息。

在此模式下，不能定义key，这一点跟upsert模式完全不同。

##### Upsert（更新插入）模式

在Upsert模式下，动态表和外部连接器交换Upsert和Delete消息。

这个模式需要一个唯一的key，通过这个key可以传递更新消息。为了正确应用消息，外部连接器需要知道这个唯一key的属性。

* 插入（Insert）和更新（Update）都被编码为Upsert消息；
* 删除（Delete）编码为Delete信息。

这种模式和Retract模式的主要区别在于，Upsert操作是用单个消息编码的，所以效率会更高。

#### 输出到Kafka

|  |
| --- |
| public class KafkaSinkTest {  public static void main(String[] args) throws Exception {  // 0. 创建流执行环境，读取数据并转换成样例类类型  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();   env.getConfig().setRestartStrategy(RestartStrategies.noRestart());  env.setParallelism(1);   // 1. 创建表执行环境  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, bsSettings);   // 2 定义到kafka的连接，创建输入表  tableEnv.connect(new Kafka()  .version("universal") // 定义版本  .topic("order") // 定义主题  .property("bootstrap.servers", "node01:9092,node02:9092,node03:9092")  )  .withFormat(new Csv())  .withSchema(new Schema()  .field("id", DataTypes.STRING())  .field("timestamp", DataTypes.STRING())  .field("category", DataTypes.STRING())  .field("areaName", DataTypes.STRING())  .field("money", DataTypes.DOUBLE())  .field("pt", DataTypes.TIMESTAMP(3))  .proctime()//使用.proctime，指定字段名定义处理时间字段,这个proctime属性只能通过附加逻辑字段，来扩展物理schema  )  .createTemporaryTable("kafkaInputTable");   // 3. 表的查询转换  Table orderTable = tableEnv.from("kafkaInputTable");   // 3.1 调用table api，做转换操作  Table resultTable = orderTable  .select($("id"),$("timestamp"),$("category"),$("areaName"),$("money"))  .filter($("areaName").isEqual("北京"));   Table aggResultTable = orderTable  .groupBy($("areaName"))  .select($("areaName"), $("id").count().as("cnt"));   tableEnv.connect(new Kafka()  .version("universal") // 定义版本  .topic("orderTest") // 定义主题  .property("bootstrap.servers", "node01:9092,node02:9092,node03:9092")  )  .withFormat(new Csv())  .withSchema(new Schema()  .field("id", DataTypes.STRING())  .field("timestamp", DataTypes.STRING())  .field("category", DataTypes.STRING())  .field("areaName", DataTypes.STRING())  .field("money", DataTypes.DOUBLE())  )  .createTemporaryTable("kafkaOutputTable");   resultTable.executeInsert("kafkaOutputTable");  } } |

#### 输出到MySql

Flink专门为Table API的jdbc连接提供了flink-jdbc连接器，我们需要先引入依赖：

|  |
| --- |
| <!--link-jdbc--> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-jdbc\_2.11</artifactId>  <version>${flink.version}</version> </dependency> |

**需求：**消费kafka数据，实时写入到mysql数据库中

* 样本数据：

|  |
| --- |
| {"id":1,"timestamp":"2020-05-08T01:03.00Z","category":"电脑","areaName":"石家庄","money":"1450"} {"id":2,"timestamp":"2020-05-08T01:01.00Z","category":"手机","areaName":"北京","money":"1450"} {"id":3,"timestamp":"2020-05-08T01:03.00Z","category":"手机","areaName":"北京","money":"8412"} {"id":4,"timestamp":"2020-05-08T05:01.00Z","category":"电脑","areaName":"上海","money":"1513"} {"id":5,"timestamp":"2020-05-08T01:03.00Z","category":"家电","areaName":"北京","money":"1550"} {"id":6,"timestamp":"2020-05-08T01:01.00Z","category":"电脑","areaName":"深圳","money":"1550"} |

操作步骤：

* 创建kafka数据源Topic

|  |
| --- |
| cd /export/servers/kafka\_2.11-0.10.0.0  bin/kafka-topics.sh --create --zookeeper node01:2181 --replication-factor 2 --partitions 3 --topic orderjson |

* Kafka生产者命令行

|  |
| --- |
| bin/kafka-console-producer.sh --broker-list node01:9092 --topic orderjson |

* 创建mysql数据表

|  |
| --- |
| **DROP** **TABLE** **IF** **EXISTS** `order\_test`**;**  **CREATE** **TABLE** `order\_test` **(**  `id` **varchar(**255**)** **NOT** **NULL,**  `**timestamp**` **varchar(**255**)** **DEFAULT** **NULL,**  `category` **varchar(**255**)** **DEFAULT** **NULL,**  `areaName` **varchar(**255**)** **DEFAULT** **NULL,**  `money` **double** **DEFAULT** **NULL,**  **PRIMARY** **KEY** **(**`id`**)**  **)** ENGINE**=**InnoDB **DEFAULT** CHARSET**=**utf8**;** |

* 代码如下

|  |
| --- |
| public class MysqlSinkTest {  public static void main(String[] args) {  // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // 01.定义Blink执行环境  // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  StreamExecutionEnvironment bsEnv = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();  StreamTableEnvironment bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings);  // or val bsTableEnv = TableEnvironment.create(bsSettings)  // 02.定义kafka输入表  String souce\_table =  "create table kafkaInputTable (" +  " id varchar," +  " `timestamp` varchar," +  " category varchar," +  " areaName varchar," +  " money double" +  " ) with (" +  " 'connector' = 'kafka'," +  " 'topic' = 'orderjson'," +  " 'properties.bootstrap.servers'='node01:9092,node02:9092,node03:9092'," +  " 'scan.startup.mode' = 'earliest-offset'," +  " 'properties.group.id' = 'testGroup'," +  " 'format' = 'json'" +  " )";  bsTableEnv.executeSql(souce\_table);   Table table = bsTableEnv.from("kafkaInputTable");  // 03.将kafka数据转换成流打印  DataStream<Row> value = bsTableEnv.toAppendStream(table, Row.class);  value.print();  // 04.定义mysql输出表  String sink\_table = "create table order\_test (" +  " id varchar," +  " `timestamp` varchar," +  " category varchar," +  " areaName varchar," +  " money double" +  ") with (" +  " 'connector' = 'jdbc'," +  " 'url' = 'jdbc:mysql://node03:3306/test?characterEncoding=utf-8&useSSL=false'," +  " 'table-name' = 'order\_test'," +  " 'driver'='com.mysql.jdbc.Driver'," +  " 'username' = 'root'," +  " 'password' = '123456'," +  " 'sink.buffer-flush.interval'='1s'," +  " 'sink.buffer-flush.max-rows'='1'," +  " 'sink.max-retries' = '5'" +  ")";  // 05.查询kafka table并插入mysql table  String insert = "INSERT INTO order\_test SELECT \* FROM kafkaInputTable";   bsTableEnv.executeSql(sink\_table);  bsTableEnv.executeSql(insert);   try {  bsEnv.execute();  } catch (Exception exception) {  exception.printStackTrace();  }  } } |

### 将表转换成DataStream

表可以转换为DataStream或DataSet。这样，自定义流处理或批处理程序就可以继续在 Table API或SQL查询的结果上运行了。

将表转换为DataStream或DataSet时，需要指定生成的数据类型，即要将表的每一行转换成的数据类型。通常，最方便的转换类型就是Row。当然，因为结果的所有字段类型都是明确的，我们也经常会用元组类型来表示。

表作为流式查询的结果，是动态更新的。所以，将这种动态查询转换成的数据流，同样需要对表的更新操作进行编码，进而有不同的转换模式。

Table API中表到DataStream有两种模式：

* 追加模式（Append Mode）：用于表只会被插入（Insert）操作更改的场景。
* 撤回模式（Retract Mode）：用于任何场景。有些类似于更新模式中Retract模式，它只有Insert和Delete两类操作。得到的数据会增加一个Boolean类型的标识位（返回的第一个字段），用它来表示到底是新增的数据（Insert），还是被删除的数据（老数据， Delete）

|  |
| --- |
| tableEnv.toAppendStream(resultTable, Row.class).print("result"); tableEnv.toRetractStream(aggResultSqlTable, Row.class).print("agg result"); |

所以，没有经过groupby之类聚合操作，可以直接用 toAppendStream 来转换；而如果经过了聚合，有更新操作，一般就必须用 toRetractDstream。

### Query的解释和执行

Table API提供了一种机制来解释（Explain）计算表的逻辑和优化查询计划。这是通过TableEnvironment.explain（table）方法或TableEnvironment.explain（）方法完成的。

explain方法会返回一个字符串，描述三个计划：

* 未优化的逻辑查询计划
* 优化后的逻辑查询计划
* 实际执行计划

我们可以在代码中查看执行计划：

|  |
| --- |
| public class ExplainSqlTest {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);   DataStream<Tuple2<Integer, String>> stream1 = env.fromElements(new Tuple2<>(1, "hello"));  DataStream<Tuple2<Integer, String>> stream2 = env.fromElements(new Tuple2<>(1, "hello"));   // explain Table API  Table table1 = tEnv.fromDataStream(stream1, $("count"), $("word"));  Table table2 = tEnv.fromDataStream(stream2, $("count"), $("word"));  Table table = table1  .where($("word").like("F%"))  .unionAll(table2);   System.out.println(table.explain());  } } |

Query的解释和执行过程，老planner和blink planner大体是一致的，又有所不同。整体来讲，Query都会表示成一个逻辑查询计划，然后分两步解释：

1. 优化查询计划

2. 解释成 DataStream 或者 DataSet程序

而Blink版本是批流统一的，所以所有的Query，只会被解释成DataStream程序；另外在批处理环境TableEnvironment下，Blink版本要到tableEnv.execute()执行调用才开始解释。

## SqlClient工具的使用

Flink 的 Table & SQL API 可以处理 SQL 语言编写的查询语句，但是这些查询需要嵌入用 Java 或 Scala 编写的表程序中。此外，这些程序在提交到集群前需要用构建工具打包。这或多或少限制了 Java/Scala 程序员对 Flink 的使用。

SQL 客户端 的目的是提供一种简单的方式来编写、调试和提交表程序到 Flink 集群上，而无需写一行 Java 或 Scala 代码。SQL 客户端命令行界面（CLI） 能够在命令行中检索和可视化分布式应用中实时产生的结果。

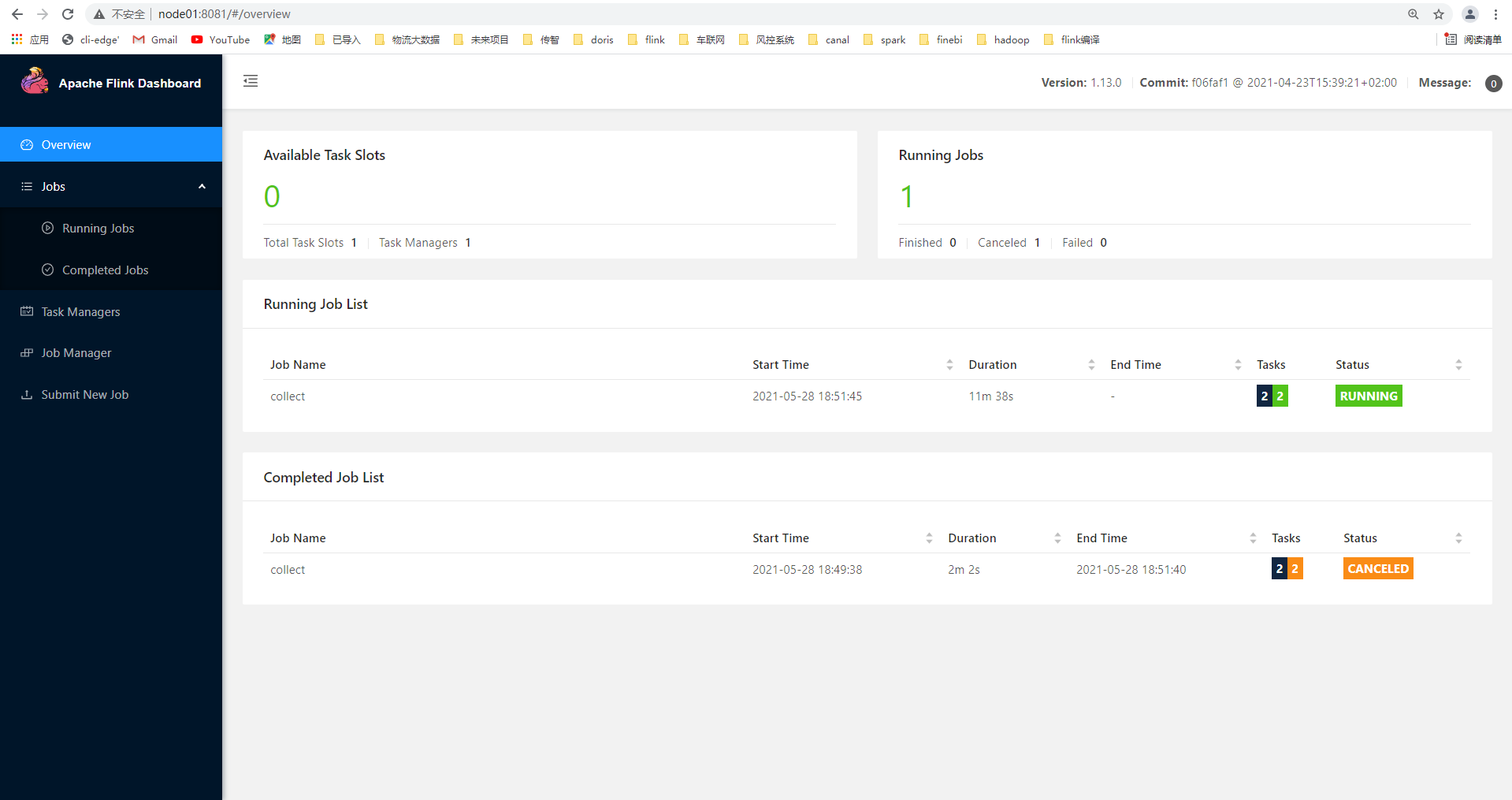


### 入门

本节介绍如何在命令行里启动（setup）和运行你的第一个 Flink SQL 程序。

SQL 客户端捆绑在常规 Flink 发行版中，因此可以直接运行。它仅需要一个正在运行的 Flink 集群就可以在其中执行表程序。有关设置 Flink 群集的更多信息，请参见[集群和部署](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/deployment/resource-providers/standalone/overview/)部分。如果仅想试用 SQL 客户端，也可以使用以下命令启动本地集群：

|  |
| --- |
| ./bin/start-cluster.sh |



#### 启动 SQL 客户端命令行界面

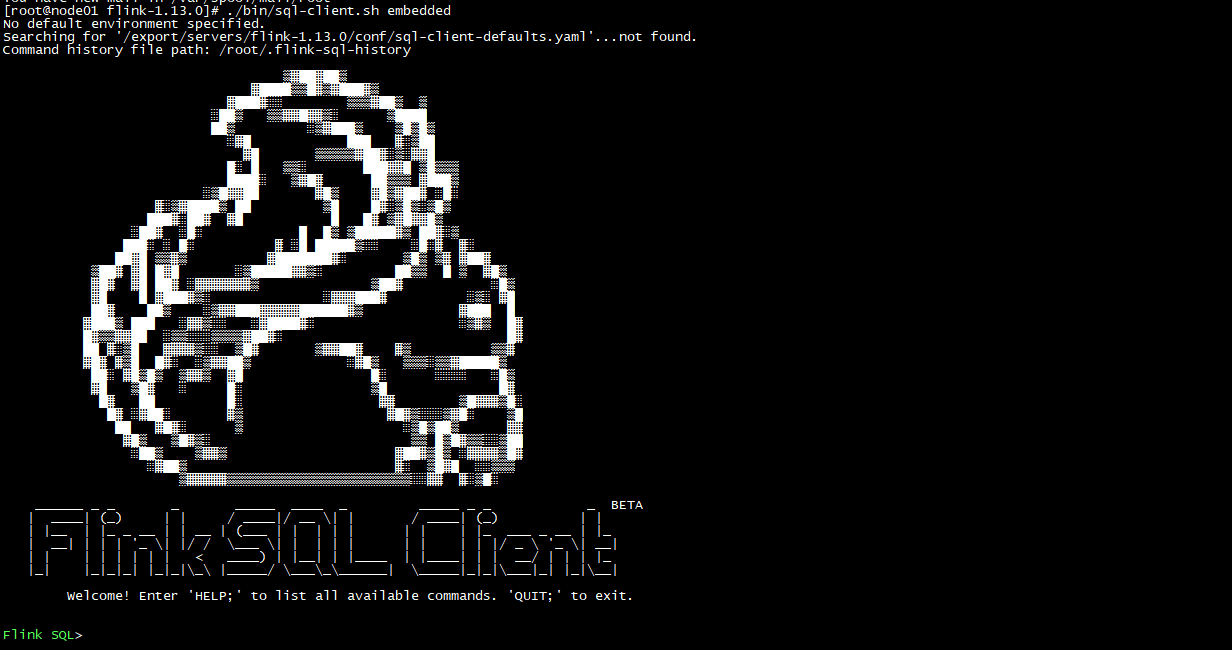
SQL Client 脚本也位于 Flink 的 bin 目录中。用户可以通过启动嵌入式 standalone 进程或通过连接到远程 SQL 客户端网关来启动 SQL 客户端命令行界面。目前仅支持 **embedded**，模式默认值**embedded**。可以通过以下方式启动 CLI：

|  |
| --- |
| ./bin/start-cluster.sh |

或者显式使用 embedded 模式:

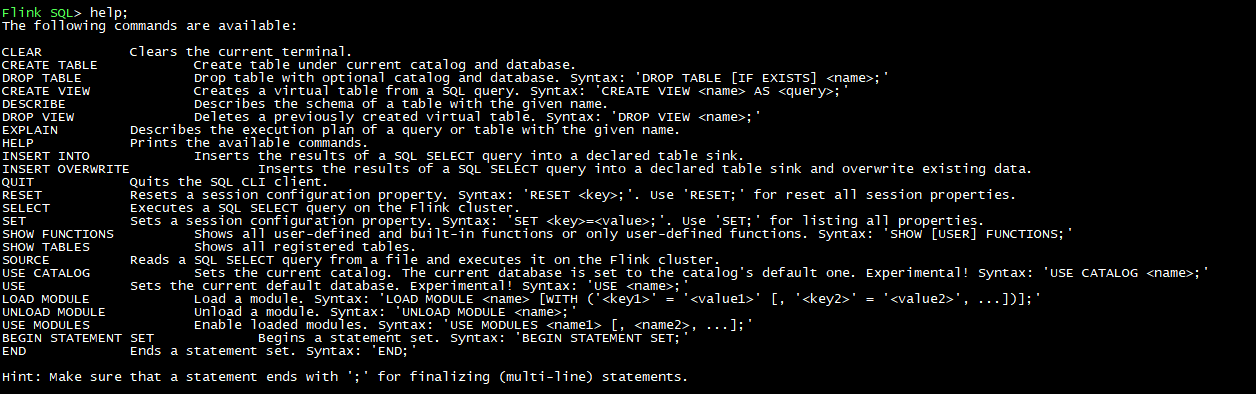
|  |
| --- |
| ./bin/sql-client.sh embedded |

启动成功 会进入 flink sql> 命令行界面 **(输入 quit; 退出)**



#### 执行 SQL 查询

命令行界面启动后，你可以使用 **help**命令列出所有可用的 SQL 语句。输入第一条 SQL 查询语句并按**Enter**键执行，可以验证你的设置及集群连接是否正确：



|  |
| --- |
| SELECT 'Hello World'; |
|  |

**默认情况下输出默认采用的是表格模式**，在上面的演示中该查询不需要 **table source**，因为只产生一行结果。CLI 将从集群中检索结果并将其可视化。按 Q 键退出结果视图。

CLI 为维护和可视化结果提供三种模式：

* 表格模式（table mode）在内存中实体化结果，并将结果用规则的分页表格可视化展示出来。执行如下命令启用：

|  |
| --- |
| SET sql-client.execution.result-mode=table; |

* 变更日志模式（changelog mode）不会实体化和可视化结果，而是由插入（+）和撤销（-）组成的持续查询产生结果流。

|  |
| --- |
| SET sql-client.execution.result-mode=changelog; |

* Tableau模式（tableau mode）更接近传统的数据库，会将执行的结果以制表的形式直接打在屏幕之上。具体显示的内容会取决于作业 执行模式的不同(execution.type)：

|  |
| --- |
| SET sql-client.execution.result-mode=tableau; |

注意当你使用这个模式运行一个流式查询的时候，Flink 会将结果持续的打印在当前的屏幕之上。如果这个流式查询的输入是有限的数据集， 那么Flink在处理完所有的数据之后，会自动的停止作业，同时屏幕上的打印也会相应的停止。如果你想提前结束这个查询，那么可以直接使用**CTRL-C**按键，这个会停掉作业同时停止屏幕上的打印。

**可以用如下查询来查看三种结果模式的运行情况：**

|  |
| --- |
| SELECT name, COUNT(\*) AS cnt FROM (VALUES ('Bob'), ('Alice'), ('Greg'), ('Bob')) AS NameTable(name) GROUP BY name; |
|  |

此查询执行一个有限字数示例：

* 变更日志模式 下，看到的结果应该类似于：

|  |
| --- |
| SET sql-client.execution.result-mode=changelog; |
|  |
| SELECT name, COUNT(\*) AS cnt FROM (VALUES ('Bob'), ('Alice'), ('Greg'), ('Bob')) AS NameTable(name) GROUP BY name; |
|  |

* 表格模式 下，可视化结果表将不断更新，直到表程序以如下内容结束：

|  |
| --- |
| SET sql-client.execution.result-mode=table; |
|  |
| SELECT name, COUNT(\*) AS cnt FROM (VALUES ('Bob'), ('Alice'), ('Greg'), ('Bob')) AS NameTable(name) GROUP BY name; |
|  |

* Tableau模式 下，如果这个查询以流的方式执行，那么将显示以下内容：

|  |
| --- |
| SET sql-client.execution.result-mode=tableau; |
|  |
| SELECT name, COUNT(\*) AS cnt FROM (VALUES ('Bob'), ('Alice'), ('Greg'), ('Bob')) AS NameTable(name) GROUP BY name; |
|  |

这几种结果模式在 SQL 查询的原型设计过程中都非常有用。这些模式的结果都存储在 SQL 客户端 的 Java 堆内存中。

为了保持 CLI 界面及时响应，变更日志模式仅显示最近的 1000 个更改。表格模式支持浏览更大的结果，这些结果仅受可用主内存和配置的[最大行数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sqlclient/" \l "sql-client-execution-max-table-result-rows)（**sql-client.execution.max-table-result.rows**）的限制。

|  |
| --- |
| **注意：**在批处理环境下执行的查询只能用表格模式或者Tableau模式进行检索。  定义查询语句后，可以将其作为长时间运行的独立 Flink 作业提交给集群。  [配置部分](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sqlclient/" \l "configuration)解释如何声明读取数据的 table source，写入数据的 sink 以及配置其他表程序属性的方法。 |

### 配置

#### SQL 客户端启动选项

可以使用以下可选的 CLI 命令启动 SQL 客户端。它们将在随后的段落中详细讨论。

|  |
| --- |
| ./bin/sql-client.sh --help  Mode "embedded" (default) submits Flink jobs from the local machine.  Syntax: [embedded] [OPTIONS]  "embedded" mode options:  -d,--defaults <environment file> Deprecated feature: the environment  properties with which every new  session is initialized. Properties  might be overwritten by session  properties.  -e,--environment <environment file> Deprecated feature: the environment  properties to be imported into the  session. It might overwrite default  environment properties.  -f,--file <script file> Script file that should be executed.  In this mode, the client will not  open an interactive terminal.  -h,--help Show the help message with  descriptions of all options.  -hist,--history <History file path> The file which you want to save the  command history into. If not  specified, we will auto-generate one  under your user's home directory.  -i,--init <initialization file> Script file that used to init the  session context. If get error in  execution, the sql client will exit.  Notice it's not allowed to add query  or insert into the init file.  -j,--jar <JAR file> A JAR file to be imported into the  session. The file might contain  user-defined classes needed for the  execution of statements such as  functions, table sources, or sinks.  Can be used multiple times.  -l,--library <JAR directory> A JAR file directory with which every  new session is initialized. The files  might contain user-defined classes  needed for the execution of  statements such as functions, table  sources, or sinks. Can be used  multiple times.  -pyarch,--pyArchives <arg> Add python archive files for job. The  archive files will be extracted to  the working directory of python UDF  worker. Currently only zip-format is  supported. For each archive file, a  target directory be specified. If the  target directory name is specified,  the archive file will be extracted to  a directory with the  specified name. Otherwise, the  archive file will be extracted to a  directory with the same name of the  archive file. The files uploaded via  this option are accessible via  relative path. '#' could be used as  the separator of the archive file  path and the target directory name.  Comma (',') could be used as the  separator to specify multiple archive  files. This option can be used to  upload the virtual environment, the  data files used in Python UDF (e.g.:  --pyArchives  file:///tmp/py37.zip,file:///tmp/data  .zip#data --pyExecutable  py37.zip/py37/bin/python). The data  files could be accessed in Python  UDF, e.g.: f = open('data/data.txt',  'r').  -pyexec,--pyExecutable <arg> Specify the path of the python  interpreter used to execute the  python UDF worker (e.g.:  --pyExecutable  /usr/local/bin/python3). The python  UDF worker depends on Python 3.6+,  Apache Beam (version == 2.27.0), Pip  (version >= 7.1.0) and SetupTools  (version >= 37.0.0). Please ensure  that the specified environment meets  the above requirements.  -pyfs,--pyFiles <pythonFiles> Attach custom files for job.  The standard resource file suffixes  such as .py/.egg/.zip/.whl or  directory are all supported. These  files will be added to the PYTHONPATH  of both the local client and the  remote python UDF worker. Files  suffixed with .zip will be extracted  and added to PYTHONPATH. Comma (',')  could be used as the separator to  specify multiple files (e.g.:  --pyFiles  file:///tmp/myresource.zip,hdfs:///$n  amenode\_address/myresource2.zip).  -pyreq,--pyRequirements <arg> Specify a requirements.txt file which  defines the third-party dependencies.  These dependencies will be installed  and added to the PYTHONPATH of the  python UDF worker. A directory which  contains the installation packages of  these dependencies could be specified  optionally. Use '#' as the separator  if the optional parameter exists  (e.g.: --pyRequirements  file:///tmp/requirements.txt#file:///  tmp/cached\_dir).  -s,--session <session identifier> The identifier for a session.  'default' is the default identifier.  -u,--update <SQL update statement> Deprecated Experimental (for testing  only!) feature: Instructs the SQL  Client to immediately execute the  given update statement after starting  up. The process is shut down after  the statement has been submitted to  the cluster and returns an  appropriate return code. Currently,  this feature is only supported for  INSERT INTO statements that declare  the target sink table.Please use  option -f to submit update statement. |

#### SQL 客户端配置

|  |  |  |  |
| --- | --- | --- | --- |
| **配置项** | **默认值** | **类型** | **说明** |
| sql-client.execution.max-table-result.rows | 1000000 | Integer | 在表模式下要缓存的行数。如果行数超过指定值，它将以FIFO样式重试该行。 |
| sql-client.execution.result-mode | TABLE | Enum  Possible values: [TABLE, CHANGELOG, TABLEAU] | 确定显示查询结果的方式。可用值为['table'，'tableau'，'changelog']。“table”模式在内存中具体化结果，并以常规的分页表表示形式将它们可视化。“changelog”模式不会具体化结果，而是可视化由连续查询生成的结果流。“tableau”模式更像是一种传统的方式，它将以tableau格式直接在屏幕上显示结果。 |
| sql-client.verbose | false | Boolean | 确定是否将详细输出输出到控制台。如果将选项设置为true，它将打印异常堆栈。否则，它只输出原因。 |

#### 使用 SQL 文件初始化会话

SQL 查询需要在其中执行的配置环境。SQL 客户端支持**-i**启动选项，以便在启动SQL 客户端时执行初始化 SQL 文件以设置环境。所谓的初始化 SQL 文件，可以使用 DDL 来定义可用的目录、表源和接收器、用户自定义函数以及其他执行和部署所需的属性。

下面给出了此类文件的示例。

|  |
| --- |
| -- 定义可用目录  CREATE CATALOG MyCatalog  WITH (  'type' = 'hive'  );  USE CATALOG MyCatalog;  -- 定义可用数据库  CREATE DATABASE MyDatabase;  USE MyDatabase;  -- 定义表格  CREATE TABLE MyTable(  MyField1 INT,  MyField2 STRING ) WITH (  'connector' = 'filesystem',  'path' = '/path/to/something',  'format' = 'csv' );  -- 定义视图  CREATE VIEW MyCustomView AS SELECT MyField2 FROM MyTable;  -- 在此处定义用户定义的函数。  CREATE FUNCTION foo.bar.AggregateUDF AS myUDF;  -- 更改表程序基本执行行为的属性。  SET table.planner = blink; -- 可选：'blink' （默认）或 'old' SET execution.runtime-mode = streaming; -- 必选：执行模式为 'batch' 或 'streaming' SET sql-client.execution.result-mode = table; -- 必选：'table', 'changelog' 或 'tableau' SET sql-client.execution.max-table-result.rows = 10000; -- 可选：'table' 模式下可维护的最大行数（默认为 1000000，小于 1 则表示无限制） SET parallelism.default = 1; -- 可选：Flink的并行度（默认为1） SET pipeline.auto-watermark-interval = 200; -- 可选：周期水印的间隔 SET pipeline.max-parallelism = 10; -- 可选：Flink的最大并行度 SET table.exec.state.ttl=1000; -- 可选：表程序的空闲状态时间 SET restart-strategy = fixed-delay;  -- 用于调整和调整表程序的配置选项。  SET table.optimizer.join-reorder-enabled = true; SET table.exec.spill-compression.enabled = true; SET table.exec.spill-compression.block-size = 128kb; |

这个配置：

* 连接到 Hive 目录并MyCatalog用作当前目录和目录MyDatabase的当前数据库，
* 定义一个MyTableSource可以从 CSV 文件中读取数据的表，
* 定义一个MyCustomView使用 SQL 查询声明虚拟表的视图，
* 定义一个myUDF可以使用类名实例化的用户定义函数，
* 在流模式下使用blink计划器运行语句，并行度为 1，
* 在table结果模式下运行探索性查询，
* 并通过配置选项围绕连接重新排序和溢出进行一些规划器调整。

使用**-i <init.sql>**选项初始化 SQL 客户端会话时，初始化 SQL 文件中允许使用以下语句：

* DDL（创建/删除/更改），
* 使用目录/数据库，
* 加载/卸载模块，
* 设置命令，
* 复位命令。

执行查询或插入语句时，请进入交互模式或使用-f选项提交SQL语句。

|  |
| --- |
| **注意：**如果 SQL Client 在初始化时遇到错误，SQL Client 将退出并显示错误信息。 |

#### 依赖关系

SQL 客户端不需要使用 Maven 或 SBT 设置 Java 项目。相反，您可以将**依赖项**作为提交到集群的常规 JAR 文件传递。您可以单独指定每个 JAR 文件（使用--jar）或定义整个库目录（使用--library）。对于连接外部系统（如 Apache Kafka）和相应数据格式（如 JSON）的连接器，Flink 提供了即用型 JAR 包。可以从 Maven 中央存储库为每个版本下载这些 JAR 文件。

### 使用 SQL Client 提交作业

SQL 客户端允许用户在交互式命令行中或使用**-f选项**执行 sql 文件提交作业。

在这两种模式下，SQL Client 都支持解析和执行 Flink 支持的所有类型的 SQL 语句。

#### 交互式命令行

在交互式命令行中，SQL 客户端读取用户输入并在获取分号 **( ;)**时执行语句。

如果语句成功执行，SQL 客户端将打印成功消息。当出现错误时，SQL Client 也会打印错误信息。默认情况下，错误消息仅包含错误原因。为了打印完整的异常堆栈以进行调试，请sql-client.verbose通过命令将 设置 为 true **SET sql-client.verbose = true;**

#### 执行 SQL 文件

SQL 客户端支持使用该-f选项执行 SQL 脚本文件。SQL Client 会一一执行SQL 脚本文件中的语句，并为每条执行的语句打印执行消息。一旦一条语句失败，SQL Client 就会存在，并且不会执行所有剩余的语句。

下面给出了此类文件的示例。

|  |
| --- |
| CREATE TEMPORARY TABLE users (  user\_id BIGINT,  user\_name STRING,  user\_level STRING,  region STRING,  PRIMARY KEY (user\_id) NOT ENFORCED ) WITH (  'connector' = 'upsert-kafka',  'topic' = 'users',  'properties.bootstrap.servers' = '...',  'key.format' = 'csv',  'value.format' = 'avro' );  -- set sync mode SET table.dml-sync=true;  -- set the job name SET pipeline.name=SqlJob;  -- set the queue that the job submit to SET yarn.application.queue=root;  -- set the job parallism SET parallism.default=100;  -- restore from the specific savepoint path SET execution.savepoint.path=/tmp/flink-savepoints/savepoint-cca7bc-bb1e257f0dab;  INSERT INTO pageviews\_enriched SELECT \* FROM pageviews AS p LEFT JOIN users FOR SYSTEM\_TIME AS OF p.proctime AS u ON p.user\_id = u.user\_id; |

这个配置：

* 定义users从CSV文件读取的时态表源，
* 设置属性，例如作业名称，
* 设置保存点路径，
* 提交从指定保存点路径加载保存点的 sql 作业。

|  |
| --- |
| **注意：**与交互模式相比，SQL Client 遇到错误会停止执行并退出。 |

#### 执行一组SQL语句

SQL Client 将每个 INSERT INTO 语句作为单个 Flink 作业执行。但是，这有时不是最佳的，因为管道的某些部分可以重复使用。SQL Client支持STATEMENT SET语法来执行一组SQL语句。这是 Table API 中 StatementSet 的等效功能。该 STATEMENT SET语法围绕一个或多个INSERT INTO语句。STATEMENT SET 块中的所有语句都经过整体优化并作为单个 Flink 作业执行。

联合优化和执行允许重用常见的中间结果，因此可以显着提高执行多个查询的效率。

语法

|  |
| --- |
| BEGIN STATEMENT SET;  -- one or more INSERT INTO statements  { INSERT INTO|OVERWRITE <select\_statement>; }+ END; |

|  |
| --- |
| **注意：**括号中的语句STATEMENT SET必须用分号 (;) 分隔。 |

|  |
| --- |
| CREATE TABLE pageviews (  user\_id BIGINT,  page\_id BIGINT,  viewtime TIMESTAMP,  proctime AS PROCTIME() ) WITH (  'connector' = 'kafka',  'topic' = 'pageviews',  'properties.bootstrap.servers' = '...',  'format' = 'avro' );  CREATE TABLE pageview (  page\_id BIGINT,  cnt BIGINT ) WITH (  'connector' = 'jdbc',  'url' = 'jdbc:mysql://localhost:3306/mydatabase',  'table-name' = 'pageview' );  CREATE TABLE uniqueview (  page\_id BIGINT,  cnt BIGINT ) WITH (  'connector' = 'jdbc',  'url' = 'jdbc:mysql://localhost:3306/mydatabase',  'table-name' = 'uniqueview' );  BEGIN STATEMENT SET;  INSERT INTO pageviews SELECT page\_id, count(1) FROM pageviews GROUP BY page\_id;  INSERT INTO uniqueview SELECT page\_id, count(distinct user\_id) FROM pageviews GROUP BY page\_id;  END; |

#### 执行 DML 语句同步/异步

默认情况下，SQL Client异步执行DML语句。这意味着，SQL Client将针对DML语句的作业提交给Flink群集，而不等待作业完成。所以SQL Client 可以同时提交多个作业。这对于通常长时间运行的流作业很有用。

SQL 客户端确保语句成功提交到集群。提交语句后，CLI 将显示有关 Flink 作业的信息。

|  |
| --- |
| Flink SQL> INSERT INTO MyTableSink SELECT *\** FROM MyTableSource; [INFO] Table update statement has been successfully submitted to the cluster: Cluster ID: StandaloneClusterId Job ID: 6f922fe5cba87406ff23ae4a7bb79044 |

|  |
| --- |
| **注意：**SQL Client 提交后不会跟踪正在运行的 Flink 作业的状态。CLI 进程可以在提交后关闭而不影响分离的查询。Flinkrestart strategy负责容错。可以使用 Flink 的 Web 界面、命令行或 REST API 取消查询。  但是，对于批处理用户，更常见的是下一个 DML 语句需要等待 util 前一个 DML 语句完成。为了同步执行 DML 语句，您可以table.dml-sync在 SQL Client 中设置 选项 true。 |

|  |
| --- |
| Flink SQL> SET table.dml-sync = true; [INFO] Session property has been set.  Flink SQL> INSERT INTO MyTableSink SELECT *\** FROM MyTableSource; [INFO] Submitting SQL update statement to the cluster... [INFO] Execute statement in sync mode. Please wait for the execution finish... [INFO] Complete execution of the SQL update statement. |

|  |
| --- |
| **注意：** 如果要终止作业，只需键入CTRL-C取消执行即可。 |

#### 从保存点启动 SQL 作业

Flink 支持使用指定的保存点启动作业。在 SQL Client 中，允许使用SETcommand 指定保存点的路径。

|  |
| --- |
| Flink SQL> SET execution.savepoint.path=/tmp/flink-savepoints/savepoint-cca7bc-bb1e257f0dab; [INFO] Session property has been set.  -- all the following DML statements will be restroed from the specified savepoint path Flink SQL> INSERT INTO ... |

当指定了保存点的路径时，Flink 会在执行以下所有 DML 语句时尝试从保存点恢复状态。

因为指定的保存点路径会影响后面所有的DML 语句，你可以使用RESET命令来重置这个配置选项，即禁止从保存点恢复。

|  |
| --- |
| Flink SQL> RESET execution.savepoint.path; [INFO] Session property has been reset. |

#### 定义自定义作业名称

SQL 客户端支持通过SET命令为查询和 DML 语句定义作业名称。

|  |
| --- |
| Flink SQL> SET pipeline.name= 'kafka-to-hive' ; [INFO] Session property has been set.  -- all the following DML statements will use the specified job name. Flink SQL> INSERT INTO ... |

因为指定的作业名会影响后面所有的查询和DML 语句，你也可以使用RESET命令来重置这个配置，即使用默认的作业名。

|  |
| --- |
| Flink SQL> RESET pipeline.name; [INFO] Session property has been reset. |

如果pipeline.name未指定该选项，SQL 客户端将为提交的作业生成一个默认名称，例如insert-into\_<sink\_table\_name>for INSERT INTOstatements。

## Catalogs

Catalog 提供了元数据信息，例如数据库、表、分区、视图以及数据库或其他外部系统中存储的函数和信息。

数据处理最关键的方面之一是管理元数据。

元数据可以是临时的，例如临时表、或者通过 TableEnvironment 注册的 UDF。

元数据也可以是持久化的，例如 Hive Metastore 中的元数据。

Catalog 提供了一个统一的API，用于管理元数据，并使其可以从 Table API 和 SQL 查询语句中来访问。

### Catalog 类型

#### GenericInMemoryCatalog

GenericInMemoryCatalog 是**基于内存实现的 Catalog**，所有元数据只在 session 的生命周期内可用。

#### JdbcCatalog

JdbcCatalog 使得用户可以将 Flink 通过 JDBC 协议连接到关系数据库。**PostgresCatalog 是当前实现的唯一一种 JDBC Catalog**。 参考 [JdbcCatalog](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/jdbc/)  获取关于配置 JDBC catalog 的详细信息。

#### HiveCatalog

HiveCatalog 有两个用途：**作为原生 Flink 元数据的持久化存储，以及作为读写现有 Hive 元数据的接口**。 Flink 的 [Hive 文档](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/overview/) 提供了有关设置 HiveCatalog 以及访问现有 Hive 元数据的详细信息。

#### 用户自定义 Catalog

Catalog 是可扩展的，用户可以通过实现 Catalog 接口来开发自定义 Catalog。 想要在 SQL CLI 中使用自定义 Catalog，用户除了需要实现自定义的 Catalog 之外，还需要为这个 Catalog 实现对应的 CatalogFactory 接口。

CatalogFactory 定义了一组属性，用于 SQL CLI 启动时配置 Catalog。 这组属性集将传递给发现服务，在该服务中，服务会尝试将属性关联到 CatalogFactory 并初始化相应的 Catalog 实例。

### 创建 Flink 表并将其注册到 Catalog

#### 使用 SQL DDL

用户可以使用 DDL 通过 Table API 或者 SQL Client 在 Catalog 中创建表：

|  |
| --- |
| public class SqlDDLDemo {  public static void main(String[] args) {  //设置当前hadoop操作的用户名  System.setProperty("HADOOP\_USER\_NAME", "root");   StreamExecutionEnvironment streamEnv = StreamExecutionEnvironment.getExecutionEnvironment();  streamEnv.setParallelism(5);   EnvironmentSettings tableEnvSettings = EnvironmentSettings.newInstance()  .useBlinkPlanner()  .inStreamingMode()  .build();   StreamTableEnvironment tableEnv = StreamTableEnvironment.create(streamEnv, tableEnvSettings);   String catalogName = "myhive";  String databaseName = "itcast";  String tableName = "test";  HiveCatalog catalog = new HiveCatalog(  catalogName, // catalog name  "default", // default database  "src/main/resources", // Hive config (hive-site.xml) directory  "2.1.1" // Hive version  );  //注册目录  tableEnv.registerCatalog(catalogName, catalog);  tableEnv.useCatalog(catalogName);   System.out.println("---------------创建数据库------------------------");  String createDbSql = "CREATE DATABASE IF NOT EXISTS "+catalogName+"."+databaseName;  tableEnv.executeSql(createDbSql);   System.out.println("---------------创建表------------------------");  tableEnv.executeSql("CREATE TABLE mytable (name STRING, age INT)");   System.out.println("---------------列出数据库中的表------------------------");  tableEnv.listTables();  } } |

#### 使用 Java/Scala

|  |
| --- |
| /\*\*  \* 用户可以用编程的方式使用Java 或者 Scala 来创建 Catalog 表。  \*/ public class JavaClientDemo {  public static void main(String[] args) throws DatabaseAlreadyExistException, TableAlreadyExistException, DatabaseNotExistException {  //设置当前hadoop操作的用户名  System.setProperty("HADOOP\_USER\_NAME", "root");   StreamExecutionEnvironment streamEnv = StreamExecutionEnvironment.getExecutionEnvironment();  streamEnv.setParallelism(5);   EnvironmentSettings tableEnvSettings = EnvironmentSettings.newInstance()  .useBlinkPlanner()  .inStreamingMode()  .build();   StreamTableEnvironment tableEnv = StreamTableEnvironment.create(streamEnv, tableEnvSettings);   String catalogName = "myhive";  String databaseName = "itcast";  String tableName = "test";  HiveCatalog catalog = new HiveCatalog(  catalogName, // catalog name  "default", // default database  "src/main/resources", // Hive config (hive-site.xml) directory  "2.1.1" // Hive version  );  //注册目录  tableEnv.registerCatalog(catalogName, catalog);  tableEnv.useCatalog(catalogName);   System.out.println("---------------创建数据库------------------------");  catalog.createDatabase(databaseName,  new CatalogDatabaseImpl(new HashMap<>(), "my comment"), true);   System.out.println("--------------创建表table-----------------------------");  // Create a catalog table  TableSchema schema = TableSchema.builder()  .field("name", DataTypes.STRING())  .field("age", DataTypes.INT())  .build();   Map<String, String> properties = new HashMap<>();  //properties.put(CatalogConfig.IS\_GENERIC, String.valueOf(true));  properties.put("connector", "COLLECTION");  catalog.createTable(  new ObjectPath(databaseName, tableName),  new CatalogTableImpl(  schema,  properties,  "my comment"),  true  );  System.out.println(tableName);  } } |

### [Catalog API](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/catalogs/" \l "catalog-api)

#### 数据库操作

|  |
| --- |
| /\*\*  \* 数据库操作  \*/ public class DatabaseOperater {  public static void main(String[] args) throws DatabaseAlreadyExistException, DatabaseNotEmptyException, DatabaseNotExistException {  //设置当前hadoop操作的用户名  System.setProperty("HADOOP\_USER\_NAME", "root");   StreamExecutionEnvironment streamEnv = StreamExecutionEnvironment.getExecutionEnvironment();  streamEnv.setParallelism(5);   EnvironmentSettings tableEnvSettings = EnvironmentSettings.newInstance()  .useBlinkPlanner()  .inStreamingMode()  .build();   StreamTableEnvironment tableEnv = StreamTableEnvironment.create(streamEnv, tableEnvSettings);   String catalogName = "myhive";  String databaseName = "itcast";  String tableName = "test";  HiveCatalog catalog = new HiveCatalog(  catalogName, // catalog name  "default", // default database  "src/main/resources", // Hive config (hive-site.xml) directory  "2.1.1" // Hive version  );  //注册目录  tableEnv.registerCatalog(catalogName, catalog);  tableEnv.useCatalog(catalogName);   System.out.println("---------------创建数据库------------------------");  catalog.createDatabase(databaseName,  new CatalogDatabaseImpl(new HashMap<>(), "my comment"), true);   System.out.println("---------------删除数据库------------------------");  catalog.dropDatabase(databaseName, true);  // System.out.println("---------------修改数据库------------------------"); // catalog.alterDatabase(databaseName, new CatalogDatabaseImpl(...), false);   System.out.println("---------------删除数据库------------------------");  CatalogDatabase database = catalog.getDatabase(databaseName);   System.out.println("---------------验证数据库是否存在------------------------");  boolean result = catalog.databaseExists(databaseName);   System.out.println("---------------在目录中列出数据库------------------------");  catalog.listDatabases();  } } |

#### 表操作

|  |
| --- |
| // 创建表 catalog.createTable(new ObjectPath("mydb", "mytable"), new CatalogTableImpl(...), false);  // 删除表 catalog.dropTable(new ObjectPath("mydb", "mytable"), false);  // 修改表 catalog.alterTable(new ObjectPath("mydb", "mytable"), new CatalogTableImpl(...), false);  // 重命名表 catalog.renameTable(new ObjectPath("mydb", "mytable"), "my\_new\_table");  // 获得表 catalog.getTable("mytable");  // 判断表是否存在 catalog.tableExists("mytable");  // 返回数据库所有表的列表 catalog.listTables("mydb"); |

#### 视图操作

|  |
| --- |
| // 创建视图 catalog.createTable(new ObjectPath("mydb", "myview"), new CatalogViewImpl(...), false);  // 删除视图 catalog.dropTable(new ObjectPath("mydb", "myview"), false);  // 修改视图 catalog.alterTable(new ObjectPath("mydb", "mytable"), new CatalogViewImpl(...), false);  // 重命名视图 catalog.renameTable(new ObjectPath("mydb", "myview"), "my\_new\_view", false);  // 获得视图 catalog.getTable("myview");  // 检查视图是否存在 catalog.tableExists("mytable");  // 获得数据库所有视图 catalog.listViews("mydb"); |

#### 分区操作

|  |
| --- |
| // 创建分区 catalog.createPartition(  new ObjectPath("mydb", "mytable"),  new CatalogPartitionSpec(...),  new CatalogPartitionImpl(...),  false);  // 删除分区 catalog.dropPartition(new ObjectPath("mydb", "mytable"), new CatalogPartitionSpec(...), false);  // 修改分区 catalog.alterPartition(  new ObjectPath("mydb", "mytable"),  new CatalogPartitionSpec(...),  new CatalogPartitionImpl(...),  false);  // 获得分区 catalog.getPartition(new ObjectPath("mydb", "mytable"), new CatalogPartitionSpec(...));  // 检查分区是否存在 catalog.partitionExists(new ObjectPath("mydb", "mytable"), new CatalogPartitionSpec(...));  // 返回表所有的分区 catalog.listPartitions(new ObjectPath("mydb", "mytable"));  // 列出给定分区规范下表的分区 catalog.listPartitions(new ObjectPath("mydb", "mytable"), new CatalogPartitionSpec(...));  // 按表达式筛选器列出表的分区 catalog.listPartitions(new ObjectPath("mydb", "mytable"), Arrays.asList(epr1, ...)); |

#### 函数操作

|  |
| --- |
| // 创建函数 catalog.createFunction(new ObjectPath("mydb", "myfunc"), new CatalogFunctionImpl(...), false);  // 删除函数 catalog.dropFunction(new ObjectPath("mydb", "myfunc"), false);  // 修改函数 catalog.alterFunction(new ObjectPath("mydb", "myfunc"), new CatalogFunctionImpl(...), false);  // 获得函数 catalog.getFunction("myfunc");  // 检查函数是否存在 catalog.functionExists("myfunc");  // 列出数据库中的函数 catalog.listFunctions("mydb"); |

### [通过 Table API 和 SQL Client 操作 Catalog](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/catalogs/" \l "%E9%80%9A%E8%BF%87-table-api-%E5%92%8C-sql-client-%E6%93%8D%E4%BD%9C-catalog)

#### 注册 Catalog

用户可以访问默认创建的内存 **Catalog default\_catalog**，这个 Catalog 默认拥有一个默认数据库 **default\_database**。 用户也可以注册其他的 Catalog 到现有的 Flink 会话中。

|  |
| --- |
| tableEnv.registerCatalog(new CustomCatalog("myCatalog")); |

#### 修改当前的 Catalog 和数据库

Flink 始终在当前的 Catalog 和数据库中寻找表、视图和 UDF。

|  |
| --- |
| tableEnv.useCatalog("myCatalog"); tableEnv.useDatabase("myDb"); |

通过提供全限定名 catalog.database.object 来访问不在当前 Catalog 中的元数据信息。

|  |
| --- |
| tableEnv.from("not\_the\_current\_catalog.not\_the\_current\_db.my\_table"); |

#### 列出可用的 Catalog

|  |
| --- |
| tableEnv.listCatalogs(); |

#### 列出可用的数据库

|  |
| --- |
| tableEnv.listDatabases(); |

#### 列出可用的表

|  |
| --- |
| tableEnv.listTables(); |

### [Flink SQL Clien读取Kafka数据流式写入Hive管理元数据](https://www.cnblogs.com/qiu-hua/p/13975404.html)

[使用Flink SQL Clien读取Kafka数据流式写入Hive（用hive 管理kafka元数据）](https://www.cnblogs.com/qiu-hua/p/13975404.html)

**版本说明：**

Kafka 0.10.0.0

Hive 2.1.1

Hadoop 2.7.5

#### Hive、kafka与Flink SQL Client整合配置

|  |
| --- |
| /flink-1.13.0  /lib  // Flink's Hive connector  flink-sql-connector-hive-2.2.0\_2.11-1.13.0.jar  flink-sql-connector-kafka\_2.11-1.13.0.jar  flink-shaded-hadoop-2-uber-2.7.5-10.0.jar  // Hive dependencies  hive-exec-2.1.1.jar  // add antlr-runtime if you need to use hive dialect  antlr-runtime-3.5.2.jar |

后三个JAR包都是Hive自带的，可以在${HIVE\_HOME}/lib目录下找到。前一个可以通过[flink官网下载](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/overview/)或者[阿里云Maven](https://maven.aliyun.com/mvn/search)搜索GAV找到并手动下载。

**注意：要将lib包分发到集群中其他flink机器上**

#### 启动相关服务并测试

* 启动zk（每台节点都需要启动）

|  |
| --- |
| zkServer.sh start |

* 启动hadoop集群

|  |
| --- |
| start-dfs.sh |

* 启动Hive

|  |
| --- |
| nohup hive --service metastore & >> nohup.out nohup hive --service hiveserver2 & >> hiveserver2.out |

* 启动kafka

|  |
| --- |
| cd /export/servers/kafka\_2.11-0.10.0.0 nohup bin/kafka-server-start.sh config/server.properties 2>&1 & |

* 创建topic

|  |
| --- |
| /export/servers/kafka\_2.11-0.10.0.0/bin/kafka-topics.sh --create --zookeeper node01:2181 --topic order --partitions 3 --replication-factor 2 |

* 启动 Flink SQL Client

|  |
| --- |
| bin/sql-client.sh embedded |

* 在Flink SQL Client中创建Hive表，指定数据源为Kafka

|  |
| --- |
| SET sql-client.verbose = true;  CREATE CATALOG myhive WITH (  'type' = 'hive',  'hive-conf-dir' = '/opt/hive-conf' );  -- set the HiveCatalog as the current catalog of the session USE CATALOG myhive;  CREATE DATABASE IF NOT EXISTS itcast; USE itcast;  CREATE TABLE IF NOT EXISTS `order`(  id INT,  category STRING,  areaName STRING,  money INT,  `timestamp` BIGINT,  eventTime AS TO\_TIMESTAMP(FROM\_UNIXTIME(`timestamp` / 1000, 'yyyy-MM-dd HH:mm:ss')), -- 事件时间  WATERMARK FOR eventTime AS eventTime - INTERVAL '10' SECOND -- 水印 ) WITH (  'connector' = 'kafka',  'topic' = 'order', -- 指定消费的topic  'scan.startup.mode' = 'latest-offset', -- 指定起始offset位置  'properties.zookeeper.connect' = 'node01:2181',  'properties.bootstrap.servers' = 'node01:9092',  'properties.group.id' = 'order\_01',  'format' = 'json',  'json.ignore-parse-errors' = 'true' ); |

* 启动Kafka，发送数据

|  |
| --- |
| bin/kafka-console-producer.sh --broker-list node01:9092 --topic order |
| {"id":1,"timestamp":1588870980000,"category":"电脑","areaName":"石家庄","money":"1450"} {"id":2,"timestamp":1588870860000,"category":"手机","areaName":"北京","money":"1450"} {"id":3,"timestamp":1588870980000,"category":"手机","areaName":"北京","money":"8412"} {"id":4,"timestamp":1588885260000,"category":"电脑","areaName":"上海","money":"1513"} {"id":5,"timestamp":1588870980000,"category":"家电","areaName":"北京","money":"1550"} {"id":6,"timestamp":1588870860000,"category":"电脑","areaName":"深圳","money":"1550"} |

* 通过Flink SQL Client查询表中的数据

|  |
| --- |
| select *\** from `order`; |
|  |
|  |

## 流处理中的概念

Flink 的 [Table API](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/tableapi/) 和 [SQL](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/overview/) 是流批统一的 API。 这意味着 Table API & SQL 在无论有限的批式输入还是无限的流式输入下，都具有相同的语义。 因为传统的关系代数以及 SQL 最开始都是为了批式处理而设计的， 关系型查询在流式场景下不如在批式场景下容易懂。

下面这些包含了概念、实际的限制，以及流式数据处理中的一些特定的配置。

* [动态表](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/dynamic_tables/): 描述了动态表的概念。
* [时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/): 解释了时间属性以及它是如何在 Table API & SQL 中使用的。
* [流上的 Join](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/): 支持的几种流上的 Join。
* [时态（temporal）表](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/versioned_tables/): 描述了时态表的概念。
* [查询配置](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/): Table API & SQL 特定的配置。

### 动态表（Dynamic Tables）

因为流处理面对的数据，是连续不断的，这和我们熟悉的关系型数据库中保存的“**表**”完全不同。所以，如果我们把流数据转换成Table，然后执行类似于table的select操作，结果就不是一成不变的，而是随着新数据的到来，会不停更新。

我们可以随着新数据的到来，不停地在之前的基础上更新结果。这样得到的表，在Flink Table API概念里，就叫做**“动态表”（Dynamic Tables）**。

动态表是Flink对流数据的Table API和SQL支持的核心概念。与表示批处理数据的静态表不同，动态表是随时间变化的。动态表可以像静态的批处理表一样进行查询，查询一个动态表会产生持续查询（Continuous Query）。连续查询永远不会终止，并会生成另一个动态表。查询（Query）会不断更新其动态结果表，以反映其动态输入表上的更改。

#### 流处理和关系代数（表，及SQL）的区别

下表比较了传统的关系代数和流处理与输入数据、执行和输出结果的关系。

|  |  |  |
| --- | --- | --- |
| **比较维度** | **关系代数/SQL** | **流处理** |
| 数据对象 | 关系(或表)是有界(多)元组集合 | 流是一个无限元组序列 |
| 数据访问 | 对批数据(例如关系数据库中的表)执行的查询可以访问完整的输入数据 | 流式查询在启动时不能访问所有数据，必须“**等待**”数据流入 |
| 查询终止条件 | 批处理查询在产生固定大小的结果后终止 | 流查询不断地根据接收到的记录更新其结果，并且始终不会结束 |

可以看到，其实关系代数（主要就是指关系型数据库中的表）和SQL，主要就是针对批处理的，这和流处理有天生的隔阂。

尽管存在这些差异，但是**使用关系查询和 SQL 处理流并不是不可能的**。高级关系数据库系统提供了一个称为 **物化视图(Materialized Views)**的特性。物化视图被定义为一条 SQL 查询，就像常规的虚拟视图一样。与虚拟视图相反，物化视图缓存查询的结果，因此在访问视图时不需要对查询进行计算。缓存的一个常见难题是防止缓存为过期的结果提供服务。当其定义查询的基表被修改时，物化视图将过期。 **即时视图维护**(Eager View Maintenance) 是一种一旦更新了物化视图的基表就立即更新视图的技术。

如果我们考虑以下问题，那么即时视图维护和流上的SQL查询之间的联系就会变得显而易见:

* 数据库表是 INSERT、UPDATE 和 DELETE DML 语句的 stream 的结果，通常称为 changelog stream
* 物化视图被定义为一条 SQL 查询。为了更新视图，查询不断地处理视图的基本关系的changelog 流
* 物化视图是流式 SQL 查询的结果

了解了这些要点之后，我们将在下一节中介绍 **动态表(Dynamic tables)**的概念。

#### 动态表 & 连续查询(Continuous Query)

动态表 是 Flink 的支持流数据的 Table API 和 SQL 的核心概念。与表示批处理数据的静态表不同，动态表是随时间变化的。可以像查询静态批处理表一样查询它们。查询动态表将生成一个 连续查询 。一个连续查询永远不会终止，结果会生成一个动态表。查询不断更新其(动态)结果表，以反映其(动态)输入表上的更改。本质上，动态表上的连续查询非常类似于定义物化视图的查询。

**需要注意的是**：连续查询的结果在语义上总是等价于以批处理模式在输入表快照上执行的相同查询的结果。

下图显示了流、动态表和连续查询之间的关系：



* 将流转换为动态表
* 在动态表上计算一个连续查询，生成一个新的动态表
* 生成的动态表被转换回流

**注意：** 动态表首先是一个逻辑概念。在查询执行期间不一定(完全)物化动态表。

##### 将流转换成表（Table）

为了处理带有关系查询的流，必须先将其转换为表。

从概念上讲，流的每个数据记录，都被解释为对结果表的插入（Insert）修改。因为流式持续不断的，而且之前的输出结果无法改变。本质上，我们其实是从一个、只有插入操作的changelog（更新日志）流，来构建一个表。

为了更好地说明动态表和持续查询的概念，我们来举一个具体的例子。

比如，我们现在的输入数据，就是用户在网站上的访问行为，数据类型（Schema）如下：

|  |
| --- |
| [  user: VARCHAR, // 用户名  cTime: TIMESTAMP, // 访问某个URL的时间戳  url: VARCHAR // 用户访问的URL  ] |

下图显示了如何将访问URL事件流，或者叫点击事件流（左侧）转换为表（右侧），当插入更多的单击流记录时，结果表将不断增长。



**注意**： 在流上定义的表在内部没有物化。

##### 连续查询（Continuous Query）

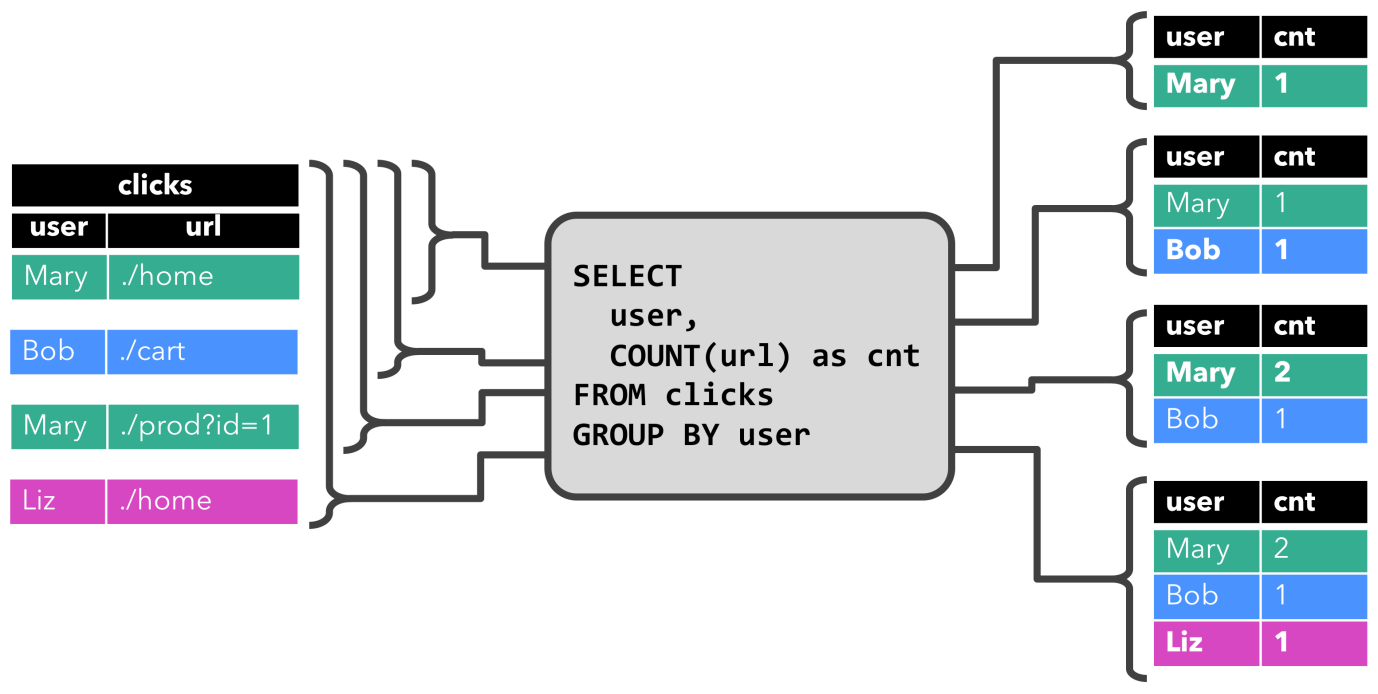
持续查询，会在动态表上做计算处理，并作为结果生成新的动态表。与批处理查询不同，连续查询从不终止，并根据输入表上的更新更新其结果表。

在任何时间点，连续查询的结果在语义上，等同于在输入表的快照上，以批处理模式执行的同一查询的结果。

在下面的示例中，我们展示了对点击事件流中的一个持续查询。

**先看第一个查询：**

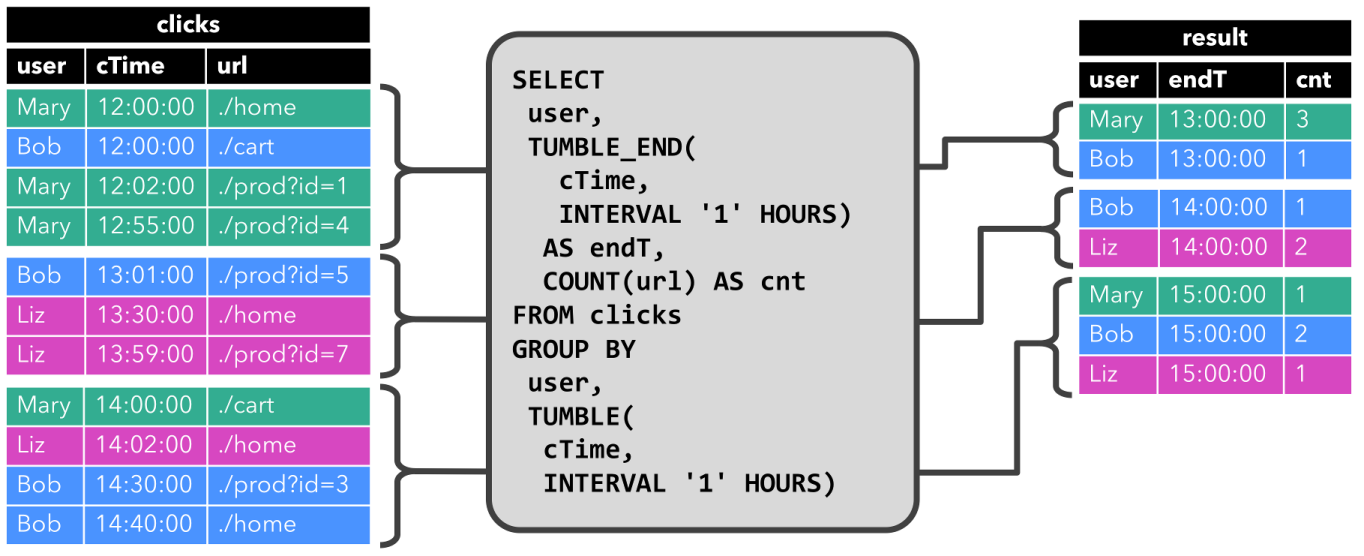
这个Query很简单，是一个分组聚合做count统计的查询。它将用户字段上的clicks表分组，并统计访问的url数。图中显示了随着时间的推移，当clicks表被其他行更新时如何计算查询。



当查询开始，clicks 表(左侧)是空的。当第一行数据被插入到 clicks 表时，查询开始计算结果表。第一行数据 [Mary,./home] 插入后，结果表(右侧，上部)由一行 [Mary, 1] 组成。当第二行 [Bob, ./cart] 插入到 clicks 表时，查询会更新结果表并插入了一行新数据 [Bob, 1]。第三行 [Mary, ./prod?id=1] 将产生已计算的结果行的更新，[Mary, 1] 更新成 [Mary, 2]。最后，当第四行数据加入 clicks 表时，查询将第三行 [Liz, 1] 插入到结果表中。

**先看第二个查询：**

第二条查询与第一条类似，但是除了用户属性之外，还将 clicks 分组至[每小时滚动窗口](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/sql/" \l "group-windows)中，然后计算 url 数量(基于时间的计算，例如基于特定[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/streaming/time_attributes.html)的窗口，后面会讨论)。同样，该图显示了不同时间点的输入和输出，以可视化动态表的变化特性。



与前面一样，左边显示了输入表 clicks。查询每小时持续计算结果并更新结果表。clicks表包含四行带有时间戳(cTime)的数据，时间戳在 12:00:00 和 12:59:59 之间。查询从这个输入计算出两个结果行(每个 user 一个)，并将它们附加到结果表中。对于 13:00:00 和 13:59:59 之间的下一个窗口，clicks 表包含三行，这将导致另外两行被追加到结果表。随着时间的推移，更多的行被添加到 click 中，结果表将被更新。

##### 将动态表转换成流

与常规的数据库表一样，动态表可以通过插入（Insert）、更新（Update）和删除（Delete）更改，进行持续的修改。将动态表转换为流或将其写入外部系统时，需要对这些更改进行编码。Flink的Table API和SQL支持三种方式对动态表的更改进行编码：

###### 仅追加（Append-only）流

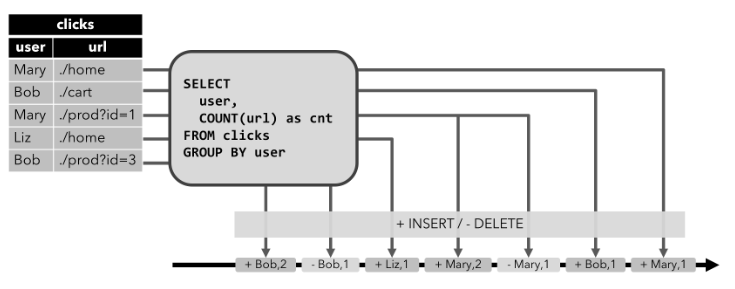
仅通过插入（Insert）更改，来修改的动态表，可以直接转换为“仅追加”流。这个流中发出的数据，就是动态表中新增的每一行。

###### 撤回（Retract）流

Retract流是包含两类消息的流，添加（Add）消息和撤回（Retract）消息。

动态表通过将INSERT 编码为add消息、DELETE 编码为retract消息、UPDATE编码为被更改行（前一行）的retract消息和更新后行（新行）的add消息，转换为retract流。

下图显示了将动态表转换为Retract流的过程。

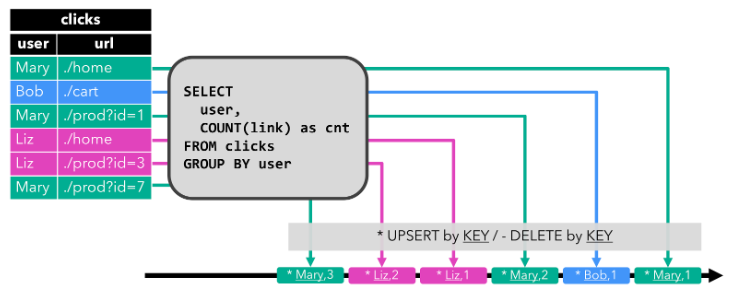


###### Upsert（更新插入）流

Upsert流包含两种类型的消息：Upsert消息和delete消息。转换为upsert流的动态表，需要有唯一的键（key）。

通过将INSERT和UPDATE更改编码为upsert消息，将DELETE更改编码为DELETE消息，就可以将具有唯一键（Unique Key）的动态表转换为流。

下图显示了将动态表转换为upsert流的过程。



这些概念我们之前都已提到过。需要注意的是，在代码里将动态表转换为DataStream时，仅支持**Append**和**Retract**流。而向外部系统输出动态表的TableSink接口，则可以有不同的实现，比如之前我们讲到的ES，就可以有Upsert模式。

### 时间特性

时间属性（Time Attribute）用一个**TIMESTAMP(int precision)**数据类型来表示，这个类型与SQL标准中的时间戳类型相对应，是Table API& SQL中专门用来表明时间属性的数据类型。**precision**为精度，表示秒以下保留几位小数点，可以是0到9的一个数字。具体而言，时间的格式为：

|  |
| --- |
| year-month-day hour:minute:second[.fractional] |

假如我们想要使用一个纳秒精度的时间，应该声明类型为**TIMESTAMP(9)**，套用上面的时间格式的话，可以表征从0000-01-01 00:00:00.000000000到9999-12-31 23:59:59.999999999。**绝大多数情况下，我们使用毫秒精度即可，即TIMESTAMP(3)**

当涉及到时间窗口，往往就要涉及到窗口的长度单位，现有的时间单位有**MILLISECOND、SECOND、MINUTE、HOUR、DAY、MONTH和YEAR**。

Flink 可以基于几种不同的 时间 概念来处理数据。

* 处理时间 指的是执行具体操作时的机器时间（大家熟知的绝对时间, 例如 Java的 System.currentTimeMillis()) ）
* 事件时间 指的是数据本身携带的时间。这个时间是在事件产生时的时间。
* 摄入时间 指的是数据进入 Flink 的时间；在系统内部，会把它当做事件时间来处理。

#### 处理时间（Processing Time）

处理时间语义下，允许表处理程序根据机器的本地时间生成结果。它是时间的最简单概念。**它既不需要提取时间戳，也不需要生成watermark**。

定义处理时间属性有三种方法：在DataStream转化时直接指定；**~~在定义Table Schema时指定；~~**在创建表的DDL中指定。

##### 创建表的DDL中指定

Processing Time使用当机器的系统时间作为处理时间，在Table API & SQL中这个字段被称为**proctime**。它不需要配置Watermark。使用时，我们在原本的Schema上添加一个虚拟的时间戳列，时间戳列由**PROCTIME()**函数计算产生。

|  |
| --- |
| public class FlinkSqlProsessingTimeTableDDL {  public static void main(String[] args) throws Exception {  //构建运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1); // 设置并行度为1方便后面进行测试  //构建表运行环境  EnvironmentSettings settings = EnvironmentSettings.newInstance()  .useBlinkPlanner()  .inStreamingMode().build();   String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, settings);  String sqlDDL = "create table InputTable (" +  " id varchar," +  " `timestamp` bigint," +  " category varchar," +  " areaName varchar," +  " money double," +  " pt AS PROCTIME()" +  " ) with (" +  " 'connector' = 'filesystem'," +  " 'path' = 'file:///"+filePath+"'," +  " 'format' = 'csv'" +  " )";   tableEnv.executeSql(sqlDDL);  Table resultTable = tableEnv.sqlQuery("select \* from InputTable ");  resultTable.printSchema();  // 打印输出  tableEnv.toAppendStream(resultTable, Row.class).print("result");  env.execute();  } } |

**注意：运行这段 DDL，必须使用 Blink Planner**

##### DataStream转化成Table时指定

由DataStream转换成表时，可以在后面指定字段名来定义Schema。在定义Schema期间，可以使用**.proctime**，定义处理时间字段。

**注意：**时间属性一定不能定义在一个已有字段上，所以它只能定义在 schema 定义的最后。

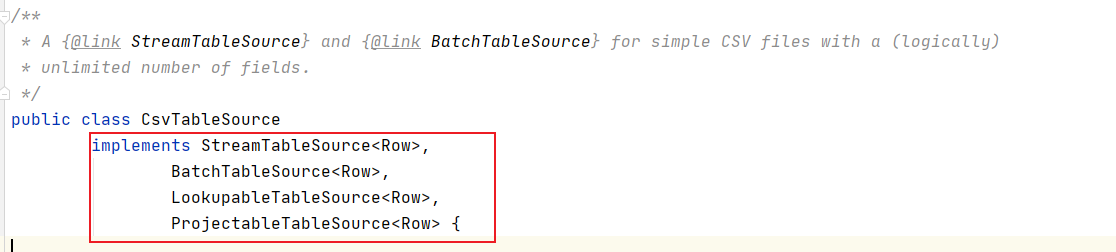
|  |
| --- |
| public class FlinkSqlProcessingTimeDataStream{  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();   env.getConfig().setRestartStrategy(RestartStrategies.noRestart());  env.setParallelism(1);   // 从文件读取数据  String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();  DataStreamSource<String> inputStream = env.readTextFile(filePath);   // map成样例类类型  SingleOutputStreamOperator<OrderInfo> dataStream = inputStream.map(new MapFunction<String, OrderInfo>() {  @Override  public OrderInfo map(String data) throws Exception {  String[] dataArray = data.split(",");  return new OrderInfo(dataArray[0], dataArray[1], dataArray[2], dataArray[3], Double.parseDouble(dataArray[4]));  }  });   // 1. 基于env创建表环境  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, bsSettings);   // 2. 基于tableEnv，将流转换成表  Table dataTable = tableEnv.fromDataStream(dataStream, $("id"), $("timestamp"),  $("category"), $("areaName"),$("money"),$("pt").proctime());   // 3. 把表转换成流，打印输出  tableEnv.toAppendStream(dataTable, Row.class).print("result");  try {  env.execute();  } catch (Exception exception) {  exception.printStackTrace();  }  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class OrderInfo {  private String id;  private String timestamp;  private String category;  private String areaName;  private Double money;  } } |

##### ~~定义Table Schema时指定（已废弃）~~

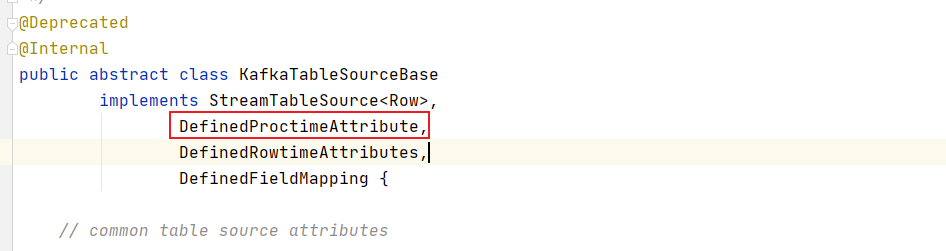
这种方法其实也很简单，只要在定义Schema的时候，加上一个新的字段，并指定成proctime就可以了。

|  |
| --- |
| public class FlinkSqlProcessingTimeTable {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();   env.getConfig().setRestartStrategy(RestartStrategies.noRestart());  env.setParallelism(1);   // 1. 创建表执行环境  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, bsSettings);   // 2. 连接外部系统，读取数据  // 2.1 读取文件数据  String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();   tableEnv.connect(new FileSystem().path(filePath))  .withFormat(new Csv()) // 定义从外部文件读取数据之后的格式化方法  .withSchema(new Schema()  .field("id", DataTypes.STRING())  .field("timestamp", DataTypes.STRING())  .field("category", DataTypes.STRING())  .field("areaName", DataTypes.STRING())  .field("money", DataTypes.DOUBLE())  //.field("pt", DataTypes.TIMESTAMP())  //.proctime() // 指定 pt字段为处理时间  ) // 定义表的结构  .createTemporaryTable("inputTable"); // 在表环境注册一张表    // 3. 表的查询转换  Table orderTable = tableEnv.from("inputTable");  // 3. 把表转换成流，打印输出  tableEnv.toAppendStream(orderTable, Row.class).print("result");  try {  env.execute();  } catch (Exception exception) {  exception.printStackTrace();  }  } } |

**注意：**使用Schema方式首先要看看它是否支持 csv 格式是不是支持的因为他底层没事进行实现。



使用Kafka的时候可以使用这种方式它底层实现了这个方法



#### 事件时间（Event Time）

Event Time时间语义**使用一条数据实际发生的时间作为时间属性**，在Table API & SQL中这个字段通常被称为**rowtime**。这种模式下多次重复计算时，计算结果是确定的。这意味着，Event Time时间语义可以保证流处理和批处理的统一

Event Time时间语义下，我们需要设置每条数据发生时的时间戳，并提供一个Watermark。Watermark表示迟于该时间的数据都作为迟到数据对待

##### 为创建表的 DDL 中指定

事件时间属性，是使用 CREATE TABLE DDL 中的 WARDMARK 语句定义的。watermark 语句，定义现有事件时间字段上的 watermark 生成表达式，该表达式将事件时间字段标记为事件时间属性。

|  |
| --- |
| public class FlinkSqlEventTimeTableDDL {  public static void main(String[] args) throws Exception {  //构建运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1); // 设置并行度为1方便后面进行测试  //构建表运行环境  EnvironmentSettings settings = EnvironmentSettings.newInstance()  .useBlinkPlanner()  .inStreamingMode().build();   String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, settings);  // 这里 FROM\_UNIXTIME 是系统内置的时间函数，用来将一个整数（秒数）转换成“YYYY-MM-DD hh:mm:ss”格式（默认，  // 也可以作为第二个 String 参数传入）的日期时间字符串（date time string）；然后再用 TO\_TIMESTAMP 将其转换成 Timestamp。  String sqlDDL = "create table InputTable (" +  " id varchar," +  " `timestamp` bigint," +  " category varchar," +  " areaName varchar," +  " money double," +  " rt AS TO\_TIMESTAMP(FROM\_UNIXTIME(`timestamp`))," +  " watermark for rt as rt - interval '1' second"+  " ) with (" +  " 'connector' = 'filesystem'," +  " 'path' = 'file:///"+filePath+"'," +  " 'format' = 'csv'" +  " )";   tableEnv.executeSql(sqlDDL);  Table resultTable = tableEnv.sqlQuery("select \* from InputTable");  resultTable.printSchema();  // 打印输出  tableEnv.toAppendStream(resultTable, Row.class).print("result");  env.execute();  } } |

在上面的DDL中，**WATERMARK**起到了定义Event Time时间属性的作用，它的基本语法规则为：

|  |
| --- |
| WATERMARK FOR rowtime\_column AS watermark\_strategy\_expression。 |

**rowtime\_column**为时间戳字段，可以是数据中的自带字段，也可以是类似**PROCTIME()**函数计算出的虚拟时间戳字段，这个字段必须是**TIMESTAMP(3)**类型。

**watermark\_strategy\_expression**定义了Watermark的生成策略，返回值必须是TIMESTAMP(3)类型。Flink提供了几种常用的策略：

* 数据自身的时间戳严格按照单调递增的形式出现，即晚到达的时间戳总比早到达的时间戳大，可以使用**WATERMARK FOR rowtime\_column AS rowtime\_column**或**WATERMARK FOR rowtime\_column AS rowtime\_column - INTERVAL '0.001' SECOND**生成Watermark。这个策略的原理是：监测所有数据时间戳，并记录时间戳最大值，在最大值基础上添加一个1毫秒的延迟作为Watermark时间。
* 数据本身是乱序到达的，Watermark在时间戳最大值的基础上延迟一定时间间隔，如果数据仍比这个时间还晚，则被定为迟到数据。我们可以使用**WATERMARK FOR rowtime\_column AS rowtime\_column - INTERVAL 'duration' timeUnit**生成Watermark。例如，WATERMARK FOR ts as ts - INTERVAL '5' SECOND定义的Watermark比时间戳最大值还延迟了5秒。这里timeUnit可以是SECOND、MINUTE或HOUR等时间单位。

**注意：**这里FROM\_UNIXTIME是系统内置的时间函数，用来将一个整数（秒数）转换成“YYYY-MM-DD hh:mm:ss”格式（默认，也可以作为第二个String参数传入）的日期时间字符串（date time string）；然后再用TO\_TIMESTAMP将其转换成Timestamp。

##### DataStream转化成Table 时指定

在DataStream转换成Table，schema的定义期间，使用.rowtime可以定义事件时间属性。**注意，必须在转换的数据流中分配时间戳和watermark**。

在将数据流转换为表时，有两种定义时间属性的方法。根据指定的.rowtime字段名是否存在于数据流的架构中，timestamp字段可以：

* 作为新字段追加到schema
* 替换现有字段

在这两种情况下，定义的事件时间戳字段，都将保存DataStream中事件时间戳的值。

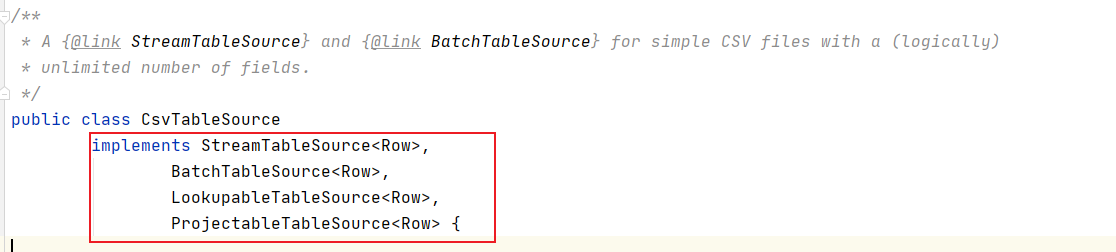
|  |
| --- |
| public class FlinkSqlEventTimeDataStream {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();  //在 Flink 1.12 中，默认的流时间特征已更改为 EventTime，因此您不再需要调用此方法来启用事件时间支持。  // 明确使用处理时间窗口和计时器可以在事件时间模式下使用。如果需要禁用水印，请使用 ExecutionConfig.setAutoWatermarkInterval（long）。  // 如果您使用的是 IngestionTime，请手动设置适当的WatermarkStrategy。  // 如果您使用基于时间特征更改行为的通用“时间窗口”操作（例如KeyedStream.timeWindow（）），请使用等效操作明确指定处理时间或事件时间。  //env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);  env.setParallelism(1);   // 从文件读取数据  String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();  DataStreamSource<String> inputStream = env.readTextFile(filePath);   // map成样例类类型  SingleOutputStreamOperator<OrderInfo> dataStream = inputStream.map(new MapFunction<String, OrderInfo>() {  @Override  public OrderInfo map(String data) throws Exception {  String[] dataArray = data.split(",");  return new OrderInfo(dataArray[0], Long.parseLong(dataArray[1]), dataArray[2], dataArray[3], Double.parseDouble(dataArray[4]));  }  }).assignTimestampsAndWatermarks(  WatermarkStrategy.<OrderInfo>forBoundedOutOfOrderness(  Duration.ofSeconds(0))  .withTimestampAssigner(new MyTimeAssigner()));   // 1. 基于env创建表环境  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, bsSettings);   //方式一：使用现有的字段作为事件时间属性  Table dataTable = tableEnv.fromDataStream(dataStream, $("id"),  $("category"), $("areaName"),$("money"),$("timestamp").rowtime());   //方式二：声明一个额外的逻辑字段作为事件时间属性  Table dataTable2 = tableEnv.fromDataStream(dataStream, $("id"),  $("category"), $("areaName"),$("money"),$("rt").rowtime());   // 3. 把表转换成流，打印输出  tableEnv.toAppendStream(dataTable, Row.class).print("方式一");  tableEnv.toAppendStream(dataTable2, Row.class).print("方式二");  try {  env.execute();  } catch (Exception exception) {  exception.printStackTrace();  }  }  public static class MyTimeAssigner implements SerializableTimestampAssigner<OrderInfo> {  /\*\*  \* 提取数据里的timestamp字段为时间戳  \* @param element event  \* @param recordTimestamp element 的当前内部时间戳，或者如果没有分配时间戳，则是一个负数  \* @return The new timestamp.  \*/  @Override  public long extractTimestamp(OrderInfo element, long recordTimestamp) {  return element.getTimestamp() \* 1000L;  }  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class OrderInfo {  private String id;  private Long timestamp;  private String category;  private String areaName;  private Double money;  } } |

##### ~~定义Table Schema时指定~~

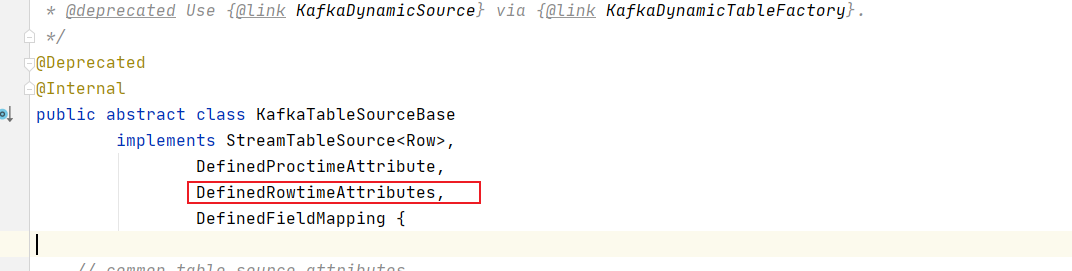
这种方法只要在定义Schema的时候，将事件时间字段，并指定成rowtime就可以了。

|  |
| --- |
| public class FlinkSqlEventTimeTable {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();   env.getConfig().setRestartStrategy(RestartStrategies.noRestart());  env.setParallelism(1);   // 1. 创建表执行环境  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, bsSettings);   // 2. 连接外部系统，读取数据  // 2.1 读取文件数据  String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();   tableEnv.connect(new FileSystem().path(filePath))  .withFormat(new Csv()) // 定义从外部文件读取数据之后的格式化方法  .withSchema(new Schema()  .field("id", DataTypes.STRING())  .field("timestamp", DataTypes.BIGINT())  .field("category", DataTypes.STRING())  .field("areaName", DataTypes.STRING())  .field("money", DataTypes.DOUBLE()) // .rowtime( // new Rowtime() // .timestampsFromField("timestamp") // 从字段使用 // .watermarksPeriodicBounded(1000) // watermark 延迟 1 秒 // )  ) // 定义表的结构  .createTemporaryTable("inputTable"); // 在表环境注册一张表    // 3. 表的查询转换  Table orderTable = tableEnv.from("inputTable");  // 3. 把表转换成流，打印输出  tableEnv.toAppendStream(orderTable, Row.class).print("result");  try {  env.execute();  } catch (Exception exception) {  exception.printStackTrace();  }  } } |

**注意：**使用Schema方式首先要看看它是否支持 csv 格式是不是支持的因为他底层没事进行实现。



使用Kafka的时候可以使用这种方式它底层实现了这个方法



### 时区

Flink 为日期和时间提供了丰富的数据类型， 包括 DATE， TIME， TIMESTAMP， TIMESTAMP\_LTZ， INTERVAL YEAR TO MONTH， INTERVAL DAY TO SECOND ([更多详情请参考 Date and Time](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/types/" \l "date-and-time))。

Flink 支持在 session （会话）级别设置时区（[更多详情请参考 table.local-time-zone](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "table-local-time-zone)）。

Flink 对多种时间类型和时区的支持使得跨时区的数据处理变得非常容易。

#### TIMESTAMP vs TIMESTAMP\_LTZ

##### TIMESTAMP 类型

* TIMESTAMP(p) 是 TIMESTAMP(p) WITHOUT TIME ZONE 的简写， 精度 p 支持的范围是0-9， 默认是6。
* TIMESTAMP 用于描述年， 月， 日， 小时， 分钟， 秒 和 小数秒对应的时间戳。
* TIMESTAMP 可以通过一个字符串来指定，例如：

|  |
| --- |
| SELECT TIMESTAMP '1970-01-01 00:00:04.001'; |
| +-------------------------+ | 1970-01-01 00:00:04.001 | +-------------------------+ |

##### TIMESTAMP\_LTZ 类型

* TIMESTAMP\_LTZ(p) 是 TIMESTAMP(p) WITH LOCAL TIME ZONE 的简写， **精度 p 支持的范围是0-9， 默认是6**。
* TIMESTAMP\_LTZ 用于描述时间线上的绝对时间点， 使用 long 保存从 epoch 至今的毫秒数， 使用int保存毫秒中的纳秒数。 epoch 时间是从 java 的标准 epoch 时间 1970-01-01T00:00:00Z 开始计算。 在计算和可视化时， 每个 TIMESTAMP\_LTZ 类型的数据都是使用的 session （会话）中配置的时区。
* TIMESTAMP\_LTZ 没有字符串表达形式因此无法通过字符串来指定， 可以通过一个 long 类型的 epoch 时间来转化(例如: 通过 Java 来产生一个 long 类型的 epoch 时间 System.currentTimeMillis())

|  |
| --- |
| Flink SQL> CREATE VIEW T1 AS SELECT TO\_TIMESTAMP\_LTZ(4001, 3); Flink SQL> SET table.local-time-zone=UTC; Flink SQL> SELECT \* FROM T1; +---------------------------+ | TO\_TIMESTAMP\_LTZ(4001, 3) | +---------------------------+ | 1970-01-01 00:00:04.001 | +---------------------------+  Flink SQL> SET table.local-time-zone=Asia/Shanghai; Flink SQL> SELECT \* FROM T1; +---------------------------+ | TO\_TIMESTAMP\_LTZ(4001, 3) | +---------------------------+ | 1970-01-01 08:00:04.001 | +---------------------------+ |

* TIMESTAMP\_LTZ 可以用于跨时区的计算，因为它是一个基于 epoch 的绝对时间点（比如上例中的 4001 毫秒）代表的就是不同时区的同一个绝对时间点。 补充一个背景知识**：在同一个时间点， 全世界所有的机器上执行 System.currentTimeMillis() 都会返回同样的值。** (比如上例中的 4001 milliseconds), 这就是绝对时间的定义。

#### 时区的作用

本地时区定义了当前 session（会话）所在的时区， 你可以在 Sql client 或者应用程序中配置。

|  |
| --- |
| EnvironmentSettings envSetting = EnvironmentSettings.newInstance().build(); TableEnvironment tEnv = TableEnvironment.create(envSetting);  // 设置为 UTC 时区 tEnv.getConfig().setLocalTimeZone(ZoneId.of("UTC"));  // 设置为上海时区 tEnv.getConfig().setLocalTimeZone(ZoneId.of("Asia/Shanghai"));  // 设置为 Los\_Angeles 时区 tEnv.getConfig().setLocalTimeZone(ZoneId.of("America/Los\_Angeles")); |

session（会话）的时区设置在 Flink SQL 中非常有用， 它的主要用法如下：

##### 确定时间函数的返回值

session （会话）中配置的时区会对以下函数生效。

* LOCALTIME
* LOCALTIMESTAMP
* CURRENT\_DATE
* CURRENT\_TIME
* CURRENT\_TIMESTAMP
* CURRENT\_ROW\_TIMESTAMP()
* NOW()
* PROCTIME()

|  |
| --- |
| Flink SQL> SET sql-client.execution.result-mode=tableau; Flink SQL> CREATE VIEW MyView1 AS SELECT LOCALTIME, LOCALTIMESTAMP, CURRENT\_DATE, CURRENT\_TIME, CURRENT\_TIMESTAMP, CURRENT\_ROW\_TIMESTAMP(), NOW(), PROCTIME(); Flink SQL> DESC MyView1; |
|  |
| Flink SQL> SET table.local-time-zone=UTC; Flink SQL> SELECT *\** FROM MyView1; |
|  |
| Flink SQL> SET table.local-time-zone=Asia/Shanghai; Flink SQL> SELECT *\** FROM MyView1; |
|  |

##### TIMESTAMP\_LTZ 字符串表示

当一个 TIMESTAMP\_LTZ 值转为 string 格式时， session 中配置的时区会生效。 例如打印这个值，将类型强制转化为 STRING 类型， 将类型强制转换为 TIMESTAMP ，将 TIMESTAMP 的值转化为 TIMESTAMP\_LTZ 类型：

|  |
| --- |
| Flink SQL> CREATE VIEW MyView2 AS SELECT TO\_TIMESTAMP\_LTZ(4001, 3) AS ltz, TIMESTAMP '1970-01-01 00:00:01.001' AS ntz; Flink SQL> DESC MyView2; |
|  |
| Flink SQL> SET table.local-time-zone=UTC; Flink SQL> SELECT *\** FROM MyView2; |
|  |
| Flink SQL> SET table.local-time-zone=Asia/Shanghai; Flink SQL> SELECT \* FROM MyView2; |
|  |
| Flink SQL> CREATE VIEW MyView3 AS SELECT ltz, CAST(ltz AS TIMESTAMP(3)), CAST(ltz AS STRING), ntz, CAST(ntz AS TIMESTAMP\_LTZ(3)) FROM MyView2;  Flink SQL> DESC MyView3; |
|  |
| Flink SQL> SELECT *\** FROM MyView3; |
|  |

#### 时间属性和时区

更多时间属性相关的详细介绍， 请参考 [Time Attribute](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/" \l "%e6%97%b6%e9%97%b4%e5%b1%9e%e6%80%a7) 。

##### 处理时间和时区

使用示例

Flink SQL 使用函数 PROCTIME() 来定义处理时间属性， 该函数返回的类型是 TIMESTAMP\_LTZ 。

|  |
| --- |
| **注意**：在 Flink1.13 之前， PROCTIME() 函数返回的类型是 TIMESTAMP ， 返回值是UTC时区下的 TIMESTAMP 。 例如：当上海的时间为 2021-03-01 12:00:00 时， PROCTIME() 显示的时间却是错误的 2021-03-01 04:00:00 。 这个问题在 Flink 1.13 中修复了， 因此用户不用再去处理时区的问题了。 |

PROCTIME() 返回的是本地时区的时间， 使用 TIMESTAMP\_LTZ 类型也可以支持夏令时时间。

|  |
| --- |
| Flink SQL> SET table.local-time-zone=UTC; Flink SQL> SELECT PROCTIME(); |
|  |
| Flink SQL> SET table.local-time-zone=Asia/Shanghai; Flink SQL> SELECT PROCTIME(); |
|  |

案例演示

* 需求：

创建表从**socket**获取数据，分别设置**UTC时区**和**Asia/Shanghai时区**查看时间字段的变化

* 实现步骤：
  + 创建表：

|  |
| --- |
| Flink SQL> CREATE TABLE MyTable1 (  item STRING,  price DOUBLE,  proctime as PROCTIME()  ) WITH (  'connector' = 'socket',  'hostname' = '127.0.0.1',  'port' = '9999',  'format' = 'csv'  ); |

* + 创建基于滚动窗口的视图：

|  |
| --- |
| CREATE VIEW MyView3 AS  SELECT  TUMBLE\_START(proctime, INTERVAL '10' MINUTES) AS window\_start,  TUMBLE\_END(proctime, INTERVAL '10' MINUTES) AS window\_end,  TUMBLE\_PROCTIME(proctime, INTERVAL '10' MINUTES) as window\_proctime,  item,  MAX(price) as max\_price  FROM MyTable1  GROUP BY TUMBLE(proctime, INTERVAL '10' MINUTES), item; |

### 时态表（Temporal Tables）

时态表（Temporal Table）是一张随时间变化的表 – 在 Flink 中称为[动态表](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/dynamic_tables/)，时态表中的每条记录都关联了一个或多个时间段，所有的 Flink 表都是时态的（动态的）。

时态表包含表的一个或多个有版本的表快照，时态表可以是一张跟踪所有变更记录的表（例如数据库表的 changelog，包含多个表快照），也可以是物化所有变更之后的表（例如数据库表，只有最新表快照）。

基本概念：

* 版本: 时态表可以划分成一系列带版本的表快照集合，表快照中的版本代表了快照中所有记录的有效区间，有效区间的开始时间和结束时间可以通过用户指定，根据时态表是否可以追踪自身的历史版本与否，时态表可以分为 版本表 和 普通表。
* 版本表: 如果时态表中的记录可以追踪和并访问它的历史版本，这种表我们称之为版本表，来自数据库的 changelog 可以定义成版本表。
* 普通表: 如果时态表中的记录仅仅可以追踪并和它的最新版本，这种表我们称之为普通表，来自数据库 或 HBase 的表可以定义成普通表。

#### 设计初衷

##### 关联一张版本表

以**订单流关联产品表**这个场景举例：

* orders 表包含了来自 Kafka 的实时订单流
* product\_changelog 表来自数据库表 products 的 changelog, 产品的价格在数据库表 products 中是随时间实时变化的。

通过主键、时间可以获取不同版本的数据，例如取不同时间的商品价格。

|  |
| --- |
| SELECT *\** FROM product\_changelog;  (changelog kind) update\_time product\_id product\_name price ================= =========== ========== ============  +(INSERT) 00:01:00 p\_001 scooter 11.11 +(INSERT) 00:02:00 p\_002 basketball 23.11 -(UPDATE\_BEFORE) 12:00:00 p\_001 scooter 11.11 +(UPDATE\_AFTER) 12:00:00 p\_001 scooter 12.99 -(UPDATE\_BEFORE) 12:00:00 p\_002 basketball 23.11  +(UPDATE\_AFTER) 12:00:00 p\_002 basketball 19.99 -(DELETE) 18:00:00 p\_001 scooter 12.99 |

表 product\_changelog 表示数据库表 products不断增长的 changelog, 比如，产品 scooter 在时间点 00:01:00的初始价格是 11.11, 在 12:00:00 的时候涨价到了 12.99, 在 18:00:00 的时候这条产品价格记录被删除。

如果我们想输出 product\_changelog 表在 10:00:00 对应的版本，表的内容如下所示：

|  |
| --- |
| update\_time product\_id product\_name price  **===========** **==========** **============** **=====**  00**:**01**:**00 p\_001 scooter 11.11  00**:**02**:**00 p\_002 basketball 23.11 |

如果我们想输出 product\_changelog 表在 13:00:00 对应的版本，表的内容如下所示：

|  |
| --- |
| update\_time product\_id product\_name price  **===========** **==========** **============** **=====**  12**:**00**:**00 p\_001 scooter 12.99  12**:**00**:**00 p\_002 basketball 19.99 |

上述例子中，products 表的版本是通过 update\_time 和 product\_id 进行追踪的，product\_id 对应 product\_changelog 表的主键，update\_time 对应事件时间。

在 Flink 中, 这由**[版本表](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/versioned_tables/" \l "%e5%a3%b0%e6%98%8e%e7%89%88%e6%9c%ac%e8%a1%a8)**表示。

##### 关联一张普通表

另一方面，某些用户案列需要连接变化的维表，该表是外部数据库表。

假设 LatestRates 是一个物化的最新汇率表 (比如：一张 HBase 表)，LatestRates 总是表示 HBase 表 Rates 的最新内容。

我们在 10:15:00 时查询到的内容如下所示：

|  |
| --- |
| 10:15:00 **>** **SELECT** **\*** **FROM** LatestRates;  currency rate  **=========** **====**  US Dollar 102  Euro 114  Yen 1 |

我们在 11:00:00 时查询到的内容如下所示：

|  |
| --- |
| 11:00:00 **>** **SELECT** **\*** **FROM** LatestRates;  currency rate  **=========** **====**  US Dollar 102  Euro 116  Yen 1 |

在 Flink 中, 这由**[普通表](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/versioned_tables/" \l "%e5%a3%b0%e6%98%8e%e6%99%ae%e9%80%9a%e8%a1%a8)**表示。

#### 时态表

Flink 使用主键约束和事件时间来定义一张版本表和版本视图。

**注意：仅 Blink planner 支持此功能。**

##### 声明版本表

在 Flink 中，定义了**主键约束和事件时间属性的表就是版本表**。

|  |
| --- |
| -- 定义一张版本表  **CREATE** **TABLE** product\_changelog **(**  product\_id STRING**,**  product\_name STRING**,**  product\_price **DECIMAL(**10**,** 4**),**  update\_time **TIMESTAMP(**3**)** METADATA **FROM** 'value.source.timestamp' VIRTUAL**,**  **PRIMARY** **KEY(**product\_id**)** **NOT** ENFORCED**,** -- (1) 定义主键约束  WATERMARK **FOR** update\_time **AS** update\_time -- (2) 通过 watermark 定义事件时间  **)** **WITH** **(**  'connector' **=** 'kafka'**,**  'topic' **=** 'products'**,**  'scan.startup.mode' **=** 'earliest-offset'**,**  'properties.bootstrap.servers' **=** 'localhost:9092'**,**  'value.format' **=** 'debezium-json'  **);** |

行 (1) 为表 product\_changelog 定义了主键,

行 (2) 把 update\_time 定义为表 product\_changelog 的事件时间，因此 product\_changelog 是一张版本表。

**注意:**METADATA FROM 'value.source.timestamp' VIRTUAL 语法的意思是从每条 changelog 中抽取 changelog 对应的数据库表中操作的执行时间，强烈推荐使用数据库表中操作的执行时间作为事件时间，否则通过时间抽取的版本可能和数据库中的版本不匹配。

##### 声明版本视图

Flink 也支持定义版本视图只要一个视图包含主键和事件时间便是一个版本视图。

假设我们有表 RatesHistory 如下所示：

|  |
| --- |
| -- 定义一张 append-only 表  **CREATE** **TABLE** RatesHistory **(**  currency\_time **TIMESTAMP(**3**),**  currency STRING**,**  rate **DECIMAL(**38**,** 10**),**  WATERMARK **FOR** currency\_time **AS** currency\_time -- 定义事件时间  **)** **WITH** **(**  'connector' **=** 'kafka'**,**  'topic' **=** 'rates'**,**  'scan.startup.mode' **=** 'earliest-offset'**,**  'properties.bootstrap.servers' **=** 'localhost:9092'**,**  'format' **=** 'json' -- 普通的 append-only 流  **)** |

表 RatesHistory 代表一个兑换日元货币汇率表（日元汇率为1），该表是不断增长的 append-only 表。 例如，欧元 兑换 日元 从 09:00:00 到 10:45:00 的汇率为 114。从 10:45:00 到 11:15:00 的汇率为 116。

|  |
| --- |
| **SELECT** **\*** **FROM** RatesHistory**;**  currency\_time currency rate  **=============** **=========** **====**  09**:**00**:**00 US Dollar 102  09**:**00**:**00 Euro 114  09**:**00**:**00 Yen 1  10**:**45**:**00 Euro 116  11**:**15**:**00 Euro 119  11**:**49**:**00 Pounds 108 |

为了在 RatesHistory 上定义版本表，Flink 支持通过**[去重查询](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/" \l "%e5%8e%bb%e9%87%8d)**定义版本视图， 去重查询可以产出一个有序的 changelog 流，去重查询能够推断主键并保留原始数据流的事件时间属性。

|  |
| --- |
| **CREATE** **VIEW** versioned\_rates **AS**  **SELECT** currency**,** rate**,** currency\_time -- (1) `currency\_time` 保留了事件时间  **FROM** **(**  **SELECT** **\*,**  **ROW\_NUMBER()** OVER **(PARTITION** **BY** currency -- (2) `currency` 是去重 query 的 unique key，可以作为主键  **ORDER** **BY** currency\_time **DESC)** **AS** **rowNum**  **FROM** RatesHistory **)**  **WHERE** **rowNum** **=** 1**;**  -- 视图 `versioned\_rates` 将会产出如下的 changelog:  **(**changelog kind**)** currency\_time currency rate  **================** **=============** **=========** **====**  **+(INSERT)** 09**:**00**:**00 US Dollar 102  **+(INSERT)** 09**:**00**:**00 Euro 114  **+(INSERT)** 09**:**00**:**00 Yen 1  **+(**UPDATE\_AFTER**)** 10**:**45**:**00 Euro 116  **+(**UPDATE\_AFTER**)** 11**:**15**:**00 Euro 119  **+(INSERT)** 11**:**49**:**00 Pounds 108  **{%** endhighlight **sql** **%}** |
| 行 `(1)` 保留了事件时间作为视图 `versioned\_rates` 的事件时间，  行 `(2)` 使得视图 `versioned\_rates` 有了主键, 因此视图 `versioned\_rates` 是一个版本视图。  视图中的去重 query 会被 Flink 优化并高效地产出 changelog stream, 产出的 changelog 保留了主键约束和事件时间。 |
| 如果我们想输出 `versioned\_rates` 表在 `11:00:00` 对应的版本，表的内容如下所示： |
| currency\_time currency rate  **=============** **==========** **====**  09**:**00**:**00 US Dollar 102  09**:**00**:**00 Yen 1  10**:**45**:**00 Euro 116 |
| 如果我们想输出 versioned\_rates 表在 12:00:00 对应的版本，表的内容如下所示： |
| currency\_time currency rate  **=============** **==========** **====**  09**:**00**:**00 US Dollar 102  09**:**00**:**00 Yen 1  10**:**45**:**00 Euro 119  11**:**49**:**00 Pounds 108 |

##### 声明普通表

普通表的声明和 Flink 建表 DDL 一致，参考 [create table](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/create/" \l "create-table) 页面获取更多如何建表的信息。

|  |
| --- |
| -- 用 DDL 定义一张 HBase 表，然后我们可以在 SQL 中将其当作一张时态表使用  -- 'currency' 列是 HBase 表中的 rowKey  **CREATE** **TABLE** LatestRates **(**  currency STRING**,**  fam1 **ROW<**rate **DOUBLE>**  **)** **WITH** **(**  'connector' **=** 'hbase-1.4'**,**  'table-name' **=** 'rates'**,**  'zookeeper.quorum' **=** 'localhost:2181'  **);** |

#### 时态表函数

时态表函数是一种过时的方式去定义时态表并关联时态表的数据，现在我们可以用时态表 DDL 去定义时态表，用[时态表 Join](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/streaming/joins.html" \l "%E6%97%B6%E6%80%81%E8%A1%A8-join) 语法去关联时态表。

时态表函数和时态表 DDL 最大的区别在于：**时态表 DDL 可以在纯 SQL 环境中使用但是时态表函数不支持，用时态表 DDL 声明的时态表支持 changelog 流和 append-only 流但时态表函数仅支持 append-only 流**。

为了访问时态表中的数据，必须传递一个**[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/streaming/time_attributes.html)**，该属性确定将要返回的表的版本。 Flink 使用[表函数](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/functions/udfs.html" \l "%E8%A1%A8%E5%80%BC%E5%87%BD%E6%95%B0)的 SQL 语法提供一种表达它的方法。

定义后，时态表函数将使用单个时间参数 timeAttribute 并返回一个行集合。 该集合包含相对于给定时间属性的所有现有主键的行的最新版本。

##### 使用示例

假设我们基于 RatesHistory 表定义了一个时态表函数，我们可以通过以下方式查询该函数 Rates(timeAttribute)：

|  |
| --- |
| **SELECT** **\*** **FROM** RatesHistory |
| currency\_time currency rate  **=============** **=========** **====**  09**:**00**:**00 US Dollar 102  09**:**00**:**00 Euro 114  09**:**00**:**00 Yen 1  10**:**45**:**00 Euro 116  11**:**15**:**00 Euro 119  11**:**49**:**00 Pounds 108 |
| **SELECT** **\*** **FROM** Rates**(**'10:15:00'**)** |
| rowtime currency rate  **=======** **=========** **====**  09**:**00**:**00 US Dollar 102  09**:**00**:**00 Euro 114  09**:**00**:**00 Yen 1 |
| **SELECT** **\*** **FROM** Rates**(**'11:00:00'**)** |
| rowtime currency rate  **========** **=========** **====**  09**:**00**:**00 US Dollar 102  10**:**45**:**00 Euro 116  09**:**00**:**00 Yen 1 |

对 Rates(timeAttribute) 的每个查询都将返回给定 timeAttribute 的 Rates 状态。

|  |
| --- |
| **注意**：当前 Flink 不支持使用常量时间属性参数直接查询时态表函数。目前，时态表函数只能在 join 中使用。  上面的示例用于为函数 Rates(timeAttribute) 返回内容提供直观信息。 |

##### 使用案例（批的方式）

* 需求描述

假设有一张订单表rateOrders和一张汇率表Rates，那么订单来自于不同的地区，所以支付的币种各不一样，那么假设需要统计每个订单在下单时候Yen币种对应的金额。

* + RateOrders 数据

|  |  |  |
| --- | --- | --- |
| **amount** | **currency** | **order\_time** |
| 7 | Euro（欧元） | 2 |
| 6 | US Dollar（美元） | 3 |
| 0.05 | Yen（日元） | 4 |
| 8 | Euro（欧元） | 5 |

* + Rates 数据

|  |  |  |
| --- | --- | --- |
| **currency** | **rate** | **rate\_time** |
| US Dollar | 102 | 1 |
| Euro | 114 | 1 |
| Yen | 1 | 1 |
| Euro | 116 | 5 |
| Euro | 117 | 7 |

* + 统计需求对应的SQL

|  |
| --- |
| **SELECT** o**.**currency**,** o**.**amount**,** r**.**rate  o**.**amount **\*** r**.**rate **AS** yen\_amount  **FROM**  Orders **AS** o**,**  **LATERAL** **TABLE** **(**Rates**(**o**.**rowtime**))** **AS** r  **WHERE** r**.**currency **=** o**.**currency |

* + 预期结果

|  |  |  |  |
| --- | --- | --- | --- |
| **currency** | **amount** | **rate** | **yen\_amount** |
| US Dollar | 1 | 102 | 102 |
| Yen | 50 | 1 | 50 |
| Euro | 2 | 114 | 228 |
| Euro | 3 | 116 | 348 |

* 实现代码：

|  |
| --- |
| public class TemporalTablesFunctionBatch {  public static void main(String[] args) throws Exception {  // 获取 stream 和 table 环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, bsSettings);  env.setParallelism(1);   //Orders 数据  List<Tuple3<Double, String, Long>> orderList = new ArrayList<>();  orderList.add(new Tuple3<>(7D, "Euro", 2L));  orderList.add(new Tuple3<>(7D, "US Dollar", 3L));  orderList.add(new Tuple3<>(0.05D, "Yen", 4L));  orderList.add(new Tuple3<>(8D, "Euro", 5L));  DataStream<Tuple3<Double, String, Long>> orderStream = env.fromCollection(orderList)  .assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple3<Double, String, Long>>forBoundedOutOfOrderness(  Duration.ofSeconds(0))  .withTimestampAssigner(new SerializableTimestampAssigner<Tuple3<Double, String, Long>>() {  @Override  public long extractTimestamp(Tuple3<Double, String, Long> element, long l) {  return element.f2 \* 1000;  }  }));   //Rates 数据  List<Tuple3<String, Integer, Long>> rateList = new ArrayList<>();  rateList.add(new Tuple3<>("US Dollar", 102, 1L));  rateList.add(new Tuple3<>("Euro", 114, 1L));  rateList.add(new Tuple3<>("Yen", 1, 1L));  rateList.add(new Tuple3<>("Euro", 116, 5L));  rateList.add(new Tuple3<>("Euro", 117, 7L));   DataStream<Tuple3<String, Integer, Long>> rateStream = env.fromCollection(rateList)  .assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple3<String, Integer, Long>>forBoundedOutOfOrderness(  Duration.ofSeconds(0))  .withTimestampAssigner(new SerializableTimestampAssigner<Tuple3<String, Integer, Long>>() {  @Override  public long extractTimestamp(Tuple3<String, Integer, Long> element, long l) {  return element.f2 \* 1000;  }  }));   //将两个数据流转换为表  Table orderTable = tableEnv.fromDataStream(orderStream, $("amount"), $("currency"), $("rowtime").rowtime());  Table rateTable = tableEnv.fromDataStream(rateStream, $("currency"), $("rate"), $("rowtime").rowtime());   //将两个表对象创建临时态表  tableEnv.createTemporaryView("Orders", orderTable);  tableEnv.createTemporaryView("RatesHistory", rateTable);   //定义一个TemporalTableFunction  TemporalTableFunction dimRate = rateTable.createTemporalTableFunction($("rowtime"), $("currency"));  //注册时态表函数  tableEnv.createTemporaryFunction("Rates", dimRate);   //关联查询  Table result = tableEnv  .sqlQuery("SELECT o.currency, o.amount, r.rate, o.amount \* r.rate AS yen\_amount " +  "FROM " +  " Orders AS o, " +  " LATERAL TABLE (Rates(o.rowtime)) AS r " +  "WHERE r.currency = o.currency");   //打印输出  DataStream resultDs = tableEnv.toAppendStream(result, Row.class);  resultDs.print();  env.execute();  } } |

##### 使用案例（流的方式）

* 需求描述

从Kafka消费事件流（browse\_event）和商品流（product\_history\_info）数据，并根据事件流中的商品id关联商品流的数据。

* + browse\_event: 事件表，某个用户在某个时刻浏览了某个商品，以及商品的价值

|  |  |  |  |
| --- | --- | --- | --- |
| **userID** | **eventTime** | **eventType** | **productID** |
| user\_001 | 2021-01-01 00:00:00 | browse | product\_005 |
| user\_001 | 2021-01-01 00:00:01 | browse | product\_005 |
| user\_001 | 2021-01-01 00:00:02 | browse | product\_005 |
| user\_001 | 2021-01-01 00:00:03 | browse | product\_005 |
| user\_001 | 2021-01-01 00:00:04 | browse | product\_005 |
| user\_001 | 2021-01-01 00:00:05 | browse | product\_005 |
| user\_001 | 2021-01-01 00:00:06 | browse | product\_005 |
| user\_002 | 2021-01-01 00:00:01 | browse | product\_003 |
| user\_002 | 2021-01-01 00:00:02 | browse | product\_003 |
| user\_002 | 2021-01-01 00:00:05 | browse | product\_003 |
| user\_002 | 2021-01-01 00:00:06 | browse | product\_003 |

* + product\_history\_info:商品基础信息表，记录了商品历史以来的基础信息。如下:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **productID** | **productName** | **productCategory** | **updatedAt** | **productPrice** |
| product\_005 | 苹果电脑 | 电脑 | 2021-01-01 0:00:00 | 20 |
| product\_005 | 苹果电脑 | 电脑 | 2021-01-01 0:00:02 | 30 |
| product\_005 | 苹果电脑 | 电脑 | 2021-01-01 0:00:05 | 40 |
| product\_003 | 华为手机 | 手机 | 2021-01-01 0:00:02 | 20 |
| product\_003 | 华为手机 | 手机 | 2021-01-01 0:00:05 | 30 |

* 实现步骤
  + 分别创建事件表和商品表的Topic

|  |
| --- |
| /export/servers/kafka\_2.11-0.10.0.0/bin/kafka-topics.sh --create --zookeeper node01:2181 --topic browseTopic --partitions 3 --replication-factor 2 |
| /export/servers/kafka\_2.11-0.10.0.0/bin/kafka-topics.sh --create --zookeeper node01:2181 --topic productHistoryInfoTopic --partitions 3 --replication-factor 2 |

* + 将以下数据分别写入到kafka指定的Topic中
    - 事件流

|  |
| --- |
| bin/kafka-console-producer.sh --broker-list node01:9092 --topic browseTopic |
| {"userID": "user\_001", "eventTime": "2021-01-01 00:00:00", "eventType": "browse", "productID": "product\_005"} {"userID": "user\_001", "eventTime": "2021-01-01 00:00:01", "eventType": "browse", "productID": "product\_005"} {"userID": "user\_001", "eventTime": "2021-01-01 00:00:02", "eventType": "browse", "productID": "product\_005"} {"userID": "user\_001", "eventTime": "2021-01-01 00:00:03", "eventType": "browse", "productID": "product\_005"} {"userID": "user\_001", "eventTime": "2021-01-01 00:00:04", "eventType": "browse", "productID": "product\_005"} {"userID": "user\_001", "eventTime": "2021-01-01 00:00:05", "eventType": "browse", "productID": "product\_005"} {"userID": "user\_001", "eventTime": "2021-01-01 00:00:06", "eventType": "browse", "productID": "product\_005"} {"userID": "user\_002", "eventTime": "2021-01-01 00:00:01", "eventType": "browse", "productID": "product\_003"} {"userID": "user\_002", "eventTime": "2021-01-01 00:00:02", "eventType": "browse", "productID": "product\_003"} {"userID": "user\_002", "eventTime": "2021-01-01 00:00:05", "eventType": "browse", "productID": "product\_003"} {"userID": "user\_002", "eventTime": "2021-01-01 00:00:06", "eventType": "browse", "productID": "product\_003"} |

* + - 商品流

|  |
| --- |
| bin/kafka-console-producer.sh --broker-list node01:9092 --topic productHistoryInfoTopic |
| {"productID":"product\_005","productName":"苹果电脑","productCategory":"电脑","updatedAt":"2021-01-01 00:00:00", "productPrice": 20} {"productID":"product\_005","productName":"苹果电脑","productCategory":"电脑","updatedAt":"2021-01-01 00:00:02", "productPrice": 30} {"productID":"product\_005","productName":"苹果电脑","productCategory":"电脑","updatedAt":"2021-01-01 00:00:05", "productPrice": 40} {"productID":"product\_003","productName":"华为手机","productCategory":"手机","updatedAt":"2021-01-01 00:00:02", "productPrice": 20} {"productID":"product\_003","productName":"华为手机","productCategory":"手机","updatedAt":"2021-01-01 00:00:05", "productPrice": 30} |

* 实现代码：

|  |
| --- |
| public class TemporalTablesFunctionStreaming {  public static void main(String[] args) throws Exception {   //1、解析命令行参数  //browse log  String kafkaBootstrapServers = "node01:9092,node02:9092,node03:9092";  String browseTopic = "browseTopic";  String browseTopicGroupID = "browseTopicGroupID11";   //product history info  String productInfoTopic = "productHistoryInfoTopic";  String productInfoGroupID = "productHistoryInfoGroupID11";   //2、设置运行环境  StreamExecutionEnvironment streamEnv = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());  //streamEnv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);  EnvironmentSettings settings = EnvironmentSettings.newInstance().inStreamingMode().useBlinkPlanner().build();  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(streamEnv, settings);  streamEnv.setParallelism(1);  // 每秒更新一次watermark  streamEnv.getConfig().setAutoWatermarkInterval(1000L);   //3、注册Kafka数据源  //注意: 为了在北京时间和时间戳之间有直观的认识，这里的UserBrowseLog中增加了一个字段eventTimeTimestamp作为eventTime的时间戳  Properties browseProperties = new Properties();  browseProperties.put("bootstrap.servers", kafkaBootstrapServers);  browseProperties.put("group.id", browseTopicGroupID);  browseProperties.put(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest");  browseProperties.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG, "false");  DataStream<UserBrowseLog> browseStream = streamEnv  .addSource(new FlinkKafkaConsumer<>(browseTopic, new SimpleStringSchema(), browseProperties))  .process(new BrowseKafkaProcessFunction())  .assignTimestampsAndWatermarks(new WatermarkStrategy<UserBrowseLog>() {  @Override  public WatermarkGenerator<UserBrowseLog> createWatermarkGenerator(  WatermarkGeneratorSupplier.Context context) {  return new WatermarkGenerator<UserBrowseLog>() {  private long maxTimestamp;  private long delay = 0;  // 每来一条数据，将这条数据与maxTimesStamp比较，看是否需要更新watermark  @Override  public void onEvent(  UserBrowseLog event,  long eventTimestamp,  WatermarkOutput output) {   maxTimestamp = Math.max(maxTimestamp, event.getEventTimeTimestamp());  }  // 周期性更新watermark  @Override  public void onPeriodicEmit(WatermarkOutput output) {  output.emitWatermark(new Watermark(maxTimestamp - delay));  }  };  }  }  // 必须指定中的timeStamp，否则报错  .withTimestampAssigner((element, recordTimestamp) -> element.getEventTimeTimestamp())  );   tableEnv.createTemporaryView("browse", browseStream, $("userID"),$("eventTimeTimestamp"),$("eventType"),$("eventTime"),$("productID"),$("browseRowtime").rowtime());  //tableEnv.toAppendStream(tableEnv.scan("browse"),Row.class).print();   //4、注册时态表(Temporal Table)  //注意: 为了在北京时间和时间戳之间有直观的认识，这里的ProductInfo中增加了一个字段updatedAtTimestamp作为updatedAt的时间戳  Properties productInfoProperties = new Properties();  productInfoProperties.put("bootstrap.servers", kafkaBootstrapServers);  productInfoProperties.put("group.id", productInfoGroupID);  productInfoProperties.put(ConsumerConfig.AUTO\_OFFSET\_RESET\_CONFIG, "earliest");  productInfoProperties.put(ConsumerConfig.ENABLE\_AUTO\_COMMIT\_CONFIG, "false");  DataStream<ProductInfo> productInfoStream = streamEnv  .addSource(new FlinkKafkaConsumer<>(productInfoTopic, new SimpleStringSchema(), productInfoProperties))  .process(new ProductInfoProcessFunction())  .assignTimestampsAndWatermarks(new WatermarkStrategy<ProductInfo>() {  @Override  public WatermarkGenerator<ProductInfo> createWatermarkGenerator(  WatermarkGeneratorSupplier.Context context) {  return new WatermarkGenerator<ProductInfo>() {  private long maxTimestamp;  private long delay = 0;   @Override  public void onEvent(  ProductInfo event,  long eventTimestamp,  WatermarkOutput output) {  maxTimestamp = Math.max(maxTimestamp, event.getUpdatedAtTimestamp());  }   @Override  public void onPeriodicEmit(WatermarkOutput output) {  output.emitWatermark(new Watermark(maxTimestamp - delay));  }  };  }  }  // 必须指定中的timeStamp，否则报错  .withTimestampAssigner((element, recordTimestamp) -> element.getUpdatedAtTimestamp())  );   tableEnv.createTemporaryView("productInfo", productInfoStream, $("productID"),$("productName"),$("productCategory"),$("productPrice"),$("updatedAt"),$("updatedAtTimestamp"),$("productInfoRowtime").rowtime());  //tableEnv.toAppendStream(tableEnv.scan("productInfo"),Row.class).print();   //设置Temporal Table的时间属性和主键  TemporalTableFunction productInfo = tableEnv.scan("productInfo").createTemporalTableFunction($("productInfoRowtime"), $("productID"));  //注册TableFunction  tableEnv.createTemporaryFunction("productInfoFunc", productInfo);   //5、运行SQL  String sql = ""  + "SELECT "  + "browse.userID, "  + "browse.eventTime, "  + "browse.eventTimeTimestamp, "  + "browse.eventType, "  + "browse.productID, "  + "productInfo.productID, "  + "productInfo.productName, "  + "productInfo.productCategory, "  + "productInfo.productPrice, "  + "productInfo.updatedAt, "  + "productInfo.updatedAtTimestamp "  + "FROM "  + " browse, "  + " LATERAL TABLE (productInfoFunc(browse.browseRowtime)) as productInfo "  + "WHERE "  + " browse.productID=productInfo.productID";   Table table = tableEnv.sqlQuery(sql);  tableEnv.toAppendStream(table, Row.class).print();   //6、开始执行  try {  streamEnv.execute();  } catch (Exception exception) {  exception.printStackTrace();  }  }    /\*\*  \* 解析Kafka数据  \*/  public static class BrowseKafkaProcessFunction extends ProcessFunction<String, UserBrowseLog> {  @Override  public void processElement(String value, Context ctx, Collector<UserBrowseLog> out) throws Exception {  try {   UserBrowseLog log = JSON.parseObject(value, UserBrowseLog.class);  // 增加一个long类型的时间戳  // 指定eventTime为yyyy-MM-dd HH:mm:ss格式的北京时间  DateTimeFormatter format = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  OffsetDateTime eventTime = LocalDateTime.parse(log.getEventTime(), format).atOffset(ZoneOffset.of("+08:00"));  // 转换成毫秒时间戳  long eventTimeTimestamp = eventTime.toInstant().toEpochMilli();  log.setEventTimeTimestamp(eventTimeTimestamp);   out.collect(log);  } catch (Exception ex) {  System.out.println("解析Kafka数据异常..." + ex);  }  }  }   /\*\*  \* 解析Kafka数据  \*/  public static class ProductInfoProcessFunction extends ProcessFunction<String, ProductInfo> {  @Override  public void processElement(String value, Context ctx, Collector<ProductInfo> out) throws Exception {  try {   ProductInfo log = JSON.parseObject(value, ProductInfo.class);   // 增加一个long类型的时间戳  // 指定eventTime为yyyy-MM-dd HH:mm:ss格式的北京时间  DateTimeFormatter format = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");  OffsetDateTime eventTime = LocalDateTime.parse(log.getUpdatedAt(), format).atOffset(ZoneOffset.of("+08:00"));  // 转换成毫秒时间戳  long eventTimeTimestamp = eventTime.toInstant().toEpochMilli();  log.setUpdatedAtTimestamp(eventTimeTimestamp);   out.collect(log);  } catch (Exception ex) {  }  }  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class UserBrowseLog implements Serializable {  private String userID;  private String eventTime;  private String eventType;  private String productID;  private Long eventTimeTimestamp;  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class ProductInfo implements Serializable {  //产品id  private String productID;  //产品名称  private String productName;  //产品类型  private String productCategory;  //更新时间  private String updatedAt;  //更新时间戳  private Long updatedAtTimestamp;  private double productPrice;  } } |

* 注意事项

|  |
| --- |
| 1. 在join时左表（左输入/探针侧）去关联一个时态表（右输入/构建侧），两边的时间语义必须相同，否则会抛出类似的异常：Non processing timeAttribute [TIME ATTRIBUTE(ROWTIME)] passed as the argument to TemporalTableFunction。 2. 基于处理时间的时态 Join 中， 如果右侧表不是可以直接查询外部系统的表而是普通的数据流，时态表函数 Join 和 时态表 Join 的语义都有问题，时态表函数 Join 仍然允许使用，但是时态表 Join 禁用了该功能。 语义问题的原因是 Join 算子没办法知道右侧时态表（构建侧）的完整快照是否到齐，这可能导致左侧的流在启动时关联不到用户期待的数据, 在生产环境中可能误导用户。 3. 在处理时间语义下的时态表函数的只会保留最新的一份数据，时间事件语义下则会保留每个水位对应的动态表。 |

#### Temporal Table JOIN vs 双流 JOIN

Temporal Table JOIN 和 双流 JOIN都可以管理State，那么他们的本质区别是什么? 那就是计算驱动的差别，Temporal Table JOIN是单边驱动，Temporal Table是被动的查询，而双流JOIN是双边驱动，两边都是主动的进行JOIN计算。

#### [Flink SQL之维表join之Temporal Table Join](https://www.cnblogs.com/qiu-hua/p/14027902.html)

维表是数仓中的一个概念，维表中的维度属性是观察数据的角度，在建设离线数仓的时候，通常是将维表与事实表进行关联构建星型模型。**在实时数仓中，同样也有维表与事实表的概念，其中事实表通常存储在kafka中，维表通常存储在外部设备中(比如MySQL，HBase)**。对于每条流式数据，可以关联一个外部维表数据源，为实时计算提供数据关联查询。维表可能是会不断变化的，在维表JOIN时，需指明这条记录关联维表快照的时刻。

需要注意是：**目前Flink SQL的维表JOIN仅支持对当前时刻维表快照的关联(处理时间语义)，而不支持事实表rowtime所对应的的维表快照(事件时间语义)。通过本文你可以了解到**：

* 如何使用Flink SQL创建表
* 如何定义Kafka数据源表
* 如何定义MySQL数据源表
* 什么是Temporal Table Join
* 维表join的案例

### [流上的 Join](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/)

Flink SQL 支持对动态表进行复杂灵活的连接操作。有几种不同类型的连接来解释可能需要的各种语义查询。

默认情况下，联接的顺序未优化。表按照它们在FROM子句中指定的顺序进行连接。您可以调整连接查询的性能，方法是首先列出更新频率最低的表，最后列出更新频率最高的表。确保以不会产生交叉联接（笛卡尔积）的顺序指定表，这不受支持并且会导致查询失败。

#### [常规 Join](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/streaming/joins.html" \l "%E5%B8%B8%E8%A7%84-join)

常规 Join 是最常用的 Join 用法。在常规 Join 中，任何新记录或对 Join 两侧表的任何更改都是可见的，并会影响最终整个 Join 的结果。例如，如果 Join 左侧插入了一条新的记录，那么它将会与 Join 右侧过去与将来的所有记录进行 Join 运算。

|  |
| --- |
| SELECT *\** FROM Orders INNER JOIN Product ON Orders.productId = Product.id |

对于流式查询，常规连接的语法是最灵活的，并且允许任何类型的更新（插入、更新、删除）输入表。

然而，常规 Join 隐含了一个重要的前提：即它需要在 Flink 的状态中永久保存 Join 两侧的数据。因而，如果 Join 操作中的一方或双方输入表持续增长的话，资源消耗也将会随之无限增长。您可以提供具有适当状态生存时间 (TTL) 的查询配置，以防止状态过大。

请注意，这可能会影响查询结果的正确性。有关详细信息，请参阅[查询配置](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "table-exec-state-ttl)。

对于流式查询，计算查询结果所需的状态可能会无限增长，具体取决于聚合类型和不同分组键的数量。请提供具有有效保留间隔的查询配置，以防止出现过多的状态。有关详细信息，请参阅[查询配置](https://ci.apache.org/docs/dev/table/streaming/query_configuration)。

##### INNER Equi-JOIN

目前仅支持 equi-join ，即 join 的联合条件至少拥有一个相等谓词。不支持任何 cross join 和 theta join。

注意： Join 的顺序没有进行优化，join 会按照 FROM 中所定义的顺序依次执行。请确保 join 所指定的表在顺序执行中不会产生不支持的 cross join （笛卡儿积）以至查询失败。

|  |
| --- |
| SELECT *\** FROM Orders INNER JOIN Product ON Orders.product\_id = Product.id |

##### OUTER Equi-JOIN

目前仅支持 equi-join ，即 join 的联合条件至少拥有一个相等谓词。不支持任何 cross join 和 theta join。

注意： Join 的顺序没有进行优化，join 会按照 FROM 中所定义的顺序依次执行。请确保 join 所指定的表在顺序执行中不会产生不支持的 cross join （笛卡儿积）以至查询失败。

|  |
| --- |
| SELECT *\** FROM Orders  LEFT JOIN Product ON Orders.productId = Product.id  SELECT *\** FROM Orders RIGHT JOIN Product ON Orders.productId = Product.id  SELECT *\** FROM Orders FULL OUTER JOIN Product ON Orders.productId = Product.id |

#### [时间区间 Join](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/streaming/joins.html" \l "%E6%97%B6%E9%97%B4%E5%8C%BA%E9%97%B4-join)

注意：Interval join （时间区间关联）是常规 join 的子集，可以使用流的方式进行处理。

Interval join需要至少一个 equi-join 谓词和一个限制了双方时间的 join 条件。例如使用两个适当的范围谓词（<, <=, >=, >），一个 BETWEEN 谓词或一个比较两个输入表中相同类型的 [时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.11/zh/dev/table/streaming/time_attributes.html) （即处理时间和事件时间）的相等谓词

比如，以下谓词是合法的 interval join 条件：

* ltime = rtime
* ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE
* ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND

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

以上示例中，所有在收到后四小时内发货的 order 会与他们相关的 shipment 进行 join。

#### 时态表 Join

##### 基于事件时间的时态 Join

时态表 Join 意味着对任意表（左输入/探针侧）去关联一个时态表（右输入/构建侧）的版本，时态表可以是一张跟踪所有变更记录的表（例如数据库表的 changelog，包含多个表快照），也可以是物化所有变更之后的表（例如数据库表，只有最新表快照）。

Flink 使用了 SQL:2011 标准引入的时态表 Join 语法，时态表 Join 的语法如下:

|  |
| --- |
| SELECT [column\_list] FROM table1 [AS <alias1>] [LEFT] JOIN table2 FOR SYSTEM\_TIME AS OF table1.{ proctime | rowtime } [AS <alias2>] ON table1.column-name1 = table2.column-name1 |

使用事件时间属性（即行时间属性），可以检索键的值，就像它在过去的某个时间点一样。这允许在共同的时间点连接两个表。版本化表将存储自上次水印以来的所有版本 - 按时间标识。

例如，假设我们有一个订单表，每个订单都有不同货币的价格。要将这个表正确地标准化为单一货币，例如美元，每个订单都需要从下订单的时间点加入正确的货币兑换率。

|  |
| --- |
| -- Create a table of orders. This is a standard -- append-only dynamic table. CREATE TABLE orders (  order\_id STRING,  price DECIMAL(32,2),  currency STRING,  order\_time TIMESTAMP(3),  WATERMARK FOR order\_time AS order\_time ) WITH (/\* ... \*/);  -- Define a versioned table of currency rates.  -- This could be from a change-data-capture -- such as Debezium, a compacted Kafka topic, or any other -- way of defining a versioned table.  CREATE TABLE currency\_rates (  currency STRING,  conversion\_rate DECIMAL(32, 2),  update\_time TIMESTAMP(3) METADATA FROM `values.source.timestamp` VIRTUAL  WATERMARK FOR update\_time AS update\_time ) WITH (  'connector' = 'upsert-kafka',  /\* ... \*/ );  SELECT   order\_id,  price,  currency,  conversion\_rate,  order\_time, FROM orders LEFT JOIN currency\_rates FOR SYSTEM TIME AS OF orders.order\_time ON orders.currency = currency\_rates.currency  order\_id price currency conversion\_rate order\_time ====== ==== ====== ============ ======== o\_001 11.11 EUR 1.14 12:00:00 o\_002 12.51 EUR 1.10 12:0600 |

**注意： event-time temporal join是由左右两侧的水印触发的；请确保连接的双方都正确设置了水印。**

**注意：事件时间临时连接需要临时连接条件的等价条件中包含的主键，例如，P.product\_id表的主键product\_changelog被约束在条件中O.product\_id = P.product\_id。**

与[常规join相比](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "regular-joins)，尽管构建方面发生了变化，但之前的时态表结果不会受到影响。与[时间区间j](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "interval-joins)oin相比，时态表连接不定义将在其中连接记录的时间窗口。

探测器端的记录总是在时间属性指定的时间与构建端的版本连接。

因此，构建端的行可能是任意旧的。随着时间的推移，不再需要的记录版本（对于给定的主键）将从状态中删除。

##### 基于处理时间的时态 Join

处理时间时态表join使用处理时间属性将行与外部版本化表中的键的最新版本相关联。

根据定义，使用处理时间属性，连接将始终返回给定键的最新值。可以将查找表视为一个简单的 HashMap<K, V>，它存储来自构建端的所有记录。这种连接的强大之处在于，当在 Flink 中将表具体化为动态表不可行时，它允许 Flink 直接针对外部系统工作。

以下处理时时态表orders联接示例显示了应与表联接的仅附加表LatestRates。 LatestRates是一个以最新速率物化的维度表（例如HBase 表）。在10:15, 10:30, 时10:52， 的内容LatestRates如下：

|  |
| --- |
| 10:15> SELECT \* FROM LatestRates;  currency rate ======== ====== US Dollar 102 Euro 114 Yen 1  10:30> SELECT \* FROM LatestRates;  currency rate ======== ====== US Dollar 102 Euro 114 Yen 1  10:52> SELECT \* FROM LatestRates;  currency rate ======== ====== US Dollar 102 Euro 116 <==== changed from 114 to 116 Yen 1 |

LastestRates有时10:15和的内容10:30是相等的。欧元汇率从 114 变为 116 10:52。

Orders是一个仅附加表，表示给定amount和给定的付款currency。例如，10:15有一个订单金额为2 Euro。

|  |
| --- |
| SELECT *\** FROM Orders;  amount currency ====== =========  2 Euro <== arrived at time 10:15  1 US Dollar <== arrived at time 10:30  2 Euro <== arrived at time 10:52 |

鉴于这些表，我们想计算所有Orders转换为通用货币的方法。

|  |
| --- |
| amount currency rate amount\*rate ====== ========= ======= ============  2 Euro 114 228 <== arrived at time 10:15  1 US Dollar 102 102 <== arrived at time 10:30  2 Euro 116 232 <== arrived at time 10:52 |

借助时态表连接，我们可以在 SQL 中表达这样的查询：

|  |
| --- |
| SELECT  o.amount, o.currency, r.rate, o.amount \* r.rate FROM  Orders AS o  JOIN LatestRates FOR SYSTEM\_TIME AS OF o.proctime AS r  ON r.currency = o.currency |

**探针端**的每条记录都将与**构建端表**的当前版本连接。在我们的示例中，查询使用处理时间概念，因此新添加的订单将始终与LatestRates执行操作时的最新版本连接。

结果对于处理时间是不确定的。处理时时间连接最常用于通过外部表（即维度表）丰富流。

与[常规join相比](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "regular-joins)，尽管构建方面发生了变化，但之前的时态表结果不会受到影响。与时间区间join相比，时态表连接不定义记录连接的时间窗口，即旧行不存储在状态中。

##### 使用案例

* 需求描述

订单表和汇率表，将汇率表设置成时态表，这样用户就可以根据订单表中的下单时间Join下单时的汇率表当时最新的维度数据

* + 订单数据

|  |  |  |  |
| --- | --- | --- | --- |
| **order\_id** | **price** | **currency** | **order\_time** |
| 1 | 29 | RMB | 2021-01-02 00:00:00 |
| 2 | 19 | RMB | 2021-01-03 00:00:00 |
| 3 | 33 | RMB | 2021-01-11 00:00:00 |
| 4 | 55 | RMB | 2021-01-21 00:00:00 |

* + 汇率数据

|  |  |  |
| --- | --- | --- |
| **currency** | **conversion\_rate** | **update\_time** |
| RMB | 114 | 2021-01-01 00:00:00 |
| RMB | 115 | 2021-01-03 00:00:00 |
| RMB | 116 | 2021-01-19 00:00:00 |
| Euro | 119 | 2021-01-03 00:00:00 |
| USD | 99 | 2021-01-03 00:00:00 |
| USD | 100 | 2021-01-03 00:00:00 |
| Euro | 118 | 2021-01-03 00:00:00 |

* + 预期结果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **order\_id** | **price** | **currency** | **conversion\_rate** | **order\_time** |
| 2 | 19 | RMB | 115 | 2021-01-03T00:00 |
| 1 | 29 | RMB | 114 | 2021-01-02T00:00 |
| 4 | 55 | RMB | 116 | 2021-01-21T00:00 |
| 3 | 33 | RMB | 115 | 2021-01-11T00:00 |

* 实现代码：

|  |
| --- |
| public class TemporalTableJoinEventTime {  public static void main(String[] args) throws Exception {  // 1.准备环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   // 2.创建TableEnvironment(Blink planner)  EnvironmentSettings settings = EnvironmentSettings.newInstance().inStreamingMode().useBlinkPlanner().build();  StreamTableEnvironment tableEnvironment = StreamTableEnvironment.create(env, settings);   // 3.文件path  String ratesHistoryPath = TableApiTest.class.getClassLoader().getResource("ratesHistory.csv").getPath();  String ratesOrderPath = TableApiTest.class.getClassLoader().getResource("rateOrder.csv").getPath();   // 4.DDL（创建一张版本表）  String ratesHistory\_ddl =  "create table currency\_rates (\n" +  " currency STRING,\n" +  " conversion\_rate DECIMAL(32, 2),\n" +  " update\_time TIMESTAMP(3),\n" +  " PRIMARY KEY (currency) NOT ENFORCED,\n" +  " WATERMARK FOR update\_time AS update\_time \n" +  ") WITH (\n" +  " 'connector.type' = 'filesystem',\n" +  " 'connector.path' = '"+ratesHistoryPath+"',\n" +  " 'format.type' = 'csv'\n" +  ")";   tableEnvironment.executeSql(ratesHistory\_ddl);    //创建一张普通表  String ratesOrder\_DDL =  "create table orders (\n" +  " order\_id STRING,\n" +  " price DECIMAL(32,2),\n" +  " currency STRING,\n" +  " order\_time TIMESTAMP(3),\n" +  " WATERMARK FOR order\_time AS order\_time \n" +  ") WITH (\n" +  " 'connector.type' = 'filesystem',\n" +  " 'connector.path' = '"+ratesOrderPath+"',\n" +  " 'format.type' = 'csv'\n" +  ")";   tableEnvironment.executeSql(ratesOrder\_DDL);    // 6.通过SQL对表的查询，生成结果表  String sql =  "SELECT \n" +  " order\_id,\n" +  " price,\n" +  " orders.currency,\n" +  " conversion\_rate,\n" +  " order\_time\n" +  " FROM orders\n" +  " LEFT JOIN currency\_rates FOR SYSTEM\_TIME AS OF orders.order\_time\n" +  " ON orders.currency = currency\_rates.currency";   Table table = tableEnvironment.sqlQuery(sql);   // 7.将table表转换为DataStream  DataStream<Tuple2<Boolean, Row>> retractStream = tableEnvironment.toRetractStream(table, Row.class);  retractStream.print();  env.execute();  } } |

#### Lookup Join

Lookup join通常用于使用从外部系统查询的数据来丰富表。连接要求一个表具有处理时间属性，另一个表由查找源连接器支持。

Lookup join使用上述**基于处理时间的时态join**语法，并使用由查找源连接器支持的正确表。

以下示例显示了指定查找联接的语法。

|  |
| --- |
| -- Customers is backed by the JDBC connector and can be used for lookup joins CREATE TEMPORARY TABLE Customers (  id INT,  name STRING,  country STRING,  zip STRING ) WITH (  'connector' = 'jdbc',  'url' = 'jdbc:mysql://mysqlhost:3306/customerdb',  'table-name' = 'customers' );  -- enrich each order with customer information SELECT o.order\_id, o.total, c.country, c.zip FROM Orders AS o  JOIN Customers FOR SYSTEM\_TIME AS OF o.proc\_time AS c  ON o.customer\_id = c.id; |

在上面的示例中，Orders 表中包含来自位于 MySQL 数据库中的 Customers 表的数据。FOR SYSTEM\_TIME AS OF带有后续处理时间属性的子句可确保在连接运算符处理行时，将Orders表的每一行与与连接谓词匹配的客户行Orders连接起来。它还可以防止Customer在将来更新连接的行时更新连接结果。

在上面的示例中，查找连接还需要强制相等连接谓词o.customer\_id = c.id。

#### [时态表函数 Join](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/streaming/joins.html" \l "%E6%97%B6%E6%80%81%E8%A1%A8%E5%87%BD%E6%95%B0-join)

将表与表函数的结果连接起来。左（外部）表的每一行都与表函数的相应调用产生的所有行连接。[用户自定义表函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/udfs/" \l "table-functions)在使用前必须注册。

##### 内连接

如果表函数调用返回空结果，则删除左（外部）表的行。

|  |
| --- |
| SELECT order\_id, res FROM Orders, LATERAL TABLE(table\_func(order\_id)) t(res) |

##### 左外连接

如果表函数调用返回空结果，则保留相应的外部行，并用空值填充结果。

当前，针对横向表的左外连接需要 ON 子句中的 TRUE 文字。

|  |
| --- |
| SELECT order\_id, res FROM Orders LEFT OUTER JOIN LATERAL TABLE(table\_func(order\_id)) t(res)  ON TRUE |

## 窗口（Windows）

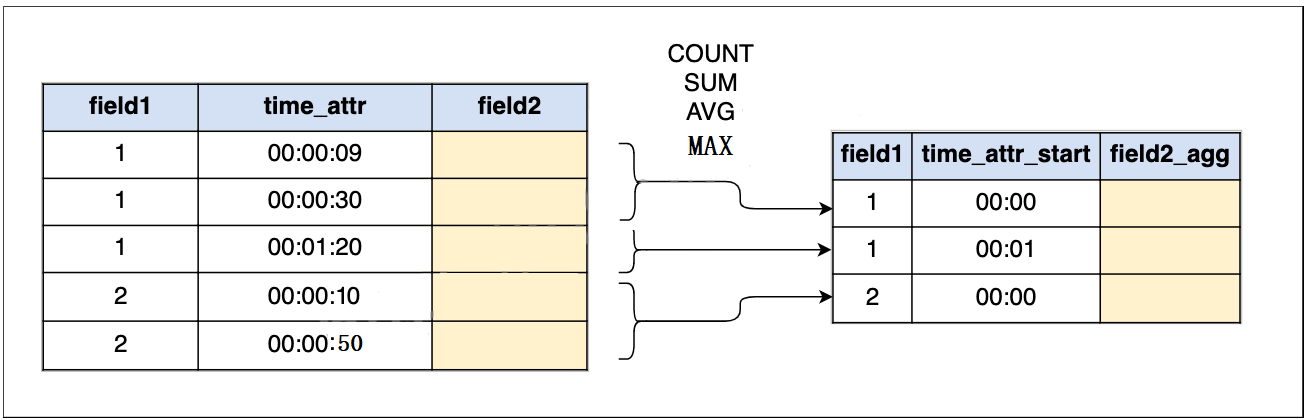
* 基于上述的时间语义，需要配合窗口操作才能发挥作用。最主要的用途就是开窗口、根据时间段做计算。
* 在Table API和SQL中，主要有两种窗口：**Group Windows**和**Over Windows**
  + Group Windows：根据时间和行计数间隔，将行聚合到有限的组（Group）中，并对每个组的数据执行一次聚合函数
  + Over Windows：针对每个输入行，计算相邻行范围内的聚合

下面我们就来看看Table API和SQL中，怎么利用时间字段做窗口操作。

### SQL中窗口的定义

#### Group Windows

GROUP BY是很多SQL用户经常使用的窗口聚合函数，**在流处理的一个时间窗口上进行GROUP BY与批处理中的非常相似**，常见的聚合操作有**COUNT、SUM、AVG、MAX**等。可见，时间窗口time\_attr\_window被作当做整个表的一个字段，用来做分组，下图展示了这个过程。



以下面例子为例：

|  |
| --- |
| **SELECT**  user\_id**,**  **COUNT(**behavior**)** **AS** behavior\_cnt**,**  TUMBLE\_END**(**proctime**,** **INTERVAL** '1' **MINUTE)** **AS** end\_ts  **FROM** user\_behavior  **GROUP** **BY** user\_id**,** TUMBLE**(**proctime**,** **INTERVAL** '1' **MINUTE)** |

这里对这个SQL语句进行分析解释。定义一个1分钟的滚动窗口，滚动窗口函数定义为：**TUMBLE(proctime, INTERVAL '1' MINUTE)**，窗口以**proctime**这个Processing Time为时间属性。这里我们使用了一个窗口分组函数，这是一个滚动窗口，如：**TUMBLE(time\_attr, interval)**，它将某个时间段内的数据都分到一组上。我们可以在**SELECT**中添加字段**TUMBLE\_START(proctime, INTERVAL '1' MINUTE)**查看窗口的的起始时间。

接下来我们将介绍几种常见的窗口分组函数。

##### 窗口表值函数（Windowing TVFs）

窗口函数是Flink中比较重要的函数，将流中的元素按照特定的逻辑生成一段时间范围，将“**流**”变成特殊的“**批**”处理。

现阶段常用的窗口：

* 滚动窗口 Tumble Windows
* 滑动窗口 Hop Windows
* 会话窗口 Session Windows**（预计Flink1.14版本支持）**
* 累计窗口 Cumulate Windows **(Flink v1.13 新增窗口）**

**Window TVF**替代了传统的[分组窗口功能](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/" \l "group-window-aggregation-deprecated)。窗口式TVF更符合SQL标准，并且功能更强大，可支持基于窗口的复杂计算，例如**Window TopN，Window Join**。但是，[分组窗口功能](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/" \l "group-window-aggregation)只能支持窗口聚合。

查看更多如何基于TVF窗口应用进一步的计算：

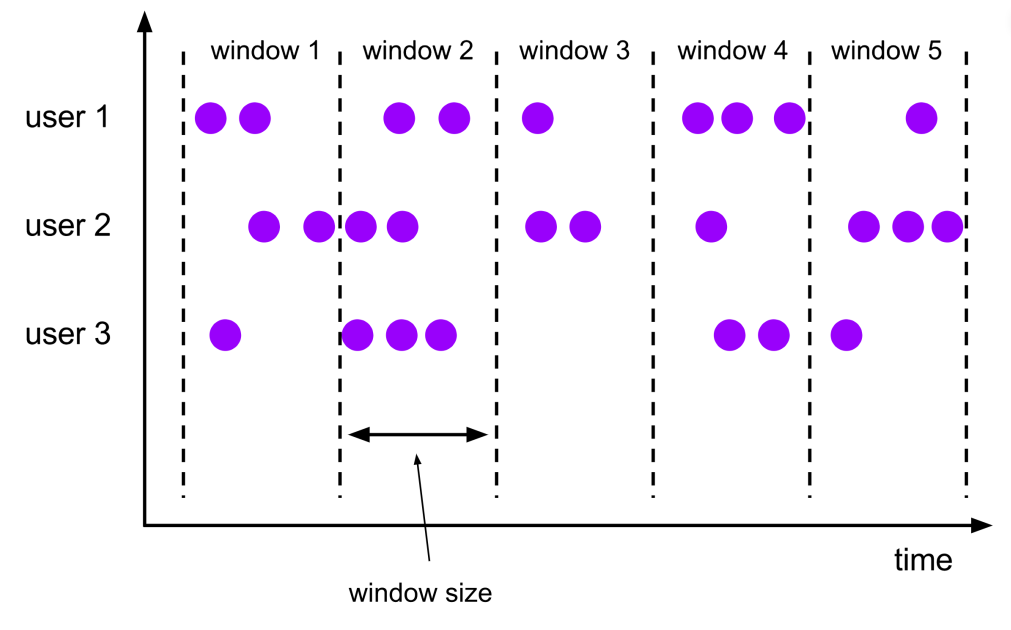
* 窗口聚合
* 窗口Top-N
* 窗口连接（**预计Flink1.14版本支持**）

##### **窗口分类函数及聚合操作**

###### 滚动窗口（Tumble Windows）

* 翻滚窗口能将数据流切分成不重叠的窗口，每一个事件只能属于一个窗口
* 翻滚窗具有固定的尺寸，不重叠。

例如，假设您指定大小为5分钟的翻滚窗口。在这种情况下，Flink将计算当前窗口，并且每五分钟启动一个新窗口，如下图所示。



TUMBLE 函数需要三个必需的参数：

|  |
| --- |
| TUMBLE(TABLE data, DESCRIPTOR(timecol), size) |

* data：一个表名。
* timecol：是一个列描述符，指示应将数据的哪个[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)列映射到翻转窗口。
* size：是指定滚动窗口宽度的持续时间。

使用示例

**需求：**

现在有一个实时数据看板，需要计算当前每10分钟GMV的总和

注意：**表必须具有时间属性，例如此表中的“bidtime”**

|  |  |  |
| --- | --- | --- |
| **bidtime** | **price** | **itme** |
| 2021-04-15 08:05:00 | 4.00 | C |
| 2021-04-15 08:07:00 | 2.00 | A |
| 2021-04-15 08:09:00 | 5.00 | D |
| 2021-04-15 08:11:00 | 3.00 | B |
| 2021-04-15 08:13:00 | 1.00 | E |
| 2021-04-15 08:17:00 | 6.00 | F |

**开窗语法：**

|  |
| --- |
| --注意：目前Flink不支持对单个窗口表值函数求值， --窗口表值函数应与聚合操作一起使用， --这个例子只是用来解释语法和表值函数产生的数据。 SELECT \* FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES));  --使用参数名称，`DATA` 必须在第一个 SELECT \* FROM TABLE(  TUMBLE(  DATA => TABLE Bid,  TIMECOL => DESCRIPTOR(bidtime),  SIZE => INTERVAL '10' MINUTES));  -- 在滚动窗口表上应用聚合 Flink SQL> SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; |

**数据分析：**

虚拟字段中获得了滚动窗口的开始时间、结束时间、开窗所属的分组时间（同一个窗口下的记录的虚拟字段结果相同）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **bidtime** | **price** | **itme** | **window\_start** | **window\_end** | **window\_time** |
| 2021-04-15 08:05:00 | 4.00 | C | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:07:00 | 2.00 | A | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:09:00 | 5.00 | D | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:11:00 | 3.00 | B | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |
| 2021-04-15 08:13:00 | 1.00 | E | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |
| 2021-04-15 08:17:00 | 6.00 | F | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |

**预期结果：**

|  |  |  |
| --- | --- | --- |
| **window\_start** | **window\_end** | **sum\_price** |
| 2021-04-15 08:00:00.000 | 2021-04-15 08:10:00.000 | 11.00 |
| 2021-04-15 08:10:00.000 | 2021-04-15 08:20:00.000 | 10.00 |

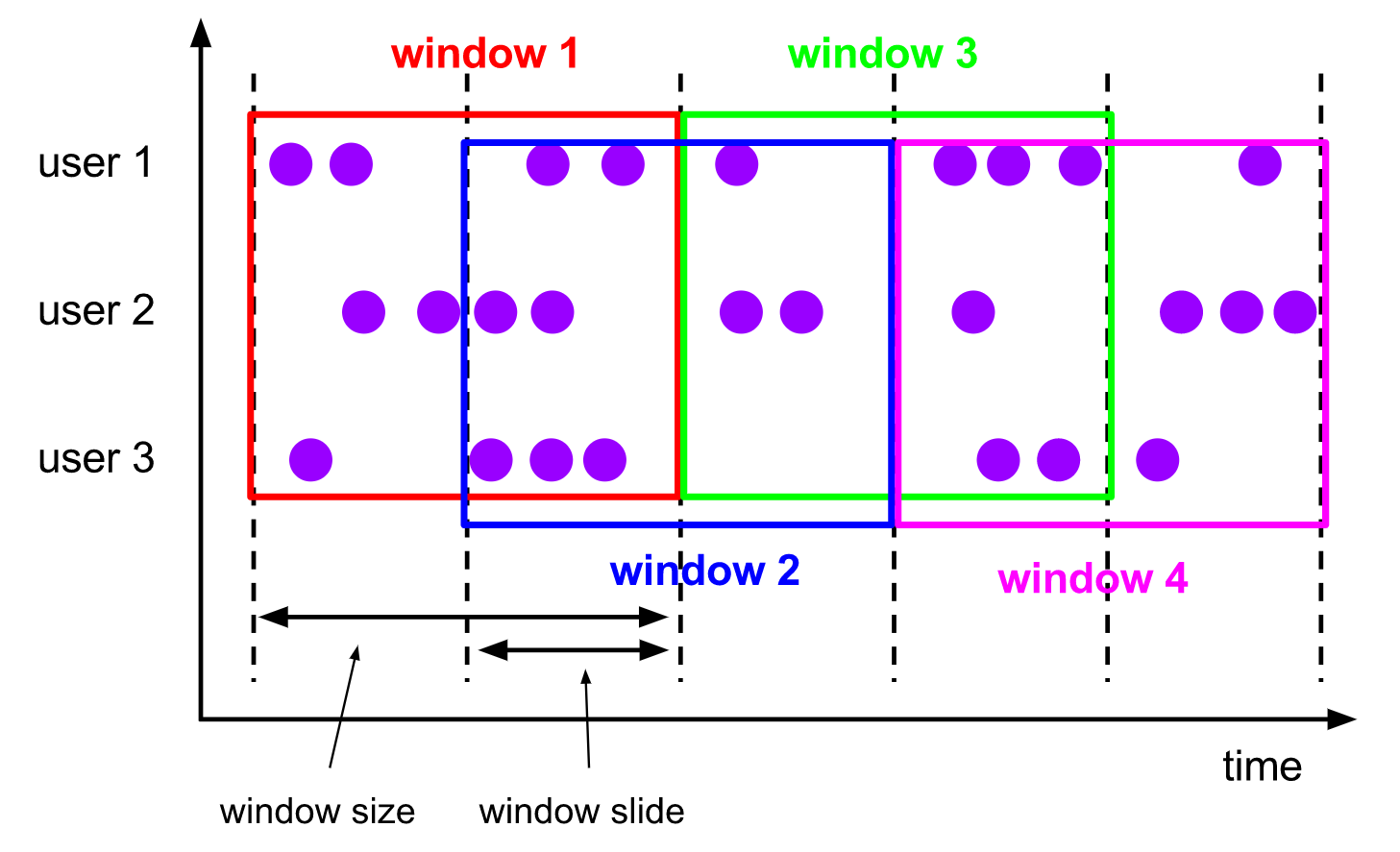
案例演示

|  |
| --- |
| public class GroupWindowsSqlTumbleExample {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);   String filePath = GroupWindowsSqlTumbleExample.class.getClassLoader().getResource("bid.csv").getPath();   // 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t  tEnv.executeSql("create table Bid(" +  "bidtime TIMESTAMP(3)," +  "price DECIMAL(10, 2), " +  "item string," +  "watermark for bidtime as bidtime - interval '1' second) " +  "with("  + "'connector' = 'filesystem',"  + "'path' = 'file:///"+filePath+"',"  + "'format' = 'csv'"  + ")");  tEnv  .sqlQuery(  "SELECT window\_start, window\_end, SUM(price) as sum\_price\n" +  " FROM TABLE(\n" +  " TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))\n" +  " GROUP BY window\_start, window\_end"  )  .execute()  .print();  } } |

###### 滑动窗口（Hop Windows）

* 滑动窗口和翻滚窗口类似，区别在于：滑动窗口可以有重叠的部分。
* 在滑动窗口中，一个元素可以对应多个窗口。

例如，您可以将大小为10分钟的窗口滑动5分钟。这样，您每隔5分钟就会得到一个窗口，其中包含最近10分钟内到达的事件，如下图所示。



HOP 接受三个必需的参数：

|  |
| --- |
| HOP(TABLE data, DESCRIPTOR(timecol), slide, size [, offset ]) |

* data：是一个表名。
* timecol：是一个列描述符，指示应将数据的哪个[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)列映射到滑动窗口。
* slide：是一个持续时间，指定了连续跳跃窗口开始之间的持续时间
* size：是指定跳变窗口宽度的持续时间

使用示例

**需求：**

每隔5min计算最近10min内的GMV的总和

**开窗语法：**

|  |
| --- |
| --注意：目前Flink不支持对单个窗口表值函数求值， --窗口表值函数应与聚合操作一起使用， --这个例子只是用来解释语法和表值函数产生的数据。 SELECT \* FROM TABLE(  HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES));  --使用参数名称，`DATA` 必须在第一个 SELECT \* FROM TABLE(  HOP(  DATA => TABLE Bid,  TIMECOL => DESCRIPTOR(bidtime),  SLIDE => INTERVAL '5' MINUTES,  SIZE => INTERVAL '10' MINUTES));  -- 在带窗口的表上应用聚合 SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; |

**数据分析：**

虚拟字段中获得了滚动窗口的开始时间、结束时间、开窗所属的分组时间（同一个窗口下的记录的虚拟字段结果相同）,为了方便开窗的阅读，按照windw\_time做了不同颜色的标记

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **bidtime** | **price** | **itme** | **window\_start** | **window\_end** | **window\_time** |
| 2021-04-15 08:05:00 | 4.00 | C | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:05:00 | 4.00 | C | 2021-04-15 08:05 | 2021-04-15 08:15 | 2021-04-15 08:14:59.999 |
| 2021-04-15 08:07:00 | 2.00 | A | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:07:00 | 2.00 | A | 2021-04-15 08:05 | 2021-04-15 08:15 | 2021-04-15 08:14:59.999 |
| 2021-04-15 08:09:00 | 5.00 | D | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:09:00 | 5.00 | D | 2021-04-15 08:05 | 2021-04-15 08:15 | 2021-04-15 08:14:59.999 |
| 2021-04-15 08:11:00 | 3.00 | B | 2021-04-15 08:05 | 2021-04-15 08:15 | 2021-04-15 08:14:59.999 |
| 2021-04-15 08:11:00 | 3.00 | B | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |
| 2021-04-15 08:13:00 | 1.00 | E | 2021-04-15 08:05 | 2021-04-15 08:15 | 2021-04-15 08:14:59.999 |
| 2021-04-15 08:13:00 | 1.00 | E | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |
| 2021-04-15 08:17:00 | 6.00 | F | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |
| 2021-04-16 01:17:00 | 6.00 | F | 2021-04-15 08:15 | 2021-04-15 08:25 | 2021-04-15 08:24:59.999 |

**预期结果：**

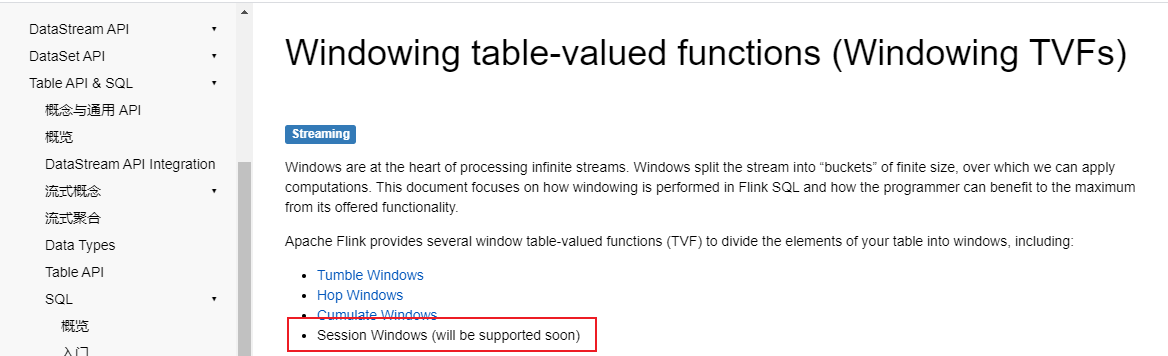
|  |  |  |
| --- | --- | --- |
| **window\_start** | **window\_end** | **sum\_price** |
| 2021-04-15 08:00:00.000 | 2021-04-15 08:10:00.000 | 11.00 |
| 2021-04-15 08:05:00.000 | 2021-04-15 08:15:00.000 | 15.00 |
| 2021-04-15 08:10:00.000 | 2021-04-15 08:20:00.000 | 10.00 |
| 2021-04-15 08:15:00.000 | 2021-04-15 08:25:00.000 | 6.00 |

案例演示

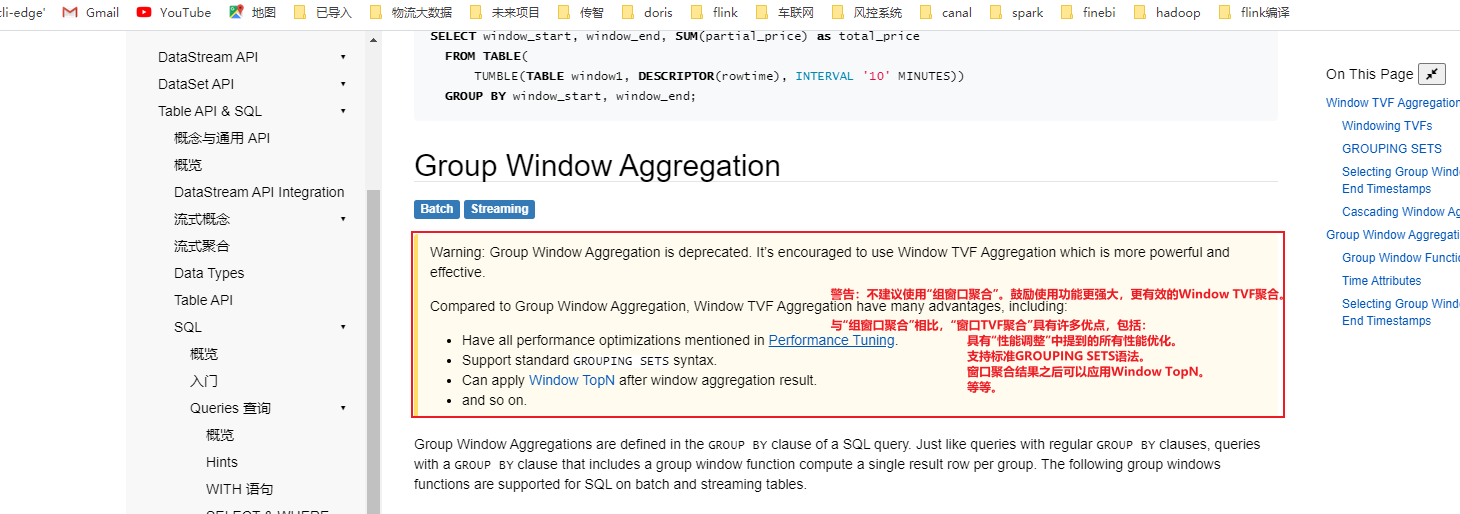
|  |
| --- |
| public class GroupWindowsSqlHopExample {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);   String filePath = GroupWindowsSqlHopExample.class.getClassLoader().getResource("bid.csv").getPath();   // 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t  tEnv.executeSql("create table Bid(" +  "bidtime TIMESTAMP(3)," +  "price DECIMAL(10, 2), " +  "item string," +  "watermark for bidtime as bidtime - interval '1' second) " +  "with("  + "'connector' = 'filesystem',"  + "'path' = 'file:///"+filePath+"',"  + "'format' = 'csv'"  + ")");  tEnv  .sqlQuery(  "SELECT window\_start, window\_end, SUM(price)\n" +  " FROM TABLE(\n" +  " HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES))\n" +  " GROUP BY window\_start, window\_end"  )  .execute()  .print();  } } |

###### 会话窗口（Session Windows，暂不支持Window TVF）

**Flink1.13版本中不支持Window TVF，预计在flink1.14版本中支持；**

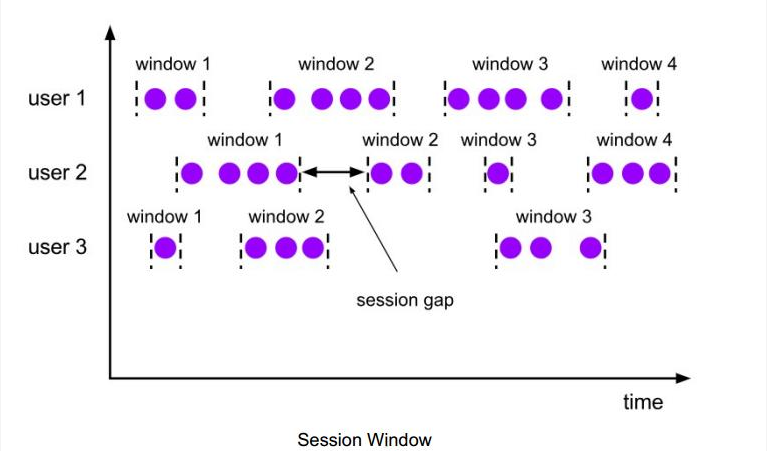


因此如果在flink1.13版本中实现会话窗口，**只能采用Group Window Aggregation**，除此之外官方建议尽可能使用Window TVF



分组窗口函数

SESSION(time\_attr, interval)：定义一个会话窗口，窗口长度是变长的，当两条数据之间的Session Gap超过了interval，这两条数据被分到两个窗口上。或者说，一个窗口等待超过interval后仍无数据进入，该窗口关闭。比如，我们定义Session Gap为3分钟，一个窗口最后一条数据之后的三分钟内没有新数据出现，则该窗口关闭，再之后的数据被归为下一个窗口。



窗口的起始结束时间

如果想查看窗口的起始结束时间，需要使用一个起始时间函数或结束时间函数。如下表所示，我们以会话窗口为例，列出常用的函数：

* SESSION\_START(time\_attr, gap-interval)：返回窗口的起始时间（**包含边界**）。例如[00:10,00:15)的窗口，返回00:10，即为此会话窗口内第一条记录的时间。
* SESSION\_END(time\_attr, gap-interval)：返回窗口的结束时间（**包含边界**）。例如[00:00,00:15)的窗口，返回 00:15，即为此会话窗口内最后一条记录的时间+<gap-interval>。
* SESSION\_ROWTIME(time\_attr, gap-interval)：返回窗口的结束时间（**不包含边界**）。例如[00:00,00:15)的窗口，返回00:14:59.999 。返回值是一个rowtime attribute，也就是可以基于该字段进行时间类型的操作，如[级联窗口](https://helpcdn.aliyun.com/document_detail/62510.html" \l "section-cwf-1kt-jhb" \o ")。该参数只能用于基于Event Time的Window。
* SESSION\_PROCTIME(time\_attr, gap-interval)：返回窗口的结束时间（**不包含边界**）。例如[00:00,00:15)的窗口，返回 00:14:59.999 。返回值是一个Proctime Attribute，也就是可以基于该字段进行时间类型的操作，如[级联窗口](https://helpcdn.aliyun.com/document_detail/62510.html" \l "section-cwf-1kt-jhb" \o ")。该参数只能用于基于Processing Time的Window。

案例演示

**需求：**

定义Session Gap为3分钟，一个窗口最后一条数据之后的三分钟内没有新数据出现，则该窗口关闭，再之后的数据被归为下一个窗口

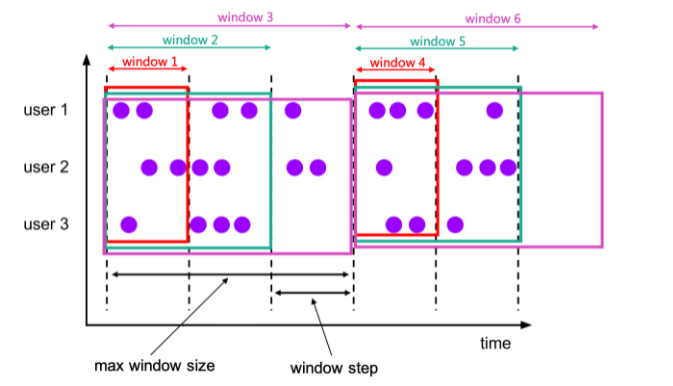
**代码实现：**

|  |
| --- |
| public class GroupWindowsSqlSessionExample {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);   String filePath = GroupWindowsSqlSessionExample.class.getClassLoader().getResource("bid.csv").getPath();   // 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t  tEnv.executeSql("create table Bid(" +  "bidtime TIMESTAMP(3)," +  "price DECIMAL(10, 2), " +  "item string," +  "watermark for bidtime as bidtime - interval '1' second) " +  "with("  + "'connector' = 'filesystem',"  + "'path' = 'file:///"+filePath+"',"  + "'format' = 'csv'"  + ")");   tEnv  .sqlQuery(  "SELECT " +  " SESSION\_START(bidtime, INTERVAL '3' minute) as wStart, " +  " SESSION\_END(bidtime, INTERVAL '3' minute) as wEnd, " +  " SUM(price) sum\_price " +  "FROM Bid " +  "GROUP BY SESSION(bidtime, INTERVAL '3' minute)"  )  .execute()  .print();  } } |

###### 累加窗口（CUMULATE Windows， Flink1.13新特性）

在某些场景下，累积窗口非常有用，例如以固定的窗口间隔以提前触发的方式滚动窗口。例如，每天的可视化图表绘制从00:00到每分钟的累积UV，而10:00处的UV代表从00:00到10:00的UV总数。这可以通过CUMULATE窗口轻松而有效地实现。

例如，你可以有1点小时的步骤1天最大尺寸的累积窗口，你会得到窗口：[00:00, 01:00)，[00:00, 02:00)，[00:00, 03:00)，...，[00:00, 24:00)为每一天。



CUMULATE 接受三个必需的参数：

|  |
| --- |
| CUMULATE(TABLE data, DESCRIPTOR(timecol), step, size) |

* TABLE 表名称
* DESCRIPTOR 表中作为开窗的时间字段名称
* step 大窗口的分割长度
* size 指定最大的那个时间窗口

使用示例

**需求：**

每10分钟，隔2分钟的累计当前的汇总结果

**开窗语法：**

|  |
| --- |
| --注意：目前Flink不支持对单个窗口表值函数求值， --窗口表值函数应与聚合操作一起使用， --这个例子只是用来解释语法和表值函数产生的数据。 SELECT \* FROM TABLE(  CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES)); --使用参数名称，`DATA` 必须在第一个 SELECT \* FROM TABLE(  CUMULATE(  DATA => TABLE Bid,  TIMECOL => DESCRIPTOR(bidtime),  STEP => INTERVAL '2' MINUTES,  SIZE => INTERVAL '10' MINUTES));  --在累积窗口表上应用聚合  SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; |

**数据分析：**

虚拟字段中获得了滚动窗口的开始时间、结束时间、开窗所属的分组时间（同一个窗口下的记录的虚拟字段结果相同）,为了方便开窗的阅读，按照windw\_time做了不同颜色的标记

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **bidtime** | **price** | **itme** | **window\_start** | **window\_end** | **window\_time** |
| 2021-04-15 08:05:00 | 4.00 | C | 2021-04-15 08:00 | 2021-04-15 08:06 | 2021-04-15 08:05:59.999 |
| 2021-04-15 08:05:00 | 4.00 | C | 2021-04-15 08:00 | 2021-04-15 08:08 | 2021-04-15 08:07:59.999 |
| 2021-04-15 08:05:00 | 4.00 | C | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:07:00 | 2.00 | A | 2021-04-15 08:00 | 2021-04-15 08:08 | 2021-04-15 08:07:59.999 |
| 2021-04-15 08:07:00 | 2.00 | A | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:09:00 | 5.00 | D | 2021-04-15 08:00 | 2021-04-15 08:10 | 2021-04-15 08:09:59.999 |
| 2021-04-15 08:11:00 | 3.00 | B | 2021-04-15 08:10 | 2021-04-15 08:12 | 2021-04-15 08:11:59.999 |
| 2021-04-15 08:11:00 | 3.00 | B | 2021-04-15 08:10 | 2021-04-15 08:14 | 2021-04-15 08:13:59.999 |
| 2021-04-15 08:11:00 | 3.00 | B | 2021-04-15 08:10 | 2021-04-15 08:16 | 2021-04-15 08:15:59.999 |
| 2021-04-15 08:11:00 | 3.00 | B | 2021-04-15 08:10 | 2021-04-15 08:18 | 2021-04-15 08:17:59.999 |
| 2021-04-15 08:11:00 | 3.00 | B | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |
| 2021-04-15 08:13:00 | 1.00 | E | 2021-04-15 08:10 | 2021-04-15 08:14 | 2021-04-15 08:13:59.999 |
| 2021-04-15 08:13:00 | 1.00 | E | 2021-04-15 08:10 | 2021-04-15 08:16 | 2021-04-15 08:15:59.999 |
| 2021-04-15 08:13:00 | 1.00 | E | 2021-04-15 08:10 | 2021-04-15 08:18 | 2021-04-15 08:17:59.999 |
| 2021-04-15 08:13:00 | 1.00 | E | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |
| 2021-04-15 08:17:00 | 6.00 | F | 2021-04-15 08:10 | 2021-04-15 08:18 | 2021-04-15 08:17:59.999 |
| 2021-04-16 01:17:00 | 6.00 | F | 2021-04-15 08:10 | 2021-04-15 08:20 | 2021-04-15 08:19:59.999 |

**预期结果：**

|  |  |  |
| --- | --- | --- |
| **window\_start** | **window\_end** | **sum\_price** |
| 2021-04-15 08:00:00.000 | 2021-04-15 08:06:00.000 | 4.00 |
| 2021-04-15 08:00:00.000 | 2021-04-15 08:08:00.000 | 6.00 |
| 2021-04-15 08:00:00.000 | 2021-04-15 08:10:00.000 | 11.00 |
| 2021-04-15 08:10:00.000 | 2021-04-15 08:12:00.000 | 3.00 |
| 2021-04-15 08:10:00.000 | 2021-04-15 08:14:00.000 | 4.00 |
| 2021-04-15 08:10:00.000 | 2021-04-15 08:16:00.000 | 4.00 |
| 2021-04-15 08:10:00.000 | 2021-04-15 08:18:00.000 | 10.00 |
| 2021-04-15 08:10:00.000 | 2021-04-15 08:20:00.000 | 10.00 |

案例演示

|  |
| --- |
| public class GroupWindowsSqlCumulateExample {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);   String filePath = GroupWindowsSqlCumulateExample.class.getClassLoader().getResource("bid.csv").getPath();   // 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t  tEnv.executeSql("create table Bid(" +  "bidtime TIMESTAMP(3)," +  "price DECIMAL(10, 2), " +  "item string," +  "watermark for bidtime as bidtime - interval '1' second) " +  "with("  + "'connector' = 'filesystem',"  + "'path' = 'file:///"+filePath+"',"  + "'format' = 'csv'"  + ")");  tEnv  .sqlQuery(  "SELECT window\_start, window\_end, SUM(price) as sum\_price\n" +  " FROM TABLE(\n" +  " CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES))\n" +  " GROUP BY window\_start, window\_end"  )  .execute()  .print();  } } |

##### **多维数据分析**

对于经常需要对数据进行多维度的聚合分析的场景，您既需要对a列做聚合，也要对b列做聚合，同时要按照a、b两列同时做聚合，因此需要多次使用UNION ALL。使用**GROUPING SETS**可以快速解决此类问题。

MaxCompute中的GROUPING SETS是对SELECT语句中GROUP BY子句的扩展，允许您采用多种方式对结果分组，而不必使用多个SELECT语句来实现这一目的。这样能够使MaxCompute的引擎给出更有效的执行计划，从而提高执行性能。

###### GROUPING SETS

窗口聚合还支持GROUPING SETS语法。分组集允许比标准描述的分组操作更为复杂的分组操作GROUP BY。按每个指定的分组集将行分别分组，并像简单GROUP BY子句一样为每个组计算聚合。

具有的窗口聚合GROUPING SETS要求window\_start和window\_end都必须在GROUP BY子句中，而不是在GROUPING SETS子句中。

**注意：**GROUPING SETS语法也可以用于非窗口函数的场景

使用示例

**准备数据**

|  |  |  |  |
| --- | --- | --- | --- |
| **userid** | **timestamp** | **price** | **category** |
| user\_001 | 1621718199 | 10.1 | 电脑 |
| user\_001 | 1621718201 | 14.1 | 手机 |
| user\_002 | 1621718202 | 82.5 | 手机 |
| user\_001 | 1621718205 | 15.6 | 电脑 |

**对数据进行分组。您可以通过以下两种方式进行分组：**

* 使用多个SELECT语句进行分组

|  |
| --- |
| SELECT window\_start, window\_end, 'NULL' as userId, 'NULL' as category, sum(price) as sum\_price FROM TABLE( TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS)) GROUP BY window\_start, window\_end UNION ALL SELECT window\_start, window\_end, userId as userId, 'NULL' as category, sum(price) as sum\_price FROM TABLE( TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS)) GROUP BY window\_start, window\_end, userId UNION ALL SELECT window\_start, window\_end,userId, category, sum(price) as sum\_price FROM TABLE( TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS)) GROUP BY window\_start, window\_end, userId, category |

* 使用GROUPING SETS进行分组

|  |
| --- |
| SELECT window\_start, window\_end, userId, category, sum(price) as sum\_price FROM TABLE(  TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS))  GROUP BY window\_start, window\_end, GROUPING SETS((userId, category), (userId), ()) |

每个子列表GROUPING SETS可以指定零个或多个列或表达式，并且其解释方式与直接在GROUP BY子句中使用的方式相同。空分组集意味着所有行都被汇总为一个组，即使没有输入行也将被输出。

在没有出现这些列的分组集的结果行中，对分组列或表达式的引用将替换为空值。

**预期结果：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **window\_start** | **window\_end** | **userId** | **category** | **sum\_price** |
| 2021-05-23 05:16:35.000 | 2021-05-23 05:16:40.000 | NULL | NULL | 10.1 |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | NULL | NULL | 96.6 |
| 2021-05-23 05:16:45.000 | 2021-05-23 05:16:50.000 | NULL | NULL | 15.6 |
| 2021-05-23 05:16:35.000 | 2021-05-23 05:16:40.000 | user\_001 | 电脑 | 10.1 |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | user\_001 | 手机 | 14.1 |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | user\_002 | 手机 | 82.5 |
| 2021-05-23 05:16:45.000 | 2021-05-23 05:16:50.000 | user\_001 | 电脑 | 15.6 |
| 2021-05-23 05:16:35.000 | 2021-05-23 05:16:40.000 | user\_001 | NULL | 10.1 |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | user\_001 | NULL | 14.1 |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | user\_002 | NULL | 82.5 |
| 2021-05-23 05:16:45.000 | 2021-05-23 05:16:50.000 | user\_001 | NULL | 15.6 |

案例演示

|  |
| --- |
| public class GroupingSetsWindowSqlExample {  public static void main(String[] args) throws Exception {  //构建运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1); // 设置并行度为1方便后面进行测试  //构建表运行环境  EnvironmentSettings settings = EnvironmentSettings.newInstance()  .useBlinkPlanner()  .inStreamingMode().build();  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);   String filePath = TableApiTest.class.getClassLoader().getResource("order.csv").getPath();  tEnv.executeSql("create table orders(" +  "userId string," +  "ts bigint," +  "price float, " +  "category varchar," +  "t as to\_timestamp(from\_unixtime(ts,'yyyy-MM-dd HH:mm:ss'))," +  "watermark for t as t - interval '5' second)" +  "with("  + "'connector' = 'filesystem',"  + "'path' = 'file:///"+filePath+"',"  + "'format' = 'csv'"  + ")");   tEnv  .sqlQuery("SELECT \* FROM orders"  )  .execute()  .print();   /\*\*  \* flinksql +I -U +U -D解释  \* +是操作后，I是插入，U是更新，D是删除  \* 如 -U是撤回前的数据，+U是更新后的数据  \*/  //原来查询方式  tEnv  .sqlQuery("SELECT window\_start, window\_end, 'NULL' as userId, 'NULL' as category, sum(price) as sum\_price\n" +  " FROM TABLE(\n" +  " TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS))\n" +  " GROUP BY window\_start, window\_end\n" +  "UNION ALL\n" +  "SELECT window\_start, window\_end, userId as userId, 'NULL' as category, sum(price) as sum\_price\n" +  " FROM TABLE(\n" +  " TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS))\n" +  " GROUP BY window\_start, window\_end, userId\n" +  "UNION ALL\n" +  "SELECT window\_start, window\_end,userId, category, sum(price) as sum\_price\n" +  " FROM TABLE(\n" +  " TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS))\n" +  " GROUP BY window\_start, window\_end, userId, category"  )  .execute()  .print();   tEnv  .sqlQuery(  "SELECT window\_start, window\_end, userId, category, sum(price) as sum\_price " +  " FROM TABLE(" +  " TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS)) " +  " GROUP BY window\_start, window\_end, GROUPING SETS((userId, category), (userId), ()) "  )  .execute()  .print();  } } |

###### ROLLUP

CUBE和ROLLUP可以认为是特殊的GROUPING SETS。

CUBE会枚举指定列的所有可能组合作为GROUPING SETS，而ROLLUP会以按层级聚合的方式产生GROUPING SETS。

ROLLUP是用于指定通用分组集类型的简写形式。它表示给定的表达式列表以及列表的所有前缀，包括空列表。

具有的窗口聚合ROLLUP要求window\_start和window\_end列都必须在GROUP BY子句中，而不是在ROLLUP子句中。

使用示例

|  |
| --- |
| GROUP BY ROLLUP(a, b, c) --等价于以下语句。 GROUPING SETS((a,b,c),(a,b),(a), ())  GROUP BY ROLLUP ( a, (b, c), d ) --等价于以下语句。 GROUPING SETS (  ( a, b, c, d ),  ( a, b, c ),  ( a ),  ( ) )  GROUP BY grouping sets((b), (c),rollup(a,b,c)) --等价于以下语句。 GROUP BY GROUPING SETS (  (b), (c),  (a,b,c), (a,b), (a), ()  ) |

案例演示

|  |
| --- |
| tEnv  .sqlQuery(  "SELECT window\_start, window\_end, userId, category, sum(price) as sum\_price " +  " FROM TABLE(" +  " TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS)) " +  " GROUP BY window\_start, window\_end, ROLLUP(userId, category)"  )  .execute()  .print(); |

###### CUBE

使用示例

|  |
| --- |
| GROUP BY CUBE(a, b, c) --等价于以下语句。 GROUPING SETS((a,b,c),(a,b),(a,c),(b,c),(a),(b),(c),())  GROUP BY CUBE ( (a, b), (c, d) ) --等价于以下语句。 GROUPING SETS (  ( a, b, c, d ),  ( a, b ),  ( c, d ),  ( ) )  GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e)) --等价于以下语句。 GROUP BY GROUPING SETS (  (a, b, c, d), (a, b, c, e),  (a, b, d), (a, b, e),  (a, c, d), (a, c, e),  (a, d), (a, e) ) |

案例演示

|  |
| --- |
| tEnv  .sqlQuery(  "SELECT window\_start, window\_end, userId, category, sum(price) as sum\_price " +  " FROM TABLE(" +  " TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS)) " +  " GROUP BY window\_start, window\_end, CUBE(userId, category)"  )  .execute()  .print(); |

###### GROUPING和GROUPING\_ID函数

GROUPING SETS结果中使用NULL充当占位符，导致您会无法区分占位符NULL与数据中真正的NULL。因此，MaxCompute为您提供了GROUPING函数。GROUPING函数接受一个列名作为参数，如果结果对应行使用了参数列做聚合，返回0，此时意味着NULL来自输入数据。否则返回1，此时意味着NULL是GROUPING SETS的占位符。

MaxCompute还提供了GROUPING\_ID函数，此函数接受一个或多个列名作为参数。结果是将参数列的GROUPING结果按照BitMap的方式组成整数。

使用示例

|  |
| --- |
| SELECT a,b,c ,COUNT(\*), GROUPING(a) ga,  GROUPING(b) gb,  GROUPING(c) gc,  GROUPING\_ID(a,b,c) groupingid FROM VALUES (1,2,3) as t(a,b,c) GROUP BY CUBE(a,b,c) |

执行结果如下：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | **b** | **c** | **\_c3** | **ga** | **gb** | **gc** | **groupingid** |
| NULL | NULL | NULL | 1 | 1 | 1 | 1 | 7 |
| NULL | NULL | 3 | 1 | 1 | 1 | 0 | 6 |
| NULL | 2 | NULL | 1 | 1 | 0 | 1 | 5 |
| NULL | 2 | 3 | 1 | 1 | 0 | 0 | 4 |
| 1 | NULL | NULL | 1 | 0 | 1 | 1 | 3 |
| 1 | NULL | 3 | 1 | 0 | 1 | 0 | 2 |
| 1 | 2 | NULL | 1 | 0 | 0 | 1 | 1 |
| 1 | 2 | 3 | 1 | 0 | 0 | 0 | 0 |

默认情况，GROUP BY列表中不被使用的列，会被填充为NULL。您可以通过GROUPING函数输出更有实际意义的值。

|  |
| --- |
| SELECT window\_start, window\_end, userId, category,  GROUPING(category) as categoryFlag, sum(price) as sum\_price,  IF(GROUPING(category) = 0, category, 'ALL') as `all` FROM TABLE(  TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS))  GROUP BY window\_start, window\_end, GROUPING SETS((userId, category), (userId)) |

返回结果：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **window\_start** | **window\_end** | **userId** | **category** | **sum\_price** | flag | all |
| 2021-05-23 05:16:35.000 | 2021-05-23 05:16:40.000 | NULL | NULL | 10.1 | 1 | ALL |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | NULL | NULL | 96.6 | 1 | ALL |
| 2021-05-23 05:16:45.000 | 2021-05-23 05:16:50.000 | NULL | NULL | 15.6 | 1 | ALL |
| 2021-05-23 05:16:35.000 | 2021-05-23 05:16:40.000 | user\_001 | 电脑 | 10.1 | 0 | 电脑 |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | user\_001 | 手机 | 14.1 | 0 | 手机 |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | user\_002 | 手机 | 82.5 | 0 | 手机 |
| 2021-05-23 05:16:45.000 | 2021-05-23 05:16:50.000 | user\_001 | 电脑 | 15.6 | 0 | 电脑 |
| 2021-05-23 05:16:35.000 | 2021-05-23 05:16:40.000 | user\_001 | NULL | 10.1 | 1 | ALL |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | user\_001 | NULL | 14.1 | 1 | ALL |
| 2021-05-23 05:16:40.000 | 2021-05-23 05:16:45.000 | user\_002 | NULL | 82.5 | 1 | ALL |
| 2021-05-23 05:16:45.000 | 2021-05-23 05:16:50.000 | user\_001 | NULL | 15.6 | 1 | ALL |

MaxCompute还提供了无参数的GROUPING\_\_ID函数，用于兼容Hive查询。

|  |
| --- |
| SELECT a,b,c ,  COUNT(\*),  GROUPING\_ID FROM VALUES (1,2,3) as t(a,b,c) GROUP BY a, b, c GROUPING SETS ((a,b,c), (a)); |

GROUPING\_ID既无输入参数，也无括号。此表达方式在MaxCompute中等价于GROUPING\_ID(a,b,c)，参数与GROUP BY的顺序一致。

**MaxCompute和Hive 2.3.0及以上版本兼容该函数，在Hive 2.3.0以下版本中该函数输出不一致，因此并不推荐您使用此函数。**

案例演示

|  |
| --- |
| tEnv  .sqlQuery(  "SELECT window\_start, window\_end, userId, category, " +  " GROUPING(category) as categoryFlag," +  " IF(GROUPING(category) = 0, category, 'ALL') as `all`, " +  " GROUPING\_ID(userId,category) groupingId," +  " sum(price) as sum\_price " +  " FROM TABLE(" +  " TUMBLE(TABLE orders, DESCRIPTOR(t), INTERVAL '5' SECONDS)) " +  " GROUP BY window\_start, window\_end, GROUPING SETS((userId, category), (userId)) "  )  .execute()  .print(); |

###### Window Top-N

Window Top-N是一个特殊的[Top-N](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/topn/)，它返回每个窗口和其他分区键的N个最小值或最大值。

对于流查询，与连续表上的常规Top-N不同，Top-N窗口不会发出中间结果，而只会发出最终结果，总N条记录位于窗口末尾。而且，不再需要窗口Top-N清除所有中间状态。因此，如果用户不需要按记录更新结果，则窗口前N个查询的性能会更好。通常，Window Top-N与[Window Aggregation](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/)一起使用。

可以使用与常规Top-N相同的语法定义Window Top-N，有关更多信息，请参见[Top-N文档](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/topn/)。除此之外，Window Top-N要求PARTITION BY子句包含window\_start和window\_end应用[Windowing TVF](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-tvf/)或[Window Aggregation](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/)关系的列。否则，优化器将无法翻译查询。

下面显示了Window Top-N语句的语法：

|  |
| --- |
| SELECT [column\_list] FROM (  SELECT [column\_list],  ROW\_NUMBER() OVER (PARTITION BY window\_start, window\_end [, col\_key1...]  ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum  FROM table\_name) -- relation applied windowing TVF WHERE rownum <= N [AND conditions] |

使用示例

以下示例显示了如何计算每10分钟营业时间窗内销售额最高的前3名供应商。

|  |
| --- |
| SELECT *\** FROM (  SELECT *\**, ROW\_NUMBER() OVER (PARTITION BY window\_start, window\_end ORDER BY price DESC) as rownum  FROM (  SELECT window\_start, window\_end, supplier\_id, *SUM*(price) as price, *COUNT*(*\**) as cnt  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end, supplier\_id  )  ) WHERE rownum <= 3; |

返回结果：

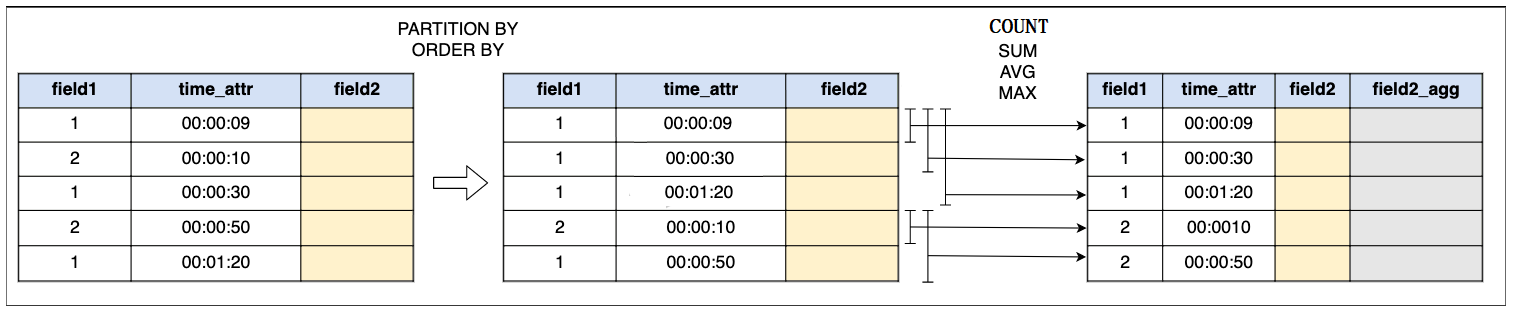
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **window\_start** | **window\_end** | **supplier\_id** | **price** | **cnt** | rownum |
| 2021-05-15 08:00:00.000 | 2021-05-15 08:10:00.000 | supplier1 | 6.00 | 10 | 1 |
| 2021-05-15 08:00:00.000 | 2021-05-15 08:10:00.000 | supplier4 | 5.00 | 96 | 2 |
| 2021-05-15 08:00:00.000 | 2021-05-15 08:10:00.000 | supplier2 | 4.00 | 15 | 3 |
| 2021-05-15 08:10:00.000 | 2021-05-15 08:20:00.000 | supplier5 | 6.00 | 10 | 1 |
| 2021-05-15 08:10:00.000 | 2021-05-15 08:20:00.000 | supplier2 | 3.00 | 14 | 2 |
| 2021-05-15 08:10:00.000 | 2021-05-15 08:20:00.000 | supplier3 | 2.00 | 82 | 3 |

案例演示

|  |
| --- |
| public class GroupWindowsSqlTopNExample {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);   String filePath = GroupWindowsSqlCumulateExample.class.getClassLoader().getResource("bid.csv").getPath();   // 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t  tEnv.executeSql("create table Bid(" +  "bidtime TIMESTAMP(3)," +  "price DECIMAL(10, 2), " +  "item string," +  "supplier\_id string," +  "watermark for bidtime as bidtime - interval '1' second) " +  "with("  + "'connector' = 'filesystem',"  + "'path' = 'file:///"+filePath+"',"  + "'format' = 'csv'"  + ")");   tEnv  .sqlQuery(  "SELECT \*\n" +  " FROM (\n" +  " SELECT \*, ROW\_NUMBER() OVER (PARTITION BY window\_start, window\_end ORDER BY price DESC) as rownum\n" +  " FROM (\n" +  " SELECT window\_start, window\_end, supplier\_id, SUM(price) as price, COUNT(\*) as cnt\n" +  " FROM TABLE(\n" +  " TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))\n" +  " GROUP BY window\_start, window\_end,supplier\_id\n" +  " )\n" +  " ) WHERE rownum <= 3"  )  .execute()  .print();  } } |

#### Over Windows

传统SQL中专门进行窗口处理的函数为OVER WINDOW。OVER WINDOW与GROUP BY有些不同，它对每一行数据都生成窗口，在窗口上进行聚合，聚合的结果会生成一个新字段。或者说，OVER WINDOW一般是一行变一行。



上图展示了OVER WINDOW的工作示意图，窗口确定的方式为：

* 先对field1做分组，相同field1的数据被分到一起，按照时间属性排序，即上图中的PARTITION BY和ORDER BY部分；
* 然后每行数据都建立一个窗口，窗口起始点是field1分组的第一行数据，结束点是当前行；
* 窗口划分好后，再对窗口内的field2字段做各类聚合操作，生成field2\_agg的新字段，常见的聚合操作有COUNT、SUM、AVG或MAX等。

从图中可以看出，每一行都有一个窗口，当前行是这个窗口的最后一行，窗口的聚合结果生成一个新的字段。具体的实现逻辑上，Flink为每一个元素维护一个窗口，为每一个元素执行一次窗口计算，完成计算后会清除过期数据。

Flink SQL中对OVER WINDOW的定义遵循了标准的SQL语法，我们先来看一下OVER WINDOW的语法结构：

|  |
| --- |
| **SELECT**  AGG\_FUNCTION**(**field2**)** OVER **(**windowDefinition2**)** **AS** field2\_agg**,**  **...**  AGG\_FUNCTION**(**fieldN**)** OVER **(**windowDefinitionN**)** **AS** fieldN\_agg  **FROM** tab1 |

其中，windowDefinition2是定义窗口的规则，包括根据哪些字段进行PARTITION BY等，在定义好的窗口上，我们使用AGG\_FUNCTION(field2)对field2字段进行聚合计算。或者我们可以使用别名来定义窗口WINDOW w AS ...：

|  |
| --- |
| **SELECT**  AGG\_FUNCTION**(**field2**)** OVER w **AS** field2\_agg**,**  **...**  **FROM** tab1  WINDOW w **AS** **(**windowDefinition**)** |

那么具体应该如何划分窗口，如何写windowDefinition呢？上图中只演示了一种窗口划分的方式，常用的窗口划分方式可以基于行，也可以基于时间段，接下来我们通过一些例子来展示窗口的划分。

##### ROWS OVER WINDOW

###### 使用示例

我们首先演示基于行来划分窗口：

|  |
| --- |
| select  itemID,  itemType,  onSellTime,  price,  sum(price) over w as sumPrice from tmall\_item  WINDOW w AS (  PARTITION BY itemType   ORDER BY onSellTime   ROWS BETWEEN UNBOUNDED preceding AND CURRENT ROW  ) |

上面的SQL语句中，WINDOW w AS (...)定义了一个名为w的窗口，它根据itemType来分组，并按照onSellTime来排序。原始数据并不是基于itemType来分组的，**PARTITION BY itemType**起到了分组的作用，相同**itemType**的数据被分到了一组，组内按照时间戳**onSellTime**来排序。这里完成了上图中最左侧表到中间表的转化。

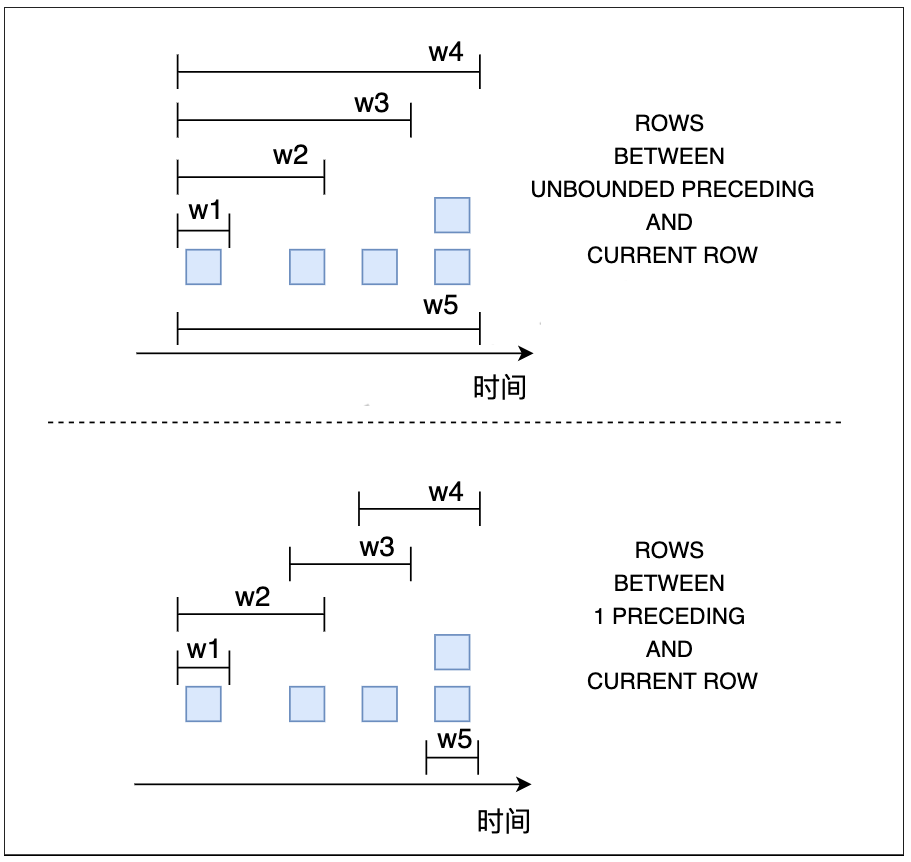
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW定义了窗口的起始和结束：

**窗口的起始点为UNBOUNDED PRECEDING，这两个SQL关键词组合在一起表示窗口起始点是数据流的最开始的行**

**CURRENT ROW表示结束点是当前行（ROWS BETWEEN ... AND ...这样的语句定义了窗口的起始和结束）**

结合分组和排序策略，这就意味着，这个窗口从数据流的第一行开始到当前行结束，按照itemType分组，按照onSellTime排序。

**注意：**目前OVER WINDOW上，Flink只支持基于时间属性的ORDER BY排序，无法基于其他字段进行排序。



上图展示了按行划分窗口的基本原理，图中上半部分使用UNBOUNDED PRECEDING表示起始位置，那么窗口是从数据流的第一个元素开始一直到当前元素；下半部分使用1 PRECEDING表示起始位置，窗口的起始点是本元素的前一个元素，我们可以把1换成其他我们想要的数字。

**图中最后两行数据从时间上虽然同时到达，但由于窗口是按行划分的，这两行数据被划分为两个窗口，这与后文提到的按时间段划分有所区别。**

如果输入数据流如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| **itemID** | **itemType** | **onSellTime** | **price** |
| ITEM001 | Electronic | 2021-05-11 10:01:00.000 | 20 |
| ITEM002 | Electronic | 2021-05-11 10:02:00.000 | 50 |
| ITEM003 | Electronic | 2021-05-11 10:03:00.000 | 30 |
| ITEM004 | Electronic | 2021-05-11 10:03:00.000 | 60 |
| ITEM005 | Electronic | 2021-05-11 10:05:00.000 | 40 |
| ITEM006 | Electronic | 2021-05-11 10:06:00.000 | 20 |
| ITEM007 | Electronic | 2021-05-11 10:07:00.000 | 70 |
| ITEM008 | Clothes | 2021-05-11 10:08:00.000 | 20 |
| ITEM009 | Clothes | 2021-05-11 10:09:00.000 | 40 |
| ITEM010 | Clothes | 2021-05-11 10:11:00.000 | 30 |

那么对于之前的SQL语句，一个查询的结果将产生下面的数据：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **itemID** | **itemType** | **onSellTime** | **price** | **sumPrice** |
| ITEM001 | Electronic | 2021-05-11 10:01:00.000 | 20.0 | 20.0 |
| ITEM002 | Electronic | 2021-05-11 10:02:00.000 | 50.0 | 70.0 |
| ITEM003 | Electronic | 2021-05-11 10:03:00.000 | 30.0 | 100.0 |
| ITEM004 | Electronic | 2021-05-11 10:03:00.000 | 60.0 | 160.0 |
| ITEM005 | Electronic | 2021-05-11 10:05:00.000 | 40.0 | 200.0 |
| ITEM006 | Electronic | 2021-05-11 10:06:00.000 | 20.0 | 220.0 |
| ITEM007 | Electronic | 2021-05-11 10:07:00.000 | 70.0 | 290.0 |
| ITEM008 | Clothes | 2021-05-11 10:08:00.000 | 20.0 | 20.0 |
| ITEM009 | Clothes | 2021-05-11 10:09:00.000 | 40.0 | 60.0 |
| ITEM010 | Clothes | 2021-05-11 10:11:00.000 | 30.0 | 90.0 |

**可以看到，对于输入的每一行数据，都有一行输出。**

总结下来，ROWS OVER WINDOW的模式应该按照下面的模式来编写SQL：

|  |
| --- |
| SELECT  field1,  AGG\_FUNCTION(field2) OVER (  [PARTITION BY (value\_expression1,..., value\_expressionN)]  ORDER BY timeAttr  ROWS  BETWEEN (UNBOUNDED | rowCount) PRECEDING AND CURRENT ROW) AS fieldName FROM tab1  -- 使用AS SELECT  field1,  AGG\_FUNCTION(field2) OVER w AS fieldName FROM tab1 WINDOW w AS (  [PARTITION BY (value\_expression1,..., value\_expressionN)]  ORDER BY timeAttr  ROWS  BETWEEN (UNBOUNDED | rowCount) PRECEDING AND CURRENT ROW ) |

**需要注意：**

|  |
| --- |
| 1. PARTITION BY是可选的，可以根据一到多个字段来对数据进行分组。 2. ORDER BY之后必须是一个时间属性，用于对数据进行排序。 3. ROWS BETWEEN ... AND ...用来界定窗口的起始结束点。UNBOUNDED PRECEDING表示整个数据流的开始作为起始点，也可以使用rowCount PRECEDING来表示当前行之前的某个元素作为起始点，rowCount是一个数字；CURRENT ROW表示当前行作为结束点。 |

###### 案例演示

* 需求描述

有一张商品销售记录表，根据商品类型统计销售金额

* + 统计销售历史总金额（Unbounded ROWS OVER Window）
  + 每隔两条销售记录作为一个窗口统计一次销售总金额（Bounded ROWS OVER Window）
* 数据来源：来自于**示例中**的销售记录数据
* 实现代码：

|  |
| --- |
| public class OverWindowSQLRows {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);   String filePath = OverWindowSQLRanges.class.getClassLoader().getResource("tmall\_item.csv").getPath();  // 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t  tEnv.executeSql("create table tmall\_item(" +  "itemID VARCHAR," +  "itemType VARCHAR," +  "eventtime VARCHAR, " +  "onSellTime AS TO\_TIMESTAMP(eventtime), " +  "price DOUBLE, " +  "WATERMARK FOR onSellTime AS onSellTime - INTERVAL '1' SECONDS) " +  "with("  + "'connector' = 'filesystem',"  + "'path' = 'file:///"+filePath+"',"  + "'format' = 'csv'"  + ")");   System.out.println("Unbounded ROWS OVER Window");  /\*\*  \* Unbounded ROWS OVER Window  \*/  tEnv  .sqlQuery(  "SELECT \n" +  " itemID,\n" +  " itemType, \n" +  " onSellTime, \n" +  " price, \n" +  " sum(price) OVER (\n" +  " PARTITION BY itemType \n" +  " ORDER BY onSellTime \n" +  " ROWS BETWEEN UNBOUNDED preceding AND CURRENT ROW) AS sumPrice\n" +  " FROM tmall\_item "  )  .execute()  .print();   tEnv  .sqlQuery(  "select " +  " itemID," +  " itemType," +  " onSellTime, \n" +  " price, \n" +  " sum(price) over w as sumPrice " +  " from tmall\_item " +  " WINDOW w AS (" +  " PARTITION BY itemType " +  " ORDER BY onSellTime " +  " ROWS BETWEEN UNBOUNDED preceding AND CURRENT ROW) "  )  .execute()  .print();   System.out.println("Bounded ROWS OVER Window");  /\*\*  \* Bounded ROWS OVER Window  \*/  tEnv  .sqlQuery(  "SELECT \n" +  " itemID,\n" +  " itemType, \n" +  " onSellTime, \n" +  " price, \n" +  " sum(price) OVER (\n" +  " PARTITION BY itemType \n" +  " ORDER BY onSellTime \n" +  " rows BETWEEN 2 preceding AND CURRENT ROW) AS sumPrice\n" +  " FROM tmall\_item "  )  .execute()  .print();  } } |

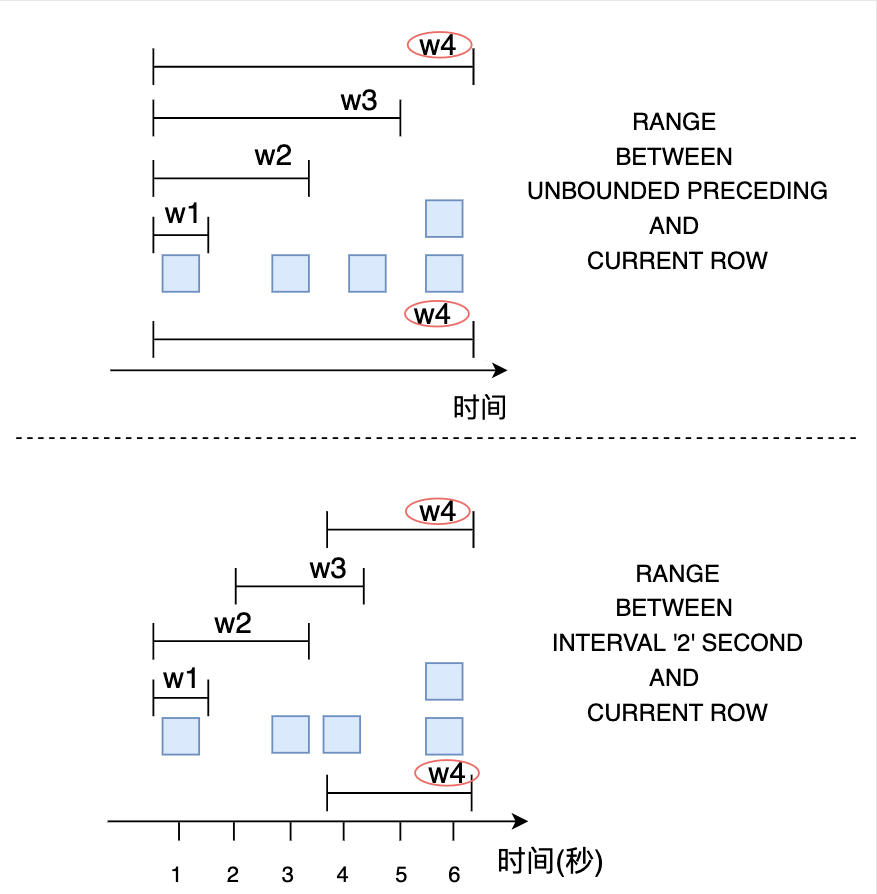
##### RANGES OVER WINDOW

###### 使用示例

第二种划分的方式是按照时间段来划分窗口，SQL中关键字为RANGE。这种窗口的结束点也是当前行，起始点是当前行之前的某个时间点。我们仍然以用户行为为例，SQL语句改为：

|  |
| --- |
| select  itemID,  itemType,  onSellTime,  price,  sum(price) over w as sumPrice from tmall\_item  WINDOW w AS (  PARTITION BY itemType  ORDER BY onSellTime  RANGE BETWEEN INTERVAL '2' MINUTE preceding AND CURRENT ROW  ) |

可以看到，与ROWS的区别在于，RANGE后面使用的是一个时间段，根据当前行的时间减去这个时间段，可以得到起始时间。



上图展示了按时间段划分窗口的基本原理，图中上半部分使用**UNBOUNDED PRECEDING**表示起始位置，与ROWS按行划分不同的是，最后两个元素虽然同时到达，但是他们被划分为一个窗口（图上半部分中的w4）；下半部分使用INTERVAL '2' SECOND表示起始位置，窗口的起始点是当前元素减去2秒，最后两个元素也被划分到了一个窗口（图下半部分中的w4）。

总结下来，RANGE OVER WINDOW的格式应该按照下面的模式来编写SQL：

|  |
| --- |
| SELECT   field1,  AGG\_FUNCTION(field2) OVER (  [PARTITION BY (value\_expression1,..., value\_expressionN)]   ORDER BY timeAttr  RANGE  BETWEEN (UNBOUNDED | timeInterval) PRECEDING AND CURRENT ROW) AS fieldName FROM tab1  -- 使用AS SELECT   field1,  AGG\_FUNCTION(field2) OVER w AS fieldName FROM tab1 WINDOW w AS (  [PARTITION BY (value\_expression1,..., value\_expressionN)]   ORDER BY timeAttr  RANGE  BETWEEN (UNBOUNDED | timeInterval) PRECEDING AND CURRENT ROW ) |

###### 案例演示

* 需求描述

有一张商品销售记录表，根据商品类型统计销售金额

* + 统计销售历史总金额（Unbounded RANGE OVER Window）
  + 每隔两分钟销售记录作为一个窗口统计一次销售总金额（Bounded RANGE OVER Window）
* 数据来源：来自于**示例中**的销售记录数据
* 实现代码：

|  |
| --- |
| public class OverWindowSQLRanges {  public static void main(String[] args) {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);   String filePath = OverWindowSQLRanges.class.getClassLoader().getResource("tmall\_item.csv").getPath();  // 作为事件时间的字段必须是 timestamp 类型, 所以根据 long 类型的 ts 计算出来一个 t  tEnv.executeSql("create table tmall\_item(" +  "itemID VARCHAR," +  "itemType VARCHAR," +  "eventtime VARCHAR, " +  "onSellTime AS TO\_TIMESTAMP(eventtime), " +  "price DOUBLE, " +  "WATERMARK FOR onSellTime AS onSellTime - INTERVAL '1' SECONDS) " +  "with("  + "'connector' = 'filesystem',"  + "'path' = 'file:///"+filePath+"',"  + "'format' = 'csv'"  + ")");    System.out.println("Unbounded ROWS OVER Window");  /\*\*  \* Unbounded ROWS OVER Window  \*/  tEnv  .sqlQuery(  "SELECT \n" +  " itemID,\n" +  " itemType, \n" +  " onSellTime, \n" +  " price, \n" +  " sum(price) OVER (\n" +  " PARTITION BY itemType \n" +  " ORDER BY onSellTime \n" +  " RANGE BETWEEN UNBOUNDED preceding AND CURRENT ROW) AS sumPrice\n" +  " FROM tmall\_item "  )  .execute()  .print();   System.out.println("Bounded ROWS OVER Window");  /\*\*  \* Bounded ROWS OVER Window  \*/  tEnv  .sqlQuery(  "SELECT \n" +  " itemID,\n" +  " itemType, \n" +  " onSellTime, \n" +  " price, \n" +  " sum(price) OVER (\n" +  " PARTITION BY itemType \n" +  " ORDER BY onSellTime \n" +  " RANGE BETWEEN INTERVAL '2' MINUTE preceding AND CURRENT ROW) AS sumPrice\n" +  " FROM tmall\_item "  )  .execute()  .print();   tEnv  .sqlQuery(  "select\n" +  " itemID,\n" +  " itemType,\n" +  " onSellTime,\n" +  " price,\n" +  " sum(price) over w as sumPrice\n" +  "from tmall\_item\n" +  " WINDOW w AS (\n" +  " PARTITION BY itemType\n" +  " ORDER BY onSellTime\n" +  " RANGE BETWEEN INTERVAL '2' MINUTE preceding AND CURRENT ROW\n" +  ") "  )  .execute()  .print();  } } |

### TableAPI中窗口的定义

#### [Group Windows](https://blog.csdn.net/weixin_42796403/article/details/114711619" \l "Group_Windows_7" \t "https://blog.csdn.net/weixin_42796403/article/details/_self)

分组窗口（Group Windows）会根据时间或行计数间隔，将行聚合到有限的组（Group）中，并对每个组的数据执行一次聚合函数。

Table API中的Group Windows都是使用.window（w:GroupWindow）子句定义的，并且必须由as子句指定一个别名。为了按窗口对表进行分组，窗口的别名必须在group by子句中，像常规的分组字段一样引用。

|  |
| --- |
| Table table **=** input  **.**window**([**GroupWindow w**].**as**(**"w"**))** //定义窗口，别名 w  **.**groupBy**(**$**(**"w"**))** //以窗口w作为分组的key  **.**select**(**$**(**"b"**).**sum**());** //聚合字段b的值，求和 |

还可以把窗口的相关信息，作为字段添加到结果表中：

|  |
| --- |
| Table table **=** input  **.**window**([**GroupWindow w**].**as**(**"w"**))** // 定义窗口，别名 w  **.**groupBy**(**$**(**"w"**),** $**(**"a"**))** // 以属性a和窗口w作为分组的key  **.**select**(**$**(**"a"**),** $**(**"b"**).**sum**());** //聚合字段b的值，求和 |

Table API提供了一组具有特定语义的预定义Window类，这些类会被转换为底层DataStream的窗口操作。

Table API支持的窗口定义，主要也是三种：滚动（Tumbling）、滑动（Sliding）和会话（Session）。

##### 滚动窗口

* 滚动窗口（Tumbling windows）要用Tumble类来定义，另外还有三个方法：
  + over：定义窗口长度
  + on：用来分组（按时间间隔）或者排序（按行数）的时间字段
  + as：别名，必须出现在后面的groupBy中
* 需求描述

每隔5秒钟统计一次每个商品类型的销售总额

* 实现代码：

|  |
| --- |
| public class GroupWindowsTableApiTumbleExample {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);   SingleOutputStreamOperator<OrderInfo> dataStream = env  .fromElements(  new OrderInfo("电脑", 1000L, 100D),  new OrderInfo("手机", 2000L, 200D),  new OrderInfo("电脑", 3000L, 300D),  new OrderInfo("手机", 4000L, 400D),  new OrderInfo("手机", 5000L, 500D),  new OrderInfo("电脑", 6000L, 600D))  .assignTimestampsAndWatermarks(  WatermarkStrategy  .<OrderInfo>forBoundedOutOfOrderness(Duration.ofSeconds(5))  .withTimestampAssigner((element, recordTimestamp) -> element.getTimestamp())  );   StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);  Table table = tableEnv  .fromDataStream(dataStream, $("category"), $("timestamp").rowtime(), $("money"));   table  .window(Tumble.over(lit(5).second()).on($("timestamp")).as("w")) // 定义滚动窗口并给窗口起一个别名  .groupBy($("category"), $("w")) // 窗口必须出现的分组字段中  .select($("category"), $("w").start().as("window\_start"), $("w").end().as("window\_end"), $("money").sum().as("total\_money"))  .execute()  .print();   env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class OrderInfo {  private String category;  private Long timestamp;  private Double money;  } } |

##### 滑动窗口

* 滑动窗口（Sliding windows）要用Slide类来定义，另外还有四个方法：
  + over：定义窗口长度
  + every：定义滑动步长
  + on：用来分组（按时间间隔）或者排序（按行数）的时间字段
  + as：别名，必须出现在后面的groupBy中
* 需求描述

每隔5秒钟统计过去10秒钟每个商品类型的销售总额

* 实现代码：

|  |
| --- |
| public class GroupWindowsTableApiSlideExample {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);   SingleOutputStreamOperator<OrderInfo> dataStream = env  .fromElements(  new OrderInfo("电脑", 1000L, 100D),  new OrderInfo("手机", 2000L, 200D),  new OrderInfo("电脑", 3000L, 300D),  new OrderInfo("手机", 4000L, 400D),  new OrderInfo("手机", 5000L, 500D),  new OrderInfo("电脑", 6000L, 600D))  .assignTimestampsAndWatermarks(  WatermarkStrategy  .<OrderInfo>forBoundedOutOfOrderness(Duration.ofSeconds(5))  .withTimestampAssigner((element, recordTimestamp) -> element.getTimestamp())  );   StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);  Table table = tableEnv  .fromDataStream(dataStream, $("category"), $("timestamp").rowtime(), $("money"));   table  .window(Slide.over(lit(10).second()).every(lit(5).second()).on($("timestamp")).as("w")) // 定义滑动窗口并给窗口起一个别名  .groupBy($("category"), $("w")) // 窗口必须出现的分组字段中  .select($("category"), $("w").start().as("window\_start"), $("w").end().as("window\_end"), $("money").sum().as("total\_money"))  .execute()  .print();  env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class OrderInfo {  private String category;  private Long timestamp;  private Double money;  } } |

##### 会话窗口

* 会话窗口（Session windows）要用Session类来定义，另外还有三个方法：
  + withGap：会话时间间隔
  + on：用来分组（按时间间隔）或者排序（按行数）的时间字段
  + as：别名，必须出现在后面的groupBy中
* 需求描述

两次的时间间隔超过6秒的基础上，没有新的订单事件这个窗口就会关闭，然后处理这个窗口区间内所产生的订单数据计算

* 实现代码：

|  |
| --- |
| public class GroupWindowsTableApiSessionExample {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);   SingleOutputStreamOperator<OrderInfo> dataStream = env  .fromElements(  new OrderInfo("电脑", 1000L, 100D),  new OrderInfo("手机", 2000L, 200D),  new OrderInfo("电脑", 3000L, 300D),  new OrderInfo("手机", 4000L, 400D),  new OrderInfo("手机", 5000L, 500D),  new OrderInfo("电脑", 6000L, 600D))  .assignTimestampsAndWatermarks(  WatermarkStrategy  .<OrderInfo>forBoundedOutOfOrderness(Duration.ofSeconds(5))  .withTimestampAssigner((element, recordTimestamp) -> element.getTimestamp())  );   StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);  Table table = tableEnv  .fromDataStream(dataStream, $("category"), $("timestamp").rowtime(), $("money"));   table  .window(Session.withGap(lit(6).second()).on($("timestamp")).as("w")) // 定义滑动窗口并给窗口起一个别名  .groupBy($("category"), $("w")) // 窗口必须出现的分组字段中  .select($("category"), $("w").start().as("window\_start"), $("w").end().as("window\_end"), $("money").sum().as("total\_money"))  .execute()  .print();  env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class OrderInfo {  private String category;  private Long timestamp;  private Double money;  } } |

#### [Over Windows](https://blog.csdn.net/weixin_42796403/article/details/114711619" \l "Over_Windows_59" \t "https://blog.csdn.net/weixin_42796403/article/details/_self)

Over window聚合是标准SQL中已有的（Over子句），可以在查询的SELECT子句中定义。Over window 聚合，会针对每个输入行，计算相邻行范围内的聚合。

Table API提供了Over类，来配置Over窗口的属性。可以在事件时间或处理时间，以及指定为时间间隔、或行计数的范围内，定义Over windows。

* 无界的over window是**使用常量指定**的
  + 基于行划分窗口UNBOUNDED\_ROW
  + 基于时间段划分窗口UNBOUNDED\_RANGE
* 有界的over window是**使用间隔的大小指定**的

代码示例如下：

* 无界的 over window

|  |
| --- |
| // 无界的事件时间over window (时间字段 "rowtime")  **.**window**(**Over partitionBy 'a orderBy 'rowtime preceding UNBOUNDED\_RANGE as 'w)  //无界的处理时间over window (时间字段"proctime")  **.**window**(**Over partitionBy 'a orderBy 'proctime preceding UNBOUNDED\_RANGE as 'w)  // 无界的事件时间Row-count over window (时间字段 "rowtime")  **.**window**(**Over partitionBy 'a orderBy 'rowtime preceding UNBOUNDED\_ROW as 'w)  //无界的处理时间Row-count over window (时间字段 "rowtime")  **.**window**(**Over partitionBy 'a orderBy 'proctime preceding UNBOUNDED\_ROW as 'w) |

* 有界的over window

|  |
| --- |
| // 有界的事件时间over window (时间字段 "rowtime"，之前1分钟)  **.**window**(**Over partitionBy 'a orderBy 'rowtime preceding 1.minutes as 'w)  // 有界的处理时间over window (时间字段 "rowtime"，之前1分钟)  **.**window**(**Over partitionBy 'a orderBy 'proctime preceding 1.minutes as 'w)  // 有界的事件时间Row-count over window (时间字段 "rowtime"，之前10行)  **.**window**(**Over partitionBy 'a orderBy 'rowtime preceding 10.rows as 'w)  // 有界的处理时间Row-count over window (时间字段 "rowtime"，之前10行)  **.**window**(**Over partitionBy 'a orderBy 'proctime preceding 10.rows as 'w) |

## 函数（Functions）

Flink Table 和 SQL内置了很多SQL中支持的函数；如果有无法满足的需要，则可以实现用户自定义的函数（UDF）来解决。

### 函数类型

Flink 中的函数有两个划分标准。

一个划分标准是：**系统（内置）函数和 Catalog 函数**。系统函数没有名称空间，只能通过其名称来进行引用。 Catalog 函数属于 Catalog 和数据库，因此它们拥有 Catalog 和数据库命名空间。 用户可以通过全/部分限定名（catalog.db.func 或 db.func）或者函数名 来对 Catalog 函数进行引用。

另一个划分标准是：**临时函数和持久化函数**。 临时函数始终由用户创建，它容易改变并且仅在会话的生命周期内有效。 持久化函数不是由系统提供，就是存储在 Catalog 中，它在会话的整个生命周期内都有效。

这两个划分标准给 Flink 用户提供了 4 种函数：

* 临时性系统函数
* 系统函数
* 临时性 Catalog 函数
* Catalog 函数

**请注意：**系统函数始终优先于 Catalog 函数解析，临时函数始终优先于持久化函数解析， 函数解析优先级如下所述

### 函数引用

用户在 Flink 中可以通过精确、模糊两种引用方式引用函数。

#### 精确函数引用

精确函数引用允许用户跨 Catalog，跨数据库调用 Catalog 函数。 例如：select mycatalog.mydb.myfunc(x) from mytable 和 select mydb.myfunc(x) from mytable。

**仅 Flink 1.10 以上版本支持。**

#### 模糊函数引用

在模糊函数引用中，用户只需在 SQL 查询中指定函数名，例如： select myfunc(x) from mytable。

### 函数解析顺序

当函数名相同，函数类型不同时，函数解析顺序才有意义。 例如：当有三个都名为 “myfunc” 的临时性 Catalog 函数，Catalog 函数，和系统函数时， 如果没有命名冲突，三个函数将会被解析为一个函数。

#### 精确函数引用

由于系统函数没有命名空间，Flink 中的精确函数引用必须 指向临时性 Catalog 函数或 Catalog 函数。

解析顺序如下：

* 临时性 catalog 函数
* Catalog 函数

#### 模糊函数引用

解析顺序如下：

* 临时性系统函数
* 系统函数
* 临时性 Catalog 函数, 在会话的当前 Catalog 和当前数据库中
* Catalog 函数, 在会话的当前 Catalog 和当前数据库中

### 系统内置函数

Flink Table API 和 SQL为用户提供了一组用于数据转换的内置函数。SQL中支持的很多函数，Table API和SQL都已经做了实现，其它还在快速开发扩展中。

以下是一些典型函数的举例，全部的内置函数，可以参考官网介绍：

<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/systemfunctions/>

|  |  |  |
| --- | --- | --- |
| **函数类型** | **SQL** | **Table API** |
| 比较函数 | value1 = value2 | ANY1 === ANY2 |
| value1 > value2 | ANY1 > ANY2 |
| 逻辑函数 | boolean1 OR boolean2 | BOOLEAN1 || BOOLEAN2 |
| boolean IS FALSE | BOOLEAN.isFalse |
| NOT boolean | !BOOLEAN |
| 算术函数 | numeric1 + numeric2 | NUMERIC1 + NUMERIC2 |
| POWER(numeric1, numeric2) | NUMERIC1.power(NUMERIC2) |
| 字符串函数 | string1 || string2 | STRING1 + STRING2 |
| UPPER(string) | STRING.upperCase() |
| CHAR\_LENGTH(string) | STRING.charLength() |
| 时间函数 | DATE string | STRING.toDate |
| TIMESTAMP string | STRING.toTimestamp |
| TIMESTAMP string | currentTime() |
| INTERVAL string range | NUMERIC.days |
|  | NUMERIC.minutes |
| 聚合函数 | COUNT(\*) | FIELD.count |
| SUM([ ALL | DISTINCT ] expression) | FIELD.sum0 |
| RANK() |  |
| ROW\_NUMBER() |  |

### UDF

自定义函数（User-defined Functions，UDF）是一种扩展开发机制，可以用来在查询语句里调用难以用其他方式表达的频繁使用或自定义的逻辑。一些系统内置函数无法解决的需求，我们可以用UDF来自定义实现。

自定义函数可以用 JVM 语言（例如 Java 或 Scala）或 Python 实现，实现者可以在 UDF 中使用任意第三方库，本文聚焦于使用 JVM 语言开发自定义函数。

#### 概述

当前 Flink 有如下几种函数：

* 标量函数 将标量值转换成一个新标量值；
* 表值函数 将标量值转换成新的行数据；
* 聚合函数 将多行数据里的标量值转换成一个新标量值；
* 表值聚合函数 将多行数据里的标量值转换成新的行数据；
* 异步表值函数 是异步查询外部数据系统的特殊函数。

注意 标量和表值函数已经使用了新的基于[数据类型](https://ci.apache.org/projects/flink/flink-docs-release-1.11/zh/dev/table/types.html)的类型系统，聚合函数仍然使用基于 TypeInformation 的旧类型系统。

以下示例展示了如何创建一个基本的标量函数，以及如何在 Table API 和 SQL 里调用这个函数。

函数用于 SQL 查询前要先经过注册；而在用于 Table API 时，函数可以先注册后调用，也可以 内联 后直接使用。

|  |
| --- |
| // 定义函数逻辑 public static class SubstringFunction extends ScalarFunction {  public String eval(String s, Integer begin, Integer end) {  return s.substring(begin, end);  } }  TableEnvironment env = TableEnvironment.create(...);  // 在 Table API 里不经注册直接“内联”调用函数 env.from("MyTable").select(call(SubstringFunction.class, $("myField"), 5, 12));  // 注册函数 env.createTemporarySystemFunction("SubstringFunction", SubstringFunction.class);  // 在 Table API 里调用注册好的函数 env.from("MyTable").select(call("SubstringFunction", $("myField"), 5, 12));  // 在 SQL 里调用注册好的函数 env.sqlQuery("SELECT SubstringFunction(myField, 5, 12) FROM MyTable"); |

对于交互式会话，还可以在使用或注册函数之前对其进行参数化，这样可以把函数 实例 而不是函数 类 用作临时函数。

为确保函数实例可应用于集群环境，参数必须是可序列化的。

|  |
| --- |
| import org.apache.flink.table.api.\*; import org.apache.flink.table.functions.ScalarFunction; import static org.apache.flink.table.api.Expressions.\*;   // 定义可参数化的函数逻辑 public static class SubstringFunction extends ScalarFunction {   private boolean endInclusive;   public SubstringFunction(boolean endInclusive) {  this.endInclusive = endInclusive;  }   public String eval(String s, Integer begin, Integer end) {  return s.substring(begin, endInclusive ? end + 1 : end);  } }  TableEnvironment env = TableEnvironment.create(...);  // 在 Table API 里不经注册直接“内联”调用函数 env.from("MyTable").select(call(new SubstringFunction(true), $("myField"), 5, 12));  // 注册函数 env.createTemporarySystemFunction("SubstringFunction", new SubstringFunction(true)); |

#### 开发指南

注意在聚合函数使用新的类型系统前，本节仅适用于标量和表值函数。

所有的自定义函数都遵循一些基本的实现原则。

##### 函数类

实现类必须继承自合适的基类之一（例如 **org.apache.flink.table.functions.ScalarFunction**）。

该类必须声明为 public ，而不是 abstract ，并且可以被全局访问。不允许使用非静态内部类或匿名类。

为了将自定义函数存储在持久化的 catalog 中，该类必须具有默认构造器，且在运行时可实例化。

##### 求值方法

基类提供了一组可以被重写的方法，例如 open()、 close() 或 isDeterministic() 。

但是，除了上述方法之外，作用于每条传入记录的主要逻辑还必须通过专门的 求值方法 来实现。

根据函数的种类，后台生成的运算符会在运行时调用诸如 eval()、accumulate() 或 retract() 之类的求值方法。

这些方法必须声明为 public ，并带有一组定义明确的参数。

常规的 JVM 方法调用语义是适用的。因此可以：

* 实现重载的方法，例如 eval(Integer) 和 eval(LocalDateTime)；
* 使用变长参数，例如 eval(Integer...);
* 使用对象继承，例如 eval(Object) 可接受 LocalDateTime 和 Integer 作为参数；
* 也可组合使用，例如 eval(Object...) 可接受所有类型的参数。

以下代码片段展示了一个重载函数的示例：

|  |
| --- |
| import org.apache.flink.table.functions.ScalarFunction;  // 有多个重载求值方法的函数 public static class SumFunction extends ScalarFunction {   public Integer eval(Integer a, Integer b) {  return a + b;  }   public Integer eval(String a, String b) {  return Integer.valueOf(a) + Integer.valueOf(b);  }   public Integer eval(Double... d) {  double result = 0;  for (double value : d)  result += value;  return (int) result;  } } |

##### 类型推导

Table（类似于 SQL 标准）是一种强类型的 API。因此，函数的参数和返回类型都必须映射到[数据类型](https://ci.apache.org/projects/flink/flink-docs-release-1.11/zh/dev/table/types.html)。

从逻辑角度看，Planner 需要知道数据类型、精度和小数位数；

从 JVM 角度来看，Planner 在调用自定义函数时需要知道如何将内部数据结构表示为 JVM 对象。

术语 类型推导 概括了意在验证输入值、派生出参数/返回值数据类型的逻辑。

Flink 自定义函数实现了自动的类型推导提取，通过反射从函数的类及其求值方法中派生数据类型。如果这种隐式的反射提取方法不成功，则可以通过使用 @DataTypeHint 和 @FunctionHint 注解相关参数、类或方法来支持提取过程，

下面展示了有关如何注解函数的例子。

如果需要更高级的类型推导逻辑，实现者可以在每个自定义函数中显式重写 getTypeInference() 方法。但是，建议使用注解方式，因为它可使自定义类型推导逻辑保持在受影响位置附近，而在其他位置则保持默认状态。

###### 自动类型推导

自动类型推导会检查函数的类和求值方法，派生出函数参数和结果的数据类型， @DataTypeHint 和 @FunctionHint 注解支持自动类型推导。

有关可以隐式映射到数据类型的类的完整列表，请参阅[数据类型](https://ci.apache.org/projects/flink/flink-docs-release-1.11/zh/dev/table/types.html" \l "%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B%E6%B3%A8%E8%A7%A3)。

**@DataTypeHint**

在许多情况下，需要支持以 内联 方式自动提取出函数参数、返回值的类型。

以下例子展示了如何使用 @DataTypeHint，详情可参考该注解类的文档。

|  |
| --- |
| import org.apache.flink.table.annotation.DataTypeHint; import org.apache.flink.table.annotation.InputGroup; import org.apache.flink.table.functions.ScalarFunction; import org.apache.flink.types.Row;  // 有多个重载求值方法的函数 public static class OverloadedFunction extends ScalarFunction {   // no hint required  public Long eval(long a, long b) {  return a + b;  }   // 定义 decimal 的精度和小数位  public @DataTypeHint("DECIMAL(12, 3)") BigDecimal eval(double a, double b) {  return BigDecimal.valueOf(a + b);  }   // 定义嵌套数据类型  @DataTypeHint("ROW<s STRING, t TIMESTAMP\_LTZ(3)>")  public Row eval(int i) {  return Row.of(String.valueOf(i), Instant.ofEpochSecond(i));  }   // 允许任意类型的符入，并输出序列化定制后的值  @DataTypeHint(value = "RAW", bridgedTo = ByteBuffer.class)  public ByteBuffer eval(@DataTypeHint(inputGroup = InputGroup.ANY) Object o) {  return MyUtils.serializeToByteBuffer(o);  } } |

**@FunctionHint**

有时我们希望一种求值方法可以同时处理多种数据类型，有时又要求对重载的多个求值方法仅声明一次通用的结果类型。

@FunctionHint 注解可以提供从入参数据类型到结果数据类型的映射，它可以在整个函数类或求值方法上注解输入、累加器和结果的数据类型。可以在类顶部声明一个或多个注解，也可以为类的所有求值方法分别声明一个或多个注解。所有的 hint 参数都是可选的，如果未定义参数，则使用默认的基于反射的类型提取。在函数类顶部定义的 hint 参数被所有求值方法继承。

以下例子展示了如何使用 @FunctionHint，详情可参考该注解类的文档。

|  |
| --- |
| // 为函数类的所有求值方法指定同一个输出类型 @FunctionHint(output = @DataTypeHint("ROW<s STRING, i INT>")) public static class OverloadedFunction extends TableFunction<Row> {   public void eval(int a, int b) {  collect(Row.of("Sum", a + b));  }   // overloading of arguments is still possible  public void eval() {  collect(Row.of("Empty args", -1));  } }  // 解耦类型推导与求值方法，类型推导完全取决于 FunctionHint @FunctionHint(  input = {@DataTypeHint("INT"), @DataTypeHint("INT")},  output = @DataTypeHint("INT") ) @FunctionHint(  input = {@DataTypeHint("BIGINT"), @DataTypeHint("BIGINT")},  output = @DataTypeHint("BIGINT") ) @FunctionHint(  input = {},  output = @DataTypeHint("BOOLEAN") ) public static class OverloadedFunction extends TableFunction<Object> {   // an implementer just needs to make sure that a method exists  // that can be called by the JVM  public void eval(Object... o) {  if (o.length == 0) {  collect(false);  }  collect(o[0]);  } } |

###### 定制类型推导

在大多数情况下，@DataTypeHint 和 @FunctionHint 足以构建自定义函数，然而通过重写 getTypeInference() 定制自动类型推导逻辑，实现者可以创建任意像系统内置函数那样有用的函数。

以下用 Java 实现的例子展示了定制类型推导的潜力，它根据字符串参数来确定函数的结果类型。该函数带有两个字符串参数：第一个参数表示要分析的字符串，第二个参数表示目标类型。

|  |
| --- |
| import org.apache.flink.table.annotation.DataTypeHint; import org.apache.flink.table.annotation.FunctionHint; import org.apache.flink.table.functions.TableFunction; import org.apache.flink.types.Row;  import org.apache.flink.table.api.DataTypes; import org.apache.flink.table.catalog.DataTypeFactory; import org.apache.flink.table.functions.ScalarFunction; import org.apache.flink.table.types.inference.TypeInference; import org.apache.flink.types.Row;  public static class LiteralFunction extends ScalarFunction {  public Object eval(String s, String type) {  switch (type) {  case "INT":  return Integer.valueOf(s);  case "DOUBLE":  return Double.valueOf(s);  case "STRING":  default:  return s;  }  }   // 禁用自动的反射式类型推导，使用如下逻辑进行类型推导  @Override  public TypeInference getTypeInference(DataTypeFactory typeFactory) {  return TypeInference.newBuilder()  // 指定输入参数的类型，必要时参数会被隐式转换  .typedArguments(DataTypes.STRING(), DataTypes.STRING())  // specify a strategy for the result data type of the function  .outputTypeStrategy(callContext -> {  if (!callContext.isArgumentLiteral(1) || callContext.isArgumentNull(1)) {  throw callContext.newValidationError("Literal expected for second argument.");  }  // 基于字符串值返回数据类型  final String literal = callContext.getArgumentValue(1, String.class).orElse("STRING");  switch (literal) {  case "INT":  return Optional.of(DataTypes.INT().notNull());  case "DOUBLE":  return Optional.of(DataTypes.DOUBLE().notNull());  case "STRING":  default:  return Optional.of(DataTypes.STRING());  }  })  .build();  } } |

##### 运行时集成

有时候自定义函数需要获取一些全局信息，或者在真正被调用之前做一些配置（setup）/清理（clean-up）的工作。自定义函数也提供了 open() 和 close() 方法，你可以重写这两个方法做到类似于 DataStream API 中 RichFunction 的功能。

open() 方法在求值方法被调用之前先调用。close() 方法在求值方法调用完之后被调用。

open() 方法提供了一个 FunctionContext，它包含了一些自定义函数被执行时的上下文信息，比如 metric group、分布式文件缓存，或者是全局的作业参数等。

下面的信息可以通过调用 FunctionContext 的对应的方法来获得：

|  |  |
| --- | --- |
| **方法** | **描述** |
| getMetricGroup() | 执行该函数的 subtask 的 Metric Group |
| getCachedFile(name) | 分布式文件缓存的本地临时文件副本 |
| getJobParameter(name, defaultValue) | 跟对应的 key 关联的全局参数值 |

下面的例子展示了如何在一个标量函数中通过 FunctionContext 来获取一个全局的任务参数：

|  |
| --- |
| import org.apache.flink.table.api.\*; import org.apache.flink.table.functions.FunctionContext; import org.apache.flink.table.functions.ScalarFunction;  public static class HashCodeFunction extends ScalarFunction {   private int factor = 0;   @Override  public void open(FunctionContext context) throws Exception {  // 获取参数 "hashcode\_factor"  // 如果不存在，则使用默认值 "12"  factor = Integer.parseInt(context.getJobParameter("hashcode\_factor", "12"));  }   public int eval(String s) {  return s.hashCode() \* factor;  } }  TableEnvironment env = TableEnvironment.create(...);  // 设置任务参数 env.getConfig().addJobParameter("hashcode\_factor", "31");  // 注册函数 env.createTemporarySystemFunction("hashCode", HashCodeFunction.class);  // 调用函数 env.sqlQuery("SELECT myField, hashCode(myField) FROM MyTable"); |

#### 标量函数（Scalar Functions）

用户定义的标量函数，可以将0、1或多个标量值，映射到新的标量值。

为了定义标量函数，必须在org.apache.flink.table.functions中扩展基类Scalar Function，并实现（一个或多个）求值（evaluation，eval）方法。标量函数的行为由求值方法决定，求值方法必须公开声明并命名为eval（直接def声明，没有override）。求值方法的参数类型和返回类型，确定了标量函数的参数和返回类型。

代码如下：

|  |
| --- |
| /\*\*  \* 自定义ScalarFunction  \*/ public class CustomScalarFunction{  public static void main(String[] args) throws Exception{  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1);  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);  //通过程序的方式来注册函数  SumFunction sumFunction = new SumFunction();  tableEnv.createTemporaryFunction("mysum", sumFunction);  Table table1 = tableEnv.sqlQuery("select mysum(1,2)");  tableEnv.toAppendStream(table1, Row.class).print();   //------------------------------------------------------   //通过sql的方式来注册函数  String className = SumFunction.class.getName();  String sql = "create temporary function default\_catalog.default\_database.mysum1" +  " as '" + className + "'";  tableEnv.executeSql(sql);  Table table2 = tableEnv.sqlQuery("select mysum1(3,4)");  tableEnv.toAppendStream(table2, Row.class).print();   //------------------------------------------------------   //列出来所有的函数，看是否包含我们定义的函数  String[] functions = tableEnv.listFunctions();  Stream.of(functions).filter(f->f.startsWith("mysum")).forEach(System.out::println);   //---------------------------------------------------   //接收非空的int或者boolean类型  StringifyFunction stringifyFunction = new StringifyFunction();  tableEnv.createTemporaryFunction("myToString", stringifyFunction);  Table table3 = tableEnv.sqlQuery("select myToString(1) ,myToString(false) ");  tableEnv.toAppendStream(table3, Row.class).print();    //接收任何类型的值，然后把它们转成string  StringifyFunction2 stringifyFunction2 = new StringifyFunction2();  tableEnv.createTemporaryFunction("myToStringAny", stringifyFunction2);  Table table4 = tableEnv.sqlQuery("select myToStringAny(1) ,myToStringAny(false),myToStringAny('aaa') ");  tableEnv.toAppendStream(table4, Row.class).print();   env.execute("CustomScalarFunction");  }   /\*\*  \* 接受两个int类型的参数，然后返回计算的sum值  \*/  public static class SumFunction extends ScalarFunction {  public Integer eval(Integer a, Integer b){  return a + b;  }  }    /\*\*  \*接收非空的int或者boolean类型,通过注解方式实现，flink 1.11版本后支持  \*/  @FunctionHint(input = @DataTypeHint("INT"))  @FunctionHint(input = @DataTypeHint("BOOLEAN"))  public static class StringifyFunction extends ScalarFunction {  public String eval(Object o) {  return o.toString();  }  }   /\*\*  \* 接收任何类型的值，然后把它们转成string  \*/  public static class StringifyFunction2 extends ScalarFunction{  public String eval(@DataTypeHint(inputGroup = InputGroup.ANY) Object o){  return o.toString();  }  } } |

#### 表函数（Table Functions）

与用户定义的标量函数类似，用户定义的表函数，可以将0、1或多个标量值作为输入参数；与标量函数不同的是，它可以返回任意数量的行作为输出，而不是单个值。

为了定义一个表函数，必须扩展org.apache.flink.table.functions中的基类TableFunction并实现（一个或多个）求值方法。表函数的行为由其求值方法决定，求值方法必须是public的，并命名为eval。求值方法的参数类型，决定表函数的所有有效参数。

返回表的类型由TableFunction的泛型类型确定。求值方法使用protected collect（T）方法发出输出行。

在Table API中，Table函数需要与.joinLateral或.leftOuterJoinLateral一起使用。

joinLateral算子，会将外部表中的每一行，与表函数（TableFunction，算子的参数是它的表达式）计算得到的所有行连接起来。

而leftOuterJoinLateral算子，则是左外连接，它同样会将外部表中的每一行与表函数计算生成的所有行连接起来；并且，对于表函数返回的是空表的外部行，也要保留下来。

在SQL中，则需要使用Lateral Table（<TableFunction>），或者带有ON TRUE条件的左连接。

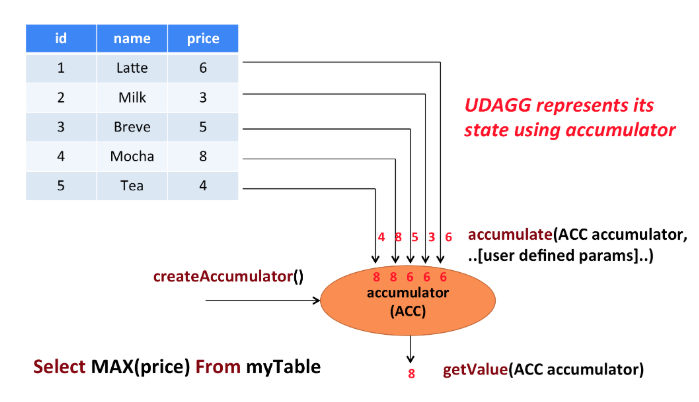
下面的代码中，我们将定义一个表函数，在表环境中注册它，并在查询中调用它。

自定义TableFunction：

|  |
| --- |
| /\*\*  \* 自定义TableFunction  \*/ public class CustomTableFunction{  public static void main(String[] args) throws Exception{  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  EnvironmentSettings bsSettings = EnvironmentSettings.newInstance()  .useBlinkPlanner()  .inStreamingMode()  .build();  StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, bsSettings);   tEnv.createTemporaryFunction("split", new Split(" "));  tEnv.createTemporaryFunction("duplicator", new DuplicatorFunction());  tEnv.createTemporaryFunction("flattens", new FlattenFunction());   List<Tuple2<Integer,String>> ordersData = new ArrayList<>();  ordersData.add(Tuple2.of(2, "Euro"));  ordersData.add(Tuple2.of(1, "US Dollar"));  ordersData.add(Tuple2.of(50, "Yen"));  ordersData.add(Tuple2.of(3, "Euro"));   DataStream<Tuple2<Integer,String>> ordersDataStream = env.fromCollection(ordersData);  Table orders = tEnv.fromDataStream(ordersDataStream, $("amount"), $("currency"), $("proctime").proctime());  tEnv.createTemporaryView("Orders", orders);   //使用left join  Table result = tEnv.sqlQuery(  "SELECT o.currency, T.word, T.length FROM Orders as o LEFT JOIN " +  "LATERAL TABLE(split(currency)) as T(word, length) ON TRUE");  tEnv.toAppendStream(result, Row.class).print("LEFT JOIN查询");   //----------------------------   String sql = "SELECT o.currency, T.word, T.length FROM Orders as o ," +  " LATERAL TABLE(split(currency)) as T(word, length)";  Table result1 = tEnv.sqlQuery(sql);  tEnv.toAppendStream(result1, Row.class).print("分词查询");   //---------------------------  //多种类型参数   String sql2 = "SELECT \* FROM Orders as o , " +  "LATERAL TABLE(duplicator(amount)) as F(word, length)," +  "LATERAL TABLE(duplicator(currency)) as T(word, length)";  Table result2 = tEnv.sqlQuery(sql2);  tEnv.toAppendStream(result2, Row.class).print("多种类型参数");   //----------------------------  //不固定参数查询   String sql3 = "SELECT \* FROM Orders as o , " +  "LATERAL TABLE(flattens(100,200,300)) as T(num)";  Table result3 = tEnv.sqlQuery(sql3);  tEnv.toAppendStream(result3, Row.class).print("不固定参数查询");   env.execute();  }   public static class Split extends TableFunction<Tuple2<String,Integer>>{  private String separator = ",";   public Split(String separator){  this.separator = separator;  }   public void eval(String str){  for (String s: str.split(separator)){  collect(new Tuple2<String,Integer>(s, s.length()));  }  }  }   /\*\*  \* 接收不固定个数的int型参数,然后将所有数据依次返回  \*/  @FunctionHint(output = @DataTypeHint("ROW< s INT>"))  public static class FlattenFunction extends TableFunction<Row>{  public void eval(Integer... args){  for (Integer i: args){  Row row = new Row(1);  row.setField(0, i);  collect(row);  }  }  }   /\*\*  \* 注册多个eval方法，接收long或者字符串类型的参数，然后将他们转成string类型  \* 通过注册指定返回值类型，flink 1.11 版本开始支持  \*/  @FunctionHint(output = @DataTypeHint("ROW< s STRING, i INT>"))  public static class DuplicatorFunction extends TableFunction<Row>{  public void eval(Integer s){  eval(String.valueOf(s));  }   public void eval(String s){  collect(Row.of(s, s.length()));  }  } } |

#### 聚合函数（Aggregate Functions）

用户自定义聚合函数（User-Defined Aggregate Functions，UDAGGs）可以把一个表中的数据，聚合成一个标量值。用户定义的聚合函数，是通过继承AggregateFunction抽象类实现的。



上图中显示了一个聚合的例子。

假设现在有一张表，包含了各种饮料的数据。该表由三列（id、name和price）、五行组成数据。现在我们需要找到表中所有饮料的最高价格，即执行max（）聚合，结果将是一个数值。

AggregateFunction的工作原理如下。

* 首先，它需要一个累加器，用来保存聚合中间结果的数据结构（状态）。可以通过调用AggregateFunction的createAccumulator（）方法创建空累加器。
* 随后，对每个输入行调用函数的accumulate（）方法来更新累加器。
* 处理完所有行后，将调用函数的getValue（）方法来计算并返回最终结果。

AggregationFunction要求必须实现的方法：

* createAccumulator()
* accumulate()
* getValue()

除了上述方法之外，还有一些可选择实现的方法。其中一些方法，可以让系统执行查询更有效率，而另一些方法，对于某些场景是必需的。例如，如果聚合函数应用在会话窗口（session group window）的上下文中，则merge（）方法是必需的。

* retract()
* merge()
* resetAccumulator()

##### 案例演示

* 需求描述

在网站性能测试中，我们经常会选择 TP50、TP95 或者 TP99 等作为性能指标。接下来我们讲讲这些指标的含义、以及在flink中如何实时统计：

TP50，top percent 50，即 50% 的数据都满足某一条件；

TP95，top percent 95，即 95% 的数据都满足某一条件；

TP99，top percent 99，即 99% 的数据都满足某一条件；

举一个例子，统计网站一分钟之内的的响应时间的TP90,正常的处理逻辑就是把这一分钟之内所有的网站的响应时间从小到大排序，然后计算出总条数count，然后计算出排名在90%的响应时间是多少（count\*0.9），就是我们要的值。

* 案例分析

对于TP指标，正常的思路可以先创建一个临时变量，里面有一个list，每来一个数据，就放到这个list里面，在getValue方法里，进行排序，取相应的TP值。

但是这种思路会有一个问题，就是如果要聚合的时间范围内，数据过多的话。就会在list存储大量的数据，会造成checkpoint过大，时间过长，最后导致程序失败。得不到正确的结果。

所以需要换一个思路，既然最后想要的是一个有序列表，那么可以把这个list结构优化一下，使用Treemap来存储，map的key就是指标，比如响应时间。value就是对应的指标出现的次数。这样getValue方法里，只需要将map的value值累加，就能得到总数count，然后计算出来相应的tp值的位置position，最后再从头累加map的value，直到累加结果大于相应的位置position，则map的key即为所求

* 开发步骤
  + 构建一个source，只是随机生成一个变量，网站的相应时间response\_time。

|  |
| --- |
| tenv.executeSql("CREATE TABLE source (\n" +  " response\_time INT,\n" +  " ts AS localtimestamp,\n" +  " WATERMARK FOR ts AS ts," +  "proctime as proctime()\n" +  ") WITH (\n" +  " 'connector' = 'datagen',\n" +  " 'rows-per-second'='1000',\n" +  " 'fields.response\_time.min'='1',\n" +  " 'fields.response\_time.max'='1000'" +  ")"); |

* + 定义一个聚合函数用的临时变量：

|  |
| --- |
| public static class TPAccum{  public Integer tp;  public Map<Integer,Integer> map = new HashMap<>(); } |

* + 实现自定义聚合函数类

|  |
| --- |
| public static class TP extends AggregateFunction<Integer,TPAccum>{   @Override  public TPAccum createAccumulator(){  return new TPAccum();  }   @Override  public Integer getValue(TPAccum acc){  if (acc.map.size() == 0){  return null;  } else {  Map<Integer,Integer> map = new TreeMap<>(acc.map);  int sum = map.values().stream().reduce(0, Integer::sum);   int tp = acc.tp;  int responseTime = 0;  int p = 0;  Double d = sum \* (tp / 100D);  for (Map.Entry<Integer,Integer> entry: map.entrySet()){  p += entry.getValue();  int position = d.intValue() - 1;  if (p >= position){  responseTime = entry.getKey();  break;  }   }  return responseTime;  }  }   public void accumulate(TPAccum acc, Integer iValue, Integer tp){  acc.tp = tp;  if (acc.map.containsKey(iValue)){  acc.map.put(iValue, acc.map.get(iValue) + 1);  } else {  acc.map.put(iValue, 1);  }  }  } |

* + 实际的查询sql如下：

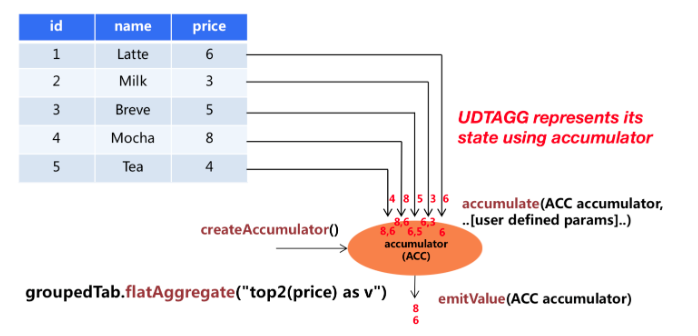
|  |
| --- |
| String sqlSelect =  "select TUMBLE\_START(proctime,INTERVAL '1' SECOND) as starttime,mytp(response\_time,50) from source" +  " group by TUMBLE(proctime,INTERVAL '1' SECOND)";  Table table = tenv.sqlQuery(sqlSelect); |

* 完整代码

|  |
| --- |
| /\*\*  \* 使用自定义聚合函数计算网站TP  \*/ public class UdafTP{  public static void main(String[] args) throws Exception{  final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  StreamTableEnvironment tenv = StreamTableEnvironment.create(env);  tenv.createTemporaryFunction("mytp", new TP());  tenv.executeSql("CREATE TABLE source (\n" +  " response\_time INT,\n" +  " ts AS localtimestamp,\n" +  " WATERMARK FOR ts AS ts," +  "proctime as proctime()\n" +  ") WITH (\n" +  " 'connector' = 'datagen',\n" +  " 'rows-per-second'='1000',\n" +  " 'fields.response\_time.min'='1',\n" +  " 'fields.response\_time.max'='1000'" +  ")");   String sqlSelect =  "select TUMBLE\_START(proctime,INTERVAL '1' SECOND) as starttime,mytp(response\_time,50) from source" +  " group by TUMBLE(proctime,INTERVAL '1' SECOND)";   Table table = tenv.sqlQuery(sqlSelect);  tenv.toAppendStream(table, Row.class).print();  env.execute();  }   public static class TP extends AggregateFunction<Integer,TPAccum>{   @Override  public TPAccum createAccumulator(){  return new TPAccum();  }   @Override  public Integer getValue(TPAccum acc){  if (acc.map.size() == 0){  return null;  } else {  Map<Integer,Integer> map = new TreeMap<>(acc.map);  int sum = map.values().stream().reduce(0, Integer::sum);   int tp = acc.tp;  int responseTime = 0;  int p = 0;  Double d = sum \* (tp / 100D);  for (Map.Entry<Integer,Integer> entry: map.entrySet()){  p += entry.getValue();  int position = d.intValue() - 1;  if (p >= position){  responseTime = entry.getKey();  break;  }   }  return responseTime;  }  }   public void accumulate(TPAccum acc, Integer iValue, Integer tp){  acc.tp = tp;  if (acc.map.containsKey(iValue)){  acc.map.put(iValue, acc.map.get(iValue) + 1);  } else {  acc.map.put(iValue, 1);  }  }   }   public static class TPAccum{  public Integer tp;  public Map<Integer,Integer> map = new HashMap<>();  }  } |

#### 表聚合函数（Table Aggregate Functions）

自定义表值聚合函数（UDTAGG）可以把一个表（一行或者多行，每行有一列或者多列）聚合成另一张表，结果中可以有多行多列。



上图展示了一个表值聚合函数的例子。假设你有一个饮料的表，这个表有 3 列，分别是 id、name 和 price，一共有 5 行。假设你需要找到价格最高的两个饮料，类似于 top2() 表值聚合函数。你需要遍历所有 5 行数据，结果是有 2 行数据的一个表。

用户自定义表值聚合函数是通过扩展 TableAggregateFunction 类来实现的。一个 TableAggregateFunction 的工作过程如下。首先，它需要一个 accumulator，这个 accumulator 负责存储聚合的中间结果。 通过调用 TableAggregateFunction 的 createAccumulator 方法来构造一个空的 accumulator。接下来，对于每一行数据，会调用 accumulate 方法来更新 accumulator。当所有数据都处理完之后，调用 emitValue 方法来计算和返回最终的结果。

下面几个 TableAggregateFunction 的方法是必须要实现的：

* createAccumulator()
* accumulate()

Flink 的类型推导在遇到复杂类型的时候可能会推导出错误的结果，比如那些非基本类型和普通的 POJO 类型的复杂类型。所以类似于 ScalarFunction 和 TableFunction，TableAggregateFunction 也提供了 TableAggregateFunction#getResultType() 和 TableAggregateFunction#getAccumulatorType() 方法来指定返回值类型和 accumulator 的类型，这两个方法都需要返回 TypeInformation。

除了上面的方法，还有几个其他的方法可以选择性的实现。有些方法可以让查询更加高效，而有些方法对于某些特定场景是必须要实现的。比如，在会话窗口（当两个会话窗口合并时会合并两个 accumulator）中使用聚合函数时，必须要实现merge() 方法。

下面几个 TableAggregateFunction 的方法在某些特定场景下是必须要实现的：

* retract() 在 bounded OVER 窗口中的聚合函数必须要实现。
* merge() 在许多批式聚合和会话窗口聚合中是必须要实现的。
* resetAccumulator() 在许多批式聚合中是必须要实现的。
* emitValue() 在批式聚合以及窗口聚合中是必须要实现的。

下面的 TableAggregateFunction 的方法可以提升流式任务的效率：

* emitUpdateWithRetract() 在 retract 模式下，该方法负责发送被更新的值。

emitValue 方法会发送所有 accumulator 给出的结果。拿 TopN 来说，emitValue 每次都会发送所有的最大的 n 个值。这在流式任务中可能会有一些性能问题。为了提升性能，用户可以实现 emitUpdateWithRetract 方法。这个方法在 retract 模式下会增量的输出结果，比如有数据更新了，我们必须要撤回老的数据，然后再发送新的数据。如果定义了 emitUpdateWithRetract 方法，那它会优先于 emitValue 方法被使用，因为一般认为 emitUpdateWithRetract 会更加高效，因为它的输出是增量的。

TableAggregateFunction 的所有方法都必须是 public 的、非 static 的，而且名字必须跟上面提到的一样。createAccumulator、getResultType 和 getAccumulatorType 这三个方法是在抽象父类 TableAggregateFunction 中定义的，而其他的方法都是约定的方法。要实现一个表值聚合函数，你必须扩展 org.apache.flink.table.functions.TableAggregateFunction，并且实现一个（或者多个）accumulate 方法。accumulate 方法可以有多个重载的方法，也可以支持变长参数。

**下面的例子展示了如何**

* 定义一个 TableAggregateFunction 来计算给定列的最大的 2 个值，
* 在 TableEnvironment 中注册函数，
* 在 Table API 查询中使用函数（当前只在 Table API 中支持 TableAggregateFunction）。

为了计算最大的 2 个值，accumulator 需要保存当前看到的最大的 2 个值。在我们的例子中，我们定义了类 Top2Accum 来作为 accumulator。Flink 的 checkpoint 机制会自动保存 accumulator，并且在失败时进行恢复，来保证精确一次的语义。

我们的 Top2 表值聚合函数（TableAggregateFunction）的 accumulate() 方法有两个输入，第一个是 Top2Accum accumulator，另一个是用户定义的输入：输入的值 v。尽管 merge() 方法在大多数聚合类型中不是必须的，我们也在样例中提供了它的实现。请注意，我们在 Scala 样例中也使用的是 Java 的基础类型，并且定义了 getResultType() 和 getAccumulatorType() 方法，因为 Flink 的类型推导对于 Scala 的类型推导支持的不是很好。

|  |
| --- |
| public class CustomTableAggregateFunction {  public static void main(String[] args) throws Exception {  //1.构建运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1); // 设置并行度为1  //2.构建TableEnv  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);   String filePath = TableApiTest.class.getClassLoader().getResource("datas.csv").getPath();  //3.构建数据源  tableEnv.connect(new FileSystem().path(filePath))  .withFormat(new Csv())  .withSchema(new Schema()  .field("id", DataTypes.STRING())  .field("name", DataTypes.STRING())  .field("price", DataTypes.INT())  ).createTemporaryTable("datas");  Table table = tableEnv.from("datas");  tableEnv.createTemporaryFunction("top2", new Top2());   Table result = table.groupBy($("id"))  .flatAggregate("top2(price) as (v, rank)")  .select("id, v, rank");    tableEnv.toRetractStream(result, Row.class).print("result");  env.execute("FlinkSqlUDFTableAggregateFunction");  }   public static class Top2Accum implements Serializable {  public Integer first;  public Integer second;  }   public static class Top2 extends TableAggregateFunction<Tuple2<Integer, Integer>, Top2Accum> {   @Override  public Top2Accum createAccumulator() {  Top2Accum acc = new Top2Accum();  acc.first = Integer.MIN\_VALUE;  acc.second = Integer.MIN\_VALUE;  return acc;  }    public void accumulate(Top2Accum acc, Integer v) {  if (v > acc.first) {  acc.second = acc.first;  acc.first = v;  } else if (v > acc.second) {  acc.second = v;  }  }   public void merge(Top2Accum acc, java.lang.Iterable<Top2Accum> iterable) {  for (Top2Accum otherAcc : iterable) {  accumulate(acc, otherAcc.first);  accumulate(acc, otherAcc.second);  }  }   public void emitValue(Top2Accum acc, Collector<Tuple2<Integer, Integer>> out) {  // emit the value and rank  if (acc.first != Integer.MIN\_VALUE) {  out.collect(Tuple2.of(acc.first, 1));  }  if (acc.second != Integer.MIN\_VALUE) {  out.collect(Tuple2.of(acc.second, 2));  }  }  } } |

下面的例子展示了如何使用 emitUpdateWithRetract 方法来只发送更新的数据。为了只发送更新的结果，accumulator 保存了上一次的最大的2个值，也保存了当前最大的2个值。注意：如果 TopN 中的 n 非常大，这种既保存上次的结果，也保存当前的结果的方式不太高效。一种解决这种问题的方式是把输入数据直接存储到 accumulator 中，然后在调用 emitUpdateWithRetract 方法时再进行计算。

|  |
| --- |
| public class CustomTableAggregateFunction2 {  public static void main(String[] args) throws Exception {  //1.构建运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1); // 设置并行度为1  //2.构建TableEnv  StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);   String filePath = TableApiTest.class.getClassLoader().getResource("datas.csv").getPath();  //3.构建数据源  tableEnv.connect(new FileSystem().path(filePath))  .withFormat(new Csv())  .withSchema(new Schema()  .field("id", DataTypes.STRING())  .field("name", DataTypes.STRING())  .field("price", DataTypes.INT())  ).createTemporaryTable("datas");  Table table = tableEnv.from("datas");  tableEnv.createTemporaryFunction("top2", new Top2());   Table result = table.groupBy($("id"))  .flatAggregate("top2(price) as (v, rank)")  .select("id, v, rank");    tableEnv.toRetractStream(result, Row.class).print("result");  env.execute("FlinkSqlUDFTableAggregateFunction");  }   /\*\*  \* Accumulator for Top2.  \*/  public static class Top2Accum {  public Integer first;  public Integer second;  public Integer oldFirst;  public Integer oldSecond;  }   /\*\*  \* The top2 user-defined table aggregate function.  \*/  public static class Top2 extends TableAggregateFunction<Tuple2<Integer, Integer>, Top2Accum> {   @Override  public Top2Accum createAccumulator() {  Top2Accum acc = new Top2Accum();  acc.first = Integer.MIN\_VALUE;  acc.second = Integer.MIN\_VALUE;  acc.oldFirst = Integer.MIN\_VALUE;  acc.oldSecond = Integer.MIN\_VALUE;  return acc;  }   public void accumulate(Top2Accum acc, Integer v) {  if (v > acc.first) {  acc.second = acc.first;  acc.first = v;  } else if (v > acc.second) {  acc.second = v;  }  }   public void emitUpdateWithRetract(Top2Accum acc, RetractableCollector<Tuple2<Integer, Integer>> out) {  if (!acc.first.equals(acc.oldFirst)) {  // if there is an update, retract old value then emit new value.  if (acc.oldFirst != Integer.MIN\_VALUE) {  out.retract(Tuple2.of(acc.oldFirst, 1));  }  out.collect(Tuple2.of(acc.first, 1));  acc.oldFirst = acc.first;  }   if (!acc.second.equals(acc.oldSecond)) {  // if there is an update, retract old value then emit new value.  if (acc.oldSecond != Integer.MIN\_VALUE) {  out.retract(Tuple2.of(acc.oldSecond, 2));  }  out.collect(Tuple2.of(acc.second, 2));  acc.oldSecond = acc.second;  }  }   public void emitValue(Top2Accum acc, Collector<Tuple2<Integer, Integer>> out) {  // emit the value and rank  if (acc.first != Integer.MIN\_VALUE) {  out.collect(Tuple2.of(acc.first, 1));  }  if (acc.second != Integer.MIN\_VALUE) {  out.collect(Tuple2.of(acc.second, 2));  }  }  } } |

## 连接到外部系统（扩展）

Flink 的 Table API 和 SQL 程序可以连接到其他外部系统，用于读写批处理和流表。表源提供对存储在外部系统（例如数据库、键值存储、消息队列或文件系统）中的数据的访问。表接收器将表发送到外部存储系统。根据源和接收器的类型，它们支持不同的格式，例如 CSV、Avro、Parquet 或 ORC。

本页介绍如何使用本机支持的连接器在Flink中注册表源和表接收器。注册源或接收器后，可以通过 Table API 和 SQL 语句访问它。

### Hive

[Apache Hive](https://hive.apache.org/) 已经成为了数据仓库生态系统中的核心。 它不仅仅是一个用于大数据分析和ETL场景的SQL引擎，同样它也是一个数据管理平台，可用于发现，定义，和演化数据。

Flink 与 Hive 的集成包含两个层面。

一是利用了 Hive 的 MetaStore 作为持久化的 Catalog，用户可通过HiveCatalog将不同会话中的 Flink 元数据存储到 Hive Metastore 中。 例如，用户可以使用HiveCatalog将其 Kafka 表或 Elasticsearch 表存储在 Hive Metastore 中，并后续在 SQL 查询中重新使用它们。

二是利用 Flink 来读写 Hive 的表。

HiveCatalog的设计提供了与 Hive 良好的兼容性，用户可以"开箱即用"的访问其已有的 Hive 数仓。 您不需要修改现有的 Hive Metastore，也不需要更改表的数据位置或分区。

**强烈建议用户使用 [Blink planner](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/overview/" \l "dependency-structure) 与 Hive 集成。**

官网地址：**<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/overview/>**

#### 支持的Hive版本

Flink 支持一下的 Hive 版本。

|  |  |
| --- | --- |
| **Hive大版本号** | **Hive小版本号** |
| 1.0 | 1.0.0、1.0.1 |
| 1.1 | 1.1.0、1.1.1 |
| 1.2 | 1.2.0、1.2.1、1.2.2 |
| 2.0 | 2.0.0、2.0.1 |
| 2.1 | 2.1.0、2.1.1 |
| 2.2 | 2.2.0 |
| 2.3 | 2.3.0、2.3.1、2.3.2、2.3.3、2.3.4、2.3.5、2.3.6 |
| 3.1 | 3.1.0、3.1.1、3.1.2 |

请注意，某些功能是否可用取决于您使用的 Hive 版本，这些限制不是由 Flink 所引起的：

* Hive 内置函数在使用 Hive-1.2.0 及更高版本时支持。
* 列约束，也就是 PRIMARY KEY 和 NOT NULL，在使用 Hive-3.1.0 及更高版本时支持。
* 更改表的统计信息，在使用 Hive-1.2.0 及更高版本时支持。
* DATE列统计信息，在使用 Hive-1.2.0 及更高版时支持。
* 使用 Hive-2.0.x 版本时不支持写入 ORC 表。

##### 依赖项

要与 Hive 集成，您需要在 Flink 下的/lib/目录中添加一些额外的依赖包， 以便通过 Table API 或 SQL Client 与 Hive 进行交互。 或者，您可以将这些依赖项放在专用文件夹中，并分别使用 Table API 程序或 SQL Client 的-C或-l选项将它们添加到 classpath 中。

Apache Hive 是基于 Hadoop 之上构建的, 首先您需要 Hadoop 的依赖，请参考 Providing Hadoop classes:

|  |
| --- |
| export HADOOP\_CLASSPATH=`hadoop classpath` |

有两种添加 Hive 依赖项的方法。第一种是使用 Flink 提供的 Hive Jar包。您可以根据使用的 Metastore 的版本来选择对应的 Hive jar。第二个方式是分别添加每个所需的 jar 包。如果您使用的 Hive 版本尚未在此处列出，则第二种方法会更适合。

|  |
| --- |
| **注意：**建议您优先使用 Flink 提供的 Hive jar 包。仅在 Flink 提供的 Hive jar 不满足您的需求时，再考虑使用分开添加 jar 包的方式。 |

**使用 Flink 提供的 Hive jar**

下表列出了所有可用的 Hive jar。您可以选择一个并放在 Flink 发行版的/lib/ 目录中。

|  |  |  |
| --- | --- | --- |
| **Metastore version** | **Maven dependency** | **SQL Client JAR** |
| 1.0.0 - 1.2.2 | flink-sql-connector-hive-1.2.2 | [Download](https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-hive-1.2.2_2.11/1.13.0/flink-sql-connector-hive-1.2.2_2.11-1.13.0.jar) |
| 2.0.0 - 2.2.0 | flink-sql-connector-hive-2.2.0 | [Download](https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-hive-2.2.0_2.11/1.13.0/flink-sql-connector-hive-2.2.0_2.11-1.13.0.jar) |
| 2.3.0 - 2.3.6 | flink-sql-connector-hive-2.3.6 | [Download](https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-hive-2.3.6_2.11/1.13.0/flink-sql-connector-hive-2.3.6_2.11-1.13.0.jar) |
| 3.0.0 - 3.1.2 | flink-sql-connector-hive-3.1.2 | [Download](https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-hive-3.1.2_2.11/1.13.0/flink-sql-connector-hive-3.1.2_2.11-1.13.0.jar) |

**用户定义的依赖项**

参考官网：**[https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/overview/#%E4%BE%9D%E8%B5%96%E9%A1%B9](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/overview/" \l "%E4%BE%9D%E8%B5%96%E9%A1%B9)**

以Hive2.1.1为例。

|  |  |
| --- | --- |
| **Hive version** | **依赖项** |
| Hive 2.1.1 | /flink-1.13.0  /lib  // Flink's Hive connector  flink-connector-hive\_2.11-1.13.0.jar  // Hive dependencies  hive-exec-2.1.0.jar  // add antlr-runtime if you need to use hive dialect  antlr-runtime-3.5.2.jar |

##### Maven 依赖

如果您在构建自己的应用程序，则需要在 mvn 文件中添加以下依赖项。 您应该在运行时添加以上的这些依赖项，而不要在已生成的 jar 文件中去包含它们。

|  |
| --- |
| <!-- Flink Dependency --> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-hive{{ site.scala\_version\_suffix }}</artifactId>  <version>{{site.version}}</version>  <scope>provided</scope> </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-table-api-java-bridge{{ site.scala\_version\_suffix }}</artifactId>  <version>{{site.version}}</version>  <scope>provided</scope> </dependency>  <!-- Hive Dependency --> <dependency>  <groupId>org.apache.hive</groupId>  <artifactId>hive-exec</artifactId>  <version>${hive.version}</version>  <scope>provided</scope> </dependency> |

#### 连接到Hive

通过 TableEnvironment 或者 YAML 配置，使用 [Catalog 接口](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/catalogs/) 和 [HiveCatalog](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/hive_catalog/)连接到现有的 Hive 集群。

请注意，虽然 HiveCatalog 不需要特定的 planner，但读写Hive表仅适用于 Blink planner。因此，强烈建议您在连接到 Hive 仓库时使用 Blink planner。

以下是如何连接到 Hive 的示例：

|  |
| --- |
| EnvironmentSettings settings = EnvironmentSettings.newInstance().useBlinkPlanner().build(); TableEnvironment tableEnv = TableEnvironment.create(settings);  String name = "myhive"; String defaultDatabase = "mydatabase"; String hiveConfDir = "/opt/hive-conf";  HiveCatalog hive = new HiveCatalog(name, defaultDatabase, hiveConfDir); tableEnv.registerCatalog("myhive", hive);  // set the HiveCatalog as the current catalog of the session tableEnv.useCatalog("myhive"); |

下表列出了通过 YAML 文件或 DDL 定义 HiveCatalog 时所支持的参数。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **必选** | **默认值** | **类型** | **描述** |
| type | 是 | (无) | String | Catalog 的类型。 创建 HiveCatalog 时，该参数必须设置为'hive'。 |
| name | 是 | (无) | String | Catalog 的名字。仅在使用 YAML file 时需要指定。 |
| hive-conf-dir | 否 | (无) | String | 指向包含 hive-site.xml 目录的 URI。 该 URI 必须是 Hadoop 文件系统所支持的类型。 如果指定一个相对 URI，即不包含 scheme，则默认为本地文件系统。如果该参数没有指定，我们会在 class path 下查找hive-site.xml。 |
| default-database | 否 | default | String | 当一个catalog被设为当前catalog时，所使用的默认当前database。 |
| hive-version | 否 | (无) | String | HiveCatalog 能够自动检测使用的 Hive 版本。我们建议不要手动设置 Hive 版本，除非自动检测机制失败。 |
| hadoop-conf-dir | 否 | (无) | String | Hadoop 配置文件目录的路径。目前仅支持本地文件系统路径。我们推荐使用 HADOOP\_CONF\_DIR 环境变量来指定 Hadoop 配置。因此仅在环境变量不满足您的需求时再考虑使用该参数，例如当您希望为每个 HiveCatalog 单独设置 Hadoop 配置时。 |

#### Hive Catalog

多年来，Hive Metastore 已经发展成为 Hadoop 生态系统中事实上的元数据中心。许多公司在其生产中使用单个 Hive Metastore 服务实例来管理其所有元数据，无论是 Hive 元数据还是非 Hive 元数据。

对于同时拥有 Hive 和 Flink 部署的用户，HiveCatalog使他们能够使用 Hive Metastore 来管理 Flink 的元数据。

对于刚刚部署 Flink 的用户，是 FlinkHiveCatalog提供的唯一一个开箱即用的持久目录。在没有持久化目录的情况下，使用[Flink SQL CREATE DDL 的](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/create/)用户必须在每个会话中重复创建元对象，例如 Kafka 表，这浪费了大量时间。HiveCatalog通过授权用户仅创建一次表和其他元对象，并在以后跨会话方便地引用和管理它们来填补这一空白。

##### 如何使用 HiveCatalog

用户可以使用 DDL 创建 Flink 元对象，然后应该立即看到它们。

HiveCatalog可用于处理两种表：Hive 兼容表和通用表。就存储层中的元数据和数据而言，兼容Hive的表是以兼容Hive的方式存储的表。因此，可以从 Hive 端查询通过 Flink 创建的 Hive 兼容表。

另一方面，通用表是 Flink 特有的。使用 来创建通用表时HiveCatalog，我们只是使用 HMS 来持久化元数据。虽然这些表对 Hive 可见，但 Hive 不太可能理解元数据。因此，在Hive中使用此类表会导致未定义的行为。

Flink 使用属性“ is\_generic ”来判断表是 Hive 兼容的还是通用的。使用 来创建表时 HiveCatalog，默认情况下将其视为通用表。如果您想创建一个与 Hive 兼容的表，请确保is\_generic在您的表属性中设置 为 false。

如上所述，不应在 Hive 中使用通用表。在 Hive CLI 中，您可以调用DESCRIBE FORMATTED一个表并通过检查is\_generic属性来决定它是否是通用的。通用表将具有is\_generic=true.

##### 支持的类型

HiveCatalog 支持泛型表的所有 Flink 类型。

对于 Hive 兼容的表，HiveCatalog需要将 Flink 数据类型映射到对应的 Hive 类型，如下表所示：

|  |  |
| --- | --- |
| **Flink Data Type** | **Hive Data Type** |
| CHAR(p) | CHAR(p) |
| VARCHAR(p) | VARCHAR(p) |
| STRING | STRING |
| BOOLEAN | BOOLEAN |
| TINYINT | TINYINT |
| SMALLINT | SMALLINT |
| INT | INT |
| BIGINT | LONG |
| FLOAT | FLOAT |
| DOUBLE | DOUBLE |
| DECIMAL(p, s) | DECIMAL(p, s) |
| DATE | DATE |
| TIMESTAMP(9) | TIMESTAMP |
| BYTES | BINARY |
| ARRAY<T> | LIST<T> |
| MAP | MAP |
| ROW | STRUCT |

关于类型映射的一些注意事项：

* Hive 的CHAR(p)最大长度为 255
* Hive 的VARCHAR(p)最大长度为 65535
* HiveMAP只支持原始键类型，而 FlinkMAP可以是任何数据类型
* UNION不支持Hive 的类型
* Hive 的TIMESTAMP精度始终为 9，不支持其他精度。另一方面，Hive UDF 可以处理TIMESTAMP精度 <= 9 的值。
* 蜂房不支持弗林克的TIMESTAMP\_WITH\_TIME\_ZONE，TIMESTAMP\_WITH\_LOCAL\_TIME\_ZONE和MULTISET
* Flink 的INTERVAL类型还不能映射到 HiveINTERVAL类型

#### [Hive Dialect](https://www.cnblogs.com/qiu-hua/p/14054118.html)

从**Flink 1.11.0** 开始，在使用 Hive dialect时，Flink 允许用户用 Hive 语法来编写 SQL 语句。通过提供与 Hive 语法的兼容性，我们旨在改善与 Hive 的互操作性，并减少用户需要在 Flink 和 Hive 之间切换来执行不同语句的情况。

##### 使用 Hive Dialect

Flink 目前支持两种 SQL 方言: default 和 hive。你需要先切换到 Hive Dialect，然后才能使用 Hive 语法编写。下面介绍如何使用 SQL 客户端和 Table API 设置方言。

还要注意，你可以为执行的每个语句动态切换方言。无需重新启动会话即可使用其他方言。

###### SQL 客户端

SQL 方言可以通过 table.sql-dialect 属性指定。因此你可以通过 SQL 客户端 yaml 文件中的 configuration 部分来设置初始方言。

|  |
| --- |
| execution: planner: blink type: batch result-mode: table  configuration: table.sql-dialect: hive |

你同样可以在 SQL 客户端启动后设置方言。

|  |
| --- |
| Flink SQL> set table.sql-dialect=hive; -- to use hive dialect [INFO] Session property has been set.  Flink SQL> set table.sql-dialect=default; -- to use default dialect [INFO] Session property has been set. |

###### Table API

你可以使用 Table API 为 TableEnvironment 设置方言。

|  |
| --- |
| EnvironmentSettings settings = EnvironmentSettings.newInstance().useBlinkPlanner()...build(); TableEnvironment tableEnv = TableEnvironment.create(settings); // to use hive dialect tableEnv.getConfig().setSqlDialect(SqlDialect.HIVE); // to use default dialect tableEnv.getConfig().setSqlDialect(SqlDialect.DEFAULT); |

##### DDL

本章节列出了 Hive 方言支持的 DDL 语句。我们主要关注语法。你可以参考[Hive文档](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL)了解每个 DDL 语句的语义。

###### CATALOG

* Show

|  |
| --- |
| SHOW CURRENT CATALOG; |

###### DATABASE

* Show

|  |
| --- |
| SHOW DATABASES; |

* Create

|  |
| --- |
| CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database\_name  [COMMENT database\_comment]  [LOCATION fs\_path]  [WITH DBPROPERTIES (property\_name=property\_value, ...)]; |

* Alter

|  |
| --- |
| **更新属性：** |
| ALTER (DATABASE|SCHEMA) database\_name SET DBPROPERTIES (property\_name=property\_value, ...); |
| **更新所有者：** |
| ALTER (DATABASE|SCHEMA) database\_name SET OWNER [USER|ROLE] user\_or\_role; |
| **更新位置：** |
| ALTER (DATABASE|SCHEMA) database\_name SET LOCATION fs\_path; |

* Drop

|  |
| --- |
| DROP (DATABASE|SCHEMA) [IF EXISTS] database\_name [RESTRICT|CASCADE]; |

* Use

|  |
| --- |
| USE database\_name; |

###### TABLE

* Show

|  |
| --- |
| SHOW TABLES; |

* Create

|  |
| --- |
| CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table\_name  [(col\_name data\_type [column\_constraint] [COMMENT col\_comment], ... [table\_constraint])]  [COMMENT table\_comment]  [PARTITIONED BY (col\_name data\_type [COMMENT col\_comment], ...)]  [  [ROW FORMAT row\_format]  [STORED AS file\_format]  ]  [LOCATION fs\_path]  [TBLPROPERTIES (property\_name=property\_value, ...)]  row\_format:  : DELIMITED [FIELDS TERMINATED BY char [ESCAPED BY char]] [COLLECTION ITEMS TERMINATED BY char]  [MAP KEYS TERMINATED BY char] [LINES TERMINATED BY char]  [NULL DEFINED AS char]  | SERDE serde\_name [WITH SERDEPROPERTIES (property\_name=property\_value, ...)]  file\_format:  : SEQUENCEFILE  | TEXTFILE  | RCFILE  | ORC  | PARQUET  | AVRO  | INPUTFORMAT input\_format\_classname OUTPUTFORMAT output\_format\_classname  column\_constraint:  : NOT NULL [[ENABLE|DISABLE] [VALIDATE|NOVALIDATE] [RELY|NORELY]]  table\_constraint:  : [CONSTRAINT constraint\_name] PRIMARY KEY (col\_name, ...) [[ENABLE|DISABLE] [VALIDATE|NOVALIDATE] [RELY|NORELY]] |

* Alter

|  |
| --- |
| **重命名：** |
| ALTER TABLE table\_name RENAME TO new\_table\_name; |
| **更新属性：** |
| ALTER TABLE table\_name SET TBLPROPERTIES (property\_name = property\_value, property\_name = property\_value, ... ); |
| **更新位置：** |
| ALTER TABLE table\_name [PARTITION partition\_spec] SET LOCATION fs\_path; |
| 如果指定了 partition\_spec，那么必须完整，即具有所有分区列的值。如果指定了，该操作将作用在对应分区上而不是表上。 |
| **更新文件格式：** |
| ALTER TABLE table\_name [PARTITION partition\_spec] SET FILEFORMAT file\_format; |
| 如果指定了 partition\_spec，那么必须完整，即具有所有分区列的值。如果指定了，该操作将作用在对应分区上而不是表上。 |
| **更新SerDe属性：** |
| ALTER TABLE table\_name [PARTITION partition\_spec] SET SERDE serde\_class\_name [WITH SERDEPROPERTIES serde\_properties];  ALTER TABLE table\_name [PARTITION partition\_spec] SET SERDEPROPERTIES serde\_properties;  serde\_properties:  : (property\_name = property\_value, property\_name = property\_value, ... ) |
| 如果指定了 partition\_spec，那么必须完整，即具有所有分区列的值。如果指定了，该操作将作用在对应分区上而不是表上。 |
| **添加分区：** |
| ALTER TABLE table\_name ADD [IF NOT EXISTS] (PARTITION partition\_spec [LOCATION fs\_path])+; |
| **删除分区：** |
| ALTER TABLE table\_name DROP [IF EXISTS] PARTITION partition\_spec[, PARTITION partition\_spec, ...]; |
| **添加/替换列：** |
| ALTER TABLE table\_name  ADD|REPLACE COLUMNS (col\_name data\_type [COMMENT col\_comment], ...)  [CASCADE|RESTRICT] |
| **更改列：** |
| ALTER TABLE table\_name CHANGE [COLUMN] col\_old\_name col\_new\_name column\_type  [COMMENT col\_comment] [FIRST|AFTER column\_name] [CASCADE|RESTRICT]; |

* Drop

|  |
| --- |
| DROP TABLE [IF EXISTS] table\_name; |

###### VIEW

* Create

|  |
| --- |
| CREATE VIEW [IF NOT EXISTS] view\_name [(column\_name, ...) ]  [COMMENT view\_comment]  [TBLPROPERTIES (property\_name = property\_value, ...)]  AS SELECT ...; |

* Alter

|  |
| --- |
| **重命名：** |
| ALTER VIEW view\_name RENAME TO new\_view\_name; |
| 注意: 变更视图只在 Table API 中有效，SQL 客户端不支持。 |
| **更新属性：** |
| ALTER VIEW view\_name SET TBLPROPERTIES (property\_name = property\_value, ... ); |
| **更新为选择：** |
| ALTER VIEW view\_name AS select\_statement; |

* Drop

|  |
| --- |
| DROP VIEW [IF EXISTS] view\_name; |

###### FUNCTION

* Show

|  |
| --- |
| SHOW FUNCTIONS; |

* Create

|  |
| --- |
| CREATE FUNCTION function\_name AS class\_name; |

* Drop

|  |
| --- |
| DROP FUNCTION [IF EXISTS] function\_name; |

##### DML & DQL Beta

Hive 方言支持常用的 Hive [DML](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML) 和 [DQL](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Select) 。 下表列出了一些 Hive 方言支持的语法。

* [SORT/CLUSTER/DISTRIBUTE BY](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+SortBy)
* [Group By](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+GroupBy)
* [Join](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Joins)
* [Union](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Union)
* [LATERAL VIEW](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+LateralView)
* [Window Functions](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+WindowingAndAnalytics)
* [SubQueries](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+SubQueries)
* [CTE](https://cwiki.apache.org/confluence/display/Hive/Common+Table+Expression)
* [INSERT INTO dest schema](https://issues.apache.org/jira/browse/HIVE-9481)
* [Implicit type conversions](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types" \l "LanguageManualTypes-AllowedImplicitConversions)

为了实现更好的语法和语义的兼容，强烈建议使用 [HiveModule](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/hive_functions/" \l "use-hive-built-in-functions-via-hivemodule) 并将其放在 Module 列表的首位，以便在函数解析时优先使用 Hive 内置函数。

Hive 方言不再支持 [Flink SQL 语法](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/) 。 若需使用 Flink 语法，请切换到 default 方言。

以下是一个使用 Hive 方言的示例。

|  |
| --- |
| Flink SQL> create catalog myhive with ('type' = 'hive', 'hive-conf-dir' = '/opt/hive-conf'); [INFO] Execute statement succeed.  Flink SQL> use catalog myhive; [INFO] Execute statement succeed.  Flink SQL> load module hive; [INFO] Execute statement succeed.  Flink SQL> use modules hive,core; [INFO] Execute statement succeed.  Flink SQL> set table.sql-dialect=hive; [INFO] Session property has been set.  Flink SQL> select explode(array(1,2,3)); -- call hive udtf +-----+ | col | +-----+ | 1 | | 2 | | 3 | +-----+ 3 rows in set  Flink SQL> create table tbl (key int,value string); [INFO] Execute statement succeed.  Flink SQL> insert overwrite table tbl values (5,'e'),(1,'a'),(1,'a'),(3,'c'),(2,'b'),(3,'c'),(3,'c'),(4,'d'); [INFO] Submitting SQL update statement to the cluster... [INFO] SQL update statement has been successfully submitted to the cluster:  Flink SQL> select \* from tbl cluster by key; -- run cluster by 2021-04-22 16:13:57,005 INFO org.apache.hadoop.mapred.FileInputFormat [] - Total input paths to process : 1 +-----+-------+ | key | value | +-----+-------+ | 1 | a | | 1 | a | | 5 | e | | 2 | b | | 3 | c | | 3 | c | | 3 | c | | 4 | d | +-----+-------+ 8 rows in set |

##### 注意

以下是使用 Hive 方言的一些注意事项。

* Hive 方言只能用于操作 Hive 对象，并要求当前 Catalog 是一个 [HiveCatalog](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/hive_catalog/) 。
* Hive 方言只支持 db.table 这种两级的标识符，不支持带有 Catalog 名字的标识符。
* 虽然所有 Hive 版本支持相同的语法，但是一些特定的功能是否可用仍取决于你使用的[Hive 版本](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/overview/" \l "%e6%94%af%e6%8c%81%e7%9a%84hive%e7%89%88%e6%9c%ac)。例如，更新数据库位置 只在 Hive-2.4.0 或更高版本支持。
* 执行 DML 和 DQL 时应该使用 [HiveModule](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/hive_functions/" \l "use-hive-built-in-functions-via-hivemodule) 。

#### Hive Read & Write

使用HiveCatalog，Apache Flink 可以用于Apache Hive Tables 的统一BATCH和STREAM处理。这意味着 Flink 可以用作 Hive 批处理引擎的性能更高的替代品，或者连续地从 Hive 表中读取和写入数据，以支持实时数据仓库应用程序。

##### 从Hive读取数据

Apache Flink 可以用于Apache Hive Tables 的统一BATCH和STREAM处理。

当作为BATCH 应用程序运行时，Flink 将在执行查询的时间点对表的状态执行其查询。

STREAMING读取将持续监视表并在新数据可用时增量获取新数据。

默认情况下，Flink 将读取有界的表。

STREAMING读取消耗已分区表和未分区表的支持。对于分区表，Flink将监视新分区的生成，并在可用时以增量方式读取它们。对于非分区表，Flink 会监控文件夹中新文件的生成，并增量读取新文件。

|  |  |  |  |
| --- | --- | --- | --- |
| **配置项** | **默认值** | **类型** | **描述** |
| streaming-source.enable | false | Boolean | 是否启用流数据源。注意：请确保每个分区/文件都应该被原子写入，否则读取可能会得到不完整的数据。 |
| streaming-source.partition.include | all | String | 选项设置要读取的分区，支持的选项有`all`和`latest`，`all`表示读取所有分区；“latest”表示按“streaming-source.partition.order”的顺序读取最新分区，“latest”仅在流式hive源表用作临时表时才有效。默认情况下，该选项为 `all`。Flink通过启用'streaming-source.enable'并将'streaming-source.partition.include'设置为'latest'来支持临时加入最新的hive分区，同时用户可以通过分配分区比较顺序和数据更新间隔配置以下分区相关的选项。 |
| streaming-source.monitor-interval | None | Duration | 连续监控分区/文件的时间间隔。注意：hive流读取的默认间隔为'1 m'，hive流临时连接的默认间隔为'60 m'，这是因为存在一个框架限制，即每个TM都会在当前hive流临时连接中访问Hive metaStore可能对metaStore产生压力的实现，这将在未来改进。 |
| streaming-source.partition-order | partition-name | String | 流源分区顺序，支持create-time、partition-time、partition-name。create-time 比较分区/文件创建时间，这不是 Hive metaStore 中的分区创建时间，而是文件系统中的文件夹/文件修改时间，如果分区文件夹以某种方式更新，例如将新文件添加到文件夹中，它会影响如何数据被消耗掉了。partition-time 比较从分区名中提取的时间。partition-name 比较分区名称的字母顺序。对于非分区表，此值应始终为“ create-time”。默认情况下，该值为分区名称。该选项与不推荐使用的选项“streaming-source.consume-order”相同。 |
| streaming-source.consume-start-offset | None | String | 流消费的开始偏移量。如何解析和比较偏移量取决于您的订单。对于创建时间和分区时间，应该是时间戳字符串 (yyyy-[m]m-[d]d [hh:mm:ss])。对于分区时间，将使用分区时间提取器从分区中提取时间。对于 partition-name，是分区名称字符串（例如 pt\_year=2020/pt\_mon=10/pt\_day=01） |

[SQL提示](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/hints/)可用于将配置应用于Hive表，而无需在Hive元存储中更改其定义。

|  |
| --- |
| SELECT \*  FROM hive\_table  /\*+ OPTIONS('streaming-source.enable'='true', 'streaming-source.consume-start-offset'='2020-05-20') \*/; |

|  |
| --- |
| **注意：**   * 监控策略是扫描当前位置路径中的所有目录/文件。许多分区可能会导致性能下降。 * 非分区表的流式读取要求将每个文件自动写入目标目录。 * 分区表的流式读取需要在 hive Metastore 的视图中原子地添加每个分区。如果没有，将消耗添加到现有分区的新数据。 * 流式读取不支持 Flink DDL 中的水印语法。这些表不能用于窗口操作符。 |

###### Reading Hive Views

Flink能够从Hive定义的视图中读取内容，但存在一些限制：

* 必须先将 Hive 目录设置为当前目录，然后才能查询视图。这可以通过tableEnv.useCatalog(...)Table API 或USE CATALOG ...SQL Client 来完成。
* Hive 和 Flink SQL 有不同的语法，例如不同的保留关键字和文字。确保视图的查询与 Flink 语法兼容。

###### Vectorized Optimization upon Read

当满足以下条件时，Flink 将自动使用 Hive 表的向量化读取：

* 格式：ORC 或 Parquet。
* 没有复杂数据类型的列，如 hive 类型：List、Map、Struct、Union。

默认情况下启用此功能。可以通过以下配置禁用它。

|  |
| --- |
| table.exec.hive.fallback-mapred-reader=true |

###### Source Parallelism Inference

默认情况下，Flink 将根据文件数和每个文件中的blocks 数推断其 Hive 读取器的最佳并行度。

Flink 允许您灵活配置并行推理的策略。您可以在TableConfig（注意这些参数影响作业的所有来源）中配置以下参数：

|  |  |  |  |
| --- | --- | --- | --- |
| **配置项** | **默认值** | **类型** | **描述** |
| table.exec.hive.infer-source-parallelism | true | Boolean | 如果为 true，则根据拆分数推断源并行度。如果为 false，则源的并行度由配置设置。 |
| table.exec.hive.infer-source-parallelism.max | 1000 | 整数 | 为源运算符设置最大推断并行度。 |

##### 时态表 Join

您可以将 Hive 表用作时态表，然后流可以通过时态join关联 Hive 表。有关[临时连接](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "temporal-joins)的更多信息，请参见[临时join](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "temporal-joins)。

Flink 支持 processing-time temporal join Hive Table，processing-time temporal join 总是加入最新版本的 temporal table。Flink 支持分区表和 Hive 非分区表的临时连接，对于分区表，Flink 支持自动跟踪 Hive 表的最新分区。

**注意：Flink 尚不支持 event-time temporal join Hive table。**

###### [Temporal Join The Latest Partition](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/hive_read_write/" \l "temporal-join-the-latest-partition)

对于随时间变化的分区表，我们可以将其作为无界流读出，如果每个分区都包含一个版本的完整数据，则该分区可以充当时态表的一个版本，时态表的版本保留数据的分区。

Flink 支持在处理时间临时连接时自动跟踪临时表的最新分区（版本），最新的分区（版本）由 'streaming-source.partition-order' 选项定义，这是使用 Hive 表最常见的用例作为 Flink 流应用程序作业中的维度表。

**注意：此功能仅在 FlinkSTREAMING模式下支持。**

下面的demo展示了一个经典的业务管道，维度表来自Hive，每天通过批处理管道作业或Flink作业更新一次，kafka流来自实时在线业务数据或日志，需要加入维度表来丰富流。

|  |
| --- |
| -- Assume the data in hive table is updated per day, every day contains the latest and complete dimension data SET table.sql-dialect=hive; CREATE TABLE dimension\_table (  product\_id STRING,  product\_name STRING,  unit\_price DECIMAL(10, 4),  pv\_count BIGINT,  like\_count BIGINT,  comment\_count BIGINT,  update\_time TIMESTAMP(3),  update\_user STRING,  ... ) PARTITIONED BY (pt\_year STRING, pt\_month STRING, pt\_day STRING) TBLPROPERTIES (  -- using default partition-name order to load the latest partition every 12h (the most recommended and convenient way)  'streaming-source.enable' = 'true',  'streaming-source.partition.include' = 'latest',  'streaming-source.monitor-interval' = '12 h',  'streaming-source.partition-order' = 'partition-name', -- option with default value, can be ignored.   -- using partition file create-time order to load the latest partition every 12h  'streaming-source.enable' = 'true',  'streaming-source.partition.include' = 'latest',  'streaming-source.partition-order' = 'create-time',  'streaming-source.monitor-interval' = '12 h'   -- using partition-time order to load the latest partition every 12h  'streaming-source.enable' = 'true',  'streaming-source.partition.include' = 'latest',  'streaming-source.monitor-interval' = '12 h',  'streaming-source.partition-order' = 'partition-time',  'partition.time-extractor.kind' = 'default',  'partition.time-extractor.timestamp-pattern' = '$pt\_year-$pt\_month-$pt\_day 00:00:00'  );  SET table.sql-dialect=default; CREATE TABLE orders\_table (  order\_id STRING,  order\_amount DOUBLE,  product\_id STRING,  log\_ts TIMESTAMP(3),  proctime as PROCTIME() ) WITH (...);   -- streaming sql, kafka temporal join a hive dimension table. Flink will automatically reload data from the -- configured latest partition in the interval of 'streaming-source.monitor-interval'.  SELECT \* FROM orders\_table AS o  JOIN dimension\_table FOR SYSTEM\_TIME AS OF o.proctime AS dim ON o.product\_id = dim.product\_id; |

###### [Temporal Join The Latest Table](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hive/hive_read_write/" \l "temporal-join-the-latest-table)

对于 Hive 表，我们可以将其作为有界流读出。在这种情况下，Hive 表只能在我们查询时跟踪其最新版本。最新版本的表保留了 Hive 表的所有数据。

当对最新的 Hive 表执行临时连接时，Hive 表将缓存在 Slot 内存中，并且流中的每条记录都通过键连接到表以确定是否找到匹配项。使用最新的 Hive 表作为临时表不需要任何额外的配置。（可选）您可以使用以下属性配置Hive表缓存的TTL。缓存过期后，会再次扫描 Hive 表以加载最新数据。

|  |  |  |  |
| --- | --- | --- | --- |
| **配置项** | **默认值** | **类型** | **描述** |
| lookup.join.cache.ttl | 60 min | Duration | Lookup join中构建表的缓存TTL（例如10分钟）。默认情况下，TTL 为 60 分钟。注意：该选项仅在查找有界 hive 表源时有效，如果您使用流式 hive 源作为临时表，请使用 'streaming-source.monitor-interval' 配置数据更新的间隔。 |

以下演示将 hive 表的所有数据加载为时态表。

|  |
| --- |
| -- Assume the data in hive table is overwrite by batch pipeline. SET table.sql-dialect=hive; CREATE TABLE dimension\_table (  product\_id STRING,  product\_name STRING,  unit\_price DECIMAL(10, 4),  pv\_count BIGINT,  like\_count BIGINT,  comment\_count BIGINT,  update\_time TIMESTAMP(3),  update\_user STRING,  ... ) TBLPROPERTIES (  'streaming-source.enable' = 'false', -- option with default value, can be ignored.  'streaming-source.partition.include' = 'all', -- option with default value, can be ignored.  'lookup.join.cache.ttl' = '12 h' );  SET table.sql-dialect=default; CREATE TABLE orders\_table (  order\_id STRING,  order\_amount DOUBLE,  product\_id STRING,  log\_ts TIMESTAMP(3),  proctime as PROCTIME() ) WITH (...);   -- streaming sql, kafka join a hive dimension table. Flink will reload all data from dimension\_table after cache ttl is expired.  SELECT \* FROM orders\_table AS o  JOIN dimension\_table FOR SYSTEM\_TIME AS OF o.proctime AS dim ON o.product\_id = dim.product\_id; |

|  |
| --- |
| **注意：**   * 每个加入的子任务都需要保留自己的 Hive 表缓存。请确保 Hive 表可以放入 TM 任务槽的内存中。 * 鼓励为streaming-source.monitor-interval（最新分区作为临时表）或lookup.join.cache.ttl（所有分区作为临时表）设置一个相对较大的值。否则，作业很容易出现性能问题，因为表需要过于频繁地更新和重新加载。 * 当前，只要缓存需要刷新，我们就简单地加载整个Hive表。没有办法区分新数据和旧数据。 |

##### 往Hive写数据

Flink支持在BATCH和STREAMING模式下从Hive写入数据。当作为BATCH 应用程序运行时，Flink 将写入 Hive 表，仅在作业完成时使这些记录可见。 BATCH写入支持追加和覆盖现有表。

|  |
| --- |
| # ------ INSERT INTO will append to the table or partition, keeping the existing data intact ------  Flink SQL> INSERT INTO mytable SELECT 'Tom', 25;  # ------ INSERT OVERWRITE will overwrite any existing data in the table or partition ------  Flink SQL> INSERT OVERWRITE mytable SELECT 'Tom', 25; |

数据也可以插入到特定的分区中。

|  |
| --- |
| # ------ Insert with static partition ------  Flink SQL> INSERT OVERWRITE myparttable PARTITION (my\_type='type\_1', my\_date='2019-08-08') SELECT 'Tom', 25;  # ------ Insert with dynamic partition ------  Flink SQL> INSERT OVERWRITE myparttable SELECT 'Tom', 25, 'type\_1', '2019-08-08';  # ------ Insert with static(my\_type) and dynamic(my\_date) partition ------  Flink SQL> INSERT OVERWRITE myparttable PARTITION (my\_type='type\_1') SELECT 'Tom', 25, '2019-08-08'; |

STREAMING不断地向 Hive 添加新数据，提交记录 - 使它们可见 - 增量。用户控制何时/如何触发具有多个属性的提交。流式写入不支持插入覆盖。

下面的示例展示了如何使用流接收器编写流查询以将数据从 Kafka 写入带有分区提交的 Hive 表，并运行批处理查询以读取该数据。

|  |
| --- |
| SET table.sql-dialect=hive; CREATE TABLE hive\_table (  user\_id STRING,  order\_amount DOUBLE ) PARTITIONED BY (dt STRING, hr STRING) STORED AS parquet TBLPROPERTIES (  'partition.time-extractor.timestamp-pattern'='$dt $hr:00:00',  'sink.partition-commit.trigger'='partition-time',  'sink.partition-commit.delay'='1 h',  'sink.partition-commit.policy.kind'='metastore,success-file' );  SET table.sql-dialect=default; CREATE TABLE kafka\_table (  user\_id STRING,  order\_amount DOUBLE,  log\_ts TIMESTAMP(3),  WATERMARK FOR log\_ts AS log\_ts - INTERVAL '5' SECOND -- Define watermark on TIMESTAMP column ) WITH (...);  -- streaming sql, insert into hive table INSERT INTO TABLE hive\_table  SELECT user\_id, order\_amount, DATE\_FORMAT(log\_ts, 'yyyy-MM-dd'), DATE\_FORMAT(log\_ts, 'HH') FROM kafka\_table;  -- batch sql, select with partition pruning SELECT \* FROM hive\_table WHERE dt='2020-05-20' and hr='12'; |

如果在TIMESTAMP\_LTZ列上定义了水印并用于partition-time提交，sink.partition-commit.watermark-time-zone则需要将其设置为会话时区，否则，提交的分区可能会在几个小时后发生。

|  |
| --- |
| SET table.sql-dialect=hive; CREATE TABLE hive\_table (  user\_id STRING,  order\_amount DOUBLE ) PARTITIONED BY (dt STRING, hr STRING) STORED AS parquet TBLPROPERTIES (  'partition.time-extractor.timestamp-pattern'='$dt $hr:00:00',  'sink.partition-commit.trigger'='partition-time',  'sink.partition-commit.delay'='1 h',  'sink.partition-commit.watermark-time-zone'='Asia/Shanghai', -- Assume user configured time zone is 'Asia/Shanghai'  'sink.partition-commit.policy.kind'='metastore,success-file' );  SET table.sql-dialect=default; CREATE TABLE kafka\_table (  user\_id STRING,  order\_amount DOUBLE,  ts BIGINT, -- time in epoch milliseconds  ts\_ltz AS TO\_TIMESTAMP\_LTZ(ts, 3),  WATERMARK FOR ts\_ltz AS ts\_ltz - INTERVAL '5' SECOND -- Define watermark on TIMESTAMP\_LTZ column ) WITH (...);  -- streaming sql, insert into hive table INSERT INTO TABLE hive\_table  SELECT user\_id, order\_amount, DATE\_FORMAT(ts\_ltz, 'yyyy-MM-dd'), DATE\_FORMAT(ts\_ltz, 'HH') FROM kafka\_table;  -- batch sql, select with partition pruning SELECT \* FROM hive\_table WHERE dt='2020-05-20' and hr='12'; |

默认情况下，对于流式写入，Flink 仅支持重命名提交者，这意味着 S3 文件系统不能支持一次流式写入。通过将以下参数配置为 false，可以实现对 S3 的 Exactly-once 写入。这将指示接收器使用 Flink 的本机编写器，但仅适用于 parquet 和 orc 文件类型。此配置设置在TableConfig并将影响作业的所有接收器。

|  |  |  |  |
| --- | --- | --- | --- |
| **配置项** | **默认值** | **类型** | **描述** |
| table.exec.hive.fallback-mapred-writer | true | boolean | 如果为false，则使用flink本机编写器来编写镶木地板和orc文件；如果是真的，使用 hadoop mapred record writer 编写 parquet 和 orc 文件。 |

#### 支持的文件格式

Flink 的 Hive 集成已经针对以下文件格式进行了测试：

* Text
* CSV
* SequenceFile
* ORC
* Parquet

### Kafka

Kafka 连接器提供从 Kafka topic 中消费和写入数据的能力。

#### 依赖

为了使用 Kafka 连接器，使用构建自动化工具（例如 Maven 或 SBT）的项目和带有 SQL JAR 包的 SQL 客户端都需要以下依赖项。

|  |  |  |
| --- | --- | --- |
| **Kafka version** | **Maven dependency** | **SQL Client JAR** |
| universal | <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-kafka\_2.11</artifactId>  <version>1.13.0</version> </dependency> | [Download](https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-kafka_2.11/1.13.0/flink-sql-connector-kafka_2.11-1.13.0.jar) |

#### 如何创建 Kafka 表

以下示例展示了如何创建 Kafka 表：

|  |
| --- |
| CREATE TABLE KafkaTable (  `user\_id` BIGINT,  `item\_id` BIGINT,  `behavior` STRING,  `ts` TIMESTAMP(3) METADATA FROM 'timestamp' ) WITH (  'connector' = 'kafka',  'topic' = 'user\_behavior',  'properties.bootstrap.servers' = 'localhost:9092',  'properties.group.id' = 'testGroup',  'scan.startup.mode' = 'earliest-offset',  'format' = 'csv' ) |

#### 可用的元数据

以下的连接器元数据可以在表定义中通过元数据列的形式获取。

R/W 列定义了一个元数据是可读的（R）还是可写的（W）。 只读列必须声明为 VIRTUAL 以在 INSERT INTO 操作中排除它们。

|  |  |  |  |
| --- | --- | --- | --- |
| **键** | **数据类型** | **描述** | **R/W** |
| topic | STRING NOT NULL | Kafka 记录的 Topic 名 | R |
| partition | INT NOT NULL | Kafka 记录的 partition ID | R |
| headers | MAP NOT NULL | 二进制 Map 类型的 Kafka 记录头（Header） | R/W |
| leader-epoch | INT NULL | Kafka 记录的 Leader epoch（如果可用） | R |
| offset | BIGINT NOT NULL | Kafka 记录在 partition 中的 offset | R |
| timestamp | TIMESTAMP\_LTZ(3) NOT NULL | Kafka 记录的时间戳 | R/W |
| timestamp-type | STRING NOT NULL | Kafka 记录的时间戳类型。可能的类型有 "NoTimestampType"， "CreateTime"（会在写入元数据时设置），或 "LogAppendTime" | R |

以下扩展的 CREATE TABLE 示例展示了使用这些元数据字段的语法：

|  |
| --- |
| CREATE TABLE KafkaTable (  `event\_time` TIMESTAMP(3) METADATA FROM 'timestamp',  `partition` BIGINT METADATA VIRTUAL,  `offset` BIGINT METADATA VIRTUAL,  `user\_id` BIGINT,  `item\_id` BIGINT,  `behavior` STRING ) WITH (  'connector' = 'kafka',  'topic' = 'user\_behavior',  'properties.bootstrap.servers' = 'localhost:9092',  'properties.group.id' = 'testGroup',  'scan.startup.mode' = 'earliest-offset',  'format' = 'csv' ); |

**格式元信息**

连接器可以读出消息格式的元数据。格式元数据的配置键以 'value.' 作为前缀。

以下示例展示了如何获取 Kafka 和 Debezium 的元数据字段：

|  |
| --- |
| CREATE TABLE KafkaTable (  `event\_time` TIMESTAMP(3) METADATA FROM 'value.source.timestamp' VIRTUAL, -- from Debezium format  `origin\_table` STRING METADATA FROM 'value.source.table' VIRTUAL, -- from Debezium format  `partition\_id` BIGINT METADATA FROM 'partition' VIRTUAL, -- from Kafka connector  `offset` BIGINT METADATA VIRTUAL, -- from Kafka connector  `user\_id` BIGINT,  `item\_id` BIGINT,  `behavior` STRING ) WITH (  'connector' = 'kafka',  'topic' = 'user\_behavior',  'properties.bootstrap.servers' = 'localhost:9092',  'properties.group.id' = 'testGroup',  'scan.startup.mode' = 'earliest-offset',  'value.format' = 'debezium-json' ); |

#### 连接器参数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **必选** | **默认值** | **类型** | **描述** |
| connector | 必选 | (无) | String | 指定使用的连接器，Kafka 连接器使用 'kafka' |
| topic | required for sink | (无) | String | 当表用作 source 时读取数据的 topic 名。亦支持用分号间隔的 topic 列表，如 'topic-1;topic-2'。注意，对 source 表而言，'topic' 和 'topic-pattern' 两个选项只能使用其中一个。当表被用作 sink 时，该配置表示写入的 topic 名。注意 sink 表不支持 topic 列表 |
| topic-pattern | 可选 | (无) | String | 匹配读取 topic 名称的正则表达式。在作业开始运行时，所有匹配该正则表达式的 topic 都将被 Kafka consumer 订阅。注意，对 source 表而言，'topic' 和 'topic-pattern' 两个选项只能使用其中一个 |
| properties.bootstrap.servers | 必选 | (无) | String | 逗号分隔的 Kafka broker 列表 |
| properties.group.id | required by source | (无) | String | Kafka source 的 consumer 组 id，对于 Kafka sink 可选填。 |
| properties.\* | 可选 | (无) | String | 可以设置和传递任意 Kafka 的配置项。后缀名必须匹配在 [Kafka 配置文档](https://kafka.apache.org/documentation/" \l "configuration) 中定义的配置键。Flink 将移除 "properties." 配置键前缀并将变换后的配置键和值传入底层的 Kafka 客户端。例如，你可以通过 'properties.allow.auto.create.topics' = 'false' 来禁用 topic 的自动创建。但是某些配置项不支持进行配置，因为 Flink 会覆盖这些配置，例如 'key.deserializer' 和 'value.deserializer' |
| format | 必选 | (无) | String | 用来序列化或反序列化 Kafka 消息的格式。 请参阅 [格式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/overview/) 页面以获取更多关于格式的细节和相关配置项。 注意：该配置项和 'value.format' 二者必需其一 |
| key.format | 可选 | （无） | String | 用来序列化和反序列化 Kafka 消息键（Key）的格式。 请参阅 [格式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/overview/) 页面以获取更多关于格式的细节和相关配置项。 注意：如果定义了键格式，则配置项 'key.fields' 也是必需的。 否则 Kafka 记录将使用空值作为键 |
| key.fields | 可选 | [] | List<String> | 表结构中用来配置消息键（Key）格式数据类型的字段列表。默认情况下该列表为空，因此消息键没有定义。 列表格式为 'field1;field2' |
| key.fields-prefix | 可选 | （无） | String | 为所有消息键（Key）格式字段指定自定义前缀，以避免与消息体（Value）格式字段重名。默认情况下前缀为空。 如果定义了前缀，表结构和配置项 'key.fields' 都需要使用带前缀的名称。 当构建消息键格式字段时，前缀会被移除，消息键格式将会使用无前缀的名称。 请注意该配置项要求必须将 'value.fields-include' 配置为 'EXCEPT\_KEY' |
| value.format | 必选 | （无） | String | 序列化和反序列化 Kafka 消息体时使用的格式。 请参阅 [格式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/overview/) 页面以获取更多关于格式的细节和相关配置项。 注意：该配置项和 'format' 二者必需其一 |
| value.fields-include | 可选 | ALL | 枚举类型  可选值：[ALL, EXCEPT\_KEY] | 定义消息体（Value）格式如何处理消息键（Key）字段的策略。 默认情况下，表结构中 'ALL' 即所有的字段都会包含在消息体格式中，即消息键字段在消息键和消息体格式中都会出现 |
| scan.startup.mode | 可选 | group-offsets | String | Kafka consumer 的启动模式。有效值为：'earliest-offset'，'latest-offset'，'group-offsets'，'timestamp' 和 'specific-offsets'。 请参阅下方 [起始消费位点](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/kafka/" \l "%E8%B5%B7%E5%A7%8B%E6%B6%88%E8%B4%B9%E4%BD%8D%E7%82%B9) 以获取更多细节。 |
| scan.startup.specific-offsets | 可选 | （无） | String | 在使用 'specific-offsets' 启动模式时为每个 partition 指定 offset，例如 'partition:0,offset:42;partition:1,offset:300' |
| scan.startup.timestamp-millis | 可选 | （无） | Long | 在使用 'timestamp' 启动模式时指定启动的时间戳（单位毫秒） |
| scan.topic-partition-discovery.interval | 可选 | （无） | Duration | Consumer 定期探测动态创建的 Kafka topic 和 partition 的时间间隔 |
| sink.partitioner | 可选 | 'default' | String | Flink partition 到 Kafka partition 的分区映射关系，可选值有：   * default：使用 Kafka 默认的分区器对消息进行分区。 * fixed：每个 Flink partition 最终对应最多一个 Kafka partition。 * round-robin：Flink partition 按轮循（round-robin）的模式对应到 Kafka partition。只有当未指定消息的消息键时生效 * 自定义 FlinkKafkaPartitioner 的子类：例如 'org.mycompany.MyPartitioner'   请参阅下方 [Sink 分区](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/kafka/" \l "sink-%E5%88%86%E5%8C%BA) 以获取更多细节。 |
| sink.semantic | 可选 | at-least-once | String | 定义 Kafka sink 的语义。有效值为 'at-least-once'，'exactly-once' 和 'none'。 |
| sink.parallelism | 可选 | （无） | Integer | 定义 Kafka sink 算子的并行度。默认情况下，并行度由框架定义为与上游串联的算子相同 |

#### 特性

##### 消息键（Key）与消息体（Value）的格式

Kafka 消息的消息键和消息体部分都可以使用某种 [格式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/overview/) 来序列化或反序列化成二进制数据。

**消息体格式**

由于 Kafka 消息中消息键是可选的，以下语句将使用消息体格式读取和写入消息，但不使用消息键格式。 'format' 选项与 'value.format' 意义相同。 所有的格式配置使用格式识别符作为前缀。

|  |
| --- |
| CREATE TABLE KafkaTable (,  `ts` TIMESTAMP(3) METADATA FROM 'timestamp',  `user\_id` BIGINT,  `item\_id` BIGINT,  `behavior` STRING ) WITH (  'connector' = 'kafka',  ...   'format' = 'json',  'json.ignore-parse-errors' = 'true' ) |

消息体格式将配置为以下的数据类型：

|  |
| --- |
| ROW<`user\_id` BIGINT, `item\_id` BIGINT, `behavior` STRING> |

**消息键和消息体格式**

以下示例展示了如何配置和使用消息键和消息体格式。 格式配置使用 'key' 或 'value' 加上格式识别符作为前缀。

|  |
| --- |
| CREATE TABLE KafkaTable (  `ts` TIMESTAMP(3) METADATA FROM 'timestamp',  `user\_id` BIGINT,  `item\_id` BIGINT,  `behavior` STRING ) WITH (  'connector' = 'kafka',  ...   'key.format' = 'json',  'key.json.ignore-parse-errors' = 'true',  'key.fields' = 'user\_id;item\_id',   'value.format' = 'json',  'value.json.fail-on-missing-field' = 'false',  'value.fields-include' = 'ALL' ) |

消息键格式包含了在 'key.fields' 中列出的字段（使用 ';' 分隔）和字段顺序。 因此将配置为以下的数据类型：

|  |
| --- |
| ROW<`user\_id` BIGINT, `item\_id` BIGINT> |

由于消息体格式配置为 'value.fields-include' = 'ALL'，所以消息键字段也会出现在消息体格式的数据类型中：

|  |
| --- |
| ROW<`user\_id` BIGINT, `item\_id` BIGINT, `behavior` STRING> |

**重名的格式字段**

如果消息键字段和消息体字段重名，连接器无法根据表结构信息将这些列区分开。 'key.fields-prefix' 配置项可以在表结构中为消息键字段指定一个唯一名称，并在配置消息键格式的时候保留原名。

以下示例展示了在消息键和消息体中同时包含 version 字段的情况：

|  |
| --- |
| CREATE TABLE KafkaTable (  `k\_version` INT,  `k\_user\_id` BIGINT,  `k\_item\_id` BIGINT,  `version` INT,  `behavior` STRING ) WITH (  'connector' = 'kafka',  ...   'key.format' = 'json',  'key.fields-prefix' = 'k\_',  'key.fields' = 'k\_version;k\_user\_id;k\_item\_id',   'value.format' = 'json',  'value.fields-include' = 'EXCEPT\_KEY' ) |

消息体格式必须配置为 'EXCEPT\_KEY' 模式。格式将被配置为以下的数据类型：

|  |
| --- |
| 消息键格式： ROW<`version` INT, `user\_id` BIGINT, `item\_id` BIGINT>  消息体格式： ROW<`version` INT, `behavior` STRING> |

##### Topic 和 Partition 的探测

topic 和 topic-pattern 配置项决定了 source 消费的 topic 或 topic 的匹配规则。topic 配置项可接受使用分号间隔的 topic 列表，例如 topic-1;topic-2。 topic-pattern 配置项使用正则表达式来探测匹配的 topic。例如 topic-pattern 设置为 test-topic-[0-9]，则在作业启动时，所有匹配该正则表达式的 topic（以 test-topic- 开头，以一位数字结尾）都将被 consumer 订阅。

为允许 consumer 在作业启动之后探测到动态创建的 topic，请将 scan.topic-partition-discovery.interval 配置为一个非负值。这将使 consumer 能够探测匹配名称规则的 topic 中新的 partition。

请参阅 [Kafka DataStream 连接器文档](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/datastream/kafka/" \l "kafka-consumer-topic-%e5%92%8c%e5%88%86%e5%8c%ba%e5%8f%91%e7%8e%b0) 以获取更多关于 topic 和 partition 探测的信息。

注意 topic 列表和 topic 匹配规则只适用于 source。对于 sink 端，Flink 目前只支持单一 topic。

##### 起始消费位点

scan.startup.mode 配置项决定了 Kafka consumer 的启动模式。有效值为：

* `group-offsets`：从 Zookeeper/Kafka 中某个指定的消费组已提交的偏移量开始。
* `earliest-offset`：从可能的最早偏移量开始。
* `latest-offset`：从最末尾偏移量开始。
* `timestamp`：从用户为每个 partition 指定的时间戳开始。
* `specific-offsets`：从用户为每个 partition 指定的偏移量开始。

默认值 group-offsets 表示从 Zookeeper/Kafka 中最近一次已提交的偏移量开始消费。

如果使用了 timestamp，必须使用另外一个配置项 scan.startup.timestamp-millis 来指定一个从格林尼治标准时间 1970 年 1 月 1 日 00:00:00.000 开始计算的毫秒单位时间戳作为起始时间。

如果使用了 specific-offsets，必须使用另外一个配置项 scan.startup.specific-offsets 来为每个 partition 指定起始偏移量， 例如，选项值 partition:0,offset:42;partition:1,offset:300 表示 partition 0 从偏移量 42 开始，partition 1 从偏移量 300 开始。

##### CDC 变更日志（Changelog） Source

Flink 原生支持使用 Kafka 作为 CDC 变更日志（changelog） source。如果 Kafka topic 中的消息是通过变更数据捕获（CDC）工具从其他数据库捕获的变更事件，则你可以使用 CDC 格式将消息解析为 Flink SQL 系统中的插入（INSERT）、更新（UPDATE）、删除（DELETE）消息。

在许多情况下，变更日志（changelog） source 都是非常有用的功能，例如将数据库中的增量数据同步到其他系统，审核日志，数据库的物化视图，时态表关联数据库表的更改历史等。

Flink 提供了几种 CDC 格式：

* [debezium](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/debezium/)
* [canal](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/canal/)
* [maxwell](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/maxwell/)

##### Sink 分区

配置项 sink.partitioner 指定了从 Flink 分区到 Kafka 分区的映射关系。 默认情况下，Flink 使用 [Kafka 默认分区器](https://github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/clients/producer/internals/DefaultPartitioner.java) 来对消息分区。默认分区器对没有消息键的消息使用 [粘性分区策略（sticky partition strategy）](https://www.confluent.io/blog/apache-kafka-producer-improvements-sticky-partitioner/) 进行分区，对含有消息键的消息使用 murmur2 哈希算法计算分区。

为了控制数据行到分区的路由，也可以提供一个自定义的 sink 分区器。‘fixed’ 分区器会将同一个 Flink 分区中的消息写入同一个 Kafka 分区，从而减少网络连接的开销。

##### 一致性保证

默认情况下，如果查询在 [启用 checkpoint](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/datastream/fault-tolerance/checkpointing/" \l "enabling-and-configuring-checkpointing)模式下执行时，Kafka sink 按照至少一次（at-lease-once）语义保证将数据写入到 Kafka topic 中。

当 Flink checkpoint 启用时，kafka 连接器可以提供精确一次（exactly-once）的语义保证。

除了启用 Flink checkpoint，还可以通过传入对应的 sink.semantic 选项来选择三种不同的运行模式：

* none：Flink 不保证任何语义。已经写出的记录可能会丢失或重复。
* at-least-once (默认设置)：保证没有记录会丢失（但可能会重复）。
* exactly-once：使用 Kafka 事务提供精确一次（exactly-once）语义。当使用事务向 Kafka 写入数据时，请将所有从 Kafka 中消费记录的应用中的 isolation.level 配置项设置成实际所需的值（read\_committed 或 read\_uncommitted，后者为默认值）。

请参阅 [Kafka 文档](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/datastream/kafka/" \l "kafka-producers-%e5%92%8c%e5%ae%b9%e9%94%99)以获取更多关于语义保证的信息。

##### Source 按分区 Watermark

Flink 对于 Kafka 支持发送按分区的 watermark。Watermark 在 Kafka consumer 中生成。 按分区 watermark 的合并方式和在流 shuffle 时合并 Watermark 的方式一致。 Source 输出的 watermark 由读取的分区中最小的 watermark 决定。 如果 topic 中的某些分区闲置，watermark 生成器将不会向前推进。 你可以在表配置中设置 ['](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "table-exec-source-idle-timeout)**[table.exec.source.idle-timeout](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "table-exec-source-idle-timeout)**['](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "table-exec-source-idle-timeout) 选项来避免上述问题。

请参阅 [Kafka watermark 策略](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/datastream/event-time/generating_watermarks/" \l "watermark-%e7%ad%96%e7%95%a5%e5%92%8c-kafka-%e8%bf%9e%e6%8e%a5%e5%99%a8)以获取更多细节。

##### 数据类型映射

Kafka 将消息键值以二进制进行存储，因此 Kafka 并不存在 schema 或数据类型。Kafka 消息使用格式配置进行序列化和反序列化，例如 csv，json，avro。 因此，数据类型映射取决于使用的格式。请参阅 [格式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/overview/) 页面以获取更多细节。

### [HBase](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/hbase/)

HBase 连接器支持读取和写入 HBase 集群。本文档介绍如何使用 HBase 连接器基于 HBase 进行 SQL 查询。

HBase 连接器在 upsert 模式下运行，可以使用 DDL 中定义的主键与外部系统交换更新操作消息。但是主键只能基于 HBase 的 rowkey 字段定义。如果没有声明主键，HBase 连接器默认取 rowkey 作为主键。

#### 依赖

为了使用HBA连接器，使用构建自动化工具（如Maven或SBT）和SQL客户机（带有SQL JAR捆绑包）的项目都需要以下依赖项。

|  |  |  |
| --- | --- | --- |
| **Hbaseversion** | **Maven dependency** | **SQL Client JAR** |
| 1.4.x | <dependency>  <groupId>org.apache.flink</groupId> <artifactId>flink-connector-hbase-1.4\_2.11</artifactId>  <version>1.13.0</version> </dependency> | [Download](https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-hbase-1.4_2.11/1.13.0/flink-sql-connector-hbase-1.4_2.11-1.13.0.jar) |
| 2.2.x | <dependency>  <groupId>org.apache.flink</groupId> <artifactId>flink-connector-hbase-2.2\_2.11</artifactId>  <version>1.13.0</version> </dependency> | [Download](https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-hbase-2.2_2.11/1.13.0/flink-sql-connector-hbase-2.2_2.11-1.13.0.jar) |

#### 如何使用 HBase 表

所有 HBase 表的列簇必须定义为 ROW 类型，字段名对应列簇名（column family），嵌套的字段名对应列限定符名（column qualifier）。用户只需在表结构中声明查询中使用的的列簇和列限定符。除了 ROW 类型的列，剩下的原子数据类型字段（比如，STRING, BIGINT）将被识别为 HBase 的 rowkey，一张表中只能声明一个 rowkey。rowkey 字段的名字可以是任意的，如果是保留关键字，需要用反引号。

|  |
| --- |
| -- 在 Flink SQL 中注册 HBase 表 "mytable" CREATE TABLE hTable (  rowkey INT,  family1 ROW<q1 INT>,  family2 ROW<q2 STRING, q3 BIGINT>,  family3 ROW<q4 DOUBLE, q5 BOOLEAN, q6 STRING>,  PRIMARY KEY (rowkey) NOT ENFORCED ) WITH (  'connector' = 'hbase-1.4',  'table-name' = 'mytable',  'zookeeper.quorum' = 'localhost:2181' );  -- 用 ROW(...) 构造函数构造列簇，并往 HBase 表写数据。 -- 假设 "T" 的表结构是 [rowkey, f1q1, f2q2, f2q3, f3q4, f3q5, f3q6] INSERT INTO hTable SELECT rowkey, ROW(f1q1), ROW(f2q2, f2q3), ROW(f3q4, f3q5, f3q6) FROM T;  -- 从 HBase 表扫描数据 SELECT rowkey, family1, family3.q4, family3.q6 FROM hTable;  -- temporal join HBase 表，将 HBase 表作为维表 SELECT \* FROM myTopic LEFT JOIN hTable FOR SYSTEM\_TIME AS OF myTopic.proctime ON myTopic.key = hTable.rowkey; |

#### 连接器参数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **必选** | **默认值** | **类型** | **描述** |
| connector | 必选 | (none) | String | 指定使用的连接器, 支持的值如下 :  hbase-1.4: 连接 HBase 1.4.x 集群  hbase-2.2: 连接 HBase 2.2.x 集群 |
| table-name | 必选 | (none) | String | 连接的 HBase 表名 |
| zookeeper.quorum | 必选 | (none) | String | HBase Zookeeper quorum 信息 |
| zookeeper.znode.parent | 可选 | /hbase | String | HBase 集群的 Zookeeper 根目录 |
| null-string-literal | 可选 | null | String | 当字符串值为 null 时的存储形式，默认存成 "null" 字符串。HBase 的 source 和 sink 的编解码将所有数据类型（除字符串外）将 null 值以空字节来存储 |
| sink.buffer-flush.max-size | 可选 | 2mb | MemorySize | 写入的参数选项。每次写入请求缓存行的最大大小。它能提升写入 HBase 数据库的性能，但是也可能增加延迟。设置为 "0" 关闭此选项 |
| sink.buffer-flush.max-rows | 可选 | 1000 | Integer | 写入的参数选项。 每次写入请求缓存的最大行数。它能提升写入 HBase 数据库的性能，但是也可能增加延迟。设置为 "0" 关闭此选项 |
| sink.buffer-flush.interval | 可选 | 1s | Duration | 写入的参数选项。刷写缓存行的间隔。它能提升写入 HBase 数据库的性能，但是也可能增加延迟。设置为 "0" 关闭此选项。注意："sink.buffer-flush.max-size" 和 "sink.buffer-flush.max-rows" 同时设置为 "0"，刷写选项整个异步处理缓存行为 |
| sink.parallelism | 可选 | (none) | Integer | 为 HBase sink operator 定义并行度。默认情况下，并行度由框架决定，和链在一起的上游 operator 一样 |
| lookup.async | 可选 | false | Boolean | 是否启用异步查找。如果为真，查找将是异步的。注意：异步方式只支持 hbase-2.2 连接器 |
| lookup.cache.max-rows | 可选 | (无) | Integer | 查找缓存的最大行数，超过这个值，最旧的行将过期。注意："lookup.cache.max-rows" 和 "lookup.cache.ttl" 必须同时被设置。默认情况下，查找缓存是禁用的 |
| lookup.cache.ttl | 可选 | (无) | Duration | 查找缓存中每一行的最大生存时间，在这段时间内，最老的行将过期。注意："lookup.cache.max-rows" 和 "lookup.cache.ttl" 必须同时被设置。默认情况下，查找缓存是禁用的。 |
| lookup.max-retries | 可选 | 3 | Integer | 查找数据库失败时的最大重试次数 |
| properties.\* | 可选 | (无) | String | 可以设置任意 HBase 的配置项。后缀名必须匹配在 [HBase 配置文档](http://hbase.apache.org/2.3/book.html" \l "hbase_default_configurations) 中定义的配置键。Flink 将移除 "properties." 配置键前缀并将变换后的配置键和值传入底层的 HBase 客户端。 例如您可以设置 'properties.hbase.security.authentication' = 'kerberos' 等kerberos认证参数。 |

#### 数据类型映射表

HBase 以字节数组存储所有数据。在读和写过程中要序列化和反序列化数据。

Flink 的 HBase 连接器利用 HBase（Hadoop) 的工具类 org.apache.hadoop.hbase.util.Bytes 进行字节数组和 Flink 数据类型转换。

Flink 的 HBase 连接器将所有数据类型（除字符串外）null 值编码成空字节。对于字符串类型，null 值的字面值由null-string-literal选项值决定。

数据类型映射表如下：

|  |  |
| --- | --- |
| **Flink 数据类型** | **HBase 转换** |
| CHAR / VARCHAR / STRING | byte[] toBytes(String s) String toString(byte[] b) |
| BOOLEAN | byte[] toBytes(boolean b) boolean toBoolean(byte[] b) |
| BINARY / VARBINARY | 返回 byte[]。 |
| DECIMAL | byte[] toBytes(BigDecimal v) BigDecimal toBigDecimal(byte[] b) |
| TINYINT | new byte[] { val } bytes[0] // returns first and only byte from bytes |
| SMALLINT | byte[] toBytes(short val) short toShort(byte[] bytes) |
| INT | byte[] toBytes(int val) int toInt(byte[] bytes) |
| BIGINT | byte[] toBytes(long val) long toLong(byte[] bytes) |
| FLOAT | byte[] toBytes(float val) float toFloat(byte[] bytes) |
| DOUBLE | byte[] toBytes(double val) double toDouble(byte[] bytes) |
| DATE | 从 1970-01-01 00:00:00 UTC 开始的天数，int 值。 |
| TIME | 从 1970-01-01 00:00:00 UTC 开始天的毫秒数，int 值。 |
| TIMESTAMP | 从 1970-01-01 00:00:00 UTC 开始的毫秒数，long 值。 |
| ARRAY | 不支持 |
| MAP / MULTISET | 不支持 |
| ROW | 不支持 |

### [DataGen](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/datagen/)

DataGen 连接器允许按数据生成规则进行读取。

DataGen 连接器可以使用[计算列语法](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/create/" \l "create-table)。 这使您可以灵活地生成记录。

DataGen 连接器是内置的。

|  |
| --- |
| **注意：**不支持复杂类型: Array，Map，Row。 请用计算列构造这些类型。 |

#### 怎么创建一个 DataGen 的表

表的有界性：当表中字段的数据全部生成完成后，source 就结束了。 因此，表的有界性取决于字段的有界性。

每个列，都有两种生成数据的方法：

* 随机生成器是默认的生成器，您可以指定随机生成的最大和最小值。char、varchar、string （类型）可以指定长度。它是无界的生成器。
* 序列生成器，您可以指定序列的起始和结束值。它是有界的生成器，当序列数字达到结束值，读取结束。

|  |
| --- |
| CREATE TABLE datagen (  f\_sequence INT,  f\_random INT,  f\_random\_str STRING,  ts AS localtimestamp,  WATERMARK FOR ts AS ts ) WITH (  'connector' = 'datagen',   -- optional options --   'rows-per-second'='5',   'fields.f\_sequence.kind'='sequence',  'fields.f\_sequence.start'='1',  'fields.f\_sequence.end'='1000',   'fields.f\_random.min'='1',  'fields.f\_random.max'='1000',   'fields.f\_random\_str.length'='10' ) |

#### 连接器参数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **必选** | **默认值** | **类型** | **描述** |
| connector | 必选 | (none) | String | 指定要使用的连接器，这里是 'datagen' |
| rows-per-second | 可选 | 10000 | Long | 每秒生成的行数，用以控制数据发出速率 |
| fields.#.kind | 可选 | random | String | 指定 '#' 字段的生成器。可以是 'sequence' 或 'random' |
| fields.#.min | 可选 | (Minimum value of type) | (Type of field) | 随机生成器的最小值，适用于数字类型 |
| fields.#.max | 可选 | (Maximum value of type) | (Type of field) | 随机生成器的最大值，适用于数字类型 |
| fields.#.length | 可选 | 100 | Integer | 随机生成器生成字符的长度，适用于 char、varchar、string |
| fields.#.start | 可选 | (none) | (Type of field) | 序列生成器的起始值 |
| fields.#.end | 可选 | (none) | (Type of field) | 序列生成器的结束值 |

### [FileSystem](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/filesystem/)

此连接器提供对[Flink FileSystem 抽象](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/deployment/filesystems/overview/)支持的文件系统中的分区文件的访问。

文件系统连接器本身包含在 Flink 中，不需要额外的依赖。需要指定相应的格式来从文件系统读取行和向文件系统写入行。

文件系统连接器允许从本地或分布式文件系统读取和写入。文件系统表可以定义为：

|  |
| --- |
| CREATE TABLE MyUserTable (  column\_name1 INT,  column\_name2 STRING,  ...  part\_name1 INT,  part\_name2 STRING ) PARTITIONED BY (part\_name1, part\_name2) WITH (  'connector' = 'filesystem', -- required: specify the connector  'path' = 'file:///path/to/whatever', -- required: path to a directory  'format' = '...', -- required: file system connector requires to specify a format,  -- Please refer to Table Formats  -- section for more details  'partition.default-name' = '...', -- optional: default partition name in case the dynamic partition  -- column value is null/empty string   -- optional: the option to enable shuffle data by dynamic partition fields in sink phase, this can greatly  -- reduce the number of file for filesystem sink but may lead data skew, the default value is false.  'sink.shuffle-by-partition.enable' = '...',  ... ) |

注意：确保包含[Flink 文件系统特定的依赖项](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/deployment/filesystems/overview/)。

#### 分区文件

Flink 的文件系统分区支持使用标准的 hive 格式。但是，它不需要使用表目录预先注册分区。根据目录结构发现和推断分区。例如，基于以下目录分区的表将被推断为包含datetime和hour分区。

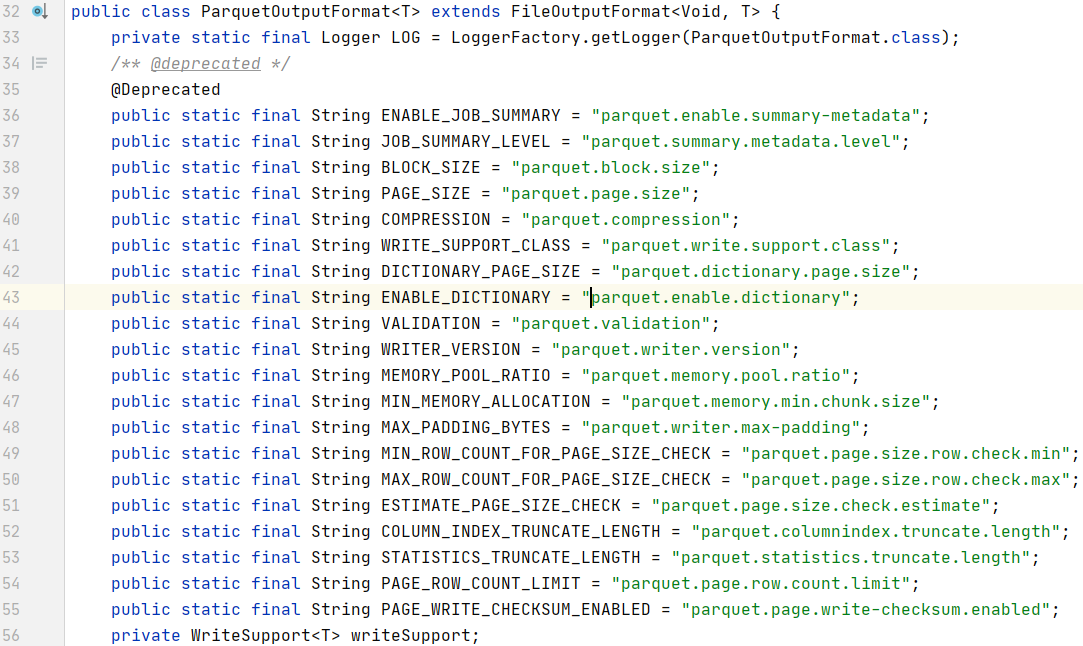
|  |
| --- |
| path └── datetime=2019-08-25  └── hour=11  ├── part-0.parquet  ├── part-1.parquet  └── hour=12  ├── part-0.parquet └── datetime=2019-08-26  └── hour=6  ├── part-0.parquet |

文件系统表支持分区插入和覆盖插入。请参阅[插入语句](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/insert/)。对分区表插入覆盖时，只会覆盖对应的分区，而不是整个表。

#### 文件格式

文件系统连接器支持多种格式：

* CSV: [RFC-4180](https://tools.ietf.org/html/rfc4180). 未压缩.
* JSON: 注意文件系统连接器的 JSON 格式不是典型的 JSON 文件，而是未压缩的[换行符分隔的 JSON](http://jsonlines.org/).
* Avro:[Apache Avro](http://avro.apache.org/). 通过配置支持压缩avro.codec.
* Parquet: [Apache Parquet](http://parquet.apache.org/). 与Hive兼容.
  + 相关配置可以看类ParquetOutputFormat，配置如下：



* Orc: [Apache Orc](http://orc.apache.org/). 与Hive兼容.
* Debezium-JSON: [debezium-json](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/debezium/).
* Canal-JSON: [canal-json](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/canal/).
* Raw: [raw](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/raw/).

#### Streaming Sink

文件系统连接器支持流式写入，基于 Flink 的[Streaming File Sink](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/datastream/streamfile_sink/)将记录写入文件。行编码格式为 csv 和 json。批量编码格式为 parquet、orc 和 avro。

可以直接写SQL，将流数据插入到非分区表中。如果是分区表，可以配置分区相关操作。有关详细信息，请参阅[分区提交](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/filesystem/filesystem.html" \l "partition-commit)。

##### Rolling Policy

分区目录内会有很多Part File，每个写某个分区的Sink子任务至少在该分区有一个文件。

|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **默认值** | **类型** | **描述** |
| sink.rolling-policy.file-size | 128MB | MemorySize | 分区文件的最大值，超过这个大小，将会启动一个新文件。 |
| sink.rolling-policy.rollover-interval | 30 m | Duration | 分区文件滚动的最大时间间隔，超过这个时间，将会新启动一个文件 |
| sink.rolling-policy.check-interval | 1 m | Duration | 一个时间间隔，定期去检查上面那个配置指定的策略下，文件是否应该滚动生成新文件 |

* 采用按批量编码格式时，in-progress文件滚动取决于Checkpoint间隔时间。
  + Checkpoint开始时由in-progress变为pending，此时不会再写入该文件，但也对读不可见
  + 通知Checkpoint完成后，该文件变为finished，对读可见。
  + 这就是个典型的两阶段提交。
* checkpoint间隔时间由execution.checkpointing.interval参数设置

##### 文件压缩

文件接收器支持文件压缩，这允许应用程序具有更小的检查点间隔，而不会生成大量文件。

|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **默认值** | **类型** | **描述** |
| auto-compaction | false | Boolean | 是否在流接收器中启用自动压缩。数据将写入临时文件。检查点完成后，检查点生成的临时文件将被压缩。临时文件在压缩前是不可见的。 |
| compaction.file-size | (none) | MemorySize | 压缩目标文件大小，默认值为滚动文件大小。 |

如果启用，文件压缩将根据目标文件大小将多个小文件合并为更大的文件。在生产中运行文件压缩时，请注意：

* 仅压缩单个检查点中的文件，即至少生成与检查点数量相同数量的文件。
* 合并前的文件是不可见的，所以文件的可见性可能是：检查点间隔+压缩时间。
* 如果压缩时间过长，则会对作业进行背压并延长检查点的时间段。

##### 分区提交

写入分区后，通常需要通知下游应用程序。例如，将分区添加到 Hive 元存储或\_SUCCESS在目录中写入文件。文件系统接收器包含允许配置自定义策略的分区提交功能。提交的行动是基于组合triggers和policies。

* 触发器：分区的提交时间可以通过水印和从分区中提取的时间来确定，也可以通过处理时间来确定。
* Policy：如何提交分区，内置策略支持提交成功文件和Metastore，也可以实现自己的策略，比如触发hive的分析生成统计信息，或者合并小文件等。

**注意：分区提交仅适用于动态分区插入。**

###### 分区提交触发器

要定义何时提交分区，提供分区提交触发器：

|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **默认值** | **类型** | **描述** |
| sink.partition-commit.trigger | process-time | String | 分区提交触发类型：  'process-time'：基于机器时间，既不需要分区时间提取，也不需要水印生成。一旦“当前系统时间”超过“分区创建系统时间”加上“延迟”，就提交分区。  'partition-time'：基于从分区值中提取的时间，需要生成水印。一旦“水印”通过“从分区值中提取的时间”加上“延迟”，就提交分区。 |
| sink.partition-commit.delay | 0 s | Duration | 直到延迟时间，分区才会提交。如果是日分区，应该是'1 d'，如果是小时分区，应该是'1 h'。 |
| sink.partition-commit.watermark-time-zone | UTC | String | 将长水印值解析为 TIMESTAMP 值的时区，解析后的水印时间戳用于与分区时间进行比较来决定分区是否应该提交。此选项仅在 `sink.partition-commit.trigger` 设置为 'partition-time' 时生效。如果此选项未正确配置，例如在 TIMESTAMP\_LTZ 列上定义了源行时间，但未配置此配置，则用户可能会在几个小时后看到提交的分区。默认值为“UTC”，表示水印在 TIMESTAMP 列上定义或未定义。如果在 TIMESTAMP\_LTZ 列上定义了水印，则水印的时区为会话时区。选项值可以是全名（例如“America/Los\_Angeles”）或自定义时区 ID（例如“GMT-08:00”）。 |

有两种类型的触发器：

* 首先是分区处理时间。它既不需要分区时间提取也不需要水印生成。根据分区创建时间和当前系统时间触发分区提交。这个触发器更通用，但不是那么精确。例如，数据延迟或故障转移会导致过早的分区提交。
* 第二个是根据从分区值和水印中提取的时间触发分区提交。这就要求你的job有水印生成，按照时间划分分区，比如每小时分区或者每天分区。

如果想让下游尽快看到分区，不管它的数据是否完整：

* 'sink.partition-commit.trigger'='process-time'（默认值）
* 'sink.partition-commit.delay'='0s' (默认值) 一旦分区中有数据，就会立即提交。注意：分区可能会被多次提交。

如果你想让下游看到分区只有在它的数据完成时，并且你的作业有水印生成，你可以从分区值中提取时间：

* 'sink.partition-commit.trigger'='partition-time'
* 'sink.partition-commit.delay'='1h' ('1h' 如果你的分区是每小时分区，取决于你的分区类型) 这是最准确的提交分区的方法，它会尽量确保提交的分区数据尽可能完整。

如果要让下游看到分区只有数据完整，但没有水印，或者无法从分区值中提取时间：

* 'sink.partition-commit.trigger'='process-time'（默认值）
* 'sink.partition-commit.delay'='1h' ('1h' 如果你的分区是每小时分区，取决于你的分区类型) 尝试准确提交分区，但数据延迟或故障转移会导致过早的分区提交。

后期数据处理：当一条记录应该写入到一个已经提交的分区时，记录会被写入到它的分区中，然后再次触发该分区的提交。

###### 分区时间提取器

时间提取器定义从分区值中提取时间。

|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **默认值** | **类型** | **描述** |
| partition.time-extractor.kind | default | String | 时间提取器从分区值中提取时间。支持默认和自定义。默认情况下，可以配置时间戳模式。对于自定义，应配置提取器类。 |
| partition.time-extractor.class | (none) | String | 用于实现 PartitionTimeExtractor 接口的提取器类 |
| partition.time-extractor.timestamp-pattern | (none) | String | “默认”构造方式允许用户使用分区字段来获取合法的时间戳模式。默认支持来自第一个字段的 'yyyy-mm-dd hh:mm:ss'。如果时间戳应该从单个分区字段'dt'中提取，可以配置：'$dt'。如果时间戳应该从多个分区字段中提取，比如'year'、'month'、'day'和'hour'，可以配置：'$year-$month-$day $hour:00:00'。如果时间戳应该从两个分区字段'dt'和'hour'中提取，可以配置：'$dt $hour:00:00' |

默认提取器基于由分区字段组成的时间戳模式。您还可以根据PartitionTimeExtractor接口指定完全自定义分区提取的实现。

|  |
| --- |
| public class HourPartTimeExtractor implements PartitionTimeExtractor {  @Override  public LocalDateTime extract(List<String> keys, List<String> values) {  String dt = values.get(0);  String hour = values.get(1);  return Timestamp.valueOf(dt + " " + hour + ":00:00").toLocalDateTime();  } } |

###### 分区提交策略

分区提交策略定义了提交分区时采取的操作。

* 第一个是 Metastore，只有 hive 表支持 Metastore 策略，文件系统通过目录结构管理分区。
* 第二个是成功文件，会在分区对应的目录中写入一个空文件。

|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **默认值** | **类型** | **描述** |
| sink.partition-commit.policy.kind | default | String | 提交分区的策略是通知下游应用该分区已完成写入，该分区已准备好被读取。Metastore：将分区添加到 Metastore。只有 hive 表支持 Metastore 策略，文件系统通过目录结构管理分区。成功文件：将“\_success”文件添加到目录。两者都可以同时配置：'metastore,success-file'。自定义：使用策略类来创建提交策略。支持配置多个策略：'metastore,success-file' |
| sink.partition-commit.policy.class | (none) | String | 用于实现 PartitionCommitPolicy 接口的分区提交策略类。仅适用于自定义提交策略 |
| sink.partition-commit.success-file.name | \_SUCCESS | String | 成功文件分区提交策略的文件名，默认为'\_SUCCESS'。 |

您可以扩展提交策略的实现，自定义提交策略实现如：

|  |
| --- |
| public class AnalysisCommitPolicy implements PartitionCommitPolicy {  private HiveShell hiveShell;   @Override  public void commit(Context context) throws Exception {  if (hiveShell == null) {  hiveShell = createHiveShell(context.catalogName());  }   hiveShell.execute(String.format(  "ALTER TABLE %s ADD IF NOT EXISTS PARTITION (%s = '%s') location '%s'",  context.tableName(),  context.partitionKeys().get(0),  context.partitionValues().get(0),  context.partitionPath()));  hiveShell.execute(String.format(  "ANALYZE TABLE %s PARTITION (%s = '%s') COMPUTE STATISTICS FOR COLUMNS",  context.tableName(),  context.partitionKeys().get(0),  context.partitionValues().get(0)));  } } |

##### 接收器并行

将文件写入外部文件系统（包括 Hive）的并行度可以通过相应的 table 选项进行配置，在流模式和批处理模式下都支持。默认情况下，并行度配置为与其最后一个上游链操作符的并行度相同。当配置了与上游并行度不同的并行度时，写入文件的操作符和压缩文件的操作符（如果使用）将应用并行度。

|  |  |  |  |
| --- | --- | --- | --- |
| **参数** | **默认值** | **类型** | **描述** |
| sink.parallelism | (none) | Integer | 将文件写入外部文件系统的并行性。该值应大于零，否则将引发异常 |

|  |
| --- |
| **注意：**目前，当且仅当上游的更改日志模式为INSERT-ONLY时，才支持配置接收器并行性。否则会抛出异常。 |

#### 完整示例

以下示例展示了如何使用文件系统连接器编写流式查询以将数据从 Kafka 写入文件系统并运行批处理查询以读取该数据。

|  |
| --- |
| CREATE TABLE kafka\_table (  user\_id STRING,  order\_amount DOUBLE,  log\_ts TIMESTAMP(3),  WATERMARK FOR log\_ts AS log\_ts - INTERVAL '5' SECOND -- Define watermark on TIMESTAMP column ) WITH (...);  CREATE TABLE fs\_table (  user\_id STRING,  order\_amount DOUBLE,  dt STRING,  `hour` STRING ) PARTITIONED BY (dt, `hour`) WITH (  'connector'='filesystem',  'path'='...',  'format'='parquet',  'sink.partition-commit.delay'='1 h',  'sink.partition-commit.policy.kind'='success-file' );  -- streaming sql, insert into file system table INSERT INTO fs\_table SELECT  user\_id,  order\_amount,  DATE\_FORMAT(log\_ts, 'yyyy-MM-dd'),  DATE\_FORMAT(log\_ts, 'HH') FROM kafka\_table;  -- batch sql, select with partition pruning SELECT \* FROM fs\_table WHERE dt='2020-05-20' and `hour`='12'; |

如果在TIMESTAMP\_LTZ列上定义了水印并用于partition-time提交，sink.partition-commit.watermark-time-zone则需要设置为会话时区，否则几个小时后可能会发生分区提交。

|  |
| --- |
| CREATE TABLE kafka\_table (  user\_id STRING,  order\_amount DOUBLE,  ts BIGINT, -- time in epoch milliseconds  ts\_ltz AS TO\_TIMESTAMP\_LTZ(ts, 3),  WATERMARK FOR ts\_ltz AS ts\_ltz - INTERVAL '5' SECOND -- Define watermark on TIMESTAMP\_LTZ column ) WITH (...);  CREATE TABLE fs\_table (  user\_id STRING,  order\_amount DOUBLE,  dt STRING,  `hour` STRING ) PARTITIONED BY (dt, `hour`) WITH (  'connector'='filesystem',  'path'='...',  'format'='parquet',  'partition.time-extractor.timestamp-pattern'='$dt $hour:00:00',  'sink.partition-commit.delay'='1 h',  'sink.partition-commit.trigger'='partition-time',  'sink.partition-commit.watermark-time-zone'='Asia/Shanghai', -- Assume user configured time zone is 'Asia/Shanghai'  'sink.partition-commit.policy.kind'='success-file' );  -- streaming sql, insert into file system table INSERT INTO fs\_table  SELECT   user\_id,   order\_amount,   DATE\_FORMAT(ts\_ltz, 'yyyy-MM-dd'),  DATE\_FORMAT(ts\_ltz, 'HH')  FROM kafka\_table;  -- batch sql, select with partition pruning SELECT \* FROM fs\_table WHERE dt='2020-05-20' and `hour`='12'; |

### [Upsert Kafka](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/upsert-kafka/)

Upsert Kafka 连接器支持以 upsert 方式从 Kafka topic 中读取数据并将数据写入 Kafka topic。

作为 source，upsert-kafka 连接器生产 changelog 流，其中每条数据记录代表一个更新或删除事件。更准确地说，数据记录中的 value 被解释为同一 key 的最后一个 value 的 UPDATE，如果有这个 key（如果不存在相应的 key，则该更新被视为 INSERT）。用表来类比，changelog 流中的数据记录被解释为 UPSERT，也称为 INSERT/UPDATE，因为任何具有相同 key 的现有行都被覆盖。另外，value 为空的消息将会被视作为 DELETE 消息。

作为 sink，upsert-kafka 连接器可以消费 changelog 流。它会将 INSERT/UPDATE\_AFTER 数据作为正常的 Kafka 消息写入，并将 DELETE 数据以 value 为空的 Kafka 消息写入（表示对应 key 的消息被删除）。Flink 将根据主键列的值对数据进行分区，从而保证主键上的消息有序，因此同一主键上的更新/删除消息将落在同一分区中。

#### 依赖

为了使用 Upsert Kafka 连接器，使用构建自动化工具（例如 Maven 或 SBT）的项目和带有 SQL JAR 包的 SQL 客户端都需要以下依赖项。

|  |  |  |
| --- | --- | --- |
| **UpsertKafkaversion** | **Maven dependency** | **SQL Client JAR** |
| universal | <dependency>  <groupId>org.apache.flink</groupId> <artifactId>flink-connector-kafka\_2.11</artifactId>  <version>1.13.0</version> </dependency> | [Download](https://repo.maven.apache.org/maven2/org/apache/flink/flink-sql-connector-kafka_2.11/1.13.0/flink-sql-connector-kafka_2.11-1.13.0.jar) |

#### 完整示例

下面的示例展示了如何创建和使用 Upsert Kafka 表：

|  |
| --- |
| CREATE TABLE pageviews\_per\_region (  user\_region STRING,  pv BIGINT,  uv BIGINT,  PRIMARY KEY (user\_region) NOT ENFORCED ) WITH (  'connector' = 'upsert-kafka',  'topic' = 'pageviews\_per\_region',  'properties.bootstrap.servers' = '...',  'key.format' = 'avro',  'value.format' = 'avro' );  CREATE TABLE pageviews (  user\_id BIGINT,  page\_id BIGINT,  viewtime TIMESTAMP,  user\_region STRING,  WATERMARK FOR viewtime AS viewtime - INTERVAL '2' SECOND ) WITH (  'connector' = 'kafka',  'topic' = 'pageviews',  'properties.bootstrap.servers' = '...',  'format' = 'json' );  -- 计算 pv、uv 并插入到 upsert-kafka sink INSERT INTO pageviews\_per\_region SELECT  user\_region,  COUNT(\*),  COUNT(DISTINCT user\_id) FROM pageviews GROUP BY user\_region; |

|  |
| --- |
| **注意：**确保在 DDL 中定义主键。 |

#### 连接器参数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **必选** | **默认值** | **类型** | **描述** |
| connector | 必选 | (none) | String | 指定要使用的连接器，Upsert Kafka 连接器使用：'upsert-kafka'。 |
| topic | 必选 | (none) | String | 用于读取和写入的 Kafka topic 名称 |
| properties.bootstrap.servers | 必选 | (none) | String | 以逗号分隔的 Kafka brokers 列表。 |
| properties.\* | 可选 | (none) | String | 该选项可以传递任意的 Kafka 参数。选项的后缀名必须匹配定义在 [Kafka 参数文档](https://kafka.apache.org/documentation/" \l "configuration)中的参数名。 Flink 会自动移除 选项名中的 "properties." 前缀，并将转换后的键名以及值传入 KafkaClient。 例如，你可以通过 'properties.allow.auto.create.topics' = 'false' 来禁止自动创建 topic。 但是，某些选项，例如'key.deserializer' 和 'value.deserializer' 是不允许通过该方式传递参数，因为 Flink 会重写这些参数的值 |
| key.format | 必选 | (none) | String | 用于对 Kafka 消息中 key 部分序列化和反序列化的格式。key 字段由 PRIMARY KEY 语法指定。支持的格式包括 'csv'、'json'、'avro'。请参考[格式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/overview/)页面以获取更多详细信息和格式参数 |
| key.fields-prefix | 可选 | (none) | String | 为键格式的所有字段定义自定义前缀，以避免与值格式的字段发生名称冲突。默认情况下，前缀为空。如果定义了自定义前缀，表架构和'key.fields'将使用前缀名称。在构造密钥格式的数据类型时，将删除前缀，并在密钥格式中使用无前缀的名称。请注意，此选项要求'value.fields-include' 必须设置为'EXCEPT\_KEY'。 |
| value.format | 必选 | (none) | String | 用于对 Kafka 消息中 value 部分序列化和反序列化的格式。支持的格式包括 'csv'、'json'、'avro'。请参考[格式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/overview/)页面以获取更多详细信息和格式参数。 |
| value.fields-include | 必选 | 'ALL' | String | 控制哪些字段应该出现在 value 中。可取值：   * ALL：消息的 value 部分将包含 schema 中所有的字段，包括定义为主键的字段。 * EXCEPT\_KEY：记录的 value 部分包含 schema 的所有字段，定义为主键的字段除外。 |
| sink.parallelism | 可选 | (none) | Integer | 定义 upsert-kafka sink 算子的并行度。默认情况下，由框架确定并行度，与上游链接算子的并行度保持一致 |
| sink.buffer-flush.max-rows | 可选 | 0 | Integer | 缓存刷新前，最多能缓存多少条记录。当 sink 收到很多同 key 上的更新时，缓存将保留同 key 的最后一条记录，因此 sink 缓存能帮助减少发往 Kafka topic 的数据量，以及避免发送潜在的 tombstone 消息。 可以通过设置为 '0' 来禁用它。默认，该选项是未开启的。注意，如果要开启 sink 缓存，需要同时设置 'sink.buffer-flush.max-rows' 和 'sink.buffer-flush.interval' 两个选项为大于零的值。 |
| sink.buffer-flush.interval | 可选 | 0 | Duration | 缓存刷新的间隔时间，超过该时间后异步线程将刷新缓存数据。当 sink 收到很多同 key 上的更新时，缓存将保留同 key 的最后一条记录，因此 sink 缓存能帮助减少发往 Kafka topic 的数据量，以及避免发送潜在的 tombstone 消息。 可以通过设置为 '0' 来禁用它。默认，该选项是未开启的。注意，如果要开启 sink 缓存，需要同时设置 'sink.buffer-flush.max-rows' 和 'sink.buffer-flush.interval' 两个选项为大于零的值。 |

#### 特性

##### 键和值格式

有关键和值格式的更多解释，请参阅[常规 Kafka 连接器](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/connectors/kafka.html" \l "key-and-value-formats)。但是，请注意，此连接器同时需要键和值格式，其中键字段源自PRIMARY KEY约束。

以下示例显示如何指定和配置键和值格式。格式选项以加号'key'或'value'加号格式标识符为前缀。

|  |
| --- |
| CREATE TABLE KafkaTable (  `ts` TIMESTAMP(3) METADATA FROM 'timestamp',  `user\_id` BIGINT,  `item\_id` BIGINT,  `behavior` STRING,  PRIMARY KEY (`user\_id`) NOT ENFORCED ) WITH (  'connector' = 'upsert-kafka',  ...   'key.format' = 'json',  'key.json.ignore-parse-errors' = 'true',   'value.format' = 'json',  'value.json.fail-on-missing-field' = 'false',  'value.fields-include' = 'EXCEPT\_KEY' ) |

##### 主键约束

Upsert Kafka 总是以 upsert 方式工作，并且需要在 DDL 中定义主键。唯一性。定义的主键将决定哪些领域出现在卡夫卡消息的键中。

##### 一致性保证

默认情况下，如果[允许检查点](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/stream/state/checkpointing.html" \l "enabling-and-configuring-checkpointing)，Upsert Kafka sink 会保证至少一次将数据插入 Kafka 话题。

这意味着，Flink 可以将具有相同密钥的重复记录写入 Kafka 主题。 但由于该连接器以 upsert 的模式工作，连接器作为源读入时，可以确保具有相同键主值下仅最后一条消息会生效。因此，upsert-kafka连接器可以[HBase沉](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/connectors/hbase.html)一样实现幂像等写入。

##### 为每个分区生成各自的水印

Flink 支持根据 Upsert Kafka 的每一个细分的数据特性发送相应的水印。当这个特性的时候，水印是在 Kafka 消费者内部生成的。数据源产生的水印是生命该消费者负责的所有部分中当前最小的水印。如果该消费者负责的部分区域是空闲的，那么整体的水印并不会前进。在这种情况下，可以通过设置。合适的**[table.exec.source.idle-timeout](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/config.html" \l "table-exec-source-idle-timeout)**来缓解这个问题。

如想获得更多细节，请[kafka水印策略](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/event_timestamps_watermarks.html" \l "watermark-strategies-and-the-kafka-connector)。

##### 数据类型映射

Upsert Kafka 用字节存储消息的密钥和值，因此没有模式或数据类型。消息按格式进行序列化和反序列化，例如：csv、json、avro。因此数据类型映射表由指定的格式确定。请参考[格式](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/connectors/formats/)页面以获取更多详细信息

## FlinkSQL的原理及调优

SQL 是数据分析中使用最广泛的语言。Flink Table API 和 SQL 使用户能够以更少的时间和精力定义高效的流分析应用程序。此外，Flink Table API 和 SQL 是高效优化过的，它集成了许多查询优化和算子优化。但并不是所有的优化都是默认开启的，因此对于某些工作负载，可以通过打开某些选项来提高性能。

在这一页，我们将介绍一些实用的优化选项以及流式聚合的内部原理，它们在某些情况下能带来很大的提升。

|  |
| --- |
| The streaming aggregation optimizations mentioned in this page are all supported for [Group Aggregations](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/group-agg/) and [Window TVF Aggregations](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/) now. |

### 配置

Table 和 SQL API 的默认配置能够确保结果准确，同时也提供可接受的性能。

根据 Table 程序的需求，可能需要调整特定的参数用于优化。例如，无界流程序可能需要保证所需的状态是有限的(请参阅 [流式概念](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/streaming/query_configuration.html)).

#### 概览

在每个 TableEnvironment 中，TableConfig 提供用于当前会话的配置项。

对于常见或者重要的配置项，TableConfig 提供带有详细注释的 getters 和 setters 方法。

对于更加高级的配置，用户可以直接访问底层的 key-value 配置项。以下章节列举了所有可用于调整 Flink Table 和 SQL API 程序的配置项。

|  |
| --- |
| **注意：**因为配置项会在执行操作的不同时间点被读取，所以推荐在实例化 TableEnvironment 后尽早地设置配置项。 |

|  |
| --- |
| // instantiate table environment TableEnvironment tEnv = ...  // access flink configuration Configuration configuration = tEnv.getConfig().getConfiguration(); // set low-level key-value options configuration.setString("table.exec.mini-batch.enabled", "true"); configuration.setString("table.exec.mini-batch.allow-latency", "5 s"); configuration.setString("table.exec.mini-batch.size", "5000"); |

|  |
| --- |
| **注意：**目前，key-value 配置项仅被 Blink planner 支持。 |

#### 执行配置

以下选项可用于优化查询执行的性能。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **场景** | **默认值** | **类型** | **描述** |
| table.exec.async-lookup.buffer-capacity | 批/流 | 100 | Integer | 异步查找联接可以触发的最大异步 i/o 操作数。 |
| table.exec.async-lookup.timeout | 批/流 | 3 min | Duration | 异步操作完成的异步超时。 |
| table.exec.disabled-operators | 批 | (none) | String | 主要用于测试。逗号分隔的操作员名称列表，每个名称代表一种禁用的操作员。可以禁用的运算符包括“NestedLoopJoin”、“ShuffleHashJoin”、“BroadcastHashJoin”、“SortMergeJoin”、“HashAgg”、“SortAgg”。默认情况下，没有操作员被禁用。 |
| table.exec.mini-batch.allow-latency | 流 | 0 ms | Duration | MiniBatch 可以使用最大延迟来缓冲输入记录。MiniBatch 是缓冲输入记录以减少状态访问的优化。MiniBatch 以允许的延迟时间间隔和达到最大缓冲记录数时触发。注意：如果 table.exec.mini-batch.enabled 设置为 true，则其值必须大于零 |
| table.exec.mini-batch.enabled | 流 | false | Boolean | 指定是否启用 MiniBatch 优化。MiniBatch 是缓冲输入记录以减少状态访问的优化。默认情况下禁用此功能。要启用此功能，用户应将此配置设置为 true。注意：如果启用了小批量，则必须设置“table.exec.mini-batch.allow-latency”和“table.exec.mini-batch.size”。 |
| table.exec.mini-batch.size | 流 | -1 | Long | MiniBatch 可以缓冲的最大输入记录数。MiniBatch 是缓冲输入记录以减少状态访问的优化。MiniBatch 以允许的延迟时间间隔和达到最大缓冲记录数时触发。注意：MiniBatch 目前仅适用于非窗口聚合。如果 table.exec.mini-batch.enabled 设置为 true，则其值必须为正 |
| table.exec.resource.default-parallelism | 批/流 | -1 | Integer | 为所有运算符（例如聚合、联接、过滤器）设置默认并行度以与并行实例一起运行。此配置的优先级高于 StreamExecutionEnvironment 的并行性（实际上，此配置覆盖了 StreamExecutionEnvironment 的并行性）。值 -1 表示未设置默认并行度，然后将回退以使用 StreamExecutionEnvironment 的并行度。 |
| table.exec.shuffle-mode | 批 | "ALL\_EDGES\_BLOCKING" | String | 设置 exec shuffle 模式。 接受的值为：   * ALL\_EDGES\_BLOCKING：所有边将使用阻塞洗牌。 * FORWARD\_EDGES\_PIPELINED：前向边缘将使用流水线洗牌，其他人阻塞。 * POINTWISE\_EDGES\_PIPELINED：逐点边缘将使用流水线洗牌，其他人阻塞。逐点边包括前向边和重新缩放边。 * ALL\_EDGES\_PIPELINED：所有边都将使用流水线洗牌。 * batch: 一样ALL\_EDGES\_BLOCKING。已弃用。 * pipelined: 一样ALL\_EDGES\_PIPELINED。已弃用。   注意：阻塞 shuffle 意味着在发送到消费者任务之前将完全生产数据。Pipelined shuffle 意味着数据将在产生后发送给消费者任务。 |
| table.exec.sink.not-null-enforcer | 批/流 | ERROR | Enum  Possible values: [ERROR, DROP] | 表上的 NOT NULL 列约束强制不能将空值插入到表中。Flink 支持“error”（默认）和“drop”强制行为。默认情况下，当空值写入 NOT NULL 列时，Flink 将检查值并抛出运行时异常。用户可以将行为更改为“删除”以静默删除此类记录而不会引发异常 |
| table.exec.sort.async-merge-enabled | 批 | true | Boolean | 是否异步合并已排序的溢出文件 |
| table.exec.sort.default-limit | 批 | -1 | Integer | 当用户在 order by 后未设置限制时的默认限制。-1 表示忽略此配置。 |
| table.exec.sort.max-num-file-handles | 批 | 128 | Integer | 外部归并排序的最大扇入。它限制了每个操作员的文件句柄数。如果太小，可能会导致中间合并。但是如果太大，就会造成同时打开的文件过多，消耗内存，导致随机读取 |
| table.exec.source.cdc-events-duplicate | 流 | false | Boolean | 指示作业中的 CDC（变更数据捕获）源是否会产生重复的变更事件，这些变更事件需要框架进行重复数据删除并获得一致的结果。CDC 源是指产生完整更改事件的源，包括 INSERT/UPDATE\_BEFORE/UPDATE\_AFTER/DELETE，例如 Debezium 格式的 Kafka 源。此配置的值默认为 false。 但是，存在重复的更改事件是一种常见情况。因为通常 CDC 工具（例如 Debezium）在发生故障转移时至少进行一次交付。因此，在异常情况下，Debezium 可能会向 Kafka 传递重复的更改事件，而 Flink 会收到重复的事件。这可能会导致 Flink 查询得到错误结果或意外异常。 因此，如果您的CDC工具是至少一次交付，建议开启此配置。启用此配置需要在 CDC 源上定义 PRIMARY KEY。主键将用于对更改事件进行重复数据删除并以额外的有状态操作符为代价生成规范化的更改日志流。 |
| table.exec.source.idle-timeout | 流 | 0 ms | Duration | 当一个源在超时时间内没有收到任何元素时，它将被标记为临时空闲。这允许下游任务推进其水印，而无需在空闲时等待来自该源的水印。默认值为 0，表示未启用检测源空闲。 |
| table.exec.spill-compression.block-size | 批 | “64 KB” | MemorySize | 溢出数据时用于进行压缩的内存大小。内存越大，压缩率越高，但作业会消耗更多的内存资源 |
| table.exec.spill-compression.enabled | 批 | true | boolean | 是否压缩溢出的数据。目前我们只支持对 sort 和 hash-agg 以及 hash-join 操作符压缩溢出数据 |
| table.exec.state.ttl | 流 | 0 ms | Duration | 指定空闲状态（即未更新的状态）将保留多长时间的最小时间间隔。状态在空闲时间少于最短时间之前永远不会被清除，并且会在空闲后的某个时间被清除。默认是从不清理状态。注意：清理状态需要额外的簿记开销。默认值为 0，这意味着它永远不会清理状态。 |
| table.exec.window-agg.buffer-size-limit | 批 | 100000 | Integer | 设置组窗口 agg 运算符中使用的窗口元素缓冲区大小限制 |

#### 优化器配置

以下配置可以用于调整查询优化器的行为以获得更好的执行计划。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **场景** | **默认值** | **类型** | **描述** |
| table.optimizer.agg-phase-strategy | 批/流 | "AUTO" | String | 聚合策略。默认AUTO，支持参数AUTO、TWO\_PHASE(使用LocalGlobal两阶段聚合)、ONE\_PHASE(仅使用Global一阶段聚合) |
| table.optimizer.distinct-agg.split.bucket-num | 流 | 1024 | Integer | 拆分不同聚合时配置桶数。该数字在第一级聚合中用于计算桶键 'hash\_code(distinct\_key) % BUCKET\_NUM'，该键用作拆分后的附加组键。 |
| table.optimizer.distinct-agg.split.enabled | 流 | false | Boolean | 告诉优化器是否将不同的聚合（例如 COUNT(DISTINCT col), SUM(DISTINCT col)）分成两个级别。第一个聚合由一个额外的键打乱，该键是使用 distinct\_key 的哈希码和桶数计算的。当不同聚合中存在数据倾斜时，这种优化非常有用，并且能够扩展作业。默认为假。 |
| table.optimizer.join-reorder-enabled | 批/流 | false | Boolean | 在优化器中启用连接重新排序。默认为禁用 |
| table.optimizer.join.broadcast-threshold | 批 | 1048576 | Long | 配置表的最大大小（以字节为单位），该表将在执行连接时广播到所有工作节点。通过将此值设置为 -1 来禁用广播 |
| table.optimizer.multiple-input-enabled | 批 | true | Boolean | 当它为真时，优化器会将带有流水线混洗的算子合并为一个多输入算子，以减少混洗并提高性能。默认值为真 |
| table.optimizer.reuse-source-enabled | 批/流 | true | Boolean | 当它为真时，优化器将尝试找出重复的表源并重用它们。这仅在 table.optimizer.reuse-sub-plan-enabled 为 true 时有效 |
| table.optimizer.reuse-sub-plan-enabled | 批/流 | true | Boolean | 当它为真时，优化器将尝试找出重复的子计划并重用它们 |
| table.optimizer.source.predicate-pushdown-enabled | 批/流 | true | Boolean | 当它为真时，优化器会将谓词下推到 FilterableTableSource 中。默认值为真 |

#### Planner 配置

以下配置可以用于调整 planner 的行为。

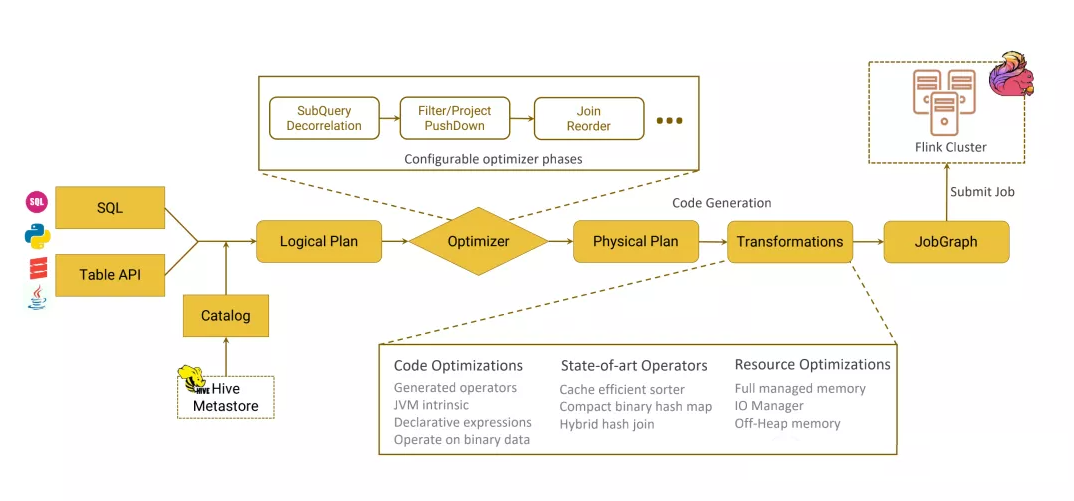
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **场景** | **默认值** | **类型** | **描述** |
| table.dml-sync | 批/流 | false | Boolean | 指定是异步执行还是同步执行 DML 作业（即插入操作）。默认情况下，执行是异步的，因此您可以同时提交多个 DML 作业。如果将此选项设置为 true，插入操作将等待作业完成。 |
| table.dynamic-table-options.enabled | 批/流 | false | Boolean | 启用或禁用用于动态指定表选项的 OPTIONS 提示，如果禁用，则在指定任何 OPTIONS 提示时将抛出异常 |
| table.generated-code.max-length | 批/流 | 64000 | Integer | 指定将生成的代码拆分为子函数调用的阈值。Java 的最大方法长度为 64 KB。如有必要，此设置允许更精细的粒度。 |
| table.local-time-zone | 批/流 | "default" | String | 本地时区定义当前会话时区 ID。它在与 <code>TIMESTAMP WITH LOCAL TIME ZONE</code> 之间转换时使用。在内部，具有本地时区的时间戳始终以 UTC 时区表示。但是，当转换为不包含时区的数据类型（例如 TIMESTAMP、TIME 或简单的 STRING）时，转换期间将使用会话时区。option 的输入要么是全名，例如“America/Los\_Angeles”，要么是自定义时区 ID，例如“GMT-08:00”。 |
| table.planner | 批/流 | BLINK | Enum  Possible values: [BLINK, OLD] | 使用“blink”规划器或“old”规划器。默认为blink planner。对于TableEnvironment，这个选项是用来构造一个TableEnvironment，但是这个选项之后就不能修改了。但是，SQL Client 没有这样的限制。注意：Flink 1.14 中将移除旧的 planner，因此该选项将过时。 |
| table.sql-dialect | 批/流 | "default" | String | SQL 方言定义了如何解析 SQL 查询。不同的 SQL 方言可能支持不同的 SQL 语法。目前支持的方言有：default 和 hive |

#### SQL Client 配置

以下配置可以用于调整 sql client 的行为。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **场景** | **默认值** | **类型** | **描述** |
| sql-client.execution.max-table-result.rows | 批/流 | 1000000 | Integer | 在表模式下要缓存的行数。如果行数超过指定值，则以 FIFO 样式重试该行。 |
| sql-client.execution.result-mode | 批/流 | TABLE | Enum  Possible values: [TABLE, CHANGELOG, TABLEAU] | 确定显示查询结果时的模式。可用值为 ['table', 'tableau', 'changelog']。“表格”模式在内存中具体化结果，并以常规的分页表格表示形式将它们可视化。“changelog”模式不会具体化结果，而是将连续查询产生的结果流可视化。'tableau' 模式更像是一种传统的方式，它会直接以 tableau 格式在屏幕上显示结果。 |
| sql-client.verbose | 批/流 | false | Boolean | 确定是否将详细输出输出到控制台。如果将选项设置为 true，它将打印异常堆栈。否则，它只输出原因。 |

### FlinkSql的工作机制



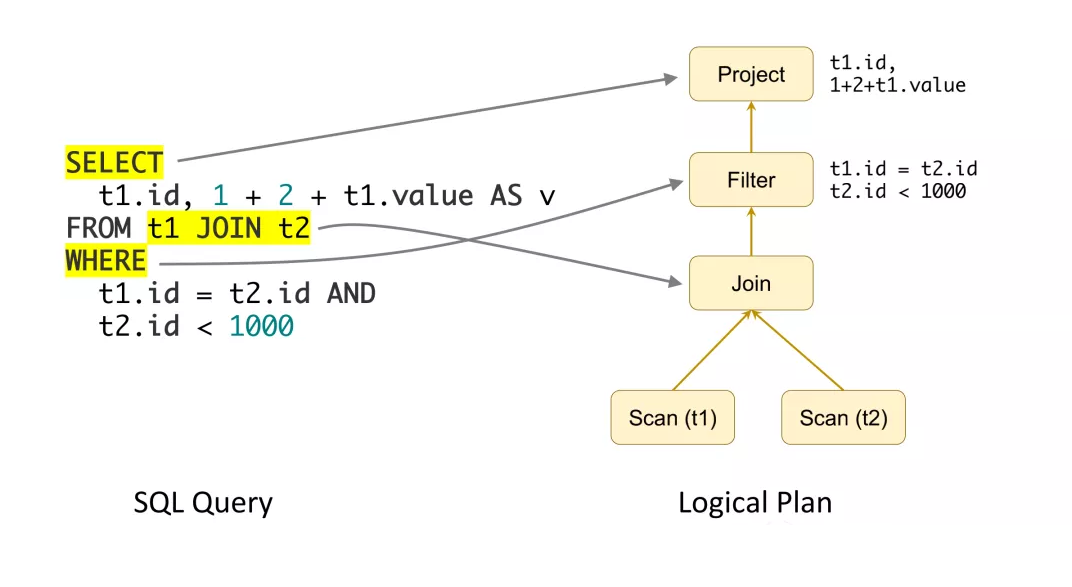
catalog支持hive 的metastore，也支持自定义Catalog。  
API到Logical plan，会有catalg参与进来-目前是可以基于hive metastore，也可以自定义，catalog会提供，比如udf参数，返回值类型，表路径等等信息。  
logical plan是优化起点，会被交给优化器optimizer进行优化，比如subquery 拆解，fliter/project下推，join recorder等，其实现过程中大量使用了calcite框架  
Physical plan使用code generation生成transformations，这里也是做了大量优化，比如Code Optimizations，state-of-art opertors，resource Optimizations等具体可以看上图。  
transformations之后就可以生成JobGraph了，可以用来提交到flink集群。  
批和流的区别重点呈现在Pysical plan和transformations。

### 批处理SQL解析过程

案例如下：

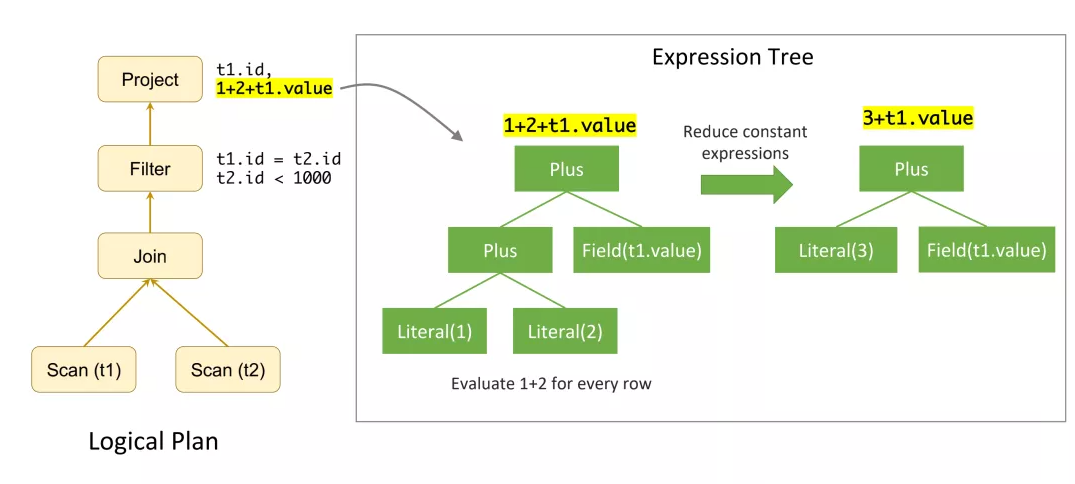
|  |
| --- |
| select  t1.id,1+2+t1.value as v from t1 join t2 where  t1.id = t2.id AND  t2.id < 1000 |

* 首先，sql表达到逻辑执行计划，select操作对应Project，join对应join，where对应fliter，该逻辑计划如下图：

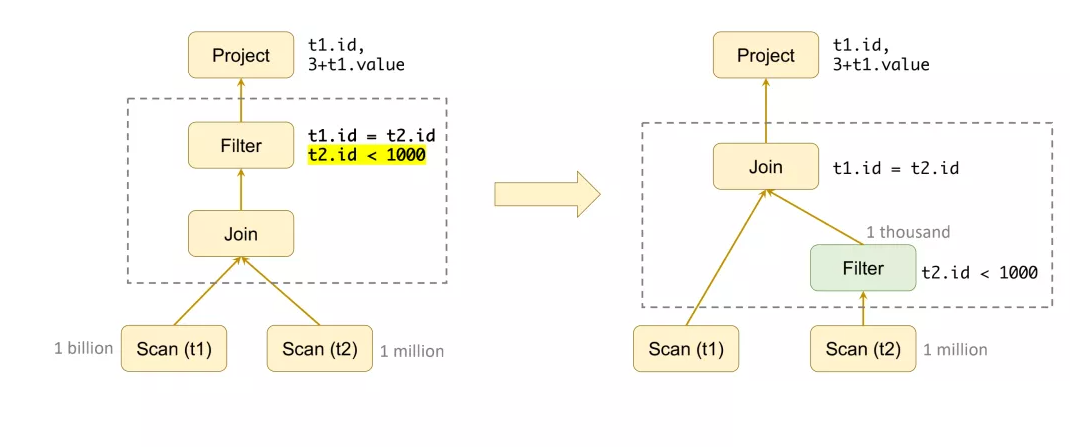


* 逻辑执行计划是优化的开始，案例中的sql优化过程如下：

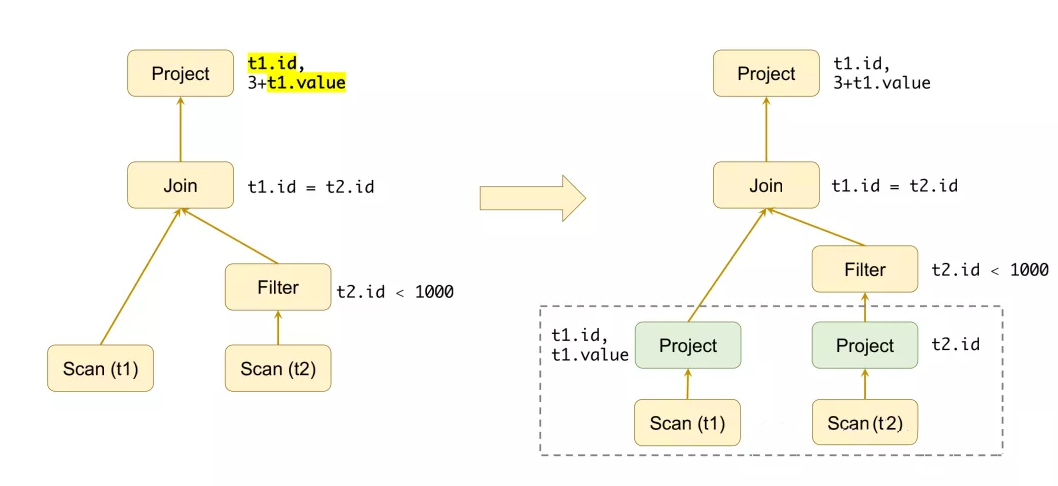
常量折叠，也即是对sql中的常量的加减乘除等操作进行预计算，避免执行过程频繁对常量重复执行加减乘除计算：



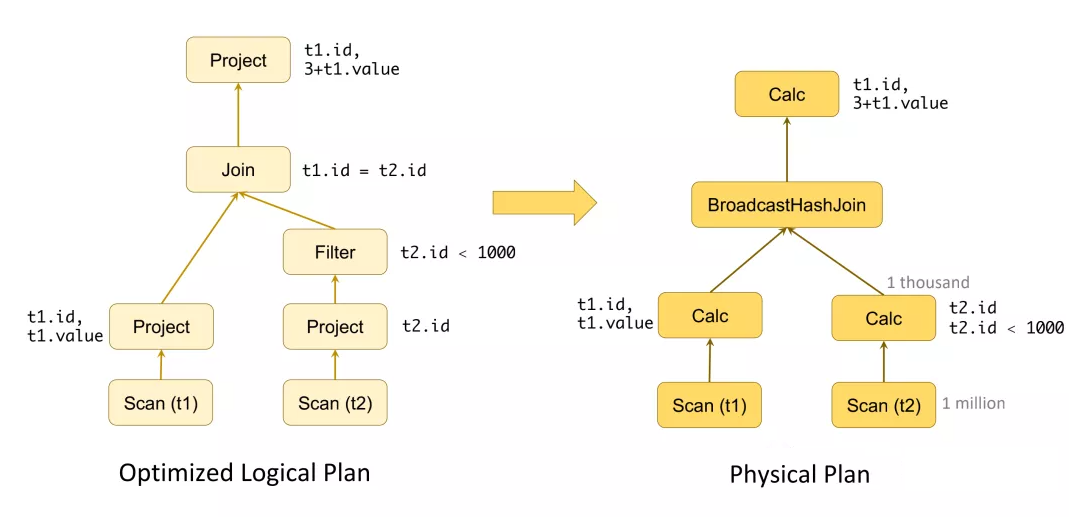
上图常量折叠前：1+2+t1.value；折叠后：3+t1.value，逻辑执行计划缩减了一个大步骤。  
filter下推执行，这里就是把t2.id<1000,下推到扫描 t2表的时候。  
假设不进行这一步优化，执行过程是：全量数据扫描，执行join操作，然后才进行fiter，这明显很浪费，id大于1000的不需要执行join操作，将fliter操作下推到join之前执行，减少了join的数据量，大大提升性能。



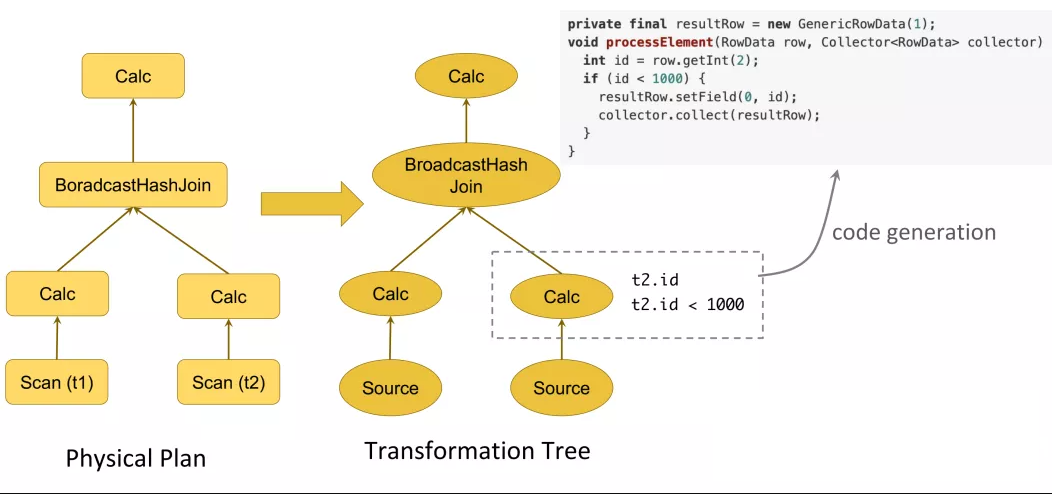
project下推执行，可以用来避免加载不需要的字段。由原来的sql可知，t1只需要加载t1.id，t1.value，t2只需要加载t2.id。假如表还有大量的其他字段，由于SQL中没用到，加载多余字段就是浪费，所以将project操作下推执行，就不需要加载无用字段。而且此时假如是列存储，只需要加载指定的列，优化更大。



* 物理执行计划生成的时候也会进行很多优化操作，如根据代价cost选择批处理join有方式(sortmergejoin，hashjoin，boradcasthashjoin)。比如这个例子，再filter下推之后，在t2.id<1000的情况下，由1 百万数据量变为了1 千条，计算cost之后，使用broadcasthashjoin最合适。

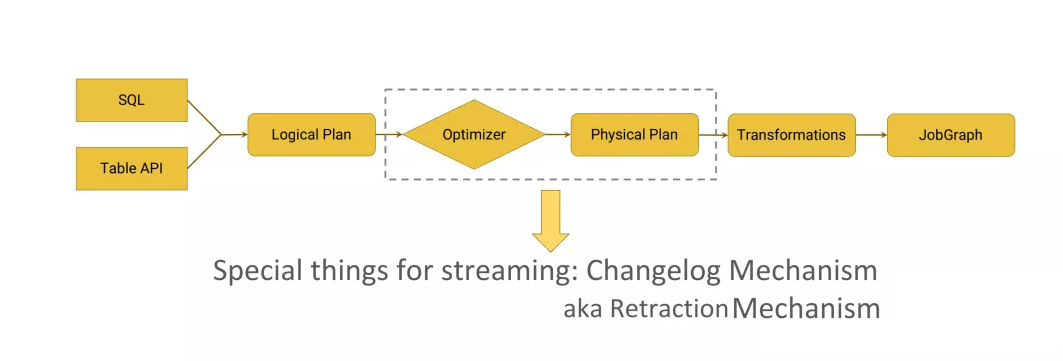


物理计划codegen过程，就是翻译成transformation tree的过程：



### 流处理SQL解析过程

flink 的流处理sql解析过程如下：



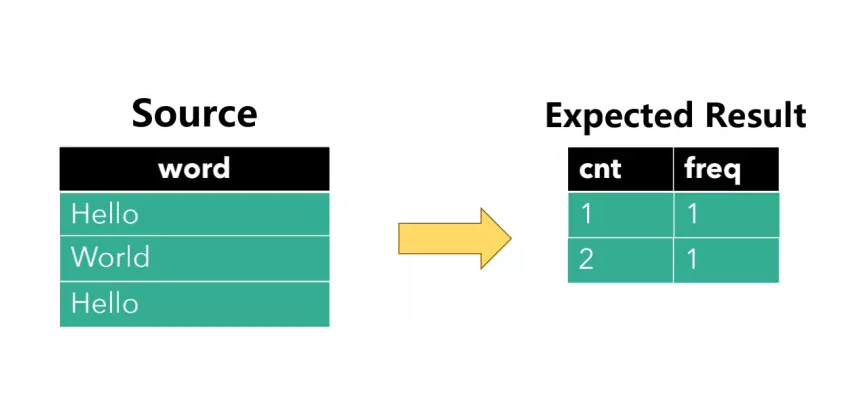
对于flink 流表的计算，在Optimizer和Physical plan过程中是需要引入changelog机制，也可以叫做retraction机制。

比如下面一条sql，表达含义就是先进行wordcount操作，得到临时表<列Row(word，cnt)>，然后统计频次cnt出现的次数。

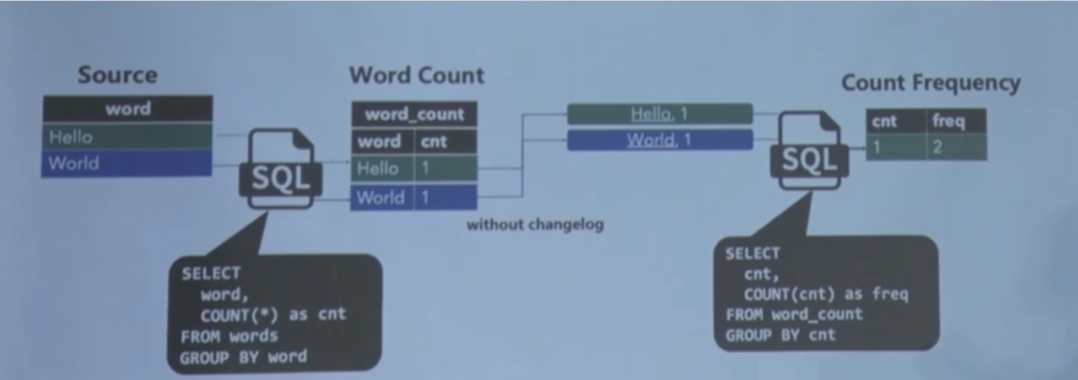
|  |
| --- |
| SELECT cnt, COUNT(cnt) as freq FROM (  SELECT word, COUNT(\*)as cnt  FROM words  GROUP BY word  ) GROUP BY cnt |

数据源先后输入的单词: hello，word ，hello。

期望的结果是cnt 值为1和2各 出现一次。

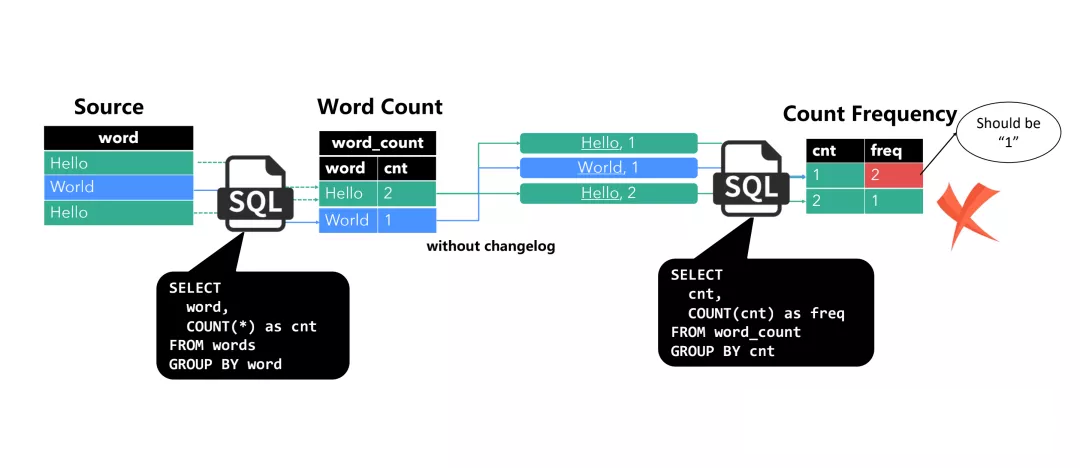


假如数据先输入了hello 和word两个词，得到计算过程及结果如下：



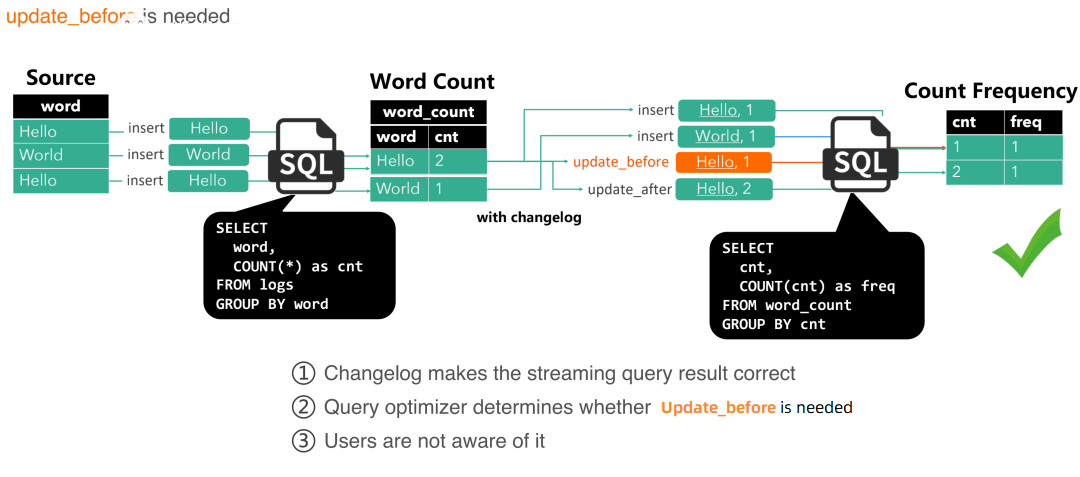
图中结果是cnt为1出现频次为2，因为word和hello各出现了一次。

此时，在输入hello，假设没有changelog机制，得到结果如下：



图中cnt 值为1出现的频次为2，cnt为2出现频次为1，这明显不符合预期，是错误的结果。

引入changelog机制，在wordcount aggregate计算结束之后，会给下游发送update\_before和update\_after两个消息，在进行cnt频率统计时，用来保证结果的正确性。



changelog机制保证了结果的正确性，同时query优化器决定者update\_before消息是否需要，并且该机制对于用户来说是无感知的。

### changelog机制介绍

#### 确定node该产生消息类型

简单来说，对于flink流处理的动态实时表，主要是有三种操作Insert，update，delete。这三种操作在transfoation之间传递的时候就是对应着三种message，下游算子接受到这三种message之后就知道该进行如何操作了，changelog机制就以此来实现的。

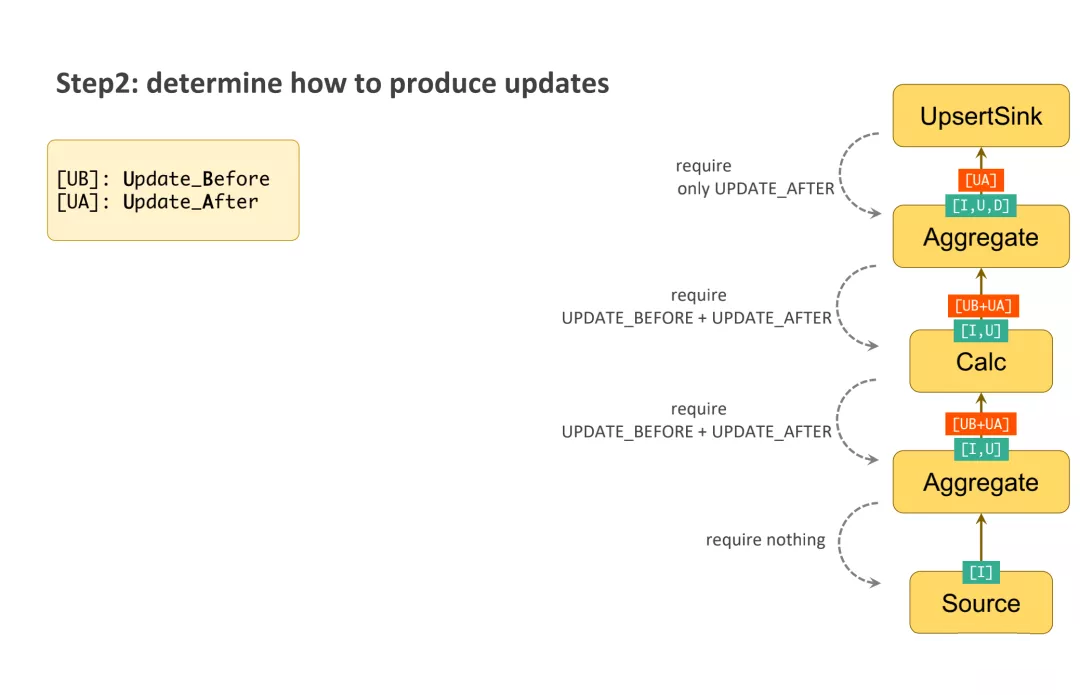


消息正向传递过程解释：

* Source到word count的Aggregate算子只产生了一种消息-INSERT message。
* word count的Aggregate之后到Calc产生了两种message，分别是新增的消息 INSERT ，更新的消息UPDATE ，Calc不会产生新的消息，直接透传消息到cnt频次统计的Aggregate算子。
* cnt频次统计的Aggregate操作向UpsertSink发送了三种消息，分别是INSERT，UPDATE，DELETE。
* 最后upsertSink就根据cnt频次统计Aggregate传递的这三种消息，做具体的操作(insert，update)，保证结果的正确性。

#### 确定update消息内容

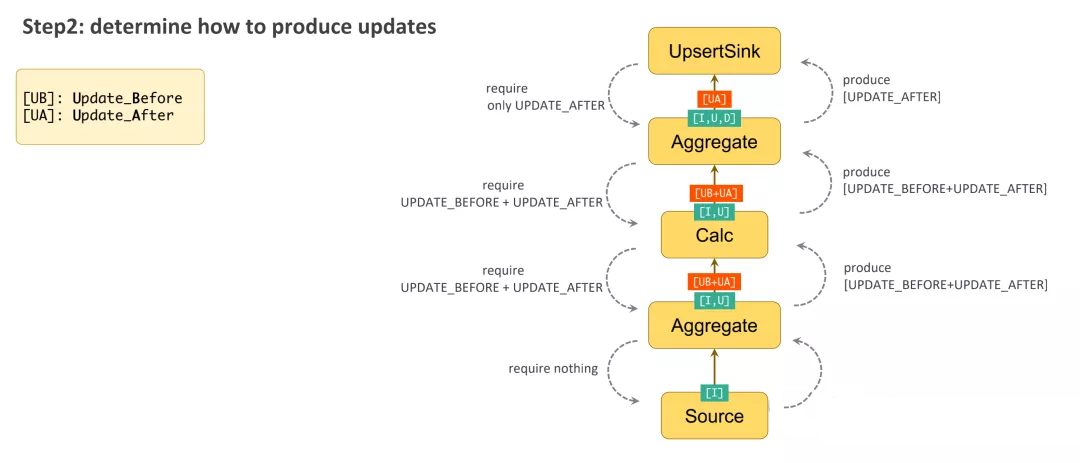
一个update消息，有两个内容：update\_before和update\_after。产生update消息的节点，可以只发送update\_after，也可以同时发送两个消息，这个决定于下游算子，推导过程可以从sink到source。



update message推导过程：

* updsertSink假设只接受Update\_After消息，那么他就会告诉频次统计的Aggregate，只发送Update\_After消息即可。sink编写时确定接受消息类型。
* Aggregate知道前一层会发送：update\_before和update\_after，而自身也需要两种消息，那么就会通知Calc节点同时发送两种消息，其实Calc节点是不会产生消息，只会透传的。
* Calc会知道wordcount的aggregate算子会产生update\_before和update\_after，而且自己也需要向下传递两个消息，所以会要求Aggregate产生update\_before和update\_after两个消息。
* wordcount的aggregate算子由于知道source不会产生update消息，所以不会要求其传递update消息。

最后就是正向传递update消息的过程，具体过程如下图右侧，source 到sink流动箭头。



经过上述过程之后，最终生成的物理计划如下：



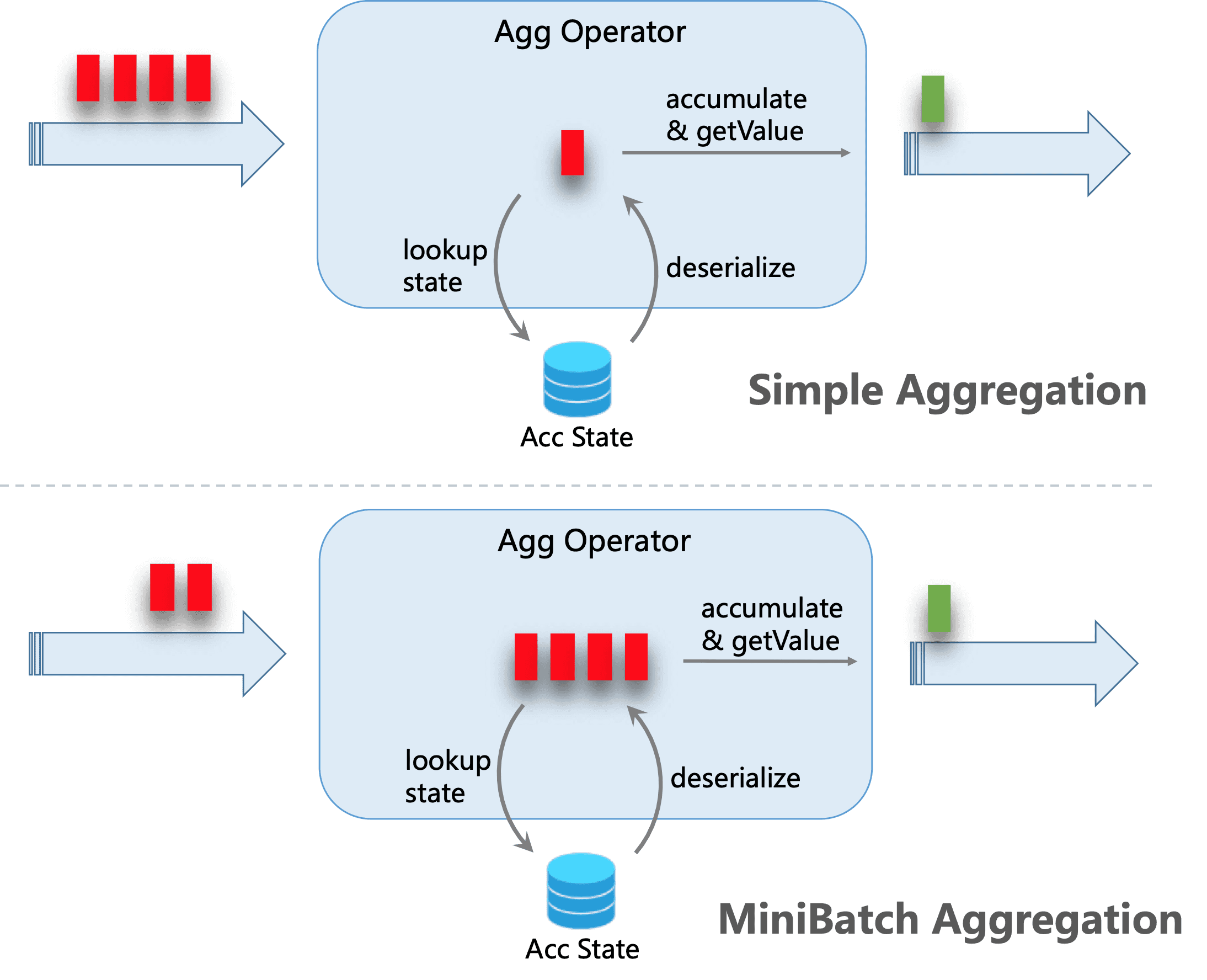
### Group Aggregate优化技巧

默认情况下，无界聚合算子是逐条处理输入的记录，即：（1）从状态中读取累加器，（2）累加/撤回记录至累加器，（3）将累加器写回状态，（4）下一条记录将再次从（1）开始处理。这种处理模式可能会增加 StateBackend 开销（尤其是对于 RocksDB StateBackend ）。此外，生产中非常常见的数据倾斜会使这个问题恶化，并且容易导致 job 发生反压。

#### 开启MicroBatch或MiniBatch（提升吞吐）

MiniBatch 聚合的核心思想是将一组输入的数据缓存在聚合算子内部的缓冲区中。当输入的数据被触发处理时，每个 key 只需一个操作即可访问状态。这样可以大大减少状态开销并获得更好的吞吐量。但是，这可能会增加一些延迟，因为它会缓冲一些记录而不是立即处理它们。这是吞吐量和延迟之间的权衡。

下图说明了 mini-batch 聚合如何减少状态操作。



* 适用场景

微批处理通过增加延迟换取高吞吐，如果您有超低延迟的要求，不建议开启微批处理。通常对于聚合的场景，微批处理可以显著的提升系统性能，建议开启。

说明 MicroBatch模式也能解决两级聚合数据抖动问题。

* 开启方式

MicroBatch和MiniBatch默认关闭，开启方式如下。

|  |
| --- |
| // instantiate table environment TableEnvironment tEnv = ...  // access flink configuration Configuration configuration = tEnv.getConfig().getConfiguration(); // set low-level key-value options configuration.setString("table.exec.mini-batch.enabled", "true"); // enable mini-batch optimization configuration.setString("table.exec.mini-batch.allow-latency", "5 s"); // use 5 seconds to buffer input records configuration.setString("table.exec.mini-batch.size", "5000"); // the maximum number of records can be buffered by each aggregate operator task |

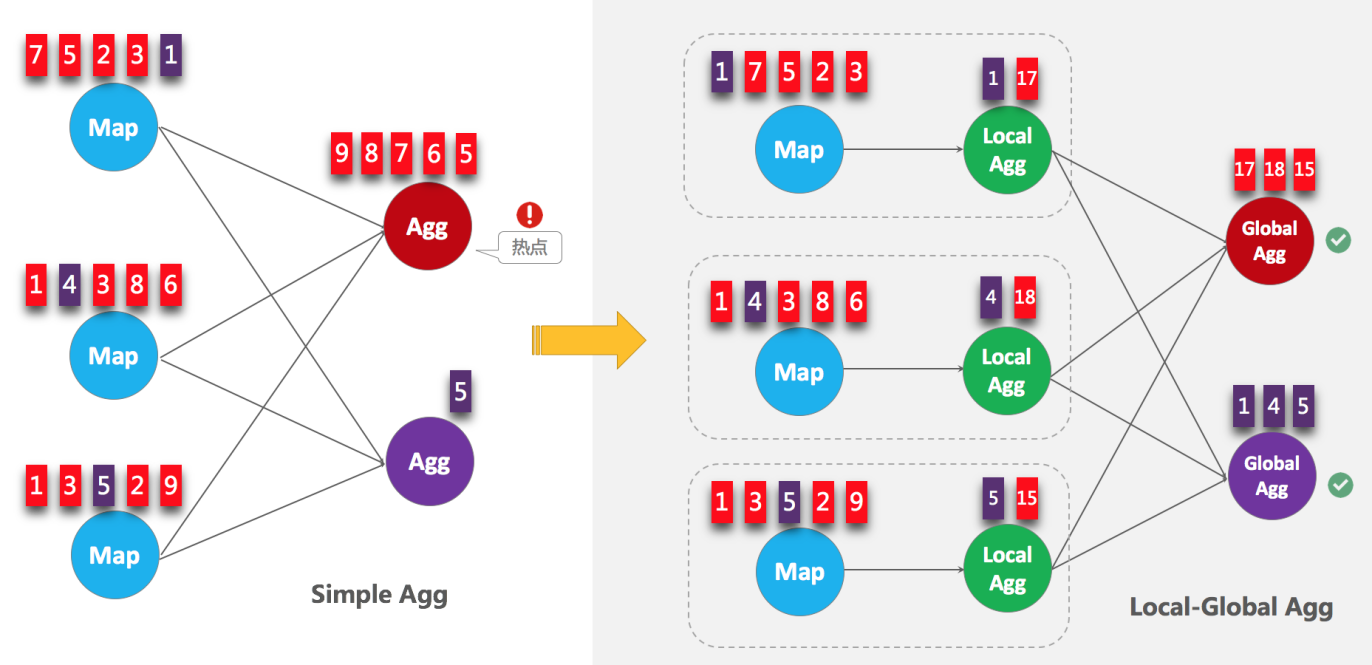
#### 开启LocalGlobal（解决常见数据热点问题）

Local-Global 聚合是为解决数据倾斜问题提出的，通过将一组聚合分为两个阶段，首先在上游进行本地聚合，然后在下游进行全局聚合，类似于 MapReduce 中的 Combine + Reduce 模式。例如，就以下 SQL 而言：

|  |
| --- |
| SELECT color, sum(id) FROM T GROUP BY color |

数据流中的记录可能会倾斜，因此某些聚合算子的实例必须比其他实例处理更多的记录，这会产生热点问题。本地聚合可以将一定数量具有相同 key 的输入数据累加到单个累加器中。全局聚合将仅接收 reduce 后的累加器，而不是大量的原始输入数据。这可以大大减少网络 shuffle 和状态访问的成本。每次本地聚合累积的输入数据量基于 mini-batch 间隔。这意味着 local-global 聚合依赖于启用了 mini-batch 优化。

下图显示了 local-global 聚合如何提高性能。



* 适用场景

LocalGlobal适用于提升如SUM、COUNT、MAX、MIN和AVG等普通聚合的性能，以及解决这些场景下的数据热点问题。

* 开启方式

下面的例子显示如何启用 local-global 聚合。

|  |
| --- |
| // instantiate table environment TableEnvironment tEnv = ...  // access flink configuration Configuration configuration = tEnv.getConfig().getConfiguration(); // set low-level key-value options configuration.setString("table.exec.mini-batch.enabled", "true"); // local-global aggregation depends on mini-batch is enabled configuration.setString("table.exec.mini-batch.allow-latency", "5 s"); configuration.setString("table.exec.mini-batch.size", "5000"); configuration.setString("table.optimizer.agg-phase-strategy", "TWO\_PHASE"); // enable two-phase, i.e. local-global aggregation |

* 判断是否生效

观察最终生成的拓扑图的节点名字中是否包含GlobalGroupAggregate或LocalGroupAggregate。

#### 开启PartialFinal（解决COUNT DISTINCT热点问题）

Local-Global 优化可有效消除常规聚合的数据倾斜，例如 SUM、COUNT、MAX、MIN、AVG。但是在处理 distinct 聚合时，其性能并不令人满意。

例如，如果我们要分析今天有多少唯一用户登录。我们可能有以下查询：

|  |
| --- |
| SELECT day, COUNT(DISTINCT user\_id) FROM T GROUP BY day |

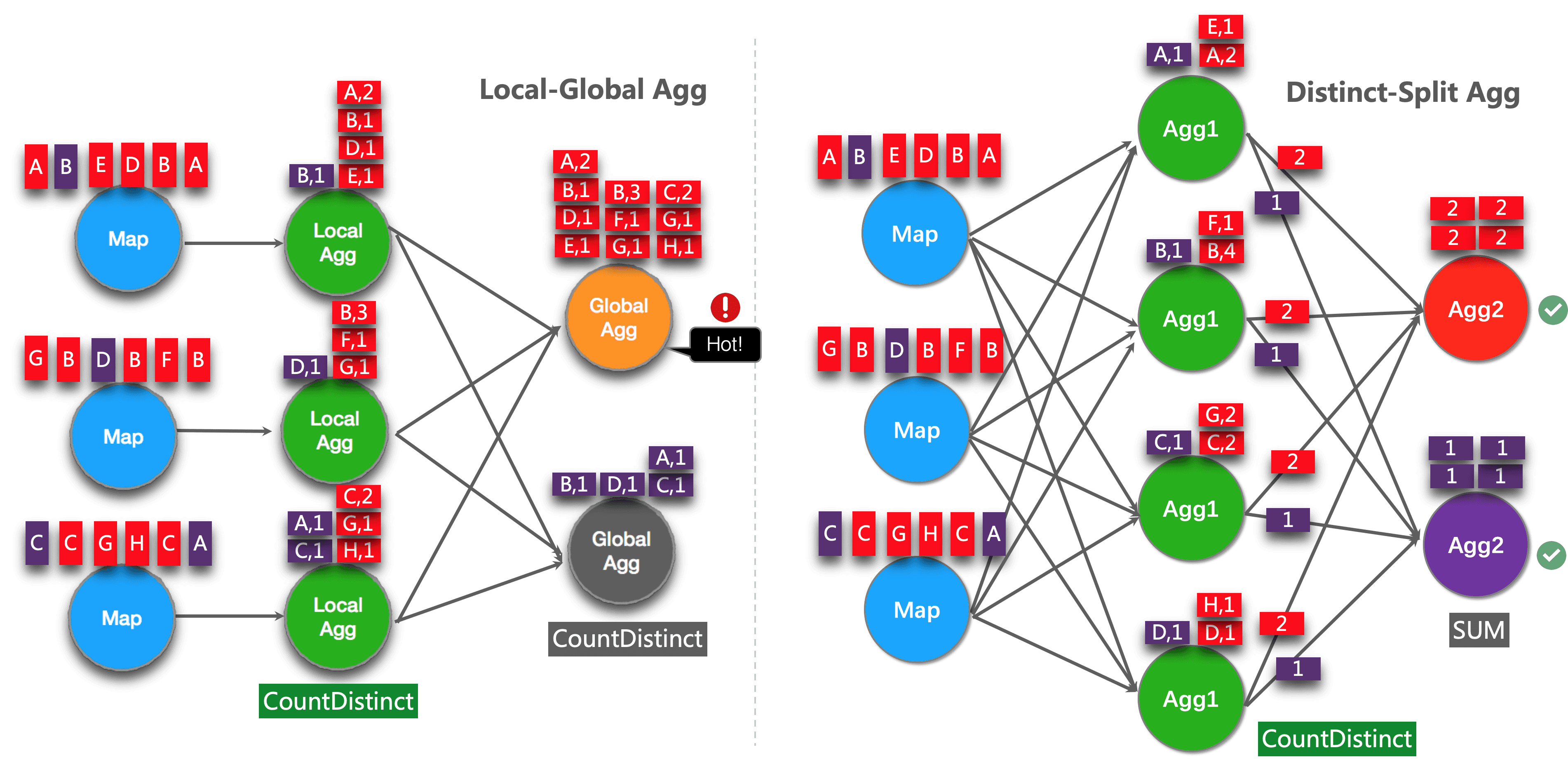
如果 distinct key （即 user\_id）的值分布稀疏，则 COUNT DISTINCT 不适合减少数据。即使启用了 local-global 优化也没有太大帮助。因为累加器仍然包含几乎所有原始记录，并且全局聚合将成为瓶颈（大多数繁重的累加器由一个任务处理，即同一天）。

这个优化的想法是将不同的聚合（例如 COUNT(DISTINCT col)）分为两个级别。第一次聚合由 group key 和额外的 bucket key 进行 shuffle。bucket key 是使用 HASH\_CODE(distinct\_key) % BUCKET\_NUM 计算的。BUCKET\_NUM 默认为1024，可以通过 table.optimizer.distinct-agg.split.bucket-num 选项进行配置。第二次聚合是由原始 group key 进行 shuffle，并使用 SUM 聚合来自不同 buckets 的 COUNT DISTINCT 值。由于相同的 distinct key 将仅在同一 bucket 中计算，因此转换是等效的。bucket key 充当附加 group key 的角色，以分担 group key 中热点的负担。bucket key 使 job 具有可伸缩性来解决不同聚合中的数据倾斜/热点。

拆分 distinct 聚合后，以上查询将被自动改写为以下查询：

|  |
| --- |
| SELECT day, SUM(cnt) FROM (  SELECT day, COUNT(DISTINCT user\_id) as cnt  FROM T  GROUP BY day, MOD(HASH\_CODE(user\_id), 1024) ) GROUP BY day |

下图显示了拆分 distinct 聚合如何提高性能（假设颜色表示 days，字母表示 user\_id）。



注意：上面是可以从这个优化中受益的最简单的示例。除此之外，Flink 还支持拆分更复杂的聚合查询，例如，多个具有不同 distinct key （例如 COUNT(DISTINCT a), SUM(DISTINCT b) ）的 distinct 聚合，可以与其他非 distinct 聚合（例如 SUM、MAX、MIN、COUNT ）一起使用。

|  |
| --- |
| **注意：**当前，拆分优化不支持包含用户定义的 AggregateFunction 聚合。 |

下面的例子显示了如何启用拆分 distinct 聚合优化。

|  |
| --- |
| // instantiate table environment TableEnvironment tEnv = ...  tEnv.getConfig() // access high-level configuration .getConfiguration() // set low-level key-value options .setString("table.optimizer.distinct-agg.split.enabled", "true"); // enable distinct agg split |

#### AGG WITH CASE WHEN改写为AGG WITH FILTER语法（提升大量COUNT DISTINCT场景性能）

在某些情况下，用户可能需要从不同维度计算 UV（独立访客）的数量，例如来自 Android 的 UV、iPhone 的 UV、Web 的 UV 和总 UV。很多人会选择 CASE WHEN，例如：

|  |
| --- |
| SELECT  day,  COUNT(DISTINCT user\_id) AS total\_uv,  COUNT(DISTINCT CASE WHEN flag IN ('android', 'iphone') THEN user\_id ELSE NULL END) AS app\_uv,  COUNT(DISTINCT CASE WHEN flag IN ('wap', 'other') THEN user\_id ELSE NULL END) AS web\_uv FROM T GROUP BY day |

但是，在这种情况下，建议使用 FILTER 语法而不是 CASE WHEN。因为 FILTER 更符合 SQL 标准，并且能获得更多的性能提升。FILTER 是用于聚合函数的修饰符，用于限制聚合中使用的值。将上面的示例替换为 FILTER 修饰符，如下所示：

|  |
| --- |
| SELECT  day,  COUNT(DISTINCT user\_id) AS total\_uv,  COUNT(DISTINCT user\_id) FILTER (WHERE flag IN ('android', 'iphone')) AS app\_uv,  COUNT(DISTINCT user\_id) FILTER (WHERE flag IN ('wap', 'other')) AS web\_uv FROM T GROUP BY day |

Flink SQL 优化器可以识别相同的 distinct key 上的不同过滤器参数。例如，在上面的示例中，三个 COUNT DISTINCT 都在 user\_id 一列上。Flink 可以只使用一个共享状态实例，而不是三个状态实例，以减少状态访问和状态大小。在某些工作负载下，可以获得显著的性能提升。

### TopN优化技巧

* TopN算法

当TopN的输入是非更新流（例如Source），TopN只有1种算法AppendRank。当TopN的输入是更新流时（例如经过了AGG或JOIN计算），TopN有3种算法，性能从高到低分别是：UpdateFastRank 、 UnaryUpdateRank和RetractRank。算法名字会显示在拓扑图的节点名字上。

* + UpdateFastRank ：最优算法。

使用该算法需要具备2个条件：

* + - 输入流有PK（Primary Key）信息，例如ORDER BY AVG。
    - 排序字段的更新是单调的，且单调方向与排序方向相反。例如，ORDER BY COUNT/COUNT\_DISTINCT/SUM（正数）DESC。

如果您要获取到优化Plan，则您需要在使用ORDER BY SUM DESC时，添加SUM为正数的过滤条件，确保total\_fee为正数。

|  |
| --- |
| insert  into print\_test SELECT  cate\_id,  seller\_id,  stat\_date,  pay\_ord\_amt --不输出rownum字段，能减小结果表的输出量。 FROM (  SELECT  \*,  ROW\_NUMBER () OVER (  PARTITION BY cate\_id,  stat\_date --注意要有时间字段，否则State过期会导致数据错乱。  ORDER  BY pay\_ord\_amt DESC  ) as rownum --根据上游sum结果排序。  FROM (  SELECT  cate\_id,  seller\_id,  stat\_date,  --重点。声明Sum的参数都是正数，所以Sum的结果是单调递增的，因此TopN能使用优化算法，只获取前100个数据。  sum (total\_fee) filter (  where  total\_fee >= 0  ) as pay\_ord\_amt  FROM  random\_test  WHERE  total\_fee >= 0  GROUP  BY cate\_name,  seller\_id,  stat\_date  ) a  WHERE  rownum <= 100  ); |

* + UnaryUpdateRank：性能仅次于UpdateFastRank的算法。使用该算法需要具备的条件是输入流中存在PK信息。
  + RetractRank：普通算法，性能最差，不建议在生产环境使用该算法。请检查输入流是否存在PK信息，如果存在，则可使用UnaryUpdateRank或UpdateFastRank算法进行优化。
* TopN优化方法
  + 无排名优化

TopN的输出结果不需要显示rownum值，仅需在最终前端显示时进行1次排序，极大地减少输入结果表的数据量。无排名优化方法详情请参见[Top-N](https://ci.apache.org/projects/flink/flink-docs-master/docs/dev/table/sql/queries/" \l "top-n" \t "https://help.aliyun.com/document_detail/_blank)。

* + 增加TopN的Cache大小

TopN为了提升性能有一个State Cache层，Cache层能提升对State的访问效率。TopN的Cache命中率的计算公式如下

|  |
| --- |
| cache\_hit = cache\_size\*parallelism/top\_n/partition\_key\_num |

例如，Top100配置缓存10000条，并发50，当您的PatitionBy的key维度较大时，例如10万级别时，Cache命中率只有10000\*50/100/100000=5%，命中率会很低，导致大量的请求都会击中State（磁盘），性能会大幅下降。因此当PartitionBy的Key维度特别大时，可以适当加大TopN的Cache Size，相对应的也建议适当加大TopN节点的Heap Memory。

|  |
| --- |
| blink.topn.cache.size: 200000 |

默认10000条，调整TopN cahce到200000，那么理论命中率能达到200000\*50/100/100000 = 100%。

* + PartitionBy的字段中要有时间类字段

例如每天的排名，要带上Day字段，否则TopN的最终结果会由于State TTL产生错乱。

### 高效去重方案

实时计算的源数据在部分场景中存在重复数据，去重成为了用户经常反馈的需求。实时计算有保留第一条（Deduplicate Keep FirstRow）和保留最后一条（Deduplicate Keep LastRow）2种去重方案。

* 语法

由于SQL上没有直接支持去重的语法，还要灵活的保留第一条或保留最后一条。因此我们使用了SQL的ROW\_NUMBER OVER WINDOW功能来实现去重语法。去重本质上是一种特殊的TopN。

|  |  |
| --- | --- |
| SELECT *\** FROM (  SELECT *\**,  ROW\_NUMBER() OVER ([PARTITION BY col1[, col2..]  ORDER BY timeAttributeCol [asc|desc]) AS rownum  FROM table\_name) WHERE rownum = 1 | |
| **参数** | **说明** |
| ROW\_NUMBER() | 计算行号的OVER窗口函数。行号从1开始计算。 |
| PARTITION BY col1[, col2..] | 可选。指定分区的列，即去重的KEYS |
| ORDER BY timeAttributeCol [asc|desc]) | 指定排序的列，必须是一个时间属性的字段（即Proctime或Rowtime）。可以指定顺序（Keep FirstRow）或者倒序 （Keep LastRow） |
| rownum | 仅支持rownum=1或rownum<=1 |

如上语法所示，去重需要两层Query：

* + 使用ROW\_NUMBER() 窗口函数来对数据根据时间属性列进行排序并标上排名。
    - 当排序字段是Proctime列时，Flink就会按照系统时间去重，其每次运行的结果是不确定的。
    - 当排序字段是Rowtime列时，Flink就会按照业务时间去重，其每次运行的结果是确定的。
  + 对排名进行过滤，只取第一条，达到了去重的目的。排序方向可以是按照时间列的顺序，也可以是倒序：
    - Deduplicate Keep FirstRow：顺序并取第一条行数据。
    - Deduplicate Keep LastRow：倒序并取第一条行数据。
* Deduplicate Keep FirstRow

保留首行的去重策略：保留KEY下第一条出现的数据，之后出现该KEY下的数据会被丢弃掉。因为STATE中只存储了KEY数据，所以性能较优，示例如下

|  |
| --- |
| SELECT *\** FROM (  SELECT *\**,  ROW\_NUMBER() OVER (PARTITION BY b ORDER BY proctime) as rowNum  FROM T ) WHERE rowNum = 1 |

以上示例是将T表按照b字段进行去重，并按照系统时间保留第一条数据。proctime在这里是源表T中的一个具有Processing Time属性的字段。如果您按照系统时间去重，也可以将proctime字段简化proctime()函数调用，可以省略proctime字段的声明。

* Deduplicate Keep LastRow

保留末行的去重策略：保留KEY下最后一条出现的数据。保留末行的去重策略性能略优于LAST\_VALUE函数，示例如下

|  |
| --- |
| SELECT *\** FROM (  SELECT *\**,  ROW\_NUMBER() OVER (PARTITION BY b, d ORDER BY rowtime DESC) as rowNum  FROM T ) WHERE rowNum = 1 |

以上示例是将T表按照b和d字段进行去重，并按照业务时间保留最后一条数据。rowtime在这里是源表T中的一个具有Event Time属性的字段。

### 高效的内置函数

在使用内置函数时，您需要注意以下几点：

* 使用内置函数替换自定义函数

实时计算的内置函数在持续的优化当中，请尽量使用内置函数替换自定义函数。实时计算对内置函数主要进行了如下优化：

* + 优化数据序列化和反序列化的耗时。
  + 新增直接对字节单位进行操作的功能。
* KEY VALUE函数使用单字符的分隔符

KEY VALUE的签名：KEYVALUE(content, keyValueSplit, keySplit, keyName)，当keyValueSplit和KeySplit是单字符，例如，冒号（:）、逗号（,）时，系统会使用优化算法，在二进制数据上直接寻找所需的keyName值，而不会将整个content进行切分，性能约提升30%

* LIKE操作注意事项
  + 如果需要进行StartWith操作，使用LIKE 'xxx%'。
  + 如果需要进行EndWith操作，使用LIKE '%xxx'。
  + 如果需要进行Contains操作，使用LIKE '%xxx%'。
  + 如果需要进行Equals操作，使用LIKE 'xxx'，等价于str = 'xxx'。
  + 如果需要匹配下划线（\_），请注意要完成转义LIKE '%seller/\_id%' ESCAPE '/'。下划线（\_）在SQL中属于单字符通配符，能匹配任何字符。如果声明为 LIKE '%seller\_id%'，则不单会匹配seller\_id，还会匹配seller#id、sellerxid或seller1id等，导致结果错误。
* 慎用正则函数（REGEXP）

正则表达式是非常耗时的操作，对比加减乘除通常有百倍的性能开销，而且正则表达式在某些极端情况下可能会进入无限循环，导致作业阻塞，具体情况请参见[Regex execution is too slow](https://stackoverflow.com/questions/4500507/regex-execution-is-too-slow-in-java" \t "https://help.aliyun.com/document_detail/_blank)，因此建议使用LIKE。正则函数包括：

* + [REGEXP](https://help.aliyun.com/document_detail/207987.htm" \l "concept-62550-zh" \o "本文为您介绍如何使用REGEXP函数，对指定的字符串执行一个正则表达式搜索，并返回一个Boolean值表示是否找到指定的匹配模式。)
  + [REGEXP\_REPLACE](https://help.aliyun.com/document_detail/207988.htm" \l "concept-62545-zh" \o "本文为您介绍如何使用REGEXP_REPLACE函数，用目标字符串替换指定字符串，并返回新的字符串。)

## SQL（参考）

本章节描述了 Flink 所支持的 SQL 语言，包括数据定义语言（Data Definition Language，DDL）、数据操纵语言（Data Manipulation Language，DML）以及查询语言。Flink 对 SQL 的支持基于实现了 SQL 标准的 [Apache Calcite](https://calcite.apache.org/)。

本页面列出了目前 Flink SQL 所支持的所有语句：

* [SELECT (Queries)](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/)
* [CREATE TABLE, DATABASE, VIEW, FUNCTION](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/create/)
* [DROP TABLE, DATABASE, VIEW, FUNCTION](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/drop/)
* [ALTER TABLE, DATABASE, FUNCTION](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/alter/)
* [INSERT](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/insert/)
* [SQL HINTS](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/hints/)
* [DESCRIBE](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/describe/)
* [EXPLAIN](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/explain/)
* [USE](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/use/)
* [SHOW](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/show/)
* [LOAD](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/load/)
* [UNLOAD](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/unload/)

### 数据类型

* 请参考专门描述该主题的页面[数据类型](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/types/)。
* 通用类型与（嵌套的）复合类型 （如：POJO、tuples、rows、Scala case 类) 都可以作为行的字段。
* 复合类型的字段任意的嵌套可被 [值访问函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/systemfunctions/" \l "value-access-functions)访问。
* 通用类型将会被视为一个黑箱，且可以被 [用户自定义函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/udfs/)传递或引用。
* 对于 DDL 语句而言，我们支持所有在[数据类型](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/types/)页面中定义的数据类型。

|  |
| --- |
| **注意**：SQL查询不支持部分数据类型（cast 表达式或字符常量值）。如：STRING, BYTES, RAW, TIME(p) WITHOUT TIME ZONE, TIME(p) WITH LOCAL TIME ZONE, TIMESTAMP(p) WITHOUT TIME ZONE, TIMESTAMP(p) WITH LOCAL TIME ZONE, ARRAY, MULTISET, ROW。 |

### 基本概念

#### Source 表

与所有 SQL 引擎一样，Flink 查询操作是在表上进行。与传统数据库不同，Flink 不在本地管理静态数据；相反，它的查询在外部表上连续运行。

Flink 数据处理流水线开始于 source 表。source 表产生在查询执行期间可以被操作的行；它们是查询时 FROM 子句中引用的表。这些表可能是 Kafka 的 topics，数据库，文件系统，或者任何其它 Flink 知道如何消费的系统。

可以通过 SQL 客户端或使用环境配置文件来定义表。SQL 客户端支持类似于传统 SQL 的[SQL DDL 命令](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/overview/)。标准 SQL DDL 用于[创建](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/create/)，[修改](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/alter/)，[删除](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/drop/)表。

Flink 支持不同的[连接器](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/overview/)和[格式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/overview/)相结合以定义表。下面是一个示例，定义一个以[CSV 文件](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/formats/csv/)作为存储格式的 source 表，其中 emp\_id，name，dept\_id 作为 CREATE 表语句中的列。

|  |
| --- |
| CREATE TABLE employee\_information (  emp\_id INT,  name VARCHAR,  dept\_id INT ) WITH (   'connector' = 'filesystem',  'path' = '/path/to/something.csv',  'format' = 'csv' ); |

可以从该表中定义一个连续查询，当新行可用时读取并立即输出它们的结果。 例如，我们可以过滤出只在部门 1 中工作的员工。

|  |
| --- |
| SELECT *\** from employee\_information WHERE DeptId = 1; |

#### 连续查询

虽然最初设计时没有考虑流语义，但 SQL 是用于构建连续数据流水线的强大工具。Flink SQL 与传统数据库查询的不同之处在于，Flink SQL 持续消费到达的行并对其结果进行更新。

一个[连续查询](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/dynamic_tables/" \l "continuous-queries)永远不会终止，并会产生一个动态表作为结果。[动态表](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/dynamic_tables/" \l "continuous-queries)是 Flink 中 Table API 和 SQL 对流数据支持的核心概念。

连续流上的聚合需要在查询执行期间不断地存储聚合的结果。例如，假设你需要从传入的数据流中计算每个部门的员工人数。查询需要维护每个部门最新的计算总数，以便在处理新行时及时输出结果。

|  |
| --- |
| SELECT   dept\_id,  COUNT(\*) as emp\_count  FROM employee\_information  GROUP BY dep\_id; |

这样的查询被认为是 有状态的。Flink 的高级容错机制将维持内部状态和一致性，因此即使遇到硬件故障，查询也始终返回正确结果。

#### Sink 表

当运行此查询时，SQL 客户端实时但是以只读方式提供输出。存储结果，作为报表或仪表板的数据来源，需要写到另一个表。这可以使用 INSERT INTO 语句来实现。本节中引用的表称为 sink 表。INSERT INTO 语句将作为一个独立查询被提交到 Flink 集群中。

|  |
| --- |
| INSERT INTO department\_counts SELECT   dept\_id,  COUNT(\*) as emp\_count  FROM employee\_information; |

提交后，它将运行并将结果直接存储到 sink 表中，而不是将结果加载到系统内存中。

### [SELECT (Queries)](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/)

SELECT 语句和 VALUES 语句需要使用 TableEnvironment 的 sqlQuery() 方法加以指定。这个方法会以 Table 的形式返回 SELECT （或 VALUE）的查询结果。

Table 可以被用于 随后的SQL 与 Table API 查询 、 转换为 DataSet 或 DataStream 或 输出到 TableSink 。SQL 与 Table API 的查询可以进行无缝融合、整体优化并翻译为单一的程序。

为了可以在 SQL 查询中访问到表，你需要先 在 TableEnvironment 中注册表 。表可以通过 TableSource、 Table、CREATE TABLE 语句、 DataStream 或 DataSet 注册。 用户也可以通过 向 TableEnvironment 中注册 catalog 的方式指定数据源的位置。

为方便起见 Table.toString() 将会在其 TableEnvironment 中自动使用一个唯一的名字注册表并返回表名。 因此， Table 对象可以如下文所示样例，直接内联到 SQL 语句中。

|  |
| --- |
| **注意**：查询若包括了不支持的 SQL 特性，将会抛出 TableException。批处理和流处理所支持的 SQL 特性将会在下述章节中列出。 |

#### 指定查询

以下示例显示如何在已注册和内联表上指定 SQL 查询。

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment(); StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);  // 从外部数据源读取 DataStream  DataStream<Tuple3<Long, String, Integer>> ds = env.addSource(...);  // 使用 SQL 查询内联的（未注册的）表 Table table = tableEnv.fromDataStream(ds, $("user"), $("product"), $("amount")); Table result = tableEnv.sqlQuery( "SELECT SUM(amount) FROM " + table + " WHERE product LIKE '%Rubber%'");  // SQL 查询一个已经注册的表 // 根据视图 "Orders" 创建一个 DataStream tableEnv.createTemporaryView("Orders", ds, $("user"), $("product"), $("amount")); // 在表上执行 SQL 查询并得到以新表返回的结果 Table result2 = tableEnv.sqlQuery( "SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'");  // 创建并注册一个 TableSink final Schema schema = new Schema() .field("product", DataTypes.STRING()) .field("amount", DataTypes.INT());  tableEnv.connect(new FileSystem().path("/path/to/file")) .withFormat(...) .withSchema(schema) .createTemporaryTable("RubberOrders");  // 在表上执行插入语句并把结果发出到 TableSink tableEnv.executeSql( "INSERT INTO RubberOrders SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'"); |

#### 执行查询

SELECT 语句或者 VALUES 语句可以通过 TableEnvironment.executeSql() 方法来执行，将选择的结果收集到本地。该方法返回 TableResult 对象用于包装查询的结果。和 SELECT 语句很像，一个 Table 对象可以通过 Table.execute() 方法执行从而将 Table 的内容收集到本地客户端。 TableResult.collect() 方法返回一个可以关闭的行迭代器。除非所有的数据都被收集到本地，否则一个查询作业永远不会结束。所以我们应该通过 CloseableIterator#close() 方法主动地关闭作业以防止资源泄露。 我们还可以通过 TableResult.print() 方法将查询结果打印到本地控制台。TableResult 中的结果数据只能被访问一次，因此一个 TableResult 实例中，collect() 方法和 print() 方法不能被同时使用。

TableResult.collect() 与 TableResult.print() 的行为在不同的 checkpointing 模式下略有不同（流作业开启 checkpointing 的方法可参考 checkpointing 配置）。

* 对于批作业或没有配置任何 checkpointing 的流作业，TableResult.collect() 与 TableResult.print() 既不保证精确一次的数据交付、也不保证至少一次的数据交付。查询结果在产生后可被客户端即刻访问，但作业失败并重启时将会报错。
* 对于配置了精准一次 checkpointing 的流作业，TableResult.collect() 与 TableResult.print() 保证端到端精确一次的数据交付。一条结果数据只有在其对应的 checkpointing 完成后才能在客户端被访问。
* 对于配置了至少一次 checkpointing 的流作业，TableResult.collect() 与 TableResult.print() 保证端到端至少一次的数据交付。查询结果在产生后可被客户端即刻访问，但同一条结果可能被多次传递给客户端。

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment(); StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, settings);  tableEnv.executeSql("CREATE TABLE Orders (`user` BIGINT, product STRING, amount INT) WITH (...)");  // execute SELECT statement TableResult tableResult1 = tableEnv.executeSql("SELECT \* FROM Orders"); // use try-with-resources statement to make sure the iterator will be closed automatically try (CloseableIterator<Row> it = tableResult1.collect()) { while(it.hasNext()) { Row row = it.next(); // handle row } }  // execute Table TableResult tableResult2 = tableEnv.sqlQuery("SELECT \* FROM Orders").execute(); tableResult2.print(); |

#### 语法

Flink 通过支持标准 ANSI SQL的 [Apache Calcite](https://calcite.apache.org/docs/reference.html)解析 SQL。

以下 BNF-语法 描述了批处理和流处理查询中所支持的 SQL 特性的超集。其中 [操作符](https://ci.apache.org/projects/flink/flink-docs-release-1.12/zh/dev/table/sql/queries.html" \l "%E6%93%8D%E4%BD%9C%E7%AC%A6) 章节展示了所支持的特性的样例，并指明了哪些特性仅适用于批处理或流处理。

|  |
| --- |
| query:  values  | {  select  | selectWithoutFrom  | query UNION [ ALL ] query  | query EXCEPT query  | query INTERSECT query  }  [ ORDER BY orderItem [, orderItem ]\* ]  [ LIMIT { count | ALL } ]  [ OFFSET start { ROW | ROWS } ]  [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY]  orderItem:  expression [ ASC | DESC ]  select:  SELECT [ ALL | DISTINCT ]  { \* | projectItem [, projectItem ]\* }  FROM tableExpression  [ WHERE booleanExpression ]  [ GROUP BY { groupItem [, groupItem ]\* } ]  [ HAVING booleanExpression ]  [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]\* ]  selectWithoutFrom:  SELECT [ ALL | DISTINCT ]  { \* | projectItem [, projectItem ]\* }  projectItem:  expression [ [ AS ] columnAlias ]  | tableAlias . \*  tableExpression:  tableReference [, tableReference ]\*  | tableExpression [ NATURAL ] [ LEFT | RIGHT | FULL ] JOIN tableExpression [ joinCondition ]  joinCondition:  ON booleanExpression  | USING '(' column [, column ]\* ')'  tableReference:  tablePrimary  [ matchRecognize ]  [ [ AS ] alias [ '(' columnAlias [, columnAlias ]\* ')' ] ]  tablePrimary:  [ TABLE ] tablePath [ dynamicTableOptions ] [systemTimePeriod] [[AS] correlationName]  | LATERAL TABLE '(' functionName '(' expression [, expression ]\* ')' ')'  | UNNEST '(' expression ')'  tablePath:  [ [ catalogName . ] schemaName . ] tableName   systemTimePeriod:  FOR SYSTEM\_TIME AS OF dateTimeExpression  dynamicTableOptions:  /\*+ OPTIONS(key=val [, key=val]\*) \*/  key:  stringLiteral  val:  stringLiteral  values:  VALUES expression [, expression ]\*  groupItem:  expression  | '(' ')'  | '(' expression [, expression ]\* ')'  | CUBE '(' expression [, expression ]\* ')'  | ROLLUP '(' expression [, expression ]\* ')'  | GROUPING SETS '(' groupItem [, groupItem ]\* ')'  windowRef:  windowName  | windowSpec  windowSpec:  [ windowName ]  '('  [ ORDER BY orderItem [, orderItem ]\* ]  [ PARTITION BY expression [, expression ]\* ]  [  RANGE numericOrIntervalExpression {PRECEDING}  | ROWS numericExpression {PRECEDING}  ]  ')'  matchRecognize:  MATCH\_RECOGNIZE '('  [ PARTITION BY expression [, expression ]\* ]  [ ORDER BY orderItem [, orderItem ]\* ]  [ MEASURES measureColumn [, measureColumn ]\* ]  [ ONE ROW PER MATCH ]  [ AFTER MATCH  ( SKIP TO NEXT ROW  | SKIP PAST LAST ROW  | SKIP TO FIRST variable  | SKIP TO LAST variable  | SKIP TO variable )  ]  PATTERN '(' pattern ')'  [ WITHIN intervalLiteral ]  DEFINE variable AS condition [, variable AS condition ]\*  ')'  measureColumn:  expression AS alias  pattern:  patternTerm [ '|' patternTerm ]\*  patternTerm:  patternFactor [ patternFactor ]\*  patternFactor:  variable [ patternQuantifier ]  patternQuantifier:  '\*'  | '\*?'  | '+'  | '+?'  | '?'  | '??'  | '{' { [ minRepeat ], [ maxRepeat ] } '}' ['?']  | '{' repeat '}' |

Flink SQL 对于标识符（表、属性、函数名）有类似于 Java 的词法约定:

* 不管是否引用标识符，都保留标识符的大小写。
* 且标识符需区分大小写。
* 与 Java 不一样的地方在于，通过反引号，可以允许标识符带有非字母的字符（如："SELECT a AS `my field` FROM t"）。

字符串文本常量需要被单引号包起来（如 SELECT 'Hello World' ）。两个单引号表示转移（如 SELECT 'It''s me.'）。字符串文本常量支持 Unicode 字符，如需明确使用 Unicode 编码，请使用以下语法：

* 使用反斜杠（\）作为转义字符（默认）：SELECT U&'\263A'
* 使用自定义的转义字符： SELECT U&'#263A' UESCAPE '#'

#### 操作符

* WITH 语句
* SELECT & WHERE
* SELECT DISTINCT
* Windowing TVF
* Window Aggregation
* Group Aggregation
* Over Aggregation
* Joins
* Set Operations
* ORDER BY clause
* LIMIT clause
* Top-N
* Window Top-N
* Deduplication
* Pattern Recognition

##### WITH 语句

**应用场景：批或者流**

WITH提供了一种编写用于更大查询的辅助语句的方法。这些语句通常被称为公共表表达式 (CTE)，可以被认为是定义仅存在于一个查询的临时视图。

WITH语句的语法是：

|  |
| --- |
| WITH <with\_item\_definition> [ , ... ] SELECT ... FROM ...;  <with\_item\_defintion>:  with\_item\_name (column\_name[, ...n]) AS ( <select\_query> ) |

以下示例定义了一个公用表表达式orders\_with\_total并在GROUP BY查询中使用它。

|  |
| --- |
| WITH orders\_with\_total AS (  SELECT order\_id, price + tax AS total  FROM Orders ) SELECT order\_id, *SUM*(total) FROM orders\_with\_total GROUP BY order\_id; |

##### SELECT & WHERE子句

**应用场景：批或者流**

该SELECT语句的一般语法是：

|  |
| --- |
| SELECT select\_list FROM table\_expression [ WHERE boolean\_expression ] |

所述table\_expression指的是数据的任何来源。它可以是现有表、视图或VALUES子句，多个现有表的连接结果或子查询。假设该表在目录中可用，以下内容将从Orders.

|  |
| --- |
| SELECT *\** FROM Orders |

该select\_list规范\*是指查询将解决所有列。但是，\*不鼓励在生产中使用 ，因为它会使查询对目录更改的稳健性降低。相反， aselect\_list可以指定可用列的子集或使用所述列进行计算。例如，如果Orders有名为order\_id, 的列price，tax您可以编写以下查询：

|  |
| --- |
| SELECT order\_id, price + tax FROM Orders |

查询也可以使用VALUES子句从内联数据中使用。每个元组对应一行，并且可以提供别名来为每一列分配名称。

|  |
| --- |
| SELECT order\_id, price FROM (VALUES (1, 2.0), (2, 3.1)) AS t (order\_id, price) |

可以根据WHERE子句过滤行。

|  |
| --- |
| SELECT price + tax FROM Orders WHERE id = 10 |

此外，可以在单行的列上调用内置和[用户定义的标量函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/udfs/)。用户定义的函数必须在使用前注册到目录中。

|  |
| --- |
| SELECT PRETTY\_PRINT(order\_id) FROM Orders |

##### SELECT DISTINCT

**应用场景：批或者流**

如果SELECT DISTINCT指定，则从结果集中删除所有重复行（每组重复项中保留一行）。

|  |
| --- |
| SELECT DISTINCT id FROM Orders |

对于流式查询，计算查询结果所需的状态可能会无限增长。状态大小取决于不同行的数量。您可以提供具有适当状态生存时间 (TTL) 的查询配置，以防止状态过大。请注意，这可能会影响查询结果的正确性。参见[查询配置](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "table-exec-state-ttl)的详细信息

##### Windowing table-valued functions (Windowing TVFs)

**应用场景：流**

Windows 是处理无限流的核心。Windows 将流拆分为有限大小的“桶”，我们可以对其进行计算。本文档重点介绍如何在 Flink SQL 中执行窗口化以及程序员如何从其提供的功能中获得最大收益。

Apache Flink 提供了几个窗口表值函数 (TVF) 来将表的元素划分为窗口，包括：

* 翻滚窗户
* 滑动窗口
* 累积窗口
* 会话窗口（**即将支持**）

请注意，每个元素在逻辑上可以属于多个窗口，具体取决于您使用的窗口表值函数。例如，HOP 窗口创建重叠窗口，其中单个元素可以分配给多个窗口。

Windowing TVF 是 Flink 定义的多态表函数（简称 PTF）。PTF 是 SQL 2016 标准的一部分，是一种特殊的表函数，但可以将表作为参数。PTF 是改变表格形状的强大功能。由于 PTF 在语义上类似于表，因此它们的调用发生在语句的FROM子句中SELECT。

窗口化 TVF 替代了传统的[分组窗口函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/" \l "group-window-aggregation-deprecated)。窗口化 TVF 更符合 SQL 标准并且更强大以支持复杂的基于窗口的计算，例如 Window TopN、Window Join。但是，[分组窗口函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/" \l "group-window-aggregation)只能支持窗口聚合。

查看更多如何应用基于加窗 TVF 的进一步计算：

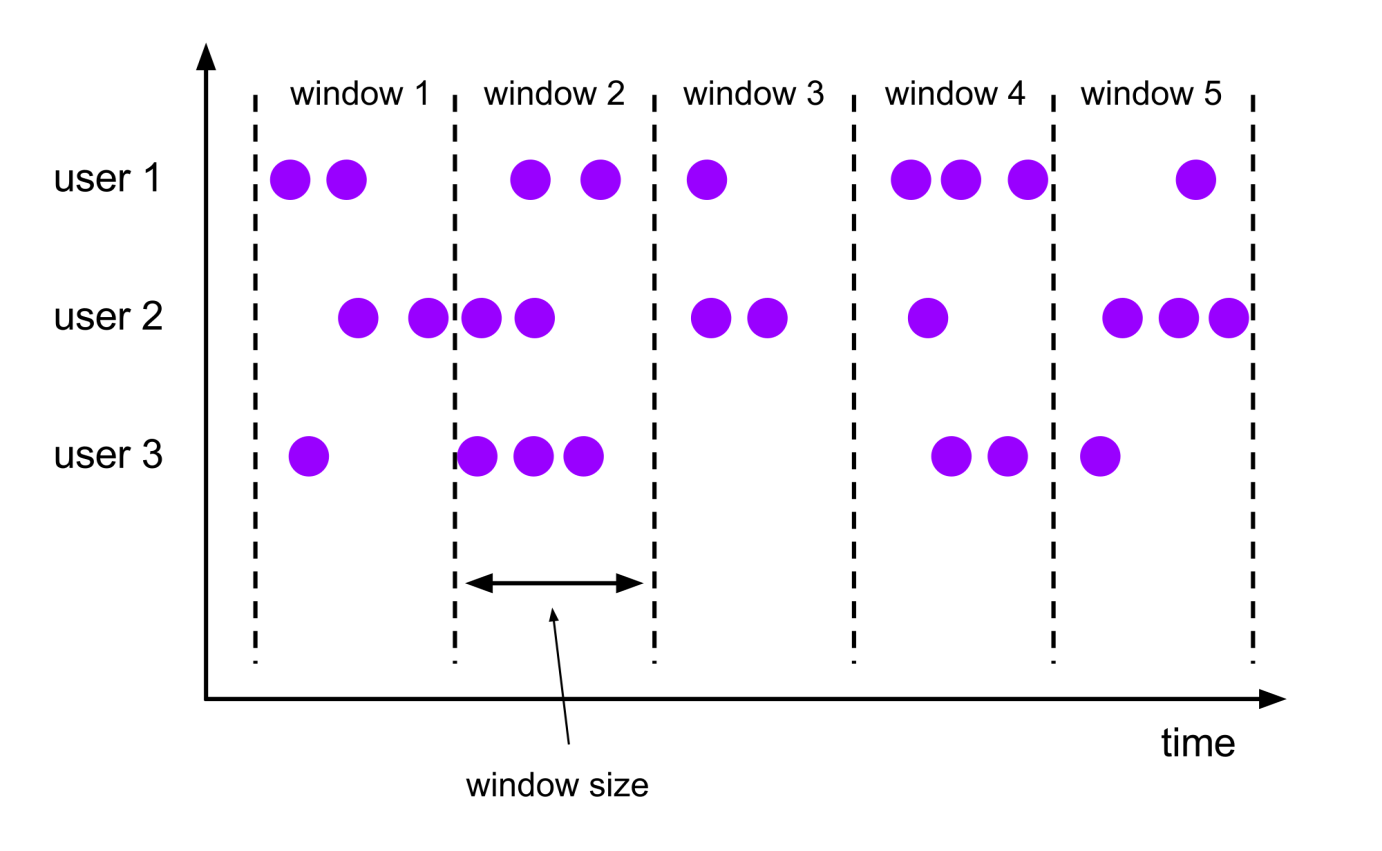
* Window Aggregation
* Window TopN
* Window Join (**即将支持**)

###### 窗口函数

Apache Flink 提供了 3 个内置的窗口 TVF**：TUMBLE、HOP和CUMULATE**. 窗口TVF的返回值是一个新的关系，包括原始关系的所有列，以及额外的3列，分别名为**“window\_start”、“window\_end”、“window\_time”，**表示分配的窗口。“window\_time”字段是窗口化TVF后窗口的[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)，可用于后续基于时间的操作。

翻滚窗口

该TUMBLE函数将每个元素分配给指定窗口大小的窗口。翻转窗口具有固定大小并且不重叠。例如，假设您指定一个大小为 5 分钟的滚动窗口。在这种情况下，Flink 将评估当前窗口，每五分钟启动一个新窗口，如下图所示。



该TUMBLE函数根据[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)列为关系的每一行分配一个窗口。的返回值TUMBLE是一个新的关系，包括原始关系的所有列以及额外的 3 列名为“window\_start”、“window\_end”、“window\_time”以指示分配的窗口。原始时间属性“timecol”将是窗口 TVF 之后的常规时间戳列。

TUMBLE 函数需要三个必需的参数：

|  |
| --- |
| TUMBLE(TABLE data, DESCRIPTOR(timecol), size) |

* data: 是一个表参数，可以与时间属性列有任何关系。
* timecol: 是一个列描述符，指示数据的哪些[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)列应该映射到滚动窗口。
* size: 是指定滚动窗口宽度的持续时间。

这是Bid表上的示例调用：

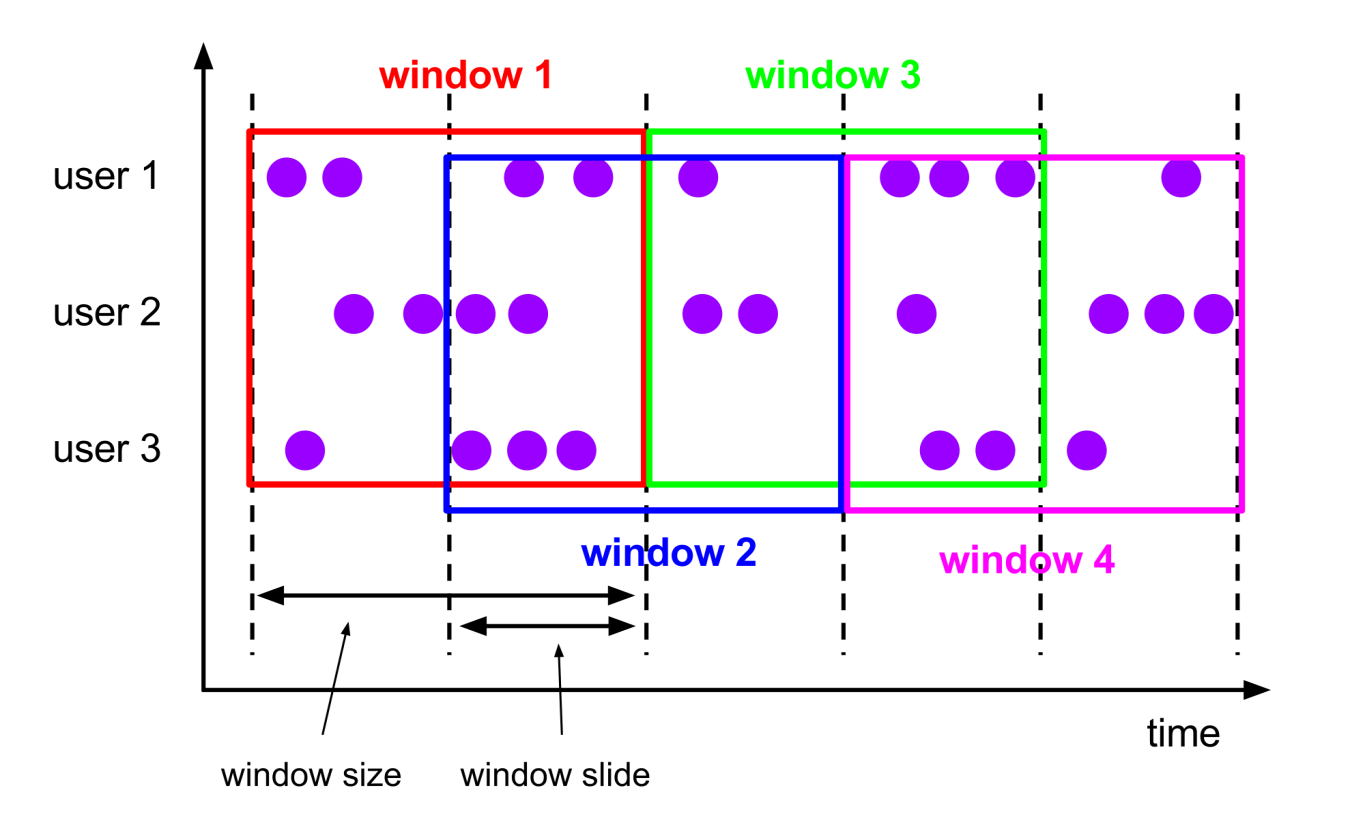
|  |
| --- |
| -- tables must have time attribute, e.g. `bidtime` in this table Flink SQL> desc Bid; +-------------+------------------------+------+-----+--------+---------------------------------+ | name | type | null | key | extras | watermark | +-------------+------------------------+------+-----+--------+---------------------------------+ | bidtime | TIMESTAMP(3) \*ROWTIME\* | true | | | `bidtime` - INTERVAL '1' SECOND | | price | DECIMAL(10, 2) | true | | | | | item | STRING | true | | | | +-------------+------------------------+------+-----+--------+---------------------------------+  Flink SQL> SELECT \* FROM Bid; +------------------+-------+------+ | bidtime | price | item | +------------------+-------+------+ | 2020-04-15 08:05 | 4.00 | C | | 2020-04-15 08:07 | 2.00 | A | | 2020-04-15 08:09 | 5.00 | D | | 2020-04-15 08:11 | 3.00 | B | | 2020-04-15 08:13 | 1.00 | E | | 2020-04-15 08:17 | 6.00 | F | +------------------+-------+------+  -- NOTE: Currently Flink doesn't support evaluating individual window table-valued function, -- window table-valued function should be used with aggregate operation, -- this example is just used for explaining the syntax and the data produced by table-valued function. Flink SQL> SELECT \* FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES)); -- or with the named params -- note: the DATA param must be the first Flink SQL> SELECT \* FROM TABLE(  TUMBLE(  DATA => TABLE Bid,  TIMECOL => DESCRIPTOR(bidtime),  SIZE => INTERVAL '10' MINUTES)); +------------------+-------+------+------------------+------------------+-------------------------+ | bidtime | price | item | window\_start | window\_end | window\_time | +------------------+-------+------+------------------+------------------+-------------------------+ | 2020-04-15 08:05 | 4.00 | C | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:07 | 2.00 | A | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:09 | 5.00 | D | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:11 | 3.00 | B | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | | 2020-04-15 08:13 | 1.00 | E | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | | 2020-04-15 08:17 | 6.00 | F | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | +------------------+-------+------+------------------+------------------+-------------------------+  -- apply aggregation on the tumbling windowed table Flink SQL> SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; +------------------+------------------+-------+ | window\_start | window\_end | price | +------------------+------------------+-------+ | 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 | +------------------+------------------+-------+ |

**注意：**为了更好地理解窗口化的行为，我们简化了时间戳值的显示以不显示尾随零，例如，如果类型为，则2020-04-15 08:05应该像2020-04-15 08:05:00.000在 Flink SQL Client 中那样显示TIMESTAMP(3)。

滑动窗口

该HOP函数将元素分配给固定长度的窗口。与TUMBLE窗口函数一样，窗口的大小由窗口大小参数配置。附加的窗口滑动参数控制跳跃窗口的启动频率。因此，如果幻灯片小于窗口大小，则跳跃窗口可能会重叠。在这种情况下，元素被分配给多个窗口。跳跃窗口也被称为“滑动窗口”。

例如，您可以有大小为 10 分钟的窗口，可滑动 5 分钟。这样，您每 5 分钟就会获得一个窗口，其中包含过去 10 分钟内到达的事件，如下图所示。



该HOP函数分配覆盖大小区间内的行的窗口，并根据[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)列移动每张幻灯片。的返回值HOP是一个新的关系，包括原始关系的所有列以及额外的 3 列名为“window\_start”、“window\_end”、“window\_time”以指示分配的窗口。原来的时间属性“timecol”在加窗TVF后会是一个常规的时间戳列。

HOP 需要三个必需的参数。

|  |
| --- |
| HOP(TABLE data, DESCRIPTOR(timecol), slide, size [, offset ]) |

* data: 是一个表参数，可以与时间属性列有任何关系。
* timecol: 是一个列描述符，指示数据的哪些[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)列应该映射到跳跃窗口。
* slide: 是指定顺序跳跃窗口开始之间的持续时间的持续时间
* size: 是指定跳跃窗口宽度的持续时间。

这是Bid表上的示例调用：

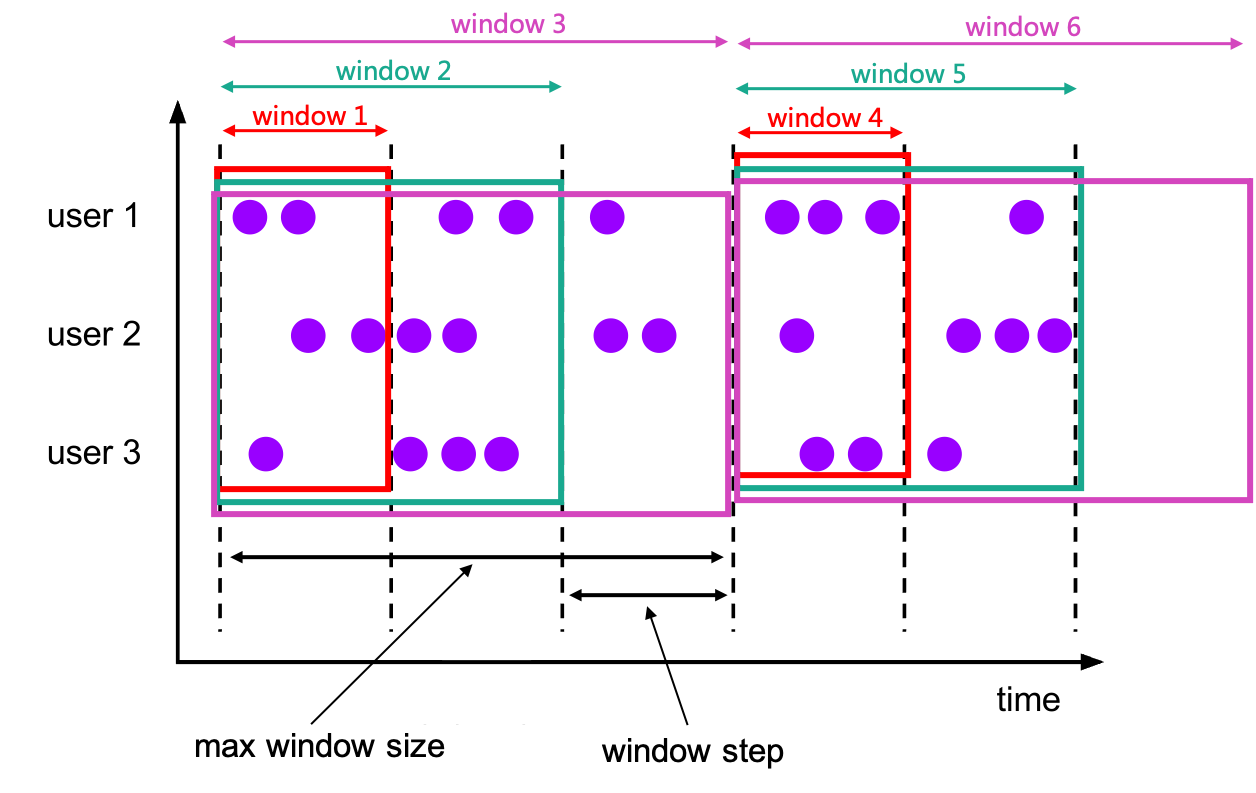
|  |
| --- |
| -- NOTE: Currently Flink doesn't support evaluating individual window table-valued function, -- window table-valued function should be used with aggregate operation, -- this example is just used for explaining the syntax and the data produced by table-valued function. > SELECT \* FROM TABLE(  HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES)); -- or with the named params -- note: the DATA param must be the first > SELECT \* FROM TABLE(  HOP(  DATA => TABLE Bid,  TIMECOL => DESCRIPTOR(bidtime),  SLIDE => INTERVAL '5' MINUTES,  SIZE => INTERVAL '10' MINUTES)); +------------------+-------+------+------------------+------------------+-------------------------+ | bidtime | price | item | window\_start | window\_end | window\_time | +------------------+-------+------+------------------+------------------+-------------------------+ | 2020-04-15 08:05 | 4.00 | C | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:05 | 4.00 | C | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 | | 2020-04-15 08:07 | 2.00 | A | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:07 | 2.00 | A | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 | | 2020-04-15 08:09 | 5.00 | D | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:09 | 5.00 | D | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 | | 2020-04-15 08:11 | 3.00 | B | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 | | 2020-04-15 08:11 | 3.00 | B | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | | 2020-04-15 08:13 | 1.00 | E | 2020-04-15 08:05 | 2020-04-15 08:15 | 2020-04-15 08:14:59.999 | | 2020-04-15 08:13 | 1.00 | E | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | | 2020-04-15 08:17 | 6.00 | F | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | | 2020-04-15 08:17 | 6.00 | F | 2020-04-15 08:15 | 2020-04-15 08:25 | 2020-04-15 08:24:59.999 | +------------------+-------+------+------------------+------------------+-------------------------+  -- apply aggregation on the hopping windowed table > SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; +------------------+------------------+-------+ | window\_start | window\_end | price | +------------------+------------------+-------+ | 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 | | 2020-04-15 08:05 | 2020-04-15 08:15 | 15.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 | | 2020-04-15 08:15 | 2020-04-15 08:25 | 6.00 | +------------------+------------------+-------+ |

累积窗口

累积窗口在某些情况下非常有用，例如在固定窗口间隔内提前触发的滚动窗口。例如，每日仪表板从 00:00 到每分钟绘制累积 UV，10:00 处的 UV 表示从 00:00 到 10:00 的 UV 总数。这可以通过 CUMULATE 窗口轻松有效地实现。

该CUMULATE函数将元素分配给覆盖初始步长间隔内的行的窗口，并每一步扩展到一个步长（保持窗口开始固定），直到最大窗口大小。您可以将CUMULATE函数视为首先应用TUMBLE最大窗口大小的窗口，然后将每个滚动窗口拆分为具有相同窗口开始和窗口结束步长差异的多个窗口。所以累积窗口确实重叠并且没有固定大小。

例如，您可以有一个 1 小时步长和 1 天最大大小的累积窗口，并且您将获得每天的窗口：[00:00, 01:00), [00:00, 02:00), [00:00, 03:00), ..., [00:00, 24:00)。



这些CUMULATE函数根据[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)列分配窗口。的返回值CUMULATE是一个新的关系，包括原始关系的所有列以及额外的 3 列名为“window\_start”、“window\_end”、“window\_time”以指示分配的窗口。原始时间属性“timecol”将是窗口 TVF 之后的常规时间戳列。

CUMULATE 需要三个必需的参数。

|  |
| --- |
| CUMULATE(TABLE data, DESCRIPTOR(timecol), step, size) |

* data: 是一个表参数，可以与时间属性列有任何关系。
* timecol: 是一个列描述符，指示数据的哪些[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)列应该映射到滚动窗口。
* step: 是指定顺序累积窗口结束之间增加的窗口大小的持续时间。
* size: 是指定累积窗口最大宽度的持续时间。size 必须是 step 的整数倍。

这是 Bid 表上的示例调用：

|  |
| --- |
| -- NOTE: Currently Flink doesn't support evaluating individual window table-valued function, -- window table-valued function should be used with aggregate operation, -- this example is just used for explaining the syntax and the data produced by table-valued function. > SELECT \* FROM TABLE(  CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES)); -- or with the named params -- note: the DATA param must be the first > SELECT \* FROM TABLE(  CUMULATE(  DATA => TABLE Bid,  TIMECOL => DESCRIPTOR(bidtime),  STEP => INTERVAL '2' MINUTES,  SIZE => INTERVAL '10' MINUTES)); +------------------+-------+------+------------------+------------------+-------------------------+ | bidtime | price | item | window\_start | window\_end | window\_time | +------------------+-------+------+------------------+------------------+-------------------------+ | 2020-04-15 08:05 | 4.00 | C | 2020-04-15 08:00 | 2020-04-15 08:06 | 2020-04-15 08:05:59.999 | | 2020-04-15 08:05 | 4.00 | C | 2020-04-15 08:00 | 2020-04-15 08:08 | 2020-04-15 08:07:59.999 | | 2020-04-15 08:05 | 4.00 | C | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:07 | 2.00 | A | 2020-04-15 08:00 | 2020-04-15 08:08 | 2020-04-15 08:07:59.999 | | 2020-04-15 08:07 | 2.00 | A | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:09 | 5.00 | D | 2020-04-15 08:00 | 2020-04-15 08:10 | 2020-04-15 08:09:59.999 | | 2020-04-15 08:11 | 3.00 | B | 2020-04-15 08:10 | 2020-04-15 08:12 | 2020-04-15 08:11:59.999 | | 2020-04-15 08:11 | 3.00 | B | 2020-04-15 08:10 | 2020-04-15 08:14 | 2020-04-15 08:13:59.999 | | 2020-04-15 08:11 | 3.00 | B | 2020-04-15 08:10 | 2020-04-15 08:16 | 2020-04-15 08:15:59.999 | | 2020-04-15 08:11 | 3.00 | B | 2020-04-15 08:10 | 2020-04-15 08:18 | 2020-04-15 08:17:59.999 | | 2020-04-15 08:11 | 3.00 | B | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | | 2020-04-15 08:13 | 1.00 | E | 2020-04-15 08:10 | 2020-04-15 08:14 | 2020-04-15 08:13:59.999 | | 2020-04-15 08:13 | 1.00 | E | 2020-04-15 08:10 | 2020-04-15 08:16 | 2020-04-15 08:15:59.999 | | 2020-04-15 08:13 | 1.00 | E | 2020-04-15 08:10 | 2020-04-15 08:18 | 2020-04-15 08:17:59.999 | | 2020-04-15 08:13 | 1.00 | E | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | | 2020-04-15 08:17 | 6.00 | F | 2020-04-15 08:10 | 2020-04-15 08:18 | 2020-04-15 08:17:59.999 | | 2020-04-15 08:17 | 6.00 | F | 2020-04-15 08:10 | 2020-04-15 08:20 | 2020-04-15 08:19:59.999 | +------------------+-------+------+------------------+------------------+-------------------------+  -- apply aggregation on the cumulating windowed table > SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; +------------------+------------------+-------+ | window\_start | window\_end | price | +------------------+------------------+-------+ | 2020-04-15 08:00 | 2020-04-15 08:06 | 4.00 | | 2020-04-15 08:00 | 2020-04-15 08:08 | 6.00 | | 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 | | 2020-04-15 08:10 | 2020-04-15 08:12 | 3.00 | | 2020-04-15 08:10 | 2020-04-15 08:14 | 4.00 | | 2020-04-15 08:10 | 2020-04-15 08:16 | 4.00 | | 2020-04-15 08:10 | 2020-04-15 08:18 | 10.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 | +------------------+------------------+-------+ |

##### Window Aggregation

###### Window TVF Aggregation

**应用场景：流**

窗口聚合被定义在GROUP BY包含“window\_start”和“window\_end”列的应用[Windowing TVF](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-tvf/)关系的子句中。就像使用常规GROUP BY子句的查询一样，使用 group by window 聚合的查询将为每个 group 计算一个结果行。

|  |
| --- |
| SELECT ... FROM <windowed\_table> -- relation applied windowing TVF GROUP BY window\_start, window\_end, ... |

与连续表上的其他聚合不同，窗口聚合不会发出中间结果，而只会发出最终结果，即窗口末尾的总聚合。此外，当不再需要时，窗口聚合会清除所有中间状态。

窗口化 TVF

Flink 支持TUMBLE、HOP和CUMULATE类型的窗口聚合，可以[在事件或处理时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)上定义。有关更多窗口函数信息，请参阅[窗口化 TVF](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-tvf/)。

以下是TUMBLE,HOP和CUMULATE窗口聚合的一些示例。

|  |
| --- |
| -- tables must have time attribute, e.g. `bidtime` in this table Flink SQL> desc Bid; +-------------+------------------------+------+-----+--------+---------------------------------+ | name | type | null | key | extras | watermark | +-------------+------------------------+------+-----+--------+---------------------------------+ | bidtime | TIMESTAMP(3) \*ROWTIME\* | true | | | `bidtime` - INTERVAL '1' SECOND | | price | DECIMAL(10, 2) | true | | | | | item | STRING | true | | | | | supplier\_id | STRING | true | | | | +-------------+------------------------+------+-----+--------+---------------------------------+  Flink SQL> SELECT \* FROM Bid; +------------------+-------+------+-------------+ | bidtime | price | item | supplier\_id | +------------------+-------+------+-------------+ | 2020-04-15 08:05 | 4.00 | C | supplier1 | | 2020-04-15 08:07 | 2.00 | A | supplier1 | | 2020-04-15 08:09 | 5.00 | D | supplier2 | | 2020-04-15 08:11 | 3.00 | B | supplier2 | | 2020-04-15 08:13 | 1.00 | E | supplier1 | | 2020-04-15 08:17 | 6.00 | F | supplier2 | +------------------+-------+------+-------------+  -- tumbling window aggregation Flink SQL> SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; +------------------+------------------+-------+ | window\_start | window\_end | price | +------------------+------------------+-------+ | 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 | +------------------+------------------+-------+  -- hopping window aggregation Flink SQL> SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  HOP(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; +------------------+------------------+-------+ | window\_start | window\_end | price | +------------------+------------------+-------+ | 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 | | 2020-04-15 08:05 | 2020-04-15 08:15 | 15.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 | | 2020-04-15 08:15 | 2020-04-15 08:25 | 6.00 | +------------------+------------------+-------+  -- cumulative window aggregation Flink SQL> SELECT window\_start, window\_end, SUM(price)  FROM TABLE(  CUMULATE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '2' MINUTES, INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; +------------------+------------------+-------+ | window\_start | window\_end | price | +------------------+------------------+-------+ | 2020-04-15 08:00 | 2020-04-15 08:06 | 4.00 | | 2020-04-15 08:00 | 2020-04-15 08:08 | 6.00 | | 2020-04-15 08:00 | 2020-04-15 08:10 | 11.00 | | 2020-04-15 08:10 | 2020-04-15 08:12 | 3.00 | | 2020-04-15 08:10 | 2020-04-15 08:14 | 4.00 | | 2020-04-15 08:10 | 2020-04-15 08:16 | 4.00 | | 2020-04-15 08:10 | 2020-04-15 08:18 | 10.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | 10.00 | +------------------+------------------+-------+ |

|  |
| --- |
| **注意：**为了更好地理解窗口化的行为，我们简化了时间戳值的显示以不显示尾随零，例如，如果类型为，则2020-04-15 08:05应该像2020-04-15 08:05:00.000在 Flink SQL Client 中那样显示TIMESTAMP(3) |

GROUPING SETS

窗口聚合也支持GROUPING SETS语法。分组集允许比标准描述的更复杂的分组操作GROUP BY。行按每个指定的分组集单独分组，并且为每个组计算聚合，就像简单GROUP BY子句一样。

具有GROUPING SETSrequirewindow\_start和window\_end列的窗口聚合必须在GROUP BY子句中，但不在GROUPING SETS子句中。

|  |
| --- |
| Flink SQL> SELECT window\_start, window\_end, supplier\_id, SUM(price) as price  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end, GROUPING SETS ((supplier\_id), ()); +------------------+------------------+-------------+-------+ | window\_start | window\_end | supplier\_id | price | +------------------+------------------+-------------+-------+ | 2020-04-15 08:00 | 2020-04-15 08:10 | (NULL) | 11.00 | | 2020-04-15 08:00 | 2020-04-15 08:10 | supplier2 | 5.00 | | 2020-04-15 08:00 | 2020-04-15 08:10 | supplier1 | 6.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | (NULL) | 10.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | supplier2 | 9.00 | | 2020-04-15 08:10 | 2020-04-15 08:20 | supplier1 | 1.00 | +------------------+------------------+-------------+-------+ |

每个子列表GROUPING SETS可以指定零个或多个列或表达式，并以与直接在GROUP BY子句中使用的方式相同的方式进行解释。空分组集意味着所有行都聚合到一个组中，即使不存在输入行也会输出。

对分组列或表达式的引用被替换为结果行中的空值，用于分组集中没有出现这些列的集合。

ROLLUP

ROLLUP是用于指定常见类型的分组集的速记符号。它表示给定的表达式列表和列表的所有前缀，包括空列表。

窗口聚合ROLLUP要求 thewindow\_start和window\_endcolumns 都必须在GROUP BY子句中，但不在ROLLUP子句中。

例如，以下查询等效于上面的查询。

|  |
| --- |
| SELECT window\_start, window\_end, supplier\_id, SUM(price) as price FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES)) GROUP BY window\_start, window\_end, ROLLUP (supplier\_id); |

CUBE

CUBE是用于指定常见类型的分组集的速记符号。它代表给定的列表及其所有可能的子集——幂集。

窗口聚合CUBE要求 thewindow\_start和window\_endcolumns 都必须在GROUP BY子句中，但不在CUBE子句中。

例如，以下两个查询是等效的。

|  |
| --- |
| SELECT window\_start, window\_end, item, supplier\_id, *SUM*(price) as price  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end, CUBE (supplier\_id, item);  SELECT window\_start, window\_end, item, supplier\_id, *SUM*(price) as price  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end, GROUPING SETS (  (supplier\_id, item),  (supplier\_id ),  ( item),  ( ) ) |

选择组窗口开始和结束时间戳

分组窗口的开始和结束时间戳可以通过分组window\_start和window\_end列来选择。

[级联窗口聚合](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/" \l "cascading-window-aggregation)

在window\_start与window\_end列是定期时间戳列，而不是时间属性。因此，它们不能用作后续基于时间的操作中的时间属性。为了传播时间属性，您需要在子句中额外添加window\_time列GROUP BY。的window\_time是所产生的第三列[加窗TVFs](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-tvf/" \l "window-functions)这是所分配的窗口的时间属性。添加window\_timeintoGROUP BY子句window\_time也使成为可以选择的组键。然后后续查询可以使用此列进行后续基于时间的操作，例如级联窗口聚合和[Window TopN](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-topn/)。

下面显示了级联窗口聚合，其中第一个窗口聚合传播第二个窗口聚合的时间属性。

|  |
| --- |
| -- tumbling 5 minutes for each supplier\_id CREATE VIEW window1 AS SELECT window\_start, window\_end, window\_time as rowtime, SUM(price) as partial\_price  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '5' MINUTES))  GROUP BY supplier\_id, window\_start, window\_end, window\_time;  -- tumbling 10 minutes on the first window SELECT window\_start, window\_end, SUM(partial\_price) as total\_price  FROM TABLE(  TUMBLE(TABLE window1, DESCRIPTOR(rowtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end; |

###### Group Window Aggregation

**应用场景：批或者流**

不推荐使用：<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/window-agg/>

##### Group Aggregation

**应用场景：批或者流**

和大多数数据系统一样，Apache Flink 支持聚合函数；内置的和用户定义的。[用户定义的函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/udfs/)必须在使用前注册到目录中。

聚合函数从多个输入行计算单个结果。例如，有聚合来计算一组行上的COUNT、SUM、AVG（平均值）、MAX（最大值）和MIN（最小值）

|  |
| --- |
| SELECT COUNT(\*) FROM Orders |

对于流式查询，重要的是要了解 Flink 运行永不终止的连续查询。相反，他们根据其输入表的更新来更新他们的结果表。对于上面的查询，每次向Orders表中插入新行时，Flink 都会输出更新的计数。

Apache Flink 支持GROUP BY用于聚合数据的标准子句。

|  |
| --- |
| SELECT COUNT(\*) FROM Orders GROUP BY order\_id |

对于流式查询，计算查询结果所需的状态可能会无限增长。状态大小取决于组的数量以及聚合函数的数量和类型。例如MIN/MAX在状态大小上很重而COUNT便宜。您可以提供具有适当状态生存时间 (TTL) 的查询配置，以防止状态过大。请注意，这可能会影响查询结果的正确性。有关详细信息，请参阅FlinkSql优化章节的**group Aggregate优化技巧**。

###### DISTINCT 聚合

不同的聚合在应用聚合函数之前删除重复值。以下示例计算不同 order\_id 的数量，而不是 Orders 表中的总行数。

|  |
| --- |
| SELECT COUNT(DISTINCT order\_id) FROM Orders |

对于流式查询，计算查询结果所需的状态可能会无限增长。状态大小主要取决于不同行的数量和维护组的时间，按窗口进行的短期组不是问题。您可以提供具有适当状态生存时间 (TTL) 的查询配置，以防止状态过大。请注意，这可能会影响查询结果的正确性。有关详细信息，请参阅查询配置。

###### GROUPING SETS

GROUPING SETS允许比标准描述的更复杂的分组操作GROUP BY。行按每个指定的分组集单独分组，并且为每个组计算聚合，就像简单GROUP BY子句一样。

|  |
| --- |
| SELECT supplier\_id, rating, *COUNT*(*\**) AS total FROM (VALUES  ('supplier1', 'product1', 4),  ('supplier1', 'product2', 3),  ('supplier2', 'product3', 3),  ('supplier2', 'product4', 4)) AS Products(supplier\_id, product\_id, rating) GROUP BY GROUPING SETS ((supplier\_id, rating), (supplier\_id), ()) |

结果：

|  |
| --- |
| +-------------+--------+-------+ | supplier\_id | rating | total | +-------------+--------+-------+ | supplier1 | 4 | 1 | | supplier1 | (NULL) | 2 | | (NULL) | (NULL) | 4 | | supplier1 | 3 | 1 | | supplier2 | 3 | 1 | | supplier2 | (NULL) | 2 | | supplier2 | 4 | 1 | +-------------+--------+-------+ |

每个子列表GROUPING SETS可以指定零个或多个列或表达式，并且解释方式与直接在GROUP BY子句中使用的方式相同。空分组集意味着所有行都聚合到一个组中，即使不存在输入行也会输出。

对分组列或表达式的引用被替换为结果行中的空值，用于分组集中没有出现这些列的集合。

对于流式查询，计算查询结果所需的状态可能会无限增长。状态大小取决于组集的数量和聚合函数的类型。您可以提供具有适当状态生存时间 (TTL) 的查询配置，以防止状态过大。请注意，这可能会影响查询结果的正确性。有关详细信息，请参阅查询配置。

ROLLUP

ROLLUP是用于指定常见类型的分组集的速记符号。它表示给定的表达式列表和列表的所有前缀，包括空列表。

例如，以下查询等效于上面的查询。

|  |
| --- |
| SELECT supplier\_id, rating, COUNT(\*) FROM (VALUES  ('supplier1', 'product1', 4),  ('supplier1', 'product2', 3),  ('supplier2', 'product3', 3),  ('supplier2', 'product4', 4)) AS Products(supplier\_id, product\_id, rating) GROUP BY ROLLUP (supplier\_id, rating) |

CUBE

CUBE是用于指定常见类型的分组集的速记符号。它代表给定的列表及其所有可能的子集——幂集。

例如，以下两个查询是等效的。

|  |
| --- |
| SELECT supplier\_id, rating, product\_id, COUNT(\*) FROM (VALUES  ('supplier1', 'product1', 4),  ('supplier1', 'product2', 3),  ('supplier2', 'product3', 3),  ('supplier2', 'product4', 4)) AS Products(supplier\_id, product\_id, rating) GROUP BY CUBE (supplier\_id, rating, product\_id)  SELECT supplier\_id, rating, product\_id, COUNT(\*) FROM (VALUES  ('supplier1', 'product1', 4),  ('supplier1', 'product2', 3),  ('supplier2', 'product3', 3),  ('supplier2', 'product4', 4)) AS Products(supplier\_id, product\_id, rating) GROUP BY GROUPING SET (  ( supplier\_id, product\_id, rating ),  ( supplier\_id, product\_id ),  ( supplier\_id, rating ),  ( supplier\_id ),  ( product\_id, rating ),  ( product\_id ),  ( rating ),  ( ) ) |

###### HAVING

HAVING消除不满足条件的组行。HAVING不同于WHERE:WHERE在GROUP BYwhileHAVING过滤器组由GROUP BY.创建的行之前过滤单个行。条件中引用的每个列必须明确引用分组列，除非它出现在聚合函数中。

|  |
| --- |
| SELECT SUM(amount) FROM Orders GROUP BY users HAVING SUM(amount) > 50 |

如果HAVING条件为真，这样的查询将发出单行，如果条件不为真，则将发出零行。

##### Over Aggregation

**应用场景：批或者流**

OVER聚合为有序行范围内的每个输入行计算聚合值。与GROUP BY聚合相反，OVER聚合不会将每个组的结果行数减少为一行。相反，OVER聚合会为每个输入行生成一个聚合值。

以下查询为每个订单计算在当前订单前一小时内收到的同一产品的所有订单的总和。

|  |
| --- |
| SELECT order\_id, order\_time, amount,  SUM(amount) OVER (  PARTITION BY product  ORDER BY order\_time  RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW  ) AS one\_hour\_prod\_amount\_sum FROM Orders |

OVER窗口的语法总结如下。

|  |
| --- |
| SELECT  agg\_func(agg\_col) OVER (  [PARTITION BY col1[, col2, ...]]  ORDER BY time\_col  range\_definition),  ... FROM ... |

您可以OVER在一个SELECT子句中定义多个窗口聚合。但是，对于流式查询，OVER由于当前限制，所有聚合的窗口必须相同。

###### ORDER BY

ORDER BY必须在单个时间属性上指定

###### PARTITION BY

OVER可以在分区表上定义窗口。在存在PARTITION BY子句的情况下，仅在其分区的行上为每个输入行计算聚合。

###### Range Definitions

范围定义指定聚合中包含的行数。范围是用一个BETWEEN子句定义的，该子句定义了下边界和上边界。这些边界之间的所有行都包含在聚合中。Flink 只支持CURRENT ROW作为上限。

有两个选项可以定义范围、ROWS间隔和RANGE间隔。

RANGE intervals

RANGE在 ORDER BY 列的值上定义了一个间隔，在 Flink 的情况下，它始终是一个时间属性。以下 RANGE 间隔定义时间属性最多比当前行少 30 分钟的所有行都包含在聚合中。

|  |
| --- |
| RANGE BETWEEN INTERVAL '30' MINUTE PRECEDING AND CURRENT ROW |

ROW intervals

ROWS间隔是基于计数的时间间隔。它准确定义了聚合中包含的行数。以下ROWS间隔定义当前行和当前行之前的 10 行（因此总共 11 行）包含在聚合中。

|  |
| --- |
| ROWS BETWEEN 10 PRECEDING AND CURRENT ROW WINDOW |

该WINDOW子句可用于OVER在该SELECT子句之外定义一个窗口。它可以使查询更具可读性，还允许我们为多个聚合重用窗口定义。

|  |
| --- |
| SELECT order\_id, order\_time, amount,  SUM(amount) OVER w AS sum\_amount,  AVG(amount) OVER w AS avg\_amount FROM Orders WINDOW w AS (  PARTITION BY product  ORDER BY order\_time  RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW) |

##### Joins

**应用场景：批或者流**

Flink SQL 支持对动态表进行复杂灵活的连接操作。有几种不同类型的连接来解释可能需要的各种语义查询。

默认情况下，联接的顺序未优化。表按照它们在FROM子句中指定的顺序进行连接。您可以调整连接查询的性能，方法是首先列出更新频率最低的表，最后列出更新频率最高的表。确保以不会产生交叉联接（笛卡尔积）的顺序指定表，这不受支持并且会导致查询失败。

###### Regular Joins（常规join）

Regular Joins是最通用的联接类型，其中任何新记录或联接任一侧的更改都是可见的，并且会影响整个联接结果。例如，如果左侧有一条新记录，它将与右侧所有以前和未来的记录连接起来。

|  |
| --- |
| SELECT *\** FROM Orders INNER JOIN Product ON Orders.productId = Product.id |

对于流式查询，常规连接的语法是最灵活的，并且允许任何类型的更新（插入、更新、删除）输入表。然而，这个操作具有重要的操作意义：它需要将连接输入的双方永远保持在 Flink 状态。因此，计算查询结果所需的状态可能会无限增长，具体取决于所有输入表和中间连接结果的不同输入行的数量。您可以提供具有适当状态生存时间 (TTL) 的查询配置，以防止状态过大。请注意，这可能会影响查询结果的正确性。有关详细信息，请参阅查询配置。

对于流式查询，计算查询结果所需的状态可能会无限增长，具体取决于聚合类型和不同分组键的数量。请提供具有有效保留间隔的查询配置，以防止状态大小过大。有关详细信息，请参阅查询配置。

INNER Equi-JOIN

返回受连接条件限制的简单笛卡尔积。目前，仅支持等值联接，即具有至少一个带有等式谓词的连接条件的联接。不支持任意交叉或 theta 连接。

|  |
| --- |
| SELECT *\** FROM Orders INNER JOIN Product ON Orders.product\_id = Product.id |

OUTER Equi-JOIN

返回符合条件的笛卡尔积中的所有行（即，通过其连接条件的所有组合行），以及连接条件与其他表的任何行都不匹配的外部表中每一行的一个副本。Flink 支持 LEFT、RIGHT 和 FULL 外连接。目前，仅支持等值联接，即，联接具有至少一个带有等式谓词的连接条件。不支持任意交叉或 theta 连接。

|  |
| --- |
| SELECT *\** FROM Orders LEFT JOIN Product ON Orders.product\_id = Product.id  SELECT *\** FROM Orders RIGHT JOIN Product ON Orders.product\_id = Product.id  SELECT *\** FROM Orders FULL OUTER JOIN Product ON Orders.product\_id = Product.id |

###### Interval Joins

返回受连接条件和时间约束限制的简单笛卡尔积。间隔连接至少需要一个等值连接谓词和一个限制双方时间的连接条件。两个合适的范围谓词可以定义这样一个条件（<、<=、>=、>）、一个 BETWEEN 谓词或一个比较两个输入的相同类型（即处理时间或事件时间）的[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)的等式谓词表。

例如，如果订单在收到订单后四小时发货，则此查询会将所有订单与其相应的发货联系起来。

|  |
| --- |
| SELECT *\** FROM Orders o, Shipments s WHERE o.id = s.order\_id AND o.order\_time BETWEEN s.ship\_time - INTERVAL '4' HOUR AND s.ship\_time |

以下谓词是有效区间连接条件的示例：

* ltime = rtime
* ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE
* ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND

对于流式查询，与常规连接相比，间隔连接仅支持具有时间属性的 append-only 表。由于时间属性是准单调递增的，因此 Flink 可以从其状态中移除旧值，而不会影响结果的正确性。

###### [Temporal Joins](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "temporal-joins)

Event Time Temporal Join

[Temporal Joins](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "temporal-joins)允许对[版本化表](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/versioned_tables/)进行连接。这意味着可以通过不断变化的元数据来丰富表格，并在某个时间点检索其值。

[Temporal Joins](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "temporal-joins)采用任意表（左输入/探测站点）并将每一行与版本化表（右输入/构建侧）中相应行的相关版本相关联。Flink 使用FOR SYSTEM\_TIME AS OF来自 SQL:2011 标准的 SQL 语法来执行此操作。

临时连接的语法如下；

|  |
| --- |
| SELECT [column\_list] FROM table1 [AS <alias1>] [LEFT] JOIN table2 FOR SYSTEM\_TIME AS OF table1.{ proctime | rowtime } [AS <alias2>] ON table1.column-name1 = table2.column-name1 |

使用事件时间属性（即行时间属性），可以检索键的值，就像它在过去的某个时间点一样。这允许在共同的时间点连接两个表。版本化表将存储自上次水印以来的所有版本 - 按时间标识。

例如，假设我们有一个订单表，每个订单都有不同货币的价格。要将这个表正确地标准化为单一货币，例如美元，每个订单都需要从下订单的时间点加入正确的货币兑换率。

|  |
| --- |
| -- Create a table of orders. This is a standard -- append-only dynamic table. CREATE TABLE orders (  order\_id STRING,  price DECIMAL(32,2),  currency STRING,  order\_time TIMESTAMP(3),  WATERMARK FOR order\_time AS order\_time ) WITH (/\* ... \*/);  -- Define a versioned table of currency rates.  -- This could be from a change-data-capture -- such as Debezium, a compacted Kafka topic, or any other -- way of defining a versioned table.  CREATE TABLE currency\_rates (  currency STRING,  conversion\_rate DECIMAL(32, 2),  update\_time TIMESTAMP(3) METADATA FROM `values.source.timestamp` VIRTUAL  WATERMARK FOR update\_time AS update\_time ) WITH (  'connector' = 'upsert-kafka',  /\* ... \*/ );  SELECT   order\_id,  price,  currency,  conversion\_rate,  order\_time, FROM orders LEFT JOIN currency\_rates FOR SYSTEM TIME AS OF orders.order\_time ON orders.currency = currency\_rates.currency  order\_id price currency conversion\_rate order\_time ====== ==== ====== ============ ======== o\_001 11.11 EUR 1.14 12:00:00 o\_002 12.51 EUR 1.10 12:0600 |

**注意**： event-time temporal join是由左右两侧的水印触发的；请确保连接的双方都正确设置了水印。

**注意**：事件时间临时连接需要临时连接条件的等价条件中包含的主键，例如，P.product\_id表的主键product\_changelog被约束在条件中O.product\_id = P.product\_id。

Processing Time Temporal Join

处理时间时态表join使用处理时间属性将行与外部版本化表中的键的最新版本相关联。

根据定义，使用处理时间属性，连接将始终返回给定键的最新值。可以将查找表视为一个简单的 HashMap<K, V>，它存储来自构建端的所有记录。这种连接的强大之处在于，当在 Flink 中将表具体化为动态表不可行时，它允许 Flink 直接针对外部系统工作。

以下处理时时态表orders联接示例显示了应与表联接的仅附加表LatestRates。 LatestRates是一个以最新速率物化的维度表（例如HBase 表）。在10:15, 10:30, 时10:52， 的内容LatestRates如下：

|  |
| --- |
| 10:15> SELECT *\** FROM LatestRates;  currency rate ======== ====== US Dollar 102 Euro 114 Yen 1  10:30> SELECT *\** FROM LatestRates;  currency rate ======== ====== US Dollar 102 Euro 114 Yen 1  10:52> SELECT *\** FROM LatestRates;  currency rate ======== ====== US Dollar 102 Euro 116 <==== changed from 114 to 116 Yen 1 |

LastestRates有时10:15和的内容10:30是相等的。欧元汇率从 114 变为 116 10:52。

Orders是一个仅附加表，表示给定amount和给定的付款currency。例如，10:15有一个订单金额为2 Euro。

|  |
| --- |
| SELECT *\** FROM Orders;  amount currency ====== =========  2 Euro <== arrived at time 10:15  1 US Dollar <== arrived at time 10:30  2 Euro <== arrived at time 10:52 |

鉴于这些表，我们想计算所有Orders转换为通用货币的方法。

|  |
| --- |
| amount currency rate amount\*rate ====== ========= ======= ============  2 Euro 114 228 <== arrived at time 10:15  1 US Dollar 102 102 <== arrived at time 10:30  2 Euro 116 232 <== arrived at time 10:52 |

借助时态表连接，我们可以在 SQL 中表达这样的查询：

|  |
| --- |
| SELECT  o.amount, o.currency, r.rate, o.amount \* r.rate FROM  Orders AS o  JOIN LatestRates FOR SYSTEM\_TIME AS OF o.proctime AS r  ON r.currency = o.currency |

探针端的每条记录都将与构建端表的当前版本连接。在我们的示例中，查询使用处理时间概念，因此新添加的订单将始终与LatestRates执行操作时的最新版本连接。

结果对于处理时间是不确定的。处理时时间连接最常用于通过外部表（即维度表）丰富流。

与[常规连接相比](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "regular-joins)，尽管构建方面发生了变化，但之前的时态表结果不会受到影响。与Interval Joins相比，时态表连接不定义记录连接的时间窗口，即旧行不存储在状态中。

###### Lookup Join

Lookup Join通常用于使用从外部系统查询的数据来丰富表。连接要求一个表具有处理时间属性，另一个表由查找源连接器支持。

Lookup Join使用上述[处理时间临时联接](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "processing-time-temporal-join)语法，并使用由查找源连接器支持的正确表。

以下示例显示了指定查找联接的语法。

|  |
| --- |
| -- Customers is backed by the JDBC connector and can be used for lookup joins CREATE TEMPORARY TABLE Customers (  id INT,  name STRING,  country STRING,  zip STRING ) WITH (  'connector' = 'jdbc',  'url' = 'jdbc:mysql://mysqlhost:3306/customerdb',  'table-name' = 'customers' );  -- enrich each order with customer information SELECT o.order\_id, o.total, c.country, c.zip FROM Orders AS o  JOIN Customers FOR SYSTEM\_TIME AS OF o.proc\_time AS c  ON o.customer\_id = c.id; |

在上面的示例中，Orders 表中包含来自位于 MySQL 数据库中的 Customers 表的数据**。FOR SYSTEM\_TIME AS OF**带有后续处理时间属性的子句可确保在连接运算符处理行时，将Orders表的每一行与与连接谓词匹配的客户行Orders连接起来。它还可以防止Customer在将来更新连接的行时更新连接结果。在上面的示例中，查找连接还需要强制相等连接谓词o.customer\_id = c.id。

###### Table Function

将表与表函数的结果连接起来。左（外部）表的每一行都与表函数的相应调用产生的所有行连接。[用户自定义表函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/udfs/" \l "table-functions)在使用前必须注册。

INNER JOIN

如果表函数调用返回空结果，则删除左（外部）表的行。

|  |
| --- |
| SELECT order\_id, res FROM Orders, LATERAL TABLE(table\_func(order\_id)) t(res) |

LEFT OUTER JOIN

如果表函数调用返回空结果，则保留相应的外部行，并用空值填充结果。当前，针对横向表的左外连接需要 ON 子句中的 TRUE 文字。

|  |
| --- |
| SELECT order\_id, res FROM Orders LEFT OUTER JOIN LATERAL TABLE(table\_func(order\_id)) t(res)  ON TRUE |

##### 集合操作

**应用场景：批或者流**

###### UNION （合并）

UNION并UNION ALL返回在任一表中找到的行。 UNION仅采用不同的行，而UNION ALL不会从结果行中删除重复项。

|  |
| --- |
| Flink SQL> create view t1(s) as values ('c'), ('a'), ('b'), ('b'), ('c'); Flink SQL> create view t2(s) as values ('d'), ('e'), ('a'), ('b'), ('b');  Flink SQL> (SELECT s FROM t1) UNION (SELECT s FROM t2); +---+ | s| +---+ | c| | a| | b| | d| | e| +---+  Flink SQL> (SELECT s FROM t1) UNION ALL (SELECT s FROM t2); +---+ | c| +---+ | c| | a| | b| | b| | c| | d| | e| | a| | b| | b| +---+ |

###### INTERSECT（相交）

INTERSECT并INTERSECT ALL返回在两个表中找到的行。 INTERSECT仅采用不同的行，而INTERSECT ALL不会从结果行中删除重复项。

|  |
| --- |
| Flink SQL> (SELECT s FROM t1) INTERSECT (SELECT s FROM t2); +---+ | s| +---+ | a| | b| +---+  Flink SQL> (SELECT s FROM t1) INTERSECT ALL (SELECT s FROM t2); +---+ | s| +---+ | a| | b| | b| +---+ |

###### EXCEPT（除了..之外）

EXCEPT并EXCEPT ALL返回在一个表中找到但在另一个表中找不到的行。 EXCEPT仅采用不同的行，而EXCEPT ALL不会从结果行中删除重复项。

|  |
| --- |
| Flink SQL> (SELECT s FROM t1) EXCEPT (SELECT s FROM t2); +---+ | s | +---+ | c | +---+  Flink SQL> (SELECT s FROM t1) EXCEPT ALL (SELECT s FROM t2); +---+ | s | +---+ | c | | c | +---+ |

###### IN

如果表达式存在于给定的表子查询中，则返回 true。子查询表必须由一列组成。此列必须与表达式具有相同的数据类型。

|  |
| --- |
| SELECT user, amount FROM Orders WHERE product IN (  SELECT product FROM NewProducts ) |

优化器将 IN 条件重写为连接和组操作。对于流式查询，计算查询结果所需的状态可能会根据不同输入行的数量无限增长。您可以提供具有适当状态生存时间 (TTL) 的查询配置，以防止状态过大。请注意，这可能会影响查询结果的正确性。有关详细信息，请参阅[查询配置](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "table-exec-state-ttl)。

###### EXISTS

|  |
| --- |
| SELECT user, amount FROM Orders WHERE product EXISTS (  SELECT product FROM NewProducts ) |

如果子查询至少返回一行，则返回 true。仅当可以在加入和组操作中重写操作时才支持。

优化器将EXISTS操作重写为连接和组操作。对于流式查询，计算查询结果所需的状态可能会根据不同输入行的数量无限增长。您可以提供具有适当状态生存时间 (TTL) 的查询配置，以防止状态过大。请注意，这可能会影响查询结果的正确性。有关详细信息，请参阅[查询配置](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "table-exec-state-ttl)。

##### OrderBy

**应用场景：批或者流**

该ORDER BY子句使结果行根据指定的表达式进行排序。如果根据最左边的表达式两行相等，则根据下一个表达式进行比较，依此类推。如果根据所有指定的表达式它们都相等，则它们以依赖于实现的顺序返回。

在流模式下运行时，表的主要排序顺序必须在[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)上升序。所有后续订单都可以自由选择。但是批处理模式没有这个限制。

|  |
| --- |
| SELECT *\** FROM Orders ORDER BY order\_time, order\_id |

##### LIMIT

**应用场景：批**

LIMIT子句约束SELECT语句返回的行数。通常，该子句与 ORDER BY 结合使用以确保结果是确定性的。

以下示例选择Orders表中的前 3 行。

|  |
| --- |
| SELECT \* FROM Orders ORDER BY orderTime LIMIT 3 |

##### Top-N

**应用场景：批或者流**

Top-N 查询要求按列排序的 N 个最小值或最大值。最小值和最大值集都被视为 Top-N 查询。在需要根据条件仅显示批处理/流表中的 N 个最底层或 N 个最顶层记录的情况下，前 N 个查询很有用。该结果集可用于进一步分析。

Flink 使用 OVER window 子句和过滤条件的组合来表达 Top-N 查询。借助 OVER windowPARTITION BY子句的强大功能，Flink 还支持按组 Top-N。

例如，每个类别实时销售额最高的前五名产品。批处理表和流表上的 SQL 支持前 N 个查询。

下面显示了 Top-N 语句的语法：

|  |
| --- |
| SELECT [column\_list] FROM (  SELECT [column\_list],  ROW\_NUMBER() OVER ([PARTITION BY col1[, col2...]]  ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum  FROM table\_name) WHERE rownum <= N [AND conditions] |

参数规格：

* ROW\_NUMBER()：根据分区内行的顺序，为每一行分配一个唯一的序列号，从一个开始。目前，我们只支持ROW\_NUMBER作为过窗函数。未来，我们将支持RANK()和DENSE\_RANK()。
* PARTITION BY col1[, col2...]: 指定分区列。每个分区都会有一个 Top-N 结果。
* ORDER BY col1 [asc|desc][, col2 [asc|desc]...]: 指定排序列。不同列的排序方向可能不同。
* WHERE rownum <= N：rownum <= NFlink 需要识别此查询是 Top-N 查询。N 代表将保留的 N 个最小或最大记录。
* [AND conditions]: where子句可以随意添加其他条件，但其他条件只能rownum <= N用AND连词组合。

|  |
| --- |
| **注意：**必须严格遵循上述模式，否则优化器将无法翻译查询。 |

**流处理模式**需注意 TopN 查询的结果会带有更新。 Flink SQL 会根据排序键对输入的流进行排序；若 top N 的记录发生了变化，变化的部分会以撤销、更新记录的形式发送到下游。 推荐使用一个支持更新的存储作为 Top-N 查询的 sink 。另外，若 top N 记录需要存储到外部存储，则结果表需要拥有相同与 Top-N 查询相同的唯一键。

Top-N 的唯一键是分区列和 rownum 列的结合，另外 Top-N 查询也可以获得上游的唯一键。以下面的任务为例，product\_id 是 ShopSales 的唯一键，然后 Top-N 的唯一键是 [category, rownum] 和 [product\_id] 。

以下示例展示了如何在流表上使用 Top-N 指定 SQL 查询。这是我们上面提到的“实时获得最大销售额的每个类别的前五名产品”的示例。

|  |
| --- |
| CREATE TABLE ShopSales (  product\_id STRING,  category STRING,  product\_name STRING,  sales BIGINT ) WITH (...);  SELECT \* FROM (  SELECT \*,  ROW\_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) AS row\_num  FROM ShopSales) WHERE row\_num <= 5 |

###### 无排名输出优化

如上文所描述，rownum 字段会作为唯一键的其中一个字段写到结果表里面，这会导致大量的结果写出到结果表。比如，当原始结果（名为 product-1001 ）从排序第九变化为排序第一时，排名 1-9 的所有结果都会以更新消息的形式发送到结果表。若结果表收到太多的数据，将会成为 SQL 任务的瓶颈。

优化方法是在 Top-N 查询的外部 SELECT 子句中省略 rownum 字段。由于前N条记录的数量通常不大，因此消费者可以自己对记录进行快速排序，因此这是合理的。去掉 rownum 字段后，上述的例子中，只有变化了的记录（ product-1001 ）需要发送到下游，从而可以节省大量的对结果表的 IO 操作。

以下的例子描述了如何以这种方式优化上述的 Top-N 查询：

|  |
| --- |
| CREATE TABLE ShopSales (  product\_id STRING,  category STRING,  product\_name STRING,  sales BIGINT ) WITH (...);  -- omit row\_num field from the output SELECT product\_id, category, product\_name, sales FROM (  SELECT \*,  ROW\_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) AS row\_num  FROM ShopSales) WHERE row\_num <= 5 |

|  |
| --- |
| **使用流处理模式时需注意：**为了使上述查询输出可以输出到外部存储并且结果正确，外部存储需要拥有与 Top-N 查询一致的唯一键。在上述的查询例子中，若 product\_id 是查询的唯一键，那么外部表必须要有 product\_id 作为其唯一键。 |

##### Window Top-N

**应用场景：流**

Window Top-N 是一个特殊的Top-N，它返回每个窗口和其他分区键的 N 个最小值或最大值。

对于流式查询，与连续表上的常规 Top-N 不同，窗口 Top-N 不会发出中间结果，而只会发出最终结果，即窗口末尾的总前 N 条记录。此外，窗口 Top-N 会在不再需要时清除所有中间状态。因此，如果用户不需要每条记录更新结果，则窗口 Top-N 查询具有更好的性能。通常，Window Top-N 与Window Aggregation一起使用。

Window Top-N 可以用与常规 Top-N 相同的语法定义，请参阅[Top-N 文档](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/topn/)以获取更多信息。除此之外，Window Top-N 需要PARTITION BY子句 containswindow\_start和window\_end应用Windowing TVF或Window Aggregation的关系的列。否则，优化器将无法翻译查询。

下面显示了 Window Top-N 语句的语法：

|  |
| --- |
| SELECT [column\_list] FROM (  SELECT [column\_list],  ROW\_NUMBER() OVER (PARTITION BY window\_start, window\_end [, col\_key1...]  ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum  FROM table\_name) -- relation applied windowing TVF WHERE rownum <= N [AND conditions] |

###### 示例

以下示例显示了如何计算每滚动 10 分钟窗口的销售额最高的前 3 名供应商。

|  |
| --- |
| -- tables must have time attribute, e.g. `bidtime` in this table Flink SQL> desc Bid; +-------------+------------------------+------+-----+--------+---------------------------------+ | name | type | null | key | extras | watermark | +-------------+------------------------+------+-----+--------+---------------------------------+ | bidtime | TIMESTAMP(3) \*ROWTIME\* | true | | | `bidtime` - INTERVAL '1' SECOND | | price | DECIMAL(10, 2) | true | | | | | item | STRING | true | | | | | supplier\_id | STRING | true | | | | +-------------+------------------------+------+-----+--------+---------------------------------+  Flink SQL> SELECT \* FROM Bid; +------------------+-------+------+-------------+ | bidtime | price | item | supplier\_id | +------------------+-------+------+-------------+ | 2020-04-15 08:05 | 4.00 | A | supplier1 | | 2020-04-15 08:06 | 4.00 | C | supplier2 | | 2020-04-15 08:07 | 2.00 | G | supplier1 | | 2020-04-15 08:08 | 2.00 | B | supplier3 | | 2020-04-15 08:09 | 5.00 | D | supplier4 | | 2020-04-15 08:11 | 2.00 | B | supplier3 | | 2020-04-15 08:13 | 1.00 | E | supplier1 | | 2020-04-15 08:15 | 3.00 | H | supplier2 | | 2020-04-15 08:17 | 6.00 | F | supplier5 | +------------------+-------+------+-------------+  Flink SQL> SELECT \*  FROM (  SELECT \*, ROW\_NUMBER() OVER (PARTITION BY window\_start, window\_end ORDER BY price DESC) as rownum  FROM (  SELECT window\_start, window\_end, supplier\_id, SUM(price) as price, COUNT(\*) as cnt  FROM TABLE(  TUMBLE(TABLE Bid, DESCRIPTOR(bidtime), INTERVAL '10' MINUTES))  GROUP BY window\_start, window\_end, supplier\_id  )  ) WHERE rownum <= 3; +------------------+------------------+-------------+-------+-----+--------+ | window\_start | window\_end | supplier\_id | price | cnt | rownum | +------------------+------------------+-------------+-------+-----+--------+ | 2020-04-15 08:00 | 2020-04-15 08:10 | supplier1 | 6.00 | 2 | 1 | | 2020-04-15 08:00 | 2020-04-15 08:10 | supplier4 | 5.00 | 1 | 2 | | 2020-04-15 08:00 | 2020-04-15 08:10 | supplier2 | 4.00 | 1 | 3 | | 2020-04-15 08:10 | 2020-04-15 08:20 | supplier5 | 6.00 | 1 | 1 | | 2020-04-15 08:10 | 2020-04-15 08:20 | supplier2 | 3.00 | 1 | 2 | | 2020-04-15 08:10 | 2020-04-15 08:20 | supplier3 | 2.00 | 1 | 3 | +------------------+------------------+-------------+-------+-----+--------+ |

|  |
| --- |
| **注意：**为了更好地理解窗口化的行为，我们简化了时间戳值的显示以不显示尾随零，例如，如果类型为，则2020-04-15 08:05应该像2020-04-15 08:05:00.000在 Flink SQL Client 中那样显示TIMESTAMP(3)。 |

###### 限制

目前，Flink 只支持在Window Aggregation之后的**Window Top-N**。将在不久的将来支持Windowing TVF后的 Window Top-N 。

##### Deduplication（去重）

**应用场景：批或者流**

去重是指对在列的集合内重复的行进行删除，只保留第一行或最后一行数据。 在某些情况下，上游的 ETL 作业不能实现精确一次的端到端，这将可能导致在故障恢复 时，sink 中有重复的记录。 由于重复的记录将影响下游分析作业的正确性（例如，SUM、COUNT）， 所以在进一步分析之前需要进行数据去重。

与 Top-N 查询相似，Flink 使用 ROW\_NUMBER() 去除重复的记录。理论上来说，去重是一个特殊的 Top-N 查询，其中 N 是 1 ，记录则是以处理时间或事件事件进行排序的。

以下代码展示了去重语句的语法：

|  |
| --- |
| SELECT [column\_list] FROM (  SELECT [column\_list],  ROW\_NUMBER() OVER ([PARTITION BY col1[, col2...]]  ORDER BY time\_attr [asc|desc]) AS rownum  FROM table\_name) WHERE rownum = 1 |

参数规格：

* ROW\_NUMBER()：为每一行分配一个唯一的序列号，从一个开始。
* PARTITION BY col1[, col2...]：指定分区列，即去重键。
* ORDER BY time\_attr [asc|desc]: 指定排序列，必须是[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)。目前 Flink 支持[处理时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/" \l "processing-time)和[事件时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/" \l "event-time)。按 ASC 排序意味着保留第一行，按 DESC 排序意味着保留最后一行。
* WHERE rownum = 1：rownum = 1Flink 需要识别此查询是重复数据删除。

|  |
| --- |
| **注意：**必须严格遵循上述模式，否则优化器将无法翻译查询。 |

以下示例显示如何在流表上使用重复数据删除指定 SQL 查询。

|  |
| --- |
| CREATE TABLE Orders (  order\_time STRING,  user STRING,  product STRING,  num BIGINT,  proctime AS PROCTIME() ) WITH (...);  -- remove duplicate rows on order\_id and keep the first occurrence row, -- because there shouldn't be two orders with the same order\_id. SELECT order\_id, user, product, num FROM (  SELECT \*,  ROW\_NUMBER() OVER (PARTITION BY order\_id ORDER BY proctime ASC) AS row\_num  FROM Orders) WHERE row\_num = 1 |

##### Pattern Recognition（模式匹配）

**应用场景：流**

搜索一组事件模式（event pattern）是一种常见的用例，尤其是在数据流情景中。Flink 提供[复杂事件处理（CEP）库](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/libs/cep/)，该库允许在事件流中进行模式检测。此外，Flink 的 SQL API 提供了一种关系式的查询表达方式，其中包含大量内置函数和基于规则的优化，可以开箱即用。

2016 年 12 月，国际标准化组织（ISO）发布了新版本的 SQL 标准，其中包括在 SQL 中的行模式识别（Row Pattern Recognition in SQL）([ISO/IEC TR 19075-5:2016](https://standards.iso.org/ittf/PubliclyAvailableStandards/c065143_ISO_IEC_TR_19075-5_2016.zip))。它允许 Flink 使用 MATCH\_RECOGNIZE 子句融合 CEP 和 SQL API，以便在 SQL 中进行复杂事件处理。

MATCH\_RECOGNIZE 子句启用以下任务：

* 使用 PARTITION BY 和 ORDER BY 子句对数据进行逻辑分区和排序。
* 使用 PATTERN 子句定义要查找的行模式。这些模式使用类似于正则表达式的语法。
* 在 DEFINE 子句中指定行模式变量的逻辑组合。
* measures 是指在 MEASURES 子句中定义的表达式，这些表达式可用于 SQL 查询中的其他部分。

下面的示例演示了基本模式识别的语法：

|  |
| --- |
| SELECT T.aid, T.bid, T.cid FROM MyTable  MATCH\_RECOGNIZE (  PARTITION BY userid  ORDER BY proctime  MEASURES  A.id AS aid,  B.id AS bid,  C.id AS cid  PATTERN (A B C)  DEFINE  A AS name = 'a',  B AS name = 'b',  C AS name = 'c' ) AS T |

本页将更详细地解释每个关键字，并演示说明更复杂的示例。

|  |
| --- |
| **注意：** Flink 的 MATCH\_RECOGNIZE 子句实现是一个完整标准子集。仅支持以下部分中记录的功能。基于社区反馈，可能会支持其他功能，请查看[已知的局限](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "known-limitations)。 |

###### 介绍和示例

安装指南

模式识别特性使用 Apache Flink 内部的 CEP 库。为了能够使用 MATCH\_RECOGNIZE 子句，需要将库作为依赖项添加到 Maven 项目中。

|  |
| --- |
| <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-cep{{ site.scala\_version\_suffix }}</artifactId>  <version>{{ site.version }}</version> </dependency> |

或者，也可以将依赖项添加到集群的 classpath（查看 [dependency section](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/datastream/project-configuration/)获取更多相关依赖信息）。

如果你想在 SQL Client 中使用 MATCH\_RECOGNIZE 子句，你无需执行任何操作，因为默认情况下包含所有依赖项。

SQL 语义

每个 MATCH\_RECOGNIZE 查询都包含以下子句：

* PARTITION BY - 定义表的逻辑分区；类似于 GROUP BY 操作。
* ORDER BY - 指定传入行的排序方式；这是必须的，因为模式依赖于顺序。
* MEASURES - 定义子句的输出；类似于 SELECT 子句。
* ONE ROW PER MATCH - 输出方式，定义每个匹配项应产生多少行。
* AFTER MATCH SKIP - 指定下一个匹配的开始位置；这也是控制单个事件可以属于多少个不同匹配项的方法。
* PATTERN - 允许使用类似于 正则表达式 的语法构造搜索的模式。
* DEFINE - 本部分定义了模式变量必须满足的条件。

|  |
| --- |
| **注意：** 目前，MATCH\_RECOGNIZE 子句只能应用于[追加表](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/dynamic_tables/" \l "update-and-append-queries)。此外，它也总是生成一个追加表。 |

示例

对于我们的示例，我们假设已经注册了一个表 Ticker。该表包含特定时间点的股票价格。

这张表的 schema 如下：

|  |
| --- |
| Ticker  |-- symbol: String # 股票的代号  |-- price: Long # 股票的价格  |-- tax: Long # 股票应纳税额  |-- rowtime: TimeIndicatorTypeInfo(rowtime) # 更改这些值的时间点 |

为了简化，我们只考虑单个股票 ACME 的传入数据。Ticker 可以类似于下表，其中的行是连续追加的。

|  |
| --- |
| symbol rowtime price tax ====== ==================== ======= ======= 'ACME' '01-Apr-11 10:00:00' 12 1 'ACME' '01-Apr-11 10:00:01' 17 2 'ACME' '01-Apr-11 10:00:02' 19 1 'ACME' '01-Apr-11 10:00:03' 21 3 'ACME' '01-Apr-11 10:00:04' 25 2 'ACME' '01-Apr-11 10:00:05' 18 1 'ACME' '01-Apr-11 10:00:06' 15 1 'ACME' '01-Apr-11 10:00:07' 14 2 'ACME' '01-Apr-11 10:00:08' 24 2 'ACME' '01-Apr-11 10:00:09' 25 2 'ACME' '01-Apr-11 10:00:10' 19 1 |

现在的任务是找出一个单一股票价格不断下降的时期。为此，可以编写如下查询：

|  |
| --- |
| SELECT \* FROM Ticker  MATCH\_RECOGNIZE (  PARTITION BY symbol  ORDER BY rowtime  MEASURES  START\_ROW.rowtime AS start\_tstamp,  LAST(PRICE\_DOWN.rowtime) AS bottom\_tstamp,  LAST(PRICE\_UP.rowtime) AS end\_tstamp  ONE ROW PER MATCH  AFTER MATCH SKIP TO LAST PRICE\_UP  PATTERN (START\_ROW PRICE\_DOWN+ PRICE\_UP)  DEFINE  PRICE\_DOWN AS  (LAST(PRICE\_DOWN.price, 1) IS NULL AND PRICE\_DOWN.price < START\_ROW.price) OR  PRICE\_DOWN.price < LAST(PRICE\_DOWN.price, 1),  PRICE\_UP AS  PRICE\_UP.price > LAST(PRICE\_DOWN.price, 1)  ) MR; |

此查询将 Ticker 表按照 symbol 列进行分区并按照 rowtime 属性进行排序。

PATTERN 子句指定我们对以下模式感兴趣：该模式具有开始事件 START\_ROW，然后是一个或多个 PRICE\_DOWN 事件，并以 PRICE\_UP 事件结束。如果可以找到这样的模式，如 AFTER MATCH SKIP TO LAST 子句所示，则从最后一个 PRICE\_UP 事件开始寻找下一个模式匹配。

DEFINE 子句指定 **PRICE\_DOWN**和 **PRICE\_UP**事件需要满足的条件。尽管不存在 START\_ROW 模式变量，但它具有一个始终被评估为 TRUE 隐式条件。

模式变量 **PRICE\_DOWN**定义为价格小于满足 PRICE\_DOWN 条件的最后一行。对于初始情况或没有满足 PRICE\_DOWN 条件的最后一行时，该行的价格应小于该模式中前一行（由 START\_ROW 引用）的价格。

模式变量 **PRICE\_UP**定义为价格大于满足 PRICE\_DOWN 条件的最后一行。

此查询为股票价格持续下跌的每个期间生成摘要行。

在查询的 **MEASURES**子句部分定义确切的输出行信息。输出行数由 ONE ROW PER MATCH 输出方式定义。

|  |
| --- |
| symbol start\_tstamp bottom\_tstamp end\_tstamp ========= ======== ============== ================== ACME 01-APR-11 10:00:04 01-APR-11 10:00:07 01-APR-11 10:00:08 |

该行结果描述了从 01-APR-11 10:00:04 开始的价格下跌期，在 01-APR-11 10:00:07 达到最低价格，到 01-APR-11 10:00:08 再次上涨。

###### 分区

可以在分区数据中寻找模式，例如单个股票行情或特定用户的趋势。这可以用 PARTITION BY 子句来表示。该子句类似于对 aggregation 使用 GROUP BY。

|  |
| --- |
| **注意：** 强烈建议对传入的数据进行分区，否则 MATCH\_RECOGNIZE 子句将被转换为非并行算子，以确保全局排序。 |

###### 事件顺序

Apache Flink 可以根据时间（处理时间或者事件时间）进行模式搜索。

如果是事件时间，则在将事件传递到内部模式状态机之前对其进行排序。所以，无论行添加到表的顺序如何，生成的输出都是正确的。而模式是按照每行中所包含的时间指定顺序计算的。

MATCH\_RECOGNIZE 子句假定升序的 [时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/) 是 ORDER BY 子句的第一个参数。

对于示例 Ticker 表，诸如 ORDER BY rowtime ASC, price DESC 的定义是有效的，但 ORDER BY price, rowtime 或者 ORDER BY rowtime DESC, price ASC 是无效的。

###### Define & Measures

DEFINE 和 MEASURES 关键字与简单 SQL 查询中的 WHERE 和 SELECT 子句具有相近的含义。

**MEASURES**子句定义匹配模式的输出中要包含哪些内容。它可以投影列并定义表达式进行计算。产生的行数取决于[输出方式](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "output-mode)设置。

**DEFINE**子句指定行必须满足的条件才能被分类到相应的[模式变量](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "defining-a-pattern)。如果没有为模式变量定义条件，则将对每一行使用计算结果为 true 的默认条件。

有关在这些子句中可使用的表达式的更详细的说明，请查看[事件流导航](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "pattern-navigation)部分。

Aggregations

Aggregations 可以在 DEFINE 和 MEASURES 子句中使用。支持内置函数和用户自定义函数。

对相应匹配项的行子集可以使用 Aggregate functions。请查看[事件流导航](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "pattern-navigation)部分以了解如何计算这些子集。

下面这个示例的任务是找出股票平均价格没有低于某个阈值的最长时间段。它展示了 MATCH\_RECOGNIZE 在 aggregation 中的可表达性。可以使用以下查询执行此任务：

|  |
| --- |
| SELECT *\** FROM Ticker  MATCH\_RECOGNIZE (  PARTITION BY symbol  ORDER BY rowtime  MEASURES  FIRST(A.rowtime) AS start\_tstamp,  LAST(A.rowtime) AS end\_tstamp,  AVG(A.price) AS avgPrice  ONE ROW PER MATCH  AFTER MATCH SKIP PAST LAST ROW  PATTERN (A+ B)  DEFINE  A AS AVG(A.price) < 15  ) MR; |

给定此查询和以下输入值：

|  |
| --- |
| symbol rowtime price tax ====== ==================== ======= ======= 'ACME' '01-Apr-11 10:00:00' 12 1 'ACME' '01-Apr-11 10:00:01' 17 2 'ACME' '01-Apr-11 10:00:02' 13 1 'ACME' '01-Apr-11 10:00:03' 16 3 'ACME' '01-Apr-11 10:00:04' 25 2 'ACME' '01-Apr-11 10:00:05' 2 1 'ACME' '01-Apr-11 10:00:06' 4 1 'ACME' '01-Apr-11 10:00:07' 10 2 'ACME' '01-Apr-11 10:00:08' 15 2 'ACME' '01-Apr-11 10:00:09' 25 2 'ACME' '01-Apr-11 10:00:10' 25 1 'ACME' '01-Apr-11 10:00:11' 30 1 |

只要事件的平均价格不超过 15，查询就会将事件作为模式变量 A 的一部分进行累积。 例如，这种限制发生在 01-Apr-11 10：00：04。接下来的时间段在 01-Apr-11 10:00:11 再次超过平均价格 15。因此，所述查询的结果将是：

|  |
| --- |
| symbol start\_tstamp end\_tstamp avgPrice ====== ============= ============ ============ ACME 01-APR-11 10:00:00 01-APR-11 10:00:03 14.5 ACME 01-APR-11 10:00:05 01-APR-11 10:00:10 13.5 |

|  |
| --- |
| **注意：**   * Aggregation 可以应用于表达式，但前提是它们引用单个模式变量。因此，SUM(A.price \* A.tax) 是有效的，而 AVG(A.price \* B.tax) 则是无效的。 * 不支持 DISTINCT aggregation。 |

###### 定义模式

MATCH\_RECOGNIZE 子句允许用户在事件流中使用功能强大、表达力强的语法搜索模式，这种语法与广泛使用的正则表达式语法有些相似。

每个模式都是由基本的构建块构造的，称为 模式变量，可以应用算子（量词和其他修饰符）到这些模块中。整个模式必须用括号括起来。

示例模式如下所示：

|  |
| --- |
| PATTERN (A B+ C\* D) |

可以使用以下算子：

* Concatenation - 像 (A B) 这样的模式意味着 A 和 B 之间的连接是严格的。因此，在它们之间不能存在没有映射到 A 或 B 的行。
* Quantifiers - 修改可以映射到模式变量的行数。
  + \* — 0 或者多行
  + + — 1 或者多行
  + ? — 0 或者 1 行
  + { n } — 严格 n 行（n > 0）
  + { n, } — n 或者更多行（n ≥ 0）
  + { n, m } — 在 n 到 m（包含）行之间（0 ≤ n ≤ m，0 < m）
  + { , m } — 在 0 到 m（包含）行之间（m > 0）

|  |
| --- |
| **注意：**不支持可能产生空匹配的模式。此类模式的示例如 PATTERN (A\*)，PATTERN (A? B\*)，PATTERN (A{0,} B{0,} C\*) 等 |

贪婪量词和勉强量词

每一个量词可以是 贪婪（默认行为）的或者 勉强 的。贪婪的量词尝试匹配尽可能多的行，而勉强的量词则尝试匹配尽可能少的行。

为了说明区别，可以通过查询查看以下示例，其中贪婪量词应用于 B 变量：

|  |
| --- |
| SELECT *\** FROM Ticker  MATCH\_RECOGNIZE(  PARTITION BY symbol  ORDER BY rowtime  MEASURES  C.price AS lastPrice  ONE ROW PER MATCH  AFTER MATCH SKIP PAST LAST ROW  PATTERN (A B\* C)  DEFINE  A AS A.price > 10,  B AS B.price < 15,  C AS C.price > 12  ) |

假设我们有以下输入：

|  |
| --- |
| symbol tax price rowtime ======= ===== ======== =====================  XYZ 1 10 2018-09-17 10:00:02  XYZ 2 11 2018-09-17 10:00:03  XYZ 1 12 2018-09-17 10:00:04  XYZ 2 13 2018-09-17 10:00:05  XYZ 1 14 2018-09-17 10:00:06  XYZ 2 16 2018-09-17 10:00:07 |

上面的模式将产生以下输出：

|  |
| --- |
| symbol lastPrice ======== ===========  XYZ 16 |

将 B\* 修改为 B\*? 的同一查询，这意味着 B\* 应该是勉强的，将产生：

|  |
| --- |
| symbol lastPrice ======== ===========  XYZ 13  XYZ 16 |

模式变量 B 只匹配价格为 12 的行，而不是包含价格为 12、13 和 14 的行。

|  |
| --- |
| **注意：**模式的最后一个变量不能使用贪婪量词。因此，不允许使用类似 (A B\*) 的模式。通过引入条件为 B 的人工状态（例如 C），可以轻松解决此问题。因此，你可以使用类似以下的查询： |

|  |
| --- |
| PATTERN (A B\* C) DEFINE  A AS condA(),  B AS condB(),  C AS NOT condB() |

|  |
| --- |
| **注意：**目前不支持可选的勉强量词（A?? 或者 A{0,1}?）。 |

时间约束

特别是对于流的使用场景，通常需要在给定的时间内完成模式。这要求限制住 Flink 在内部必须保持的状态总体大小（即已经过期的状态就不需要再维护了），即使在贪婪的量词的情况下也是如此。

因此，Flink SQL 支持附加的（非标准 SQL）WITHIN 子句来定义模式的时间约束。子句可以在 PATTERN 子句之后定义，并以毫秒为间隔进行解析。

如果潜在匹配的第一个和最后一个事件之间的时间长于给定值，则不会将这种匹配追加到结果表中。

|  |
| --- |
| **注意：**通常鼓励使用 WITHIN 子句，因为它有助于 Flink 进行有效的内存管理。一旦达到阈值，即可修剪基础状态。然而，WITHIN 子句不是 SQL 标准的一部分。时间约束处理的方法已被提议将来可能会改变。 |

下面的示例查询说明了 WITHIN 子句的用法：

|  |
| --- |
| SELECT *\** FROM Ticker  MATCH\_RECOGNIZE(  PARTITION BY symbol  ORDER BY rowtime  MEASURES  C.rowtime AS dropTime,  A.price - C.price AS dropDiff  ONE ROW PER MATCH  AFTER MATCH SKIP PAST LAST ROW  PATTERN (A B\* C) WITHIN INTERVAL '1' HOUR  DEFINE  B AS B.price > A.price - 10  C AS C.price < A.price - 10  ) |

该查询检测到在 1 小时的间隔内价格下降了 10。

假设该查询用于分析以下股票数据：

|  |
| --- |
| symbol rowtime price tax ====== ==================== ======= ======= 'ACME' '01-Apr-11 10:00:00' 20 1 'ACME' '01-Apr-11 10:20:00' 17 2 'ACME' '01-Apr-11 10:40:00' 18 1 'ACME' '01-Apr-11 11:00:00' 11 3 'ACME' '01-Apr-11 11:20:00' 14 2 'ACME' '01-Apr-11 11:40:00' 9 1 'ACME' '01-Apr-11 12:00:00' 15 1 'ACME' '01-Apr-11 12:20:00' 14 2 'ACME' '01-Apr-11 12:40:00' 24 2 'ACME' '01-Apr-11 13:00:00' 1 2 'ACME' '01-Apr-11 13:20:00' 19 1 |

查询将生成以下结果：

|  |
| --- |
| symbol dropTime dropDiff ====== ==================== ============= 'ACME' '01-Apr-11 13:00:00' 14 |

结果行代表价格从 15（在01-Apr-11 12:00:00）下降到 1（在01-Apr-11 13:00:00）。dropDiff 列包含价格差异。

请注意，即使价格也下降了较高的值，例如，下降了 11（在 01-Apr-11 10:00:00 和 01-Apr-11 11:40:00 之间），这两个事件之间的时间差大于 1 小时。因此，它们不会产生匹配。

###### 输出方式

输出方式 描述每个找到的匹配项应该输出多少行。SQL 标准描述了两种方式：

* ALL ROWS PER MATCH
* ONE ROW PER MATCH

目前，唯一支持的输出方式是 ONE ROW PER MATCH，它将始终为每个找到的匹配项生成一个输出摘要行。

输出行的 schema 将是按特定顺序连接 [partitioning columns] + [measures columns]。

以下示例显示了所定义的查询的输出：

|  |
| --- |
| SELECT *\** FROM Ticker  MATCH\_RECOGNIZE(  PARTITION BY symbol  ORDER BY rowtime  MEASURES  FIRST(A.price) AS startPrice,  LAST(A.price) AS topPrice,  B.price AS lastPrice  ONE ROW PER MATCH  PATTERN (A+ B)  DEFINE  A AS LAST(A.price, 1) IS NULL OR A.price > LAST(A.price, 1),  B AS B.price < LAST(A.price)  ) |

对于以下输入行：

|  |
| --- |
| symbol tax price rowtime ======== ===== ======== =====================  XYZ 1 10 2018-09-17 10:00:02  XYZ 2 12 2018-09-17 10:00:03  XYZ 1 13 2018-09-17 10:00:04  XYZ 2 11 2018-09-17 10:00:05 |

该查询将生成以下输出：

|  |
| --- |
| symbol startPrice topPrice lastPrice ======== ============ ========== ===========  XYZ 10 13 11 |

该模式识别由 symbol 列分区。即使在 MEASURES 子句中未明确提及，分区列仍会添加到结果的开头。

###### 模式导航

DEFINE 和 MEASURES 子句允许在（可能）匹配模式的行列表中进行导航。

本节讨论用于声明条件或产生输出结果的导航。

引用模式变量

引用模式变量 允许引用一组映射到 DEFINE 或 MEASURES 子句中特定模式变量的行。

例如，如果我们尝试将当前行与 A 进行匹配，则表达式 A.price 描述了目前为止已映射到 A 的一组行加上当前行。如果 DEFINE/MEASURES 子句中的表达式需要一行（例如 a.price 或 a.price > 10），它将选择属于相应集合的最后一个值。

如果没有指定模式变量（例如 SUM(price)），则表达式引用默认模式变量 \*，该变量引用模式中的所有变量。换句话说，它创建了一个列表，其中列出了迄今为止映射到任何变量的所有行以及当前行。

**示例**

对于更全面的示例，可以查看以下模式和相应的条件：

|  |
| --- |
| PATTERN (A B+) DEFINE  A AS A.price >= 10,  B AS B.price > A.price AND SUM(price) < 100 AND SUM(B.price) < 80 |

下表描述了如何为每个传入事件计算这些条件。

该表由以下列组成：

* # - 行标识符，用于唯一标识列表中的传入行 [A.price]/[B.price]/[price]。
* price - 传入行的价格。
* [A.price]/[B.price]/[price] - 描述 DEFINE 子句中用于计算条件的行列表。
* Classifier - 当前行的分类器，指示该行映射到的模式变量。
* A.price/B.price/SUM(price)/SUM(B.price) - 描述了这些表达式求值后的结果。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **#** | **price** | **Classifier** | **[A.price]** | **[B.price]** | **[price]** | **A.price** | **B.price** | **SUM(price)** | **SUM(B.price)** |
| #1 | 10 | -> A | #1 | - | - | 10 | - | - | - |
| #2 | 15 | -> B | #1 | #2 | #1, #2 | 10 | 15 | 25 | 15 |
| #3 | 20 | -> B | #1 | #2, #3 | #1, #2, #3 | 10 | 20 | 45 | 35 |
| #4 | 31 | -> B | #1 | #2, #3, #4 | #1, #2, #3,#4 | 10 | 31 | 76 | 66 |
| #5 | 35 |  | #1 | #2, #3, #4, #5 | #1, #2, #3, #4, #5 | 10 | 35 | 111 | 101 |

从表中可以看出，第一行映射到模式变量 A，随后的行映射到模式变量 B。但是，最后一行不满足 B 条件，因为所有映射行 SUM(price) 的总和与 B 中所有行的总和都超过了指定的阈值。

Logical Offsets

Logical offsets 在映射到指定模式变量的事件启用导航。这可以用两个相应的函数表示：

|  |  |
| --- | --- |
| **Offset functions** | **描述** |
| ```text LAST(variable.field, n) ``` | 返回映射到变量最后 n 个元素的事件中的字段值。计数从映射的最后一个元素开始。 |
| ```text FIRST(variable.field, n) ``` | 返回映射到变量的第 n 个元素的事件中的字段值。计数从映射的第一个元素开始。 |

**示例**

对于更全面的示例，可以参考以下模式和相应的条件：

|  |
| --- |
| PATTERN (A B+) DEFINE  A AS A.price >= 10,  B AS (LAST(B.price, 1) IS NULL OR B.price > LAST(B.price, 1)) AND  (LAST(B.price, 2) IS NULL OR B.price > 2 \* LAST(B.price, 2)) |

下表描述了如何为每个传入事件计算这些条件。

该表包括以下列：

* price - 传入行的价格。
* Classifier - 当前行的分类器，指示该行映射到的模式变量。
* LAST(B.price, 1)/LAST(B.price, 2) - 描述对这些表达式求值后的结果。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **price** | **Classifier** | **LAST(B.price, 1)** | **LAST(B.price, 2)** | **Comment** |
| 10 | -> A |  |  |  |
| 15 | -> B | null | null | 注意 LAST(B.price, 1) 为空，因为仍然没有映射到 B。 |
| 20 | -> B | 15 | null |  |
| 31 | -> B | 20 | 15 |  |
| 35 |  | 31 | 20 | 因为 35 < 2 \* 20 没有映射。 |

将默认模式变量与 logical offsets 一起使用也可能很有意义。

在这种情况下，offset 会包含到目前为止映射的所有行：

|  |  |  |  |
| --- | --- | --- | --- |
| PATTERN (A B? C) DEFINE  B AS B.price < 20,  C AS LAST(price, 1) < C.price | | | |
| **price** | **Classifier** | **LAST(B.price, 1)** | **Comment** |
| 10 | ->A |  |  |
| 15 | -> B |  |  |
| 20 | -> C | 15 | LAST(price, 1) 被计算为映射到 B 变量的行的价格 |

如果第二行没有映射到 B 变量，则会得到以下结果：

|  |  |  |  |
| --- | --- | --- | --- |
| **price** | **Classifier** | **LAST(B.price, 1)** | **Comment** |
| 10 | ->A |  |  |
| 20 | -> C | 10 | LAST(price, 1) 被计算为映射到 A 变量的行的价格 |

也可以在 FIRST/LAST 函数的第一个参数中使用多个模式变量引用。这样，可以编写访问多个列的表达式。但是，它们都必须使用相同的模式变量。换句话说，必须在一行中计算 LAST/FIRST 函数的值。

因此，可以使用 LAST(A.price \* A.tax)，但不允许使用类似 LAST(A.price \* B.tax) 的表达式。

###### 匹配后的策略

AFTER MATCH SKIP 子句指定在找到完全匹配后从何处开始新的匹配过程。

有四种不同的策略：

* SKIP PAST LAST ROW - 在当前匹配的最后一行之后的下一行继续模式匹配。
* SKIP TO NEXT ROW - 继续从匹配项开始行后的下一行开始搜索新匹配项。
* SKIP TO LAST variable - 恢复映射到指定模式变量的最后一行的模式匹配。
* SKIP TO FIRST variable - 在映射到指定模式变量的第一行继续模式匹配。

这也是一种指定单个事件可以属于多少个匹配项的方法。例如，使用 SKIP PAST LAST ROW 策略，每个事件最多只能属于一个匹配项。

**示例**

为了更好地理解这些策略之间的差异，我们可以看看下面的例子。

对于以下输入行：

|  |
| --- |
| symbol tax price rowtime ======== ===== ======= =====================  XYZ 1 7 2018-09-17 10:00:01  XYZ 2 9 2018-09-17 10:00:02  XYZ 1 10 2018-09-17 10:00:03  XYZ 2 5 2018-09-17 10:00:04  XYZ 2 10 2018-09-17 10:00:05  XYZ 2 7 2018-09-17 10:00:06  XYZ 2 14 2018-09-17 10:00:07 |

我们使用不同的策略评估以下查询：

|  |
| --- |
| SELECT *\** FROM Ticker  MATCH\_RECOGNIZE(  PARTITION BY symbol  ORDER BY rowtime  MEASURES  SUM(A.price) AS sumPrice,  FIRST(rowtime) AS startTime,  LAST(rowtime) AS endTime  ONE ROW PER MATCH  [AFTER MATCH STRATEGY]  PATTERN (A+ C)  DEFINE  A AS SUM(A.price) < 30  ) |

该查询返回映射到 A 的总体匹配的第一个和最后一个时间戳所有行的价格之和。

查询将根据使用的 AFTER MATCH 策略产生不同的结果：

**AFTER MATCH SKIP PAST LAST ROW**

|  |
| --- |
| symbol sumPrice startTime endTime ====== ======== ================ ==================  XYZ 26 2018-09-17 10:00:01 2018-09-17 10:00:04  XYZ 17 2018-09-17 10:00:05 2018-09-17 10:00:07 |

第一个结果与 #1，#2，#3，#4 行匹配。

第二个结果与 #5，#6, #7 行匹配。

**AFTER MATCH SKIP TO NEXT ROW [#](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "after-match-skip-to-next-row)**

|  |
| --- |
| symbol sumPrice startTime endTime ==== ========= ================ ===================  XYZ 26 2018-09-17 10:00:01 2018-09-17 10:00:04  XYZ 24 2018-09-17 10:00:02 2018-09-17 10:00:05  XYZ 25 2018-09-17 10:00:03 2018-09-17 10:00:06  XYZ 22 2018-09-17 10:00:04 2018-09-17 10:00:07  XYZ 17 2018-09-17 10:00:05 2018-09-17 10:00:07 |

同样，第一个结果与 #1，#2，#3，#4 行匹配。

与上一个策略相比，下一个匹配再次包含 #2 行匹配。因此，第二个结果与 #2，#3，#4，#5 行匹配。

第三个结果与 #3，#4，#5, #6 行匹配。

第四个结果与 #4，#5，#6, #7 行匹配。

最后一个结果与 #5，#6, #7 行匹配。

**AFTER MATCH SKIP TO LAST A [#](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "after-match-skip-to-last-a)**

|  |
| --- |
| symbol sumPrice startTime endTime ====== ======== ============= =====================  XYZ 26 2018-09-17 10:00:01 2018-09-17 10:00:04  XYZ 25 2018-09-17 10:00:03 2018-09-17 10:00:06  XYZ 17 2018-09-17 10:00:05 2018-09-17 10:00:07 |

同样，第一个结果与 #1，#2，#3，#4 行匹配。

与前一个策略相比，下一个匹配只包含 #3 行（对应 A）用于下一个匹配。因此，第二个结果与 #3，#4，#5, #6 行匹配。

最后一个结果与 #5，#6, #7 行匹配。

**AFTER MATCH SKIP TO FIRST A [#](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "after-match-skip-to-first-a)**

这种组合将产生一个运行时异常，因为人们总是试图在上一个开始的地方开始一个新的匹配。这将产生一个无限循环，因此是禁止的。

必须记住，在 SKIP TO FIRST/LAST variable 策略的场景下，可能没有映射到该变量的行（例如，对于模式 A\*）。在这种情况下，将抛出一个运行时异常，因为标准要求一个有效的行来继续匹配。

###### 时间属性

为了在 MATCH\_RECOGNIZE 之上应用一些后续查询，可能需要使用[时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)。有两个函数可供选择：

|  |  |
| --- | --- |
| **functions** | **描述** |
| MATCH\_ROWTIME() | 返回映射到给定模式的最后一行的时间戳。  结果属性是[行时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/)，可用于后续基于时间的操作，例如 [interval joins](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "interval-joins)和 [group window or over window aggregations](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "aggregations)。 |
| MATCH\_PROCTIME() | 返回[处理时间属性](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/concepts/time_attributes/" \l "processing-time)，该属性可用于随后的基于时间的操作，例如 [interval joins](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/joins/" \l "interval-joins)和 [group window or over window aggregations](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "aggregations)。 |

###### 控制内存消耗

在编写 MATCH\_RECOGNIZE 查询时，内存消耗是一个重要的考虑因素，因为潜在匹配的空间是以宽度优先的方式构建的。鉴于此，我们必须确保模式能够完成。最好使用映射到匹配项的合理数量的行，因为它们必须内存相适。

例如，该模式不能有没有接受每一行上限的量词。这种模式可以是这样的：

|  |
| --- |
| PATTERN (A B+ C) DEFINE  A as A.price > 10,  C as C.price > 20 |

查询将每个传入行映射到 B 变量，因此永远不会完成。可以纠正此查询，例如，通过否定 C 的条件：

|  |
| --- |
| PATTERN (A B+ C) DEFINE  A as A.price > 10,  B as B.price <= 20,  C as C.price > 20 |

或者使用 贪婪量词和勉强量词：

|  |
| --- |
| PATTERN (A B+? C) DEFINE  A as A.price > 10,  C as C.price > 20 |

|  |
| --- |
| **注意：**MATCH\_RECOGNIZE 子句未使用配置的 [state retention time](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/config/" \l "idle-state-retention-time)。为此，可能需要使用 WITHIN [子句](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "time-constraint)。 |

###### 已知的局限

Flink 对 MATCH\_RECOGNIZE 子句实现是一项长期持续的工作，目前尚不支持 SQL 标准的某些功能。

不支持的功能包括：

* 模式表达式：
  + Pattern groups - 这意味着量词不能应用于模式的子序列。因此，(A (B C)+) 不是有效的模式。
  + Alterations - 像 PATTERN((A B | C D) E)这样的模式，这意味着在寻找 E 行之前必须先找到子序列 A B 或者 C D。
  + PERMUTE operator - 这等同于它应用于所示的所有变量的排列 PATTERN (PERMUTE (A, B, C)) = PATTERN (A B C | A C B | B A C | B C A | C A B | C B A)。
  + Anchors - ^, $，表示分区的开始/结束，在流上下文中没有意义，将不被支持。
  + Exclusion - PATTERN ({- A -} B) 表示将查找 A，但是不会参与输出。这只适用于 ALL ROWS PER MATCH 方式。
  + Reluctant optional quantifier - PATTERN A?? 只支持贪婪的可选量词。
* ALL ROWS PER MATCH 输出方式 - 为参与创建匹配项的每一行产生一个输出行。这也意味着：
  + MEASURES 子句唯一支持的语义是 FINAL
  + CLASSIFIER 函数，尚不支持返回行映射到的模式变量。
* SUBSET - 它允许创建模式变量的逻辑组，并在 DEFINE 和 MEASURES 子句中使用这些组。
* Physical offsets - PREV/NEXT，它为所有可见事件建立索引，而不是仅将那些映射到模式变量的事件编入索引（如 [logical offsets](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sql/queries/match_recognize/" \l "logical-offsets) 的情况）。
* 提取时间属性 - 目前无法为后续基于时间的操作提取时间属性。
* MATCH\_RECOGNIZE 仅 SQL 支持。Table API 中没有等效项。
* Aggregations:
  + 不支持 distinct aggregations。

### CREATE 语句

CREATE 语句用于向当前或指定的 [Catalog](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/catalogs/) 中注册表、视图或函数。注册后的表、视图和函数可以在 SQL 查询中使用。

目前 Flink SQL 支持下列 CREATE 语句：

* CREATE TABLE
* CREATE DATABASE
* CREATE VIEW
* CREATE FUNCTION

#### 执行 CREATE 语句

可以使用 TableEnvironment 中的 executeSql() 方法执行 CREATE 语句。 若 CREATE 操作执行成功，executeSql() 方法返回 ‘OK’，否则会抛出异常。

以下的例子展示了如何在 TableEnvironment 中执行一个 CREATE 语句。

|  |
| --- |
| EnvironmentSettings settings = EnvironmentSettings.newInstance()... TableEnvironment tableEnv = TableEnvironment.create(settings);  // 对已注册的表进行 SQL 查询 // 注册名为 “Orders” 的表 tableEnv.executeSql("CREATE TABLE Orders (`user` BIGINT, product STRING, amount INT) WITH (...)"); // 在表上执行 SQL 查询，并把得到的结果作为一个新的表 Table result = tableEnv.sqlQuery( "SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'");  // 对已注册的表进行 INSERT 操作 // 注册 TableSink tableEnv.executeSql("CREATE TABLE RubberOrders(product STRING, amount INT) WITH (...)"); // 在表上执行 INSERT 语句并向 TableSink 发出结果 tableEnv.executeSql( "INSERT INTO RubberOrders SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'"); |

#### CREATE TABLE

|  |
| --- |
| CREATE TABLE [IF NOT EXISTS] [catalog\_name.][db\_name.]table\_name  (  { <physical\_column\_definition> | <metadata\_column\_definition> | <computed\_column\_definition> }[ , ...n]  [ <watermark\_definition> ]  [ <table\_constraint> ][ , ...n]  )  [COMMENT table\_comment]  [PARTITIONED BY (partition\_column\_name1, partition\_column\_name2, ...)]  WITH (key1=val1, key2=val2, ...)  [ LIKE source\_table [( <like\_options> )] ]   <physical\_column\_definition>:  column\_name column\_type [ <column\_constraint> ] [COMMENT column\_comment]   <column\_constraint>:  [CONSTRAINT constraint\_name] PRIMARY KEY NOT ENFORCED  <table\_constraint>:  [CONSTRAINT constraint\_name] PRIMARY KEY (column\_name, ...) NOT ENFORCED  <metadata\_column\_definition>:  column\_name column\_type METADATA [ FROM metadata\_key ] [ VIRTUAL ]  <computed\_column\_definition>:  column\_name AS computed\_column\_expression [COMMENT column\_comment]  <watermark\_definition>:  WATERMARK FOR rowtime\_column\_name AS watermark\_strategy\_expression  <source\_table>:  [catalog\_name.][db\_name.]table\_name  <like\_options>: {  { INCLUDING | EXCLUDING } { ALL | CONSTRAINTS | PARTITIONS }  | { INCLUDING | EXCLUDING | OVERWRITING } { GENERATED | OPTIONS | WATERMARKS }  }[, ...] |

根据指定的表名创建一个表，如果同名表已经在 catalog 中存在了，则无法注册。

##### Columns

###### Physical / Regular Columns

物理列是数据库中已知的常规列。它们定义物理数据中字段的名称、类型和顺序。可以在物理列之间声明其他类型的列，但不会影响最终的物理模式。

以下语句创建一个仅包含常规列的表：

|  |
| --- |
| CREATE TABLE MyTable (  `user\_id` BIGINT,  `name` STRING ) WITH (  ... ); |

###### Metadata Columns

元数据列是 SQL 标准的扩展，允许访问连接器和/或表格每一行的格式特定字段。元数据列由METADATA关键字指示。例如，元数据列可用于读取和写入 Kafka 记录的时间戳，以进行基于时间的操作。该[连接器和格式的文档](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/overview/)列出了每一个组件中的可用元数据字段。但是，在表的架构中声明元数据列是可选的。

以下语句创建一个表，其中包含一个引用元数据字段的附加元数据列timestamp：

|  |
| --- |
| CREATE TABLE MyTable (  `user\_id` BIGINT,  `name` STRING,  `record\_time` TIMESTAMP\_LTZ(3) METADATA FROM 'timestamp' -- reads and writes a Kafka record's timestamp ) WITH (  'connector' = 'kafka'  ... ); |

每个元数据字段都由基于字符串的键标识，并具有记录的数据类型。例如，Kafka 连接器公开了一个具有键timestamp和数据类型的元数据字段TIMESTAMP\_LTZ(3) ，可用于读取和写入记录。

在上面的示例中，元数据列record\_time成为表架构的一部分，可以像常规列一样进行转换和存储：

|  |
| --- |
| INSERT INTO MyTable SELECT user\_id, name, record\_time + INTERVAL '1' SECOND FROM MyTable; |

为方便起见，FROM如果应将列名用作标识元数据键，则可以省略该子句：

|  |
| --- |
| CREATE TABLE MyTable (  `user\_id` BIGINT,  `name` STRING,  `timestamp` TIMESTAMP\_LTZ(3) METADATA *-- use column name as metadata key* ) WITH (  'connector' = 'kafka'  ... ); |

为方便起见，如果列的数据类型与元数据字段的数据类型不同，运行时将执行显式转换。当然，这需要两种数据类型兼容。

|  |
| --- |
| CREATE TABLE MyTable (  `user\_id` BIGINT,  `name` STRING,  `timestamp` BIGINT METADATA *-- cast the timestamp as BIGINT* ) WITH (  'connector' = 'kafka'  ... ); |

默认情况下，规划器假定元数据列可用于读取和写入。但是，在许多情况下，外部系统提供的只读元数据字段多于可写字段。因此，可以使用VIRTUAL关键字从持久化中排除元数据列。

|  |
| --- |
| CREATE TABLE MyTable (  `timestamp` BIGINT METADATA, -- part of the query-to-sink schema  `offset` BIGINT METADATA VIRTUAL, -- not part of the query-to-sink schema  `user\_id` BIGINT,  `name` STRING, ) WITH (  'connector' = 'kafka'  ... ); |

在上面的示例中，offset是只读元数据列，并从查询到接收器模式中排除。因此，源到查询模式（for SELECT）和查询到接收器（for INSERT INTO）模式不同：

|  |
| --- |
| source-to-query schema: MyTable(`timestamp` BIGINT, `offset` BIGINT, `user\_id` BIGINT, `name` STRING)  query-to-sink schema: MyTable(`timestamp` BIGINT, `user\_id` BIGINT, `name` STRING) |

###### Computed Columns

计算列是使用语法生成的虚拟列column\_name AS computed\_column\_expression。

计算列计算可以引用同一表中声明的其他列的表达式。可以访问物理列和元数据列。列本身并不物理存储在表中。列的数据类型是从给定的表达式自动派生的，不必手动声明。

规划器将计算列转换为源后的常规投影。对于优化或[水印策略下推](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/sourcessinks/)，评估可能会分布在运算符之间，执行多次，或者如果给定查询不需要则跳过。

例如，计算列可以定义为：

|  |
| --- |
| CREATE TABLE MyTable (  `user\_id` BIGINT,  `price` DOUBLE,  `quantity` DOUBLE,  `cost` AS price \* quanitity, -- evaluate expression and supply the result to queries ) WITH (  'connector' = 'kafka'  ... ); |

表达式可以包含列、常量或函数的任意组合。表达式不能包含子查询。

计算列在flink通常用于定义时间属性 的CREATE TABLE语句。

* 一个处理时间属性可以很容易地通过被定义proc AS PROCTIME()使用系统的PROCTIME()功能。
* 一个事件时间属性时间戳可以是之前预处理WATERMARK声明。例如，如果原始字段不是TIMESTAMP(3)类型或嵌套在 JSON 字符串中，则可以使用计算列。

与虚拟元数据列类似，计算列被排除在持久化之外。因此，计算列不能作为INSERT INTO语句的目标。因此，源到查询模式（for SELECT）和查询到接收器（for INSERT INTO）模式不同：

|  |
| --- |
| source-to-query schema: MyTable(`user\_id` BIGINT, `price` DOUBLE, `quantity` DOUBLE, `cost` DOUBLE)  query-to-sink schema: MyTable(`user\_id` BIGINT, `price` DOUBLE, `quantity` DOUBLE) |

##### WATERMARK

WATERMARK 定义了表的事件时间属性，其形式为 WATERMARK FOR rowtime\_column\_name AS watermark\_strategy\_expression 。

rowtime\_column\_name 把一个现有的列定义为一个为表标记事件时间的属性。该列的类型必须为 TIMESTAMP(3)，且是 schema 中的顶层列，它也可以是一个计算列。

watermark\_strategy\_expression 定义了 watermark 的生成策略。它允许使用包括计算列在内的任意非查询表达式来计算 watermark ；表达式的返回类型必须是 TIMESTAMP(3)，表示了从 Epoch 以来的经过的时间。 返回的 watermark 只有当其不为空且其值大于之前发出的本地 watermark 时才会被发出（以保证 watermark 递增）。每条记录的 watermark 生成表达式计算都会由框架完成。 框架会定期发出所生成的最大的 watermark ，如果当前 watermark 仍然与前一个 watermark 相同、为空、或返回的 watermark 的值小于最后一个发出的 watermark ，则新的 watermark 不会被发出。 Watermark 根据 [pipeline.auto-watermark-interval](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/deployment/config/" \l "pipeline-auto-watermark-interval) 中所配置的间隔发出。 若 watermark 的间隔是 0ms ，那么每条记录都会产生一个 watermark，且 watermark 会在不为空并大于上一个发出的 watermark 时发出。

使用事件时间语义时，表必须包含事件时间属性和 watermark 策略。

Flink 提供了几种常用的 watermark 策略。

* 严格递增时间戳： WATERMARK FOR rowtime\_column AS rowtime\_column。

发出到目前为止已观察到的最大时间戳的 watermark ，时间戳大于最大时间戳的行被认为没有迟到。

* 递增时间戳： WATERMARK FOR rowtime\_column AS rowtime\_column - INTERVAL '0.001' SECOND。

发出到目前为止已观察到的最大时间戳减 1 的 watermark ，时间戳大于或等于最大时间戳的行被认为没有迟到。

* 有界乱序时间戳： WATERMARK FOR rowtime\_column AS rowtime\_column - INTERVAL 'string' timeUnit。

发出到目前为止已观察到的最大时间戳减去指定延迟的 watermark ，例如， WATERMARK FOR rowtime\_column AS rowtime\_column - INTERVAL '5' SECOND 是一个 5 秒延迟的 watermark 策略。

|  |
| --- |
| CREATE TABLE Orders (  `user` BIGINT,  product STRING,  order\_time TIMESTAMP(3),  WATERMARK FOR order\_time AS order\_time - INTERVAL '5' SECOND ) WITH ( . . . ); |

##### PRIMARY KEY

主键作为Flink 优化的一种提示信息。主键限制一张表或视图的某个（些）列是唯一的并且不包含 Null 值。 主键声明的列都是非 nullable 的。因此主键可以被用作表行级别的唯一标识。

主键可以和列的定义一起声明，也可以独立声明为表的限制属性，不管是哪种方式，主键都不可以重复定义，否则 Flink 会报错。

**有效性检查**

SQL 标准主键限制可以有两种模式：ENFORCED 或者 NOT ENFORCED。 它申明了是否输入/出数据会做合法性检查（是否唯一）。Flink 不存储数据因此只支持 NOT ENFORCED 模式，即不做检查，用户需要自己保证唯一性。

Flink 假设声明了主键的列都是不包含 Null 值的，Connector 在处理数据时需要自己保证语义正确。

|  |
| --- |
| **注意：**在 CREATE TABLE 语句中，创建主键会修改列的 nullable 属性，主键声明的列默认都是非 Nullable 的。 |

##### PARTITIONED BY

根据指定的列对已经创建的表进行分区。若表使用 filesystem sink ，则将会为每个分区创建一个目录。

##### WITH选项

表属性用于创建 table source/sink ，一般用于寻找和创建底层的连接器。

表达式 key1=val1 的键和值必须为字符串文本常量。请参考 [连接外部系统](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/connectors/table/overview/)了解不同连接器所支持的属性。

|  |
| --- |
| **注意：**在 CREATE TABLE 语句中，创建主键会修改列的 nullable 属性，主键声明的列默认都是非 Nullable 的。  表名可以为以下三种格式：   * catalog\_name.db\_name.table\_name * db\_name.table\_name * table\_name   使用catalog\_name.db\_name.table\_name 的表将会与名为 “catalog\_name” 的 catalog 和名为 “db\_name” 的数据库一起注册到 metastore 中。使用 db\_name.table\_name 的表将会被注册到当前执行的 table environment 中的 catalog 且数据库会被命名为 “db\_name”；对于 table\_name, 数据表将会被注册到当前正在运行的catalog和数据库中。  使用 CREATE TABLE 语句注册的表均可用作 table source 和 table sink。 在被 DML 语句引用前，我们无法决定其实际用于 source 或 sink。 |

##### LIKE

LIKE 子句来源于两种 SQL 特性的变体/组合（Feature T171，“表定义中的 LIKE 语法” 和 Feature T173，“表定义中的 LIKE 语法扩展”）。LIKE 子句可以基于现有表的定义去创建新表，并且可以扩展或排除原始表中的某些部分。与 SQL 标准相反，LIKE 子句必须在 CREATE 语句中定义，并且是基于 CREATE 语句的更上层定义，这是因为 LIKE 子句可以用于定义表的多个部分，而不仅仅是 schema 部分。

你可以使用该子句，重用（或改写）指定的连接器配置属性或者可以向外部表添加 watermark 定义，例如可以向 Apache Hive 中定义的表添加 watermark 定义。

示例如下：

|  |
| --- |
| CREATE TABLE Orders (  `user` BIGINT,  product STRING,  order\_time TIMESTAMP(3) ) WITH (   'connector' = 'kafka',  'scan.startup.mode' = 'earliest-offset' );  CREATE TABLE Orders\_with\_watermark (  -- 添加 watermark 定义  WATERMARK FOR order\_time AS order\_time - INTERVAL '5' SECOND  ) WITH (  -- 改写 startup-mode 属性  'scan.startup.mode' = 'latest-offset' ) LIKE Orders; |

结果表 Orders\_with\_watermark 等效于使用以下语句创建的表：

|  |
| --- |
| CREATE TABLE Orders\_with\_watermark (  `user` BIGINT,  product STRING,  order\_time TIMESTAMP(3),  WATERMARK FOR order\_time AS order\_time - INTERVAL '5' SECOND  ) WITH (  'connector' = 'kafka',  'scan.startup.mode' = 'latest-offset' ); |

表属性的合并逻辑可以用 like options 来控制。

可以控制合并的表属性如下：

* CONSTRAINTS - 主键和唯一键约束
* GENERATED - 计算列
* OPTIONS - 连接器信息、格式化方式等配置项
* PARTITIONS - 表分区信息
* WATERMARKS - watermark 定义

并且有三种不同的表属性合并策略：

* INCLUDING - 新表包含源表（source table）所有的表属性，如果和源表的表属性重复则会直接失败，例如新表和源表存在相同 key 的属性。
* EXCLUDING - 新表不包含源表指定的任何表属性。
* OVERWRITING - 新表包含源表的表属性，但如果出现重复项，则会用新表的表属性覆盖源表中的重复表属性，例如，两个表中都存在相同 key 的属性，则会使用当前语句中定义的 key 的属性值。

并且你可以使用 INCLUDING/EXCLUDING ALL 这种声明方式来指定使用怎样的合并策略，例如使用 EXCLUDING ALL INCLUDING WATERMARKS，那么代表只有源表的 WATERMARKS 属性才会被包含进新表。

示例如下：

|  |
| --- |
| -- 存储在文件系统的源表 CREATE TABLE Orders\_in\_file (  `user` BIGINT,  product STRING,  order\_time\_string STRING,  order\_time AS to\_timestamp(order\_time)   ) PARTITIONED BY (`user`)  WITH (   'connector' = 'filesystem',  'path' = '...' );  -- 对应存储在 kafka 的源表 CREATE TABLE Orders\_in\_kafka (  -- 添加 watermark 定义  WATERMARK FOR order\_time AS order\_time - INTERVAL '5' SECOND  ) WITH (  'connector' = 'kafka',  ... ) LIKE Orders\_in\_file (  -- 排除需要生成 watermark 的计算列之外的所有内容。  -- 去除不适用于 kafka 的所有分区和文件系统的相关属性。  EXCLUDING ALL  INCLUDING GENERATED ); |

如果未提供 like 配置项（like options），默认将使用 INCLUDING ALL OVERWRITING OPTIONS 的合并策略。

|  |
| --- |
| **注意：**   * 您无法选择物理列的合并策略，当物理列进行合并时就如使用了 INCLUDING 策略。 * 源表 source\_table 可以是一个组合 ID。您可以指定不同 catalog 或者 DB 的表作为源表: 例如，my\_catalog.my\_db.MyTable 指定了源表 MyTable 来源于名为 MyCatalog 的 catalog 和名为 my\_db 的 DB ，my\_db.MyTable 指定了源表 MyTable 来源于当前 catalog 和名为 my\_db 的 DB。 |

#### CREATE CATALOG

|  |
| --- |
| CREATE CATALOG catalog\_name  WITH (key1=val1, key2=val2, ...) |

创建具有给定目录属性的目录。如果已经存在同名目录，则抛出异常。

有选项

数据库属性一般用于存储关于这个数据库额外的信息。 表达式 key1=val1 中的键和值都需要是字符串文本常量。

#### CREATE VIEW

|  |
| --- |
| CREATE [TEMPORARY] VIEW [IF NOT EXISTS] [catalog\_name.][db\_name.]view\_name  [{columnName [, columnName ]\* }] [COMMENT view\_comment]  AS query\_expression |

根据给定的 query 语句创建一个视图。若数据库中已经存在同名视图会抛出异常.

**TEMPORARY**

创建一个有 catalog 和数据库命名空间的临时视图，并覆盖原有的视图。

**IF NOT EXISTS**

若该视图已经存在，则不会进行任何操作。

#### CREATE FUNCTION

|  |
| --- |
| CREATE [TEMPORARY|TEMPORARY SYSTEM] FUNCTION  [IF NOT EXISTS] [[catalog\_name.]db\_name.]function\_name  AS identifier [LANGUAGE JAVA|SCALA|PYTHON] |

创建一个有 catalog 和数据库命名空间的 catalog function ，需要指定一个 **identifier** ，可指定 language tag 。 若 catalog 中，已经有同名的函数注册了，则无法注册。

如果 language tag 是 JAVA 或者 SCALA ，则 identifier 是 UDF 实现类的全限定名。关于 JAVA/SCALA UDF 的实现，请参考 [自定义函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/udfs/)。

**TEMPORARY**

创建一个有 catalog 和数据库命名空间的临时 catalog function ，并覆盖原有的 catalog function 。

**TEMPORARY SYSTEM**

创建一个没有数据库命名空间的临时系统 catalog function ，并覆盖系统内置的函数。

**IF NOT EXISTS**

若该函数已经存在，则不会进行任何操作。

**LANGUAGE JAVA||SCALA||PYTHON**

Language tag 用于指定 Flink runtime 如何执行这个函数。目前，只支持 JAVA, SCALA 和 PYTHON，且函数的默认语言为 JAVA。

### DROP 语句

DROP 语句用于从当前或指定的 Catalog 中删除一个已经注册的表、视图或函数。

Flink SQL 目前支持以下 DROP 语句：

* DROP TABLE
* DROP DATABASE
* DROP VIEW
* DROP FUNCTION

#### 执行 DROP 语句

可以使用 TableEnvironment 中的 executeSql() 方法执行 DROP 语句。 若 DROP 操作执行成功，executeSql() 方法返回 ‘OK’，否则会抛出异常。

以下的例子展示了如何在 TableEnvironment 中执行一个 DROP 语句。

|  |
| --- |
| EnvironmentSettings settings = EnvironmentSettings.newInstance()... TableEnvironment tableEnv = TableEnvironment.create(settings);  // 注册名为 “Orders” 的表 tableEnv.executeSql("CREATE TABLE Orders (`user` BIGINT, product STRING, amount INT) WITH (...)");  // 字符串数组： ["Orders"] String[] tables = tableEnv.listTables(); // or tableEnv.executeSql("SHOW TABLES").print();  // 从 catalog 删除 “Orders” 表 tableEnv.executeSql("DROP TABLE Orders");  // 空字符串数组 String[] tables = tableEnv.listTables(); // or tableEnv.executeSql("SHOW TABLES").print(); |

#### DROP TABLE

|  |
| --- |
| DROP TABLE [IF EXISTS] [catalog\_name.][db\_name.]table\_name |

根据给定的表名删除某个表。若需要删除的表不存在，则抛出异常。

**IF EXISTS**

表不存在时不会进行任何操作。

#### DROP DATABASE

|  |
| --- |
| DROP DATABASE [IF EXISTS] [catalog\_name.]db\_name [ (RESTRICT | CASCADE) ] |

根据给定的表名删除数据库。若需要删除的数据库不存在会抛出异常 。

**IF EXISTS**

若数据库不存在，不执行任何操作。

**RESTRICT**

当删除一个非空数据库时，会触发异常。（默认为开）

**CASCADE**

删除一个非空数据库时，把相关联的表与函数一并删除。

#### DROP VIEW

|  |
| --- |
| DROP [TEMPORARY] VIEW [IF EXISTS] [catalog\_name.][db\_name.]view\_name |

删除一个有 catalog 和数据库命名空间的视图。若需要删除的视图不存在，则会产生异常。

**TEMPORARY**

删除一个有 catalog 和数据库命名空间的临时视图。

**IF EXISTS**

若视图不存在，则不会进行任何操作。

#### DROP FUNCTION

|  |
| --- |
| DROP [TEMPORARY|TEMPORARY SYSTEM] FUNCTION [IF EXISTS] [catalog\_name.][db\_name.]function\_name; |

删除一个有 catalog 和数据库命名空间的 catalog function。若需要删除的函数不存在，则会产生异常。

**TEMPORARY**

删除一个有 catalog 和数据库命名空间的临时 catalog function。

**TEMPORARY SYSTEM**

删除一个没有数据库命名空间的临时系统函数。

**IF EXISTS**

若函数不存在，则不会进行任何操作。

### ALTER 语句

ALTER 语句用于修改一个已经在 Catalog 中注册的表、视图或函数定义。

Flink SQL 目前支持以下 ALTER 语句：

* ALTER TABLE
* ALTER DATABASE
* ALTER FUNCTION

#### 执行 ALTER 语句

可以使用 TableEnvironment 中的 executeSql() 方法执行 ALTER 语句。 若 ALTER 操作执行成功，executeSql() 方法返回 ‘OK’，否则会抛出异常。

以下的例子展示了如何在 TableEnvironment 中执行一个 ALTER 语句。

|  |
| --- |
| EnvironmentSettings settings = EnvironmentSettings.newInstance()... TableEnvironment tableEnv = TableEnvironment.create(settings);  // 注册名为 “Orders” 的表 tableEnv.executeSql("CREATE TABLE Orders (`user` BIGINT, product STRING, amount INT) WITH (...)");  // 字符串数组： ["Orders"] String[] tables = tableEnv.listTables(); // or tableEnv.executeSql("SHOW TABLES").print();  // 把 “Orders” 的表名改为 “NewOrders” tableEnv.executeSql("ALTER TABLE Orders RENAME TO NewOrders;");  // 字符串数组：["NewOrders"] String[] tables = tableEnv.listTables(); // or tableEnv.executeSql("SHOW TABLES").print(); |

#### ALTER TABLE

* 重命名表

|  |
| --- |
| ALTER TABLE [catalog\_name.][db\_name.]table\_name RENAME TO new\_table\_name |

把原有的表名更改为新的表名。

* 设置或修改表属性

|  |
| --- |
| ALTER TABLE [catalog\_name.][db\_name.]table\_name SET (key1=val1, key2=val2, ...) |

为指定的表设置一个或多个属性。若个别属性已经存在于表中，则使用新的值覆盖旧的值。

#### ALTER DATABASE

|  |
| --- |
| ALTER DATABASE [catalog\_name.]db\_name SET (key1=val1, key2=val2, ...) |

在数据库中设置一个或多个属性。若个别属性已经在数据库中设定，将会使用新值覆盖旧值。

#### ALTER FUNCTION

|  |
| --- |
| ALTER [TEMPORARY|TEMPORARY SYSTEM] FUNCTION  [IF EXISTS] [catalog\_name.][db\_name.]function\_name  AS identifier [LANGUAGE JAVA|SCALA|PYTHON] |

修改一个有 catalog 和数据库命名空间的 catalog function ，需要指定一个新的 identifier ，可指定 language tag 。若函数不存在，删除会抛出异常。

如果 language tag 是 JAVA 或者 SCALA ，则 identifier 是 UDF 实现类的全限定名。关于 JAVA/SCALA UDF 的实现，请参考 [自定义函数](https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/table/functions/udfs/)。

**TEMPORARY**

创建一个有 catalog 和数据库命名空间的临时 catalog function ，并覆盖原有的 catalog function 。

**TEMPORARY SYSTEM**

创建一个没有数据库命名空间的临时系统 catalog function ，并覆盖系统内置的函数。

**IF NOT EXISTS**

若该函数已经存在，则不会进行任何操作。

**LANGUAGE JAVA||SCALA||PYTHON**

Language tag 用于指定 Flink runtime 如何执行这个函数。目前，只支持 JAVA, SCALA 和 PYTHON，且函数的默认语言为 JAVA。

### INSERT 语句

INSERT 语句用来向表中添加行。

#### 执行 INSERT 语句

单条 INSERT 语句，可以使用 TableEnvironment 中的 executeSql() 方法执行。executeSql() 方法执行 INSERT 语句时会立即提交一个 Flink 作业，并且返回一个 TableResult 对象，通过该对象可以获取 JobClient 方便的操作提交的作业。 多条 INSERT 语句，使用 TableEnvironment 中的 createStatementSet 创建一个 StatementSet 对象，然后使用 StatementSet 中的 addInsertSql() 方法添加多条 INSERT 语句，最后通过 StatementSet 中的 execute() 方法来执行。

以下的例子展示了如何在 TableEnvironment 中执行一条 INSERT 语句，或者通过 StatementSet 执行多条 INSERT 语句。

|  |
| --- |
| EnvironmentSettings settings = EnvironmentSettings.newInstance()... TableEnvironment tEnv = TableEnvironment.create(settings);  // 注册一个 "Orders" 源表，和 "RubberOrders" 结果表 tEnv.executeSql("CREATE TABLE Orders (`user` BIGINT, product VARCHAR, amount INT) WITH (...)"); tEnv.executeSql("CREATE TABLE RubberOrders(product VARCHAR, amount INT) WITH (...)");  // 运行一条 INSERT 语句，将源表的数据输出到结果表中 TableResult tableResult1 = tEnv.executeSql( "INSERT INTO RubberOrders SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'"); // 通过 TableResult 来获取作业状态 System.out.println(tableResult1.getJobClient().get().getJobStatus());  //---------------------------------------------------------------------------- // 注册一个 "GlassOrders" 结果表用于运行多 INSERT 语句 tEnv.executeSql("CREATE TABLE GlassOrders(product VARCHAR, amount INT) WITH (...)");  // 运行多条 INSERT 语句，将原表数据输出到多个结果表中 StatementSet stmtSet = tEnv.createStatementSet(); // `addInsertSql` 方法每次只接收单条 INSERT 语句 stmtSet.addInsertSql( "INSERT INTO RubberOrders SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'"); stmtSet.addInsertSql( "INSERT INTO GlassOrders SELECT product, amount FROM Orders WHERE product LIKE '%Glass%'"); // 执行刚刚添加的所有 INSERT 语句 TableResult tableResult2 = stmtSet.execute(); // 通过 TableResult 来获取作业状态 System.out.println(tableResult1.getJobClient().get().getJobStatus()); |

#### 将 SELECT 查询数据插入表中

通过 INSERT 语句，可以将查询的结果插入到表中

##### 语法

|  |
| --- |
| INSERT { INTO | OVERWRITE } [catalog\_name.][db\_name.]table\_name [PARTITION part\_spec] select\_statement  part\_spec:  (part\_col\_name1=val1 [, part\_col\_name2=val2, ...]) |

**OVERWRITE**

INSERT OVERWRITE 将会覆盖表中或分区中的任何已存在的数据。否则，新数据会追加到表中或分区中。

**PARTITION**

PARTITION 语句应该包含需要插入的静态分区列与值。

##### 示例

|  |
| --- |
| -- 创建一个分区表 CREATE TABLE country\_page\_view (user STRING, cnt INT, date STRING, country STRING) PARTITIONED BY (date, country) WITH (...)  -- 追加行到该静态分区中 (date='2019-8-30', country='China') INSERT INTO country\_page\_view PARTITION (date='2019-8-30', country='China')  SELECT user, cnt FROM page\_view\_source;  -- 追加行到分区 (date, country) 中，其中 date 是静态分区 '2019-8-30'；country 是动态分区，其值由每一行动态决定 INSERT INTO country\_page\_view PARTITION (date='2019-8-30')  SELECT user, cnt, country FROM page\_view\_source;  -- 覆盖行到静态分区 (date='2019-8-30', country='China') INSERT OVERWRITE country\_page\_view PARTITION (date='2019-8-30', country='China')  SELECT user, cnt FROM page\_view\_source;  -- 覆盖行到分区 (date, country) 中，其中 date 是静态分区 '2019-8-30'；country 是动态分区，其值由每一行动态决定 INSERT OVERWRITE country\_page\_view PARTITION (date='2019-8-30')  SELECT user, cnt, country FROM page\_view\_source; |

#### 将值插入表中

通过 INSERT 语句，也可以直接将值插入到表中

##### 语法

|  |
| --- |
| INSERT { INTO | OVERWRITE } [catalog\_name.][db\_name.]table\_name VALUES values\_row [, values\_row ...]  values\_row:  : (val1 [, val2, ...]) |

**OVERWRITE**

INSERT OVERWRITE 将会覆盖表中的任何已存在的数据。否则，新数据会追加到表中。

##### 示例

|  |
| --- |
| CREATE TABLE students (name STRING, age INT, gpa DECIMAL(3, 2)) WITH (...);  INSERT INTO students  VALUES ('fred flintstone', 35, 1.28), ('barney rubble', 32, 2.32); |

### DESCRIBE Statements

DESCRIBE 语句用来描述一张表或者视图的 Schema。

#### 执行 DESCRIBE 语句

DESCRIBE 语句可以通过 TableEnvironment 的 executeSql() 执行。 若 DESCRIBE 操作执行成功，executeSql() 方法返回该表的 Schema，否则会抛出异常。

以下的例子展示了如何在 TableEnvironment 中执行一个 DESCRIBE 语句。

|  |
| --- |
| EnvironmentSettings settings = EnvironmentSettings.newInstance()... TableEnvironment tableEnv = TableEnvironment.create(settings);  // register a table named "Orders" tableEnv.executeSql( "CREATE TABLE Orders (" + " `user` BIGINT NOT NULl," + " product VARCHAR(32)," + " amount INT," + " ts TIMESTAMP(3)," + " ptime AS PROCTIME()," + " PRIMARY KEY(`user`) NOT ENFORCED," + " WATERMARK FOR ts AS ts - INTERVAL '1' SECONDS" + ") with (...)");  // print the schema tableEnv.executeSql("DESCRIBE Orders").print();  // print the schema tableEnv.executeSql("DESC Orders").print(); |

上面例子的结果是：

|  |
| --- |
| +---------+----------------------------------+-------+-----------+-----------------+----------------------------+ | name | type | null | key | computed column | watermark | +---------+----------------------------------+-------+-----------+-----------------+----------------------------+ | user | BIGINT | false | PRI(user) | | | | product | VARCHAR(32) | true | | | | | amount | INT | true | | | | | ts | TIMESTAMP(3) \*ROWTIME\* | true | | | `ts` - INTERVAL '1' SECOND | | ptime | TIMESTAMP(3) NOT NULL \*PROCTIME\* | false | | PROCTIME() | | +---------+----------------------------------+-------+-----------+-----------------+----------------------------+ 5 rows in set |

#### 语法

|  |
| --- |
| { DESCRIBE | DESC } [catalog\_name.][db\_name.]table\_name |

### EXPLAIN Statements

EXPLAIN 语句用来解释一条 query 语句或者 INSERT 语句的逻辑计划和优化后的计划。

#### 运行一条 EXPLAIN 语句

EXPLAIN 语句可以通过 TableEnvironment 的 executeSql() 执行。 若 EXPLAIN 操作执行成功，executeSql() 方法返回解释的结果，否则会抛出异常。

以下的例子展示了如何在 TableEnvironment 中执行一条 EXPLAIN 语句。

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment(); StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);  // register a table named "Orders" tEnv.executeSql("CREATE TABLE MyTable1 (`count` bigint, word VARCHAR(256)) WITH (...)"); tEnv.executeSql("CREATE TABLE MyTable2 (`count` bigint, word VARCHAR(256)) WITH (...)");  // explain SELECT statement through TableEnvironment.explainSql() String explanation = tEnv.explainSql( "SELECT `count`, word FROM MyTable1 WHERE word LIKE 'F%' " + "UNION ALL " + "SELECT `count`, word FROM MyTable2"); System.out.println(explanation);  // explain SELECT statement through TableEnvironment.executeSql() TableResult tableResult = tEnv.executeSql( "EXPLAIN PLAN FOR " + "SELECT `count`, word FROM MyTable1 WHERE word LIKE 'F%' " + "UNION ALL " + "SELECT `count`, word FROM MyTable2"); tableResult.print(); |

执行 EXPLAIN 语句后的结果为：

|  |
| --- |
| == Abstract Syntax Tree == LogicalUnion(all=[true])  LogicalFilter(condition=[LIKE($1, \_UTF-16LE'F%')])  FlinkLogicalTableSourceScan(table=[[default\_catalog, default\_database, MyTable1]], fields=[count, word])  FlinkLogicalTableSourceScan(table=[[default\_catalog, default\_database, MyTable2]], fields=[count, word])    == Optimized Logical Plan == DataStreamUnion(all=[true], union all=[count, word])  DataStreamCalc(select=[count, word], where=[LIKE(word, \_UTF-16LE'F%')])  TableSourceScan(table=[[default\_catalog, default\_database, MyTable1]], fields=[count, word])  TableSourceScan(table=[[default\_catalog, default\_database, MyTable2]], fields=[count, word])  == Physical Execution Plan == Stage 1 : Data Source  content : collect elements with CollectionInputFormat  Stage 2 : Data Source  content : collect elements with CollectionInputFormat   Stage 3 : Operator  content : from: (count, word)  ship\_strategy : REBALANCE   Stage 4 : Operator  content : where: (LIKE(word, \_UTF-16LE'F%')), select: (count, word)  ship\_strategy : FORWARD   Stage 5 : Operator  content : from: (count, word)  ship\_strategy : REBALANCE |

#### 语法

|  |
| --- |
| EXPLAIN PLAN FOR <query\_statement\_or\_insert\_statement> |

### USE 语句

USE 语句用来设置当前的 catalog 或者 database。

#### 运行一个 USE 语句

可以使用 TableEnvironment 中的 executeSql() 方法执行 USE 语句。 若 USE 操作执行成功，executeSql() 方法返回 ‘OK’，否则会抛出异常。

以下的例子展示了如何在 TableEnvironment 中执行一个 USE 语句。

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment(); StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);  // create a catalog tEnv.executeSql("CREATE CATALOG cat1 WITH (...)"); tEnv.executeSql("SHOW CATALOGS").print(); // +-----------------+ // | catalog name | // +-----------------+ // | default\_catalog | // | cat1 | // +-----------------+  // change default catalog tEnv.executeSql("USE CATALOG cat1");  tEnv.executeSql("SHOW DATABASES").print(); // databases are empty // +---------------+ // | database name | // +---------------+ // +---------------+  // create a database tEnv.executeSql("CREATE DATABASE db1 WITH (...)"); tEnv.executeSql("SHOW DATABASES").print(); // +---------------+ // | database name | // +---------------+ // | db1 | // +---------------+  // change default database tEnv.executeSql("USE db1");  // change module resolution order and enabled status tEnv.executeSql("USE MODULES hive"); tEnv.executeSql("SHOW FULL MODULES").print(); // +-------------+-------+ // | module name | used | // +-------------+-------+ // | hive | true | // | core | false | // +-------------+-------+ |

#### USE CATLOAG

|  |
| --- |
| USE CATALOG catalog\_name |

设置当前的 catalog。所有后续命令未显式指定 catalog 的将使用此 catalog。如果指定的的 catalog 不存在，则抛出异常。默认的当前 catalog 是 default\_catalog。

#### USE MODULES

|  |
| --- |
| USE MODULES module\_name1[, module\_name2, ...] |

使用声明的顺序设置启用的模块。所有后续命令将解析启用模块内的元数据（函数/用户定义类型/规则等）并遵循解析顺序。模块在加载时默认使用。如果USE MODULES语句未使用，加载的模块将被禁用。默认加载和启用的模块是core.

#### USE

|  |
| --- |
| USE [catalog\_name.]database\_name |

设置当前的 database。所有后续命令未显式指定 database 的将使用此 database。如果指定的的 database 不存在，则抛出异常。默认的当前 database 是 default\_database。

### SHOW 语句

SHOW 语句用于列出所有的 catalog，或者列出当前 catalog 中所有的 database，或者列出当前 catalog 和当前 database 的所有表或视图，或者列出当前正在使用的 catalog 和 database, 或者列出当前 catalog 和当前 database 中所有的 function，包括：系统 function 和用户定义的 function，或者仅仅列出当前 catalog 和当前 database 中用户定义的 function，或者列出当前环境所有激活的 module，或者列出当前环境所有加载的 module 及激活状态。

目前 Flink SQL 支持下列 SHOW 语句：

* SHOW CATALOGS
* SHOW CURRENT CATALOG
* SHOW DATABASES
* SHOW CURRENT DATABASE
* SHOW TABLES
* SHOW VIEWS
* SHOW FUNCTIONS
* SHOW MODULES
* SHOW FULL MODULES

#### 执行 SHOW 语句

可以使用 TableEnvironment 中的 executeSql() 方法执行 SHOW 语句。 若 SHOW 操作执行成功，executeSql() 方法返回所有对象，否则会抛出异常。

以下的例子展示了如何在 TableEnvironment 中执行一个 SHOW 语句。

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment(); StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);  // show catalogs tEnv.executeSql("SHOW CATALOGS").print(); // +-----------------+ // | catalog name | // +-----------------+ // | default\_catalog | // +-----------------+  // show current catalog tEnv.executeSql("SHOW CURRENT CATALOG").print(); // +----------------------+ // | current catalog name | // +----------------------+ // | default\_catalog | // +----------------------+  // show databases tEnv.executeSql("SHOW DATABASES").print(); // +------------------+ // | database name | // +------------------+ // | default\_database | // +------------------+  // show current database tEnv.executeSql("SHOW CURRENT DATABASE").print(); // +-----------------------+ // | current database name | // +-----------------------+ // | default\_database | // +-----------------------+  // create a table tEnv.executeSql("CREATE TABLE my\_table (...) WITH (...)"); // show tables tEnv.executeSql("SHOW TABLES").print(); // +------------+ // | table name | // +------------+ // | my\_table | // +------------+  // create a view tEnv.executeSql("CREATE VIEW my\_view AS ..."); // show views tEnv.executeSql("SHOW VIEWS").print(); // +-----------+ // | view name | // +-----------+ // | my\_view | // +-----------+  // show functions tEnv.executeSql("SHOW FUNCTIONS").print(); // +---------------+ // | function name | // +---------------+ // | mod | // | sha256 | // | ... | // +---------------+  // create a user defined function tEnv.executeSql("CREATE FUNCTION f1 AS ..."); // show user defined functions tEnv.executeSql("SHOW USER FUNCTIONS").print(); // +---------------+ // | function name | // +---------------+ // | f1 | // | ... | // +---------------+  // show modules tEnv.executeSql("SHOW MODULES").print(); // +-------------+ // | module name | // +-------------+ // | core | // +-------------+  // show full modules tEnv.executeSql("SHOW FULL MODULES").print(); // +-------------+-------+ // | module name | used | // +-------------+-------+ // | core | true | // | hive | false | // +-------------+-------+ |

#### SHOW CATALOGS

|  |
| --- |
| SHOW CATALOGS |

展示所有的 catalog。

#### SHOW CURRENT CATALOG

|  |
| --- |
| SHOW CURRENT CATALOG |

显示当前正在使用的 catalog。

#### SHOW DATABASES

|  |
| --- |
| SHOW DATABASES |

展示当前 catalog 中所有的 database。

#### SHOW CURRENT DATABASE

|  |
| --- |
| SHOW CURRENT DATABASE |

显示当前正在使用的 database。

#### SHOW TABLES

|  |
| --- |
| SHOW TABLES |

展示当前 catalog 和当前 database 中所有的表。

#### SHOW VIEWS

|  |
| --- |
| SHOW VIEWS |

展示当前 catalog 和当前 database 中所有的视图。

#### SHOW FUNCTIONS

|  |
| --- |
| SHOW [USER] FUNCTIONS |

展示当前 catalog 和当前 database 中所有的 function，包括：系统 function 和用户定义的 function。

USER 仅仅展示当前 catalog 和当前 database 中用户定义的 function。

#### SHOW MODULES

|  |
| --- |
| SHOW [FULL] MODULES |

展示当前环境激活的所有 module。

FULL 展示当前环境加载的所有 module 及激活状态。

### LOAD Statements

LOAD 语句用于加载内置或用户定义的模块。

#### 运行 LOAD 语句

LOAD语句可以与被执行executeSql()的方法TableEnvironment。executeSql()对于成功的 LOAD 操作，该方法返回“OK”；否则会抛出异常。

以下示例显示了如何在TableEnvironment.

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment(); StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);  // load a hive module tEnv.executeSql("LOAD MODULE hive WITH ('hive-version' = '3.1.2')"); tEnv.executeSql("SHOW MODULES").print(); // +-------------+ // | module name | // +-------------+ // | core | // | hive | // +-------------+ |

#### 加载模块

以下语法概述了可用的语法：

|  |
| --- |
| LOAD MODULE module\_name [WITH ('key1' = 'val1', 'key2' = 'val2', ...)] |

module\_name是一个简单的标识符。它区分大小写并且应该与模块工厂中定义的模块类型相同，因为它用于执行模块发现。Properties('key1' = 'val1', 'key2' = 'val2', ...)是一个映射，包含一组键值对（除了 key 'type'）并传递给发现服务以实例化相应的模块。

### UNLOAD Statements

UNLOAD 语句用于卸载内置或用户定义的模块。

#### 运行 UNLOAD 语句

UNLOAD语句可以与被执行executeSql()的方法TableEnvironment。executeSql()对于成功的 LOAD 操作，该方法返回“OK”；否则会抛出异常。

以下示例显示了如何在TableEnvironment.

|  |
| --- |
| StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment(); StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);  // unload a core module tEnv.executeSql("UNLOAD MODULE core"); tEnv.executeSql("SHOW MODULES").print(); // Empty set |

#### 卸载模块

以下语法概述了可用的语法：

|  |
| --- |
| UNLOAD MODULE module\_name |

### SET Statements

SET 语句用于修改配置或列出配置。

#### 运行 SET 语句

SET语句可以在SQL CLI 中执行。

以下示例显示如何SET在 SQL CLI 中运行语句。

|  |
| --- |
| Flink SQL> SET table.planner = blink; [INFO] Session property has been set.  Flink SQL> SET; table.planner=blink; |

#### 语法

|  |
| --- |
| SET (key = value)? |

如果未指定键和值，则只打印所有属性。否则，设置具有指定值的键。

### RESET Statements

RESET 语句用于将配置重置为默认值。

#### 运行 RESET 语句

RESET语句可以在SQL CLI 中执行。

以下示例显示如何RESET在 SQL CLI 中运行语句。

|  |
| --- |
| Flink SQL> RESET table.planner; [INFO] Session property has been reset.  Flink SQL> RESET; [INFO] All session properties have been set to their default values. |

#### 语法

|  |
| --- |
| RESET (key)? |

如果未指定键，则将所有属性重置为默认值。否则，将指定的键重置为默认值。