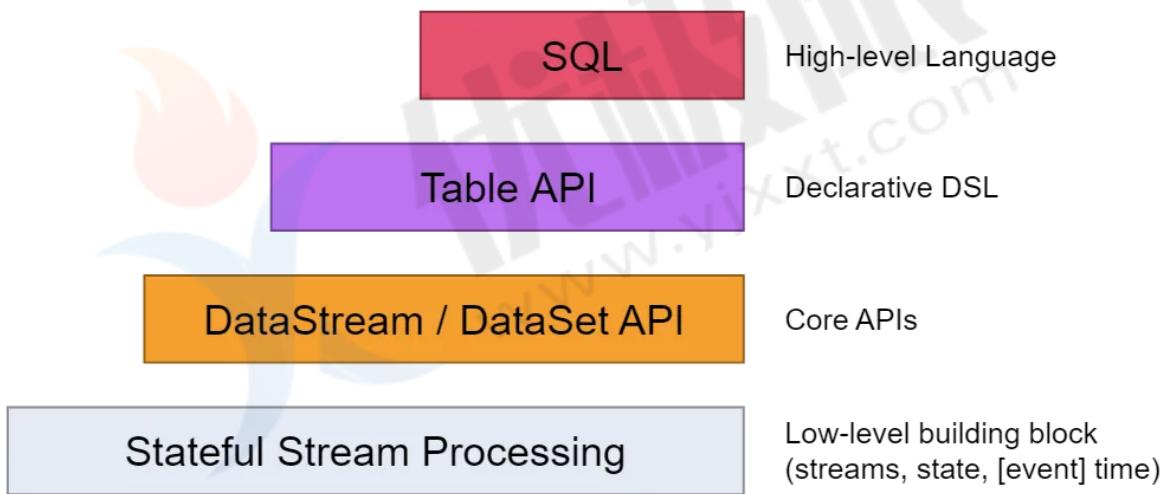


# Flink Table & Sql 1.15.2

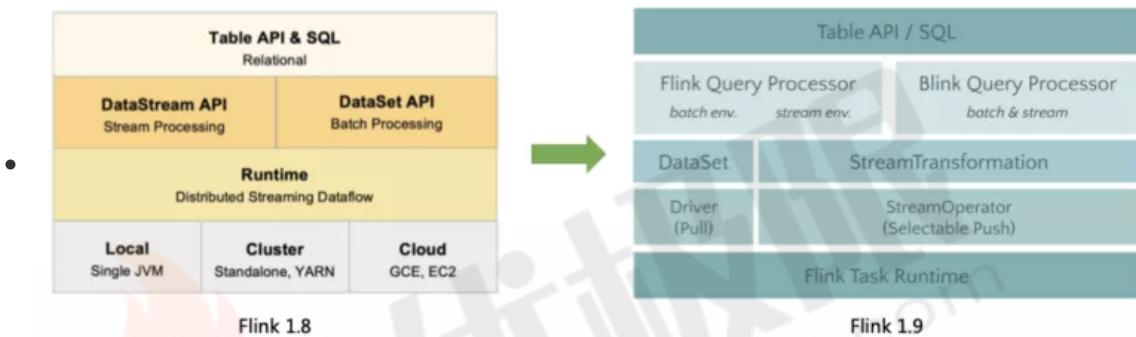


## 1. FlinkSQL 基础知识

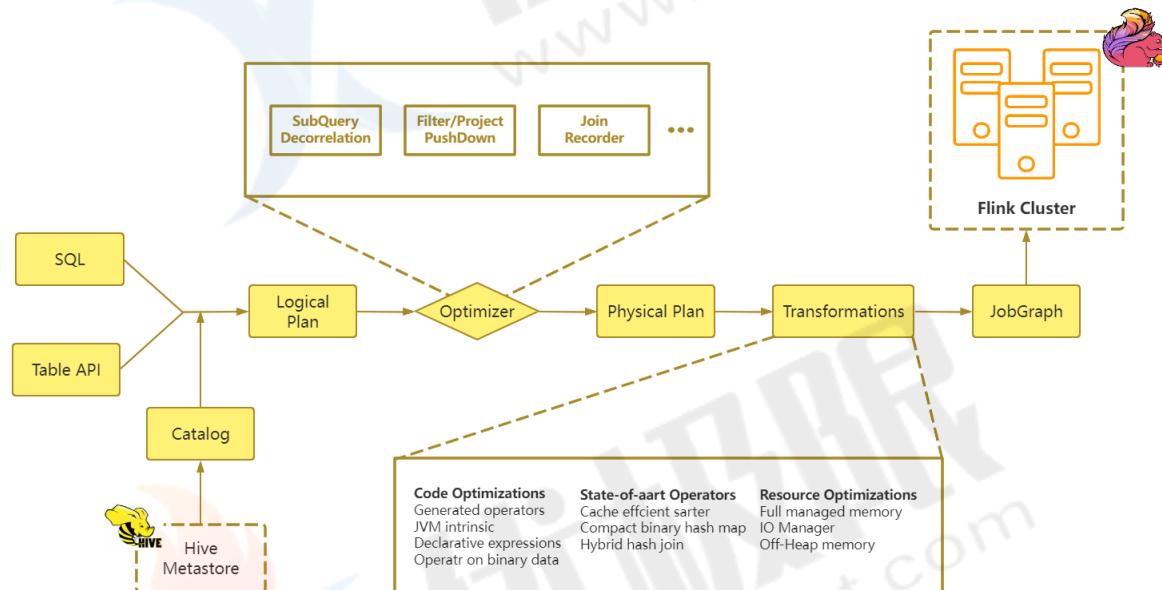
- Flink 的 Table API & SQL 程序可以连接到其他外部系统，用于读取和写入批处理和流式表；
- Flink SQL 是 Flink 实时计算为简化计算模型，降低用户使用实时计算门槛而设计的一套符合标准 SQL 语义的开发语言。
- Flink SQL 是面向用户的 API 层，在我们传统的流式计算领域，比如 Storm、Spark Streaming 都会提供一些 Function 或者 Datastream API，用户通过 Java 或 Scala 写业务逻辑，这种方式虽然灵活，但有一些不足，比如具备一定门槛且调优较难，随着版本的不断更新，API 也出现了很多不兼容的地方。
- 在这个背景下，毫无疑问，SQL 就成了我们最佳选择，之所以选择将 SQL 作为核心 API，是因为其具有几个非常重要的特点：
  - SQL 属于设定式语言，用户只要表达清楚需求即可，不需要了解具体做法；
  - SQL 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；
  - SQL 易于理解，不同行业和领域的人都懂，学习成本较低；
  - SQL 非常稳定，在数据库 30 多年的历史中，SQL 本身变化较少；
  - 流与批的统一，Flink 底层 Runtime 本身就是一个流与批统一的引擎，而 SQL 可以做到 API 层的流与批统一。

### 1.1. 整体架构

- 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
- 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！



## 1.2. 工作流程



- 工作流程如下：
  - SQL 和 Table 在进入 Flink 以后会转化成统一的数据结构表达形式，即 Logical Plan。
  - 其中，Catalog 会提供一些原数据信息，用于后续的优化。
  - Logical Plan 是优化的路口，经过一系列的优化规则后，Flink 会把初始的 Logical Plan 优化为 Physical Plan
  - Physical Plan 通过 Code Generation 机制翻译为 Transformation，最后转换成 JobGraph，用于提交到 Flink 的集群做分布式的执行。
  - 可以看到，整个流程并没有单独的流处理和批处理的路径，因为这些优化的过程和扩建都是共享的

## 1.3. 编程模型

- Maven依赖

```
<!-- Flink SQL -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-common</artifactId>
  <version>${flink.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner_2.12</artifactId>
  <version>${flink.version}</version>
</dependency>
<dependency>
```

```

<groupId>org.apache.flink</groupId>
<artifactId>flink-table-api-java</artifactId>
<version>${flink.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-api-scala_2.12</artifactId>
    <version>${flink.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-api-java-bridge</artifactId>
    <version>${flink.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-api-scala-bridge_2.12</artifactId>
    <version>${flink.version}</version>
</dependency>

```

- 编程模型

- 创建FlinkSql 运行环境
- 将数据源定义（映射）成表（视图）
- 执行sql 语义的查询（sql 语法或者 tableapi）
- 将查询结果输出到目标表

```

import org.apache.flink.table.api.*;
import org.apache.flink.connector.datagen.table.DataGenOptions;

// Create a TableEnvironment for batch or streaming execution.
// See the "Create a TableEnvironment" section for details.
TableEnvironment tableEnv = TableEnvironment.create(/*...*/);

// Create a source table
tableEnv.createTemporaryTable("SourceTable",
    TableDescriptor.forConnector("datagen")
        .schema(Schema.newBuilder()
            .column("f0", DataTypes.STRING())
            .build())
        .option(DataGenOptions.ROWS_PER_SECOND,
    100)
        .build());

// Create a sink table (using SQL DDL)
tableEnv.executeSql("CREATE TEMPORARY TABLE SinkTable WITH ('connector' = 'blackhole') LIKE SourceTable");

// Create a Table object from a Table API query
Table table2 = tableEnv.from("SourceTable");

// Create a Table object from a SQL query
Table table3 = tableEnv.sqlQuery("SELECT * FROM SourceTable");

// Emit a Table API result Table to a TableSink, same for SQL result
TableResult tableResult = table2.insertInto("SinkTable").execute();

```

## 2. FlinkSQL 通用API



### 2.1. 运行环境

- `TableEnvironment` 是 Table API 和 SQL 的核心概念。它负责:
  - 在内部的 catalog 中注册 `Table`
  - 注册外部的 catalog
  - 加载可插拔模块
  - 执行 SQL 查询
  - 注册自定义函数 (scalar、table 或 aggregation)
  - `DataStream` 和 `Table` 之间的转换(面向 `StreamTableEnvironment` )
- `Table` 总是与特定的 `TableEnvironment` 绑定。不能在同一条查询中使用不同 `TableEnvironment` 中的表，例如，对它们进行 join 或 union 操作。
- 不论输入数据源是流式的还是批式的，Table API 和 SQL 查询都会被转换成 `DataStream` 程序。
- 创建方式1

```
//设置环境配置和创建 Flink Table 环境
EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .instreamingMode()
    .build();
TableEnvironment tableEnvironment = TableEnvironment.create(settings);
```

- 创建方式2

```
//直接获取流运行环境和创建Flink Table环境
StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tableEnvironment =
StreamTableEnvironment.create(environment);
```

### 2.2. 创建表

- `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 名称、数据库名称以及对象名称。
  - 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。

#### 2.2.1. 常规表分类

- 临时表 (Temporary Table)
  - 与单个 Flink 会话 (session) 的生命周期相关。
  - 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。
- 永久表 (Permanent Table)
  - 在多个 Flink 会话和群集 (cluster) 中可见。
  - 永久表需要 catalog (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。
- 屏蔽特性 (Shadowing)
  - 使用与已存在的永久表相同的标识符去注册临时表。临时表会屏蔽永久表，并且只要临时表存在，永久表就无法访问。所有使用该标识符的查询都将作用于临时表。
  - 这可能对实验 (experimentation) 有用。它允许先对一个临时表进行完全相同的查询。
  - 例如只有一个子集的数据，或者数据是不确定的。一旦验证了查询的正确性，就可以对实际的生产表进行查询。

## 2.2.2. fromDataStream

- 之前的所有学习都是基于流，如果能将流转换成表就比较润了！
- 想要将一个 DataStream 转换成表也很简单，可以通过调用表环境的 fromDataStream()方法来实现，返回的就是一个 Table 对象

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
// 获取表环境
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
// 读取数据源
SingleOutputStreamOperator<Event> eventStream = env.addSource(...);
// 将数据流转换成表
Table eventTable = tableEnv.fromDataStream(eventStream);
```

- 如果流里是POJO对象，那么表的一行就是一个对象，表的列名就是对象的属性名，也可以自己选择对象的部分属性来组成表，然后使用as进行重命名

```
Table table = tableEnv.fromDataStream(stream, $("user").as("myUser"),
$("url").as("myUrl"));
```

- `fromDataStream` (DataStream<T> dataStream) Table  
`fromDataStream` (DataStream<T> dataStream, Schema schema) Table  
`fromDataStream` (DataStream<T> dataStream, String fields) Table  
`fromDataStream` (DataStream<T> dataStream, Expression... fields) Table
- Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor [Next Tip](#)

- \$ 在Java中的应用
  - Flink对Scala的支持性真的非常棒，让只能用Java的人泪流满面
  - 引用: import org.apache.flink.table.api.Expressions;
    - 调用: Expressions.\$("columnName")
  - 引用: import static org.apache.flink.table.api.Expressions.\$;
    - 调用: \$("columnName")

## 2.2.3. createTable

- 表总是通过三元标识符注册，包括 catalog 名、数据库名和表名。

- catalog\_name.database\_name.object\_name
- 用户可以指定一个 catalog 和数据库作为“当前catalog”和“当前数据库”。
- 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。

```

TableEnvironment tEnv = ...;
tEnv.useCatalog("custom_catalog");
tEnv.useDatabase("custom_database");

Table table = ...;

// register the view named 'exampleview' in the catalog named
// 'custom_catalog'
// in the database named 'custom_database'
tableEnv.createTemporaryView("exampleview", table);

// register the view named 'exampleview' in the catalog named
// 'custom_catalog'
// in the database named 'other_database'
tableEnv.createTemporaryView("other_database.exampleview", table);

// register the view named 'example.view' in the catalog named
// 'custom_catalog'
// in the database named 'custom_database'
tableEnv.createTemporaryView(`example.view`, table);

// register the view named 'exampleview' in the catalog named
// 'other_catalog'
// in the database named 'other_database'
tableEnv.createTemporaryView("other_catalog.other_database.exampleview",
table);

```

<code>createTable(String path, TableDescriptor descriptor)</code>	<code>void</code>
<code>createTemporaryTable(String path, TableDescriptor descriptor)</code>	<code>void</code>
Press Enter to insert, Tab to replace Next Tip	
⋮	

- TableDescriptor的四大核心要素
  - Connector 连接器
  - Format 数据格式
  - Schema 表结构
  - Option 连接器参数

#### 2.2.4. createTemporaryView

- 如果我们希望直接在 SQL 中引用这张表，就需要调用表环境的 createTemporaryView()方法来创建虚拟视图
- 调用 createTemporaryView()方法创建虚拟表，传入的两个参数
  - 第一个依然是注册的表名
  - 第二个可以直接就是DataStream
  - 之后传入多个参数，用来指定表中的字段

```

tableEnv.createTemporaryView("EventTable", eventStream,
$("timestamp").as("ts"), $("url"));

```

<code>createTemporaryView(String path, DataStream&lt;T&gt; dataStream, Schema schema)</code>	void
<code>createTemporaryView(String path, Table view)</code>	void
<code>createTemporaryView(String path, DataStream&lt;T&gt; dataStream)</code>	void
<code>createTemporaryView(String path, DataStream&lt;T&gt; dataStream, String fields)</code>	void
<code>createTemporaryView(String path, DataStream&lt;T&gt; dataStream, Expression... fields)</code>	void
Press Ctrl+ . to choose the selected [or first] suggestion and insert a dot afterwards Next Tip	

## 2.3. 数据类型

### 2.3.1. 原子类型

- DataStream 中支持的数据类型，Table 中也是都支持的
- 在 Flink 中，基础数据类型（Integer、Double、String）和通用数据类型（不可再拆分的数据类型）统一称作“原子类型”。
- 原子类型的 DataStream，转换之后就成了只有一列的Table，列字段（field）的数据类型可以由原子类型推断出。
- 另外还可以在 fromDataStream()方法里增加参数，用来重新命名列字段

```
StreamTableEnvironment tableEnv = ...;
DataStream<Long> stream = ...;
// 将数据流转换成动态表，动态表只有一个字段，重命名为 myLong
Table table = tableEnv.fromDataStream(stream, $("myLong"));
```

### 2.3.2. Tuple 类型

- 当原子类型不做重命名时，默认的字段名就是“f0”，容易想到，这其实就是将原子类型看作了一元组 Tuple1 的处理结果
- Table 支持 Flink 中定义的元组类型 Tuple，对应在表中字段名默认就是元组中元素的属性名 f0、f1、f2...。
- 所有字段都可以被重新排序，也可以提取其中的一部分字段。字段还可以通过调用表达式的 as()方法来进行重命名

```
StreamTableEnvironment tableEnv = ...;
DataStream<Tuple2<Long, Integer>> stream = ...;
// 将数据流转换成只包含 f1 字段的表
Table table = tableEnv.fromDataStream(stream, $("f1"));
// 将数据流转换成包含 f0 和 f1 字段的表，在表中 f0 和 f1 位置交换
Table table = tableEnv.fromDataStream(stream, $("f1"), $("f0"));
// 将 f1 字段命名为 myInt，f0 命名为 myLong
Table table = tableEnv.fromDataStream(stream, $("f1").as("myInt"),
    $("f0").as("myLong"));
```

### 2.3.3. POJO 类型

- Flink 也支持多种数据类型组合成的“复合类型”，最典型的就是简单 Java 对象（POJO 类型）。
- 由于 POJO 中已经定义好了可读性强的字段名，这种类型的数据流转换成 Table 就显得无比顺畅了
- 将 POJO 类型的 DataStream 转换成 Table，如果不指定字段名称，就会直接使用原始 POJO 类型中的字段名称。
- POJO 中的字段同样可以被重新排序、提取和重命名

```

StreamTableEnvironment tableEnv = ...;
DataStream<Event> stream = ...;
Table table = tableEnv.fromDataStream(stream);
Table table = tableEnv.fromDataStream(stream, $("user"));
Table table = tableEnv.fromDataStream(stream, $("user").as("myUser"),
$("url").as("myUrl"));

```

### 2.3.4. Row 类型

- Flink 中还定义了一个在关系型表中更加通用的数据类型——行 (Row)，它是 Table 中数据的基本组织形式。
- Row 类型也是一种复合类型，它的长度固定，而且无法直接推断出每个字段的类型，所以在使用时必须指明具体的类型信息；
- 在创建 Table 时调用的 CREATE 语句就会将所有的字段名称和类型指定，这在 Flink 中被称为表的“模式结构” (Schema) 。
- 除此之外，Row 类型还附加了一个属性 RowKind，用来表示当前行在更新操作中的类型。
- 这样 Row 就可以用来表示更新日志流 (changelog stream) 中的数据，从而架起了 Flink 中流和表的转换桥梁
- 所以在更新日志流中，元素的类型必须是 Row，而且需要调用 ofKind() 方法来指定更新类型

```

DataStream<Row> dataStream =
env.fromElements(
Row.ofKind(RowKind.INSERT, "Alice", 12),
Row.ofKind(RowKind.INSERT, "Bob", 5),
Row.ofKind(RowKind.UPDATE_BEFORE, "Alice", 12),
Row.ofKind(RowKind.UPDATE_AFTER, "Alice", 100));
// 将更新日志流转换为表
Table table = tableEnv.fromChangelogStream(dataStream);

```

## 2.4. 查询表

### 2.4.1. Table API

- Table API 是关于 Scala 和 Java 的集成语言式查询 API。与 SQL 相反，Table API 的查询不是由字符串指定，而是在宿主语言中逐步构建。
- Table API 是基于 Table 类的，该类表示一个表，并提供使用关系操作的方法。这些方法返回一个新的 Table 对象，该对象表示对输入 Table 进行关系操作的结果。
- 例如 table.groupBy(...).select()，其中 groupBy(...) 指定 table 的分组，而 select(...) 在 table 分组上的投影。
- 文档 Table API 说明了所有流处理和批处理表支持的 Table API 算子。

```

// get a TableEnvironment
TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// register Orders table

// scan registered Orders table
Table orders = tableEnv.from("Orders");
// compute revenue for all customers from France
Table revenue = orders

```

```

.filter($"cCountry").isEqual("FRANCE"))
.groupBy($"cID"), $"cName")
.select($"cID"), $"cName"), $"revenue").sum().as("revSum"));

// emit or convert Table
// execute query

```

## 2.4.2. SQL

- Flink SQL 是基于实现了SQL标准的 Apache Calcite 的。SQL 查询由常规字符串指定。
- 文档SQL 描述了Flink对流处理和批处理表的SQL支持。

```

// get a TableEnvironment
TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// register Orders table

// compute revenue for all customers from France
Table revenue = tableEnv.sqlQuery(
    "SELECT CID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY CID, cName"
);

// 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"
);

// emit or convert Table
// execute query

```

## 2.4.3. 混搭方式

- Table API 和 SQL 查询的混用非常简单因为它们都返回 Table 对象：
  - 可以在 SQL 查询返回的 Table 对象上定义 Table API 查询。
  - 在 TableEnvironment 中注册的结果表可以在 SQL 查询的 FROM 子句中引用，通过这种方法就可以在 Table API 查询的结果上定义 SQL 查询。
- 代码实现

```

import com.yjxxt.pojo.Emp;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.codehaus.jackson.map.ObjectMapper;

```

```

import static org.apache.flink.table.api.Expressions.$;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello04APIDQL {
    public static void main(String[] args) {
        //运行环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        StreamTableEnvironment tableEnvironment =
StreamTableEnvironment.create(environment);

        //Pojo类型
        DataStreamSource<String> empSource =
environment.readTextFile("data/emp.txt");
        DataStream<Emp> empStream = empSource.map(line -> new
ObjectMapper().readValue(line, Emp.class));
        Table empTable = tableEnvironment.fromDataStream(empStream);

        //查询数据: TableAPI--查询20部门员工信息的 编号 姓名 薪资 部门编号
        empTable.filter($"deptno").isEqual(20)
            .select($"empno", $"ename", $"sal", $"deptno")
            .execute()
            .print();

        //查询数据: SQL--查询20部门员工信息的 编号 姓名 薪资 部门编号
        tableEnvironment.createTemporaryView("t_emp", empTable);
        tableEnvironment.sqlQuery("select empno,ename,sal,deptno from
t_emp where deptno = 20")
            .execute()
            .print();

        //查询数据: 混合方式--查询20部门员工信息的 编号 姓名 薪资 部门编号
        //SQL查询中可以直接使用table的信息
        //SQL查询结果返回的又是一个Table
        tableEnvironment.sqlQuery("select * from " +
empTable.toString() + " where deptno = 20")
            .filter($"job").isEqual("ANALYST")
            .select($"ename", $"job")
            .execute()
            .print();
    }
}

```

## 2.5. 输出表

### 2.5.1. toDataStream

- 将一个 Table 对象转换成 DataStream 非常简单，只要直接调用表环境的方法 toDataStream()就可以了。
- 得到的流只是一个 仅追加流(只有插入) ，只有插入操作
- // 这里需要将要转换的 Table 对象作为参数传入  
tableEnv.toDataStream(myTable).print();

## 2.5.2. insertInto

- Table 通过写入 TableSink 输出。TableSink 是一个通用接口，包括：
  - 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)
  - 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。
  - Insert模式:
    - AppendStreamTable，只能执行insert动作，例如窗口聚合结果，每个窗口的结果都是唯一的，不会影响之前窗口的输出结果
  - Redo模式:
    - 对应RetractStreamTableSink /UpsertStreamTableSink，除了执行Insert动作，还可执行Update/Delete动作，也就是结果可更新
    - RetractStreamTableSink与UpsertStreamTableSink的区别主要在于消息编码格式不同，如果产生一条结果数据需要更新
      - RetractStreamTableSink需要编码两条消息Delete与Insert
      - UpsertStreamTableSink只需要编码成为一条upsert消息即可
- Table.insertInto(String tableName) 定义了一个完整的端到端管道将源表中的数据传输到一个被注册的输出表中。
  - 该方法通过名称在 catalog 中查找输出表并确认 Table schema 和输出表 schema 一致。
  - 可以通过方法 TablePipeline.explain() 和 TablePipeline.execute() 分别来解释和执行一个数据流管道。

```
// get a TableEnvironment
TableEnvironment tableEnv = ...;

// create an output Table
tableEnvironment.executeSql(
    "CREATE TABLE print_table (\n" +
    "  f0 STRING,\n" +
    "  f1 STRING,\n" +
    "  f2 STRING,\n" +
    "  f3 STRING\n" +
    ") WITH (\n" +
    "  'connector' = 'print'\n" +
    ")");

// compute a result Table using Table API operators and/or SQL queries
Table result = ...;

// Prepare the insert into pipeline
```

```

TablePipeline pipeline = result.insertInto("print_table");

// Print explain details
pipeline.printExplain();

// emit the result Table to the registered TableSink
pipeline.execute();

```

## 3. FlinkSQL Connector

- Connector 通常是用于对接外部存储建表（源表或目标表）时的映射器、桥接器
- Connector 本质上是对 Flink 的 Table Source /Table Sink 算子的封装；
- 连接器使用的核心要素
  - 导入连接器 jar 包依赖
  - 指定连接器类型名
  - 指定连接器所需的参数（不同连接器有不同的参数配置需求）
  - 获取连接器所提供的元数据
- FlinkSQL目前支持的 Format
  - <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/connectors/table/overview/>

Name	Version	Source	Sink
Filesystem		Bounded and Unbounded Scan, Lookup	Streaming Sink, Batch Sink
Elasticsearch	6.x & 7.x	Not supported	Streaming Sink, Batch Sink
Apache Kafka	0.10+	Unbounded Scan	Streaming Sink, Batch Sink
Amazon Kinesis Data Streams		Unbounded Scan	Streaming Sink
JDBC		Bounded Scan, Lookup	Streaming Sink, Batch Sink
Apache HBase	1.4.x & 2.2.x	Bounded Scan, Lookup	Streaming Sink, Batch Sink
Apache Hive	Supported Versions	Unbounded Scan, Bounded Scan, Lookup	Streaming Sink, Batch Sink

### 3.1. Kafka Connector

- <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/connectors/table/kafka/>
- Kafka 连接器提供从 Kafka topic 中消费和写入数据的能力。
- maven依赖

```

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka</artifactId>
  <version>1.15.2</version>
</dependency>

```

- 代码实现

- ```

CREATE TABLE flink_kafka_source (\\n` +
    "  `deptno` INT,\\n` +
    "  `dname` STRING,\\n` +
    "  `loc` STRING\\n` +
  ") WITH (\\n` +
    "  'connector' = 'kafka',\\n` +
    "  'topic' = 'topic_kafka_source',\\n` +
    "  'properties.bootstrap.servers' =
  'node01:9092,node02:9092,node03:9092',\\n` +
    "  'properties.group.id' = 'yjxxtliyi',\\n` +
    "  'scan.startup.mode' = 'earliest-offset',\\n` +
    "  'format' = 'csv'\\n` +
  ");

```

- 可用元数据

| 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                                                                               | R   |
|           | offset          | BIGINT NOT NULL                                                                        | R   |
|           | timestamp       | TIMESTAMP_LTZ(3) NOT NULL                                                              | R/W |
|           | timestamp-type  | STRING NOT NULL                                                                        | R   |
|           |                 | 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"。 |     |
|           |                 |                                                                                        |     |

- 配置参数

| Option                        | Description                                                                                      |
|-------------------------------|--------------------------------------------------------------------------------------------------|
| connector                     | 指定要使用的连接器，供 Kafka 使用 'kafka'                                                                     |
| topic                         | 支持通过分号分隔主题的源主题列表，将表用作数据源                                                                         |
| topic-pattern                 | 要读取的主题名称模式的正则表达式                                                                                 |
| properties.bootstrap.servers  | 逗号分隔的 Kafka 代理服务列表                                                                               |
| properties.*                  | 传给 kafka-client 的（会自动去除 properties.前缀）                                                           |
| fomat/key.format/value.format | 用于反序列化和序列化 Kafka 消息的值部分的格式。                                                                      |
| key.fields                    | 默认情况下，此列表为空，该列表应如下所示 'field1;field2'。                                                            |
| scan.startup.mode             | 消费 kafka 的启动模式可取： earliest-offset , latest-offset , group-offsets , timestamp , specific-offsets |
| scan.startup.specific-offsets | 'specific-offsets' 在启动模式的情况下为每个分区指定偏移量                                                           |
| sink.partitioner              | 默认 default， 可取 fixed , round-robin , 自定义类 'com.yjxxt.MyPartitioner'                              |
| sink.delivery-guarantee       | 可取 at-least-once , exactly-once , none                                                           |
| sink.parallelism              | 默认为 none， 根据算子 chain 决定                                                                          |

## 3.2. JDBC Connector

- <https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/connectors/table/jdbc/>
- JDBC 连接器允许使用 JDBC 驱动向任意类型的关系型数据库读取或者写入数据。
- 如果在 DDL 中定义了主键，JDBC sink 将以 upsert 模式与外部系统交换 UPDATE/DELETE 消息；否则，它将以 append 模式与外部系统交换消息且不支持消费 UPDATE/DELETE 消息。
- maven 依赖

```

    <dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-jdbc</artifactId>
        <version>1.15.2</version>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.18</version>
    </dependency>

```

- 代码示例

- ```

CREATE TABLE flink_mysql_source (
    "empno" INT, \n" +
    "ename" STRING, \n" +
    "job" STRING, \n" +
    "mgr" INT, \n" +
    "hiredate" DATE, \n" +
    "sal" DECIMAL(10, 2), \n" +
    "comm" DECIMAL(10, 2), \n" +
    "deptno" INT, \n" +
    PRIMARY KEY (empno) NOT ENFORCED\n" +
) WITH (\n" +
    'connector' = 'jdbc', \n" +
    'url' = 'jdbc:mysql://localhost:3306/scott?\nserverTimezone=UTF&characterEncoding=utf8&useUnicode=true&useSSL=false'\n, \n" +
    'driver' = 'com.mysql.cj.jdbc.Driver', \n" +
    'username' = 'root', \n" +
    'password' = '123456', \n" +
    'table-name' = 'emp'\n" +
);

```

- 连接器参数

参数	是否必填	默认值	类型	描述
connector	必填	(none)	String	指定使用什么类型的连接器，这里应该是'jdbc'。
url	必填	(none)	String	JDBC 数据库 url。
table-name	必填	(none)	String	连接到 JDBC 表的名称。
driver	可选	(none)	String	用于连接到此 URL 的 JDBC 驱动类名，如果不设置，将自动从 URL 中推导。
username	可选	(none)	String	JDBC 用户名。如果指定了 'username' 和 'password' 中的任一参数，则两者必须都被指定。
password	可选	(none)	String	JDBC 密码。
connection.max-retry-timeout	可选	60s	Duration	最大重试超时时间，以秒为单位且不应该小于 1 秒。
scan.partition.column	可选	(none)	String	用于将输入进行分区的列名。请参阅下面的 <a href="#">分区扫描</a> 部分了解更多详情。
scan.partition.num	可选	(none)	Integer	分区数。
scan.partition.lower-bound	可选	(none)	Integer	第一个分区的最小值。
scan.partition.upper-bound	可选	(none)	Integer	最后一个分区的最大值。
scan.fetch-size	可选	0	Integer	每次循环读取时应该从数据库中获取的行数。如果指定的值为 '0'，则该配置项会被忽略。
scan.auto-commit	可选	true	Boolean	在 JDBC 驱动程序上设置 auto-commit 标志，它决定了每个语句是否在事务中自动提交。有些 JDBC 驱动程序，特别是 Postgres，可能需要将此设置为 false 以便流化结果。
lookup.cache.max-rows	可选	(none)	Integer	lookup cache 的最大行数，若超过该值，则最老的行记录将会过期。默认情况下，lookup cache 是未开启的。请参阅下面的 <a href="#">Lookup Cache</a> 部分了解更多详情。
lookup.cache.ttl	可选	(none)	Duration	lookup cache 中每一行记录的最大存活时间，若超过该时间，则最老的行记录将会过期。默认情况下，lookup cache 是未开启的。请参阅下面的 <a href="#">Lookup Cache</a> 部分了解更多详情。

- 数据类型映射

- Flink 支持连接到多个使用方言 (dialect) 的数据库，如 MySQL、Oracle、PostgreSQL、Derby 等。

<a href="#">MySQL type</a>	<a href="#">Oracle type</a>	<a href="#">PostgreSQL type</a>	<a href="#">Flink SQL type</a>
TINYINT			TINYINT
SMALLINT TINYINT UNSIGNED		SMALLINT INT2 SMALLSERIAL SERIAL2	SMALLINT
INT MEDIUMINT SMALLINT UNSIGNED		INTEGER SERIAL	INT
BIGINT INT UNSIGNED		BIGINT BIGSERIAL	BIGINT
BIGINT UNSIGNED			DECIMAL(20, 0)
BIGINT		BIGINT	BIGINT
FLOAT	BINARY_FLOAT	REAL FLOAT4	FLOAT
DOUBLE DOUBLE PRECISION	BINARY_DOUBLE	FLOAT8 DOUBLE PRECISION	DOUBLE
NUMERIC(p, s) DECIMAL(p, s)	SMALLINT FLOAT(s) DOUBLE PRECISION REAL NUMBER(p, s)	NUMERIC(p, s) DECIMAL(p, s)	DECIMAL(p, s)
BOOLEAN TINYINT(1)		BOOLEAN	BOOLEAN
DATE	DATE	DATE	DATE
TIME [(p)]	DATE	TIME [(p)] [WITHOUT TIMEZONE]	TIME [(p)] [WITHOUT TIMEZONE]
DATETIME [(p)]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]	TIMESTAMP [(p)] [WITHOUT TIMEZONE]
CHAR(n) VARCHAR(n) TEXT	CHAR(n) VARCHAR(n) CLOB	CHAR(n) CHARACTER(n) VARCHAR(n) CHARACTER VARYING(n) TEXT	STRING

<a href="#">MySQL type</a>	<a href="#">Oracle type</a>	<a href="#">PostgreSQL type</a>	<a href="#">Flink SQL type</a>
BINARY VARBINARY BLOB	RAW(s) BLOB	BYTEA	BYTES
		ARRAY	ARRAY

### 3.3. DataGen Connector

- <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/connectors/table/datagen/>
- DataGen 连接器允许按数据生成规则进行读取。
- DataGen 连接器是内置的。
- 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、binary、varbinary、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。
- 代码实现

```

o  public static void main(String[] args) {
    //运行环境
    StreamExecutionEnvironment environment =
    StreamExecutionEnvironment.getExecutionEnvironment();
    environment.setParallelism(1);
    StreamTableEnvironment tableEnvironment =
    StreamTableEnvironment.create(environment);

    tableEnvironment.executeSql("CREATE TABLE table_datagen(\n" +
        "f_sequence INT,\n" +
        "f_random INT,\n" +
        "f_random_str STRING,\n" +
        "ts AS localtimestamp\n" +
        "WATERMARK FOR ts AS ts\n" +
        ") WITH(\n" +
        "'connector' = 'datagen',\n" +
        "'rows-per-second' = '5',\n" +
        "'fields.f_sequence.kind' =
    'sequence',\n" +
        "'fields.f_sequence.start' = '1',\n" +
        "'fields.f_sequence.end' = '1000',\n" +
        "'fields.f_random.min' = '1',\n" +
        "'fields.f_random.max' = '1000',\n" +
        "'fields.f_random_str.length' = '10'\n"
        +
        ")");
    tableEnvironment.sqlQuery("select * from
    datagen").execute().print();
}

```

- 配置参数

参数	是否必选	默认值	数据类型	描述
connector	必须	(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.#.max-past	可选	0	Duration	随机生成器生成相对当前时间向过去偏移的最大值，适用于 timestamp 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、binary、varbinary、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。

### 3.4. Upsert Kafka Connector

- Upsert Kafka Connector允许用户以upsert的方式从Kafka主题读取数据或将数据写入Kafka主题。
- 在某些场景中输出（更新）结果的时候，需要将 Kafka 消息记录的 key 当成主键处理，用来确定一条数据是应该作为插入、删除还是更新记录来处理。
- 使用 upsert-kafka connector，必须在创建表时定义主键，并为键（key.format）和值（value.format）指定序列化反序列化格式。
- 作为 source
  - upsert-kafka Connector会生产一个changelog流，其中每条数据记录都表示一个更新或删除事件。
  - 如果不存在对应的key，则视为INSERT操作。
  - 如果已经存在了相对应的key，则该key对应的value值为最后一次更新的值。

- 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 为空的消息将被视作为 DELETE 消息。
- 作为 sink
  - upsert-kafka Connector 会消费一个 changelog 流。
  - 将 INSERT / UPDATE\_AFTER 数据作为正常的 Kafka 消息值写入(即 INSERT 和 UPDATE 操作，都会进行正常写入)
  - 如果是更新，则同一个 key 会存储多条数据，但在读取该表数据时，只保留最后一次更新的值
  - 并将 DELETE 数据以 value 为空的 Kafka 消息写入
  - 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。
- 代码实现

```

○ CREATE TABLE pageviews_per_region (
    userid STRING,
    pv BIGINT,
    uv BIGINT,
    PRIMARY KEY (userid) NOT ENFORCED
) WITH (
    'connector' = 'upsert-kafka',
    'topic' = 'pageviews_per_region',
    'properties.bootstrap.servers' = '...',
    'key.format' = 'csv',
    'value.format' = 'csv'
);

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;

```

- 连接器参数

参数	是否必选	默认值	数据类型	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 <a href="#">Kafka 参数文档</a> 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 <a href="#">格式</a> 页面以获取更多详细信息和格式参数。
key.fields-prefix	optional	(none)	String	Defines a custom prefix for all fields of the key format to avoid name clashes with fields of the value format. By default, the prefix is empty. If a custom prefix is defined, both the table schema and 'key.fields' will work with prefixed names. When constructing the data type of the key format, the prefix will be removed and the non-prefixed names will be used within the key format. Please note that this option requires that 'value.fields-include' must be set to '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 <a href="#">格式</a> 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。

## 3.5. FileSystem Connector

- <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/connectors/table/filesystem/>

- 在 Flink 中包含了该文件系统连接器，不需要添加额外的依赖。

```

○ <dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-files</artifactId>
    <version>${flink.version}</version>
</dependency>
```

- 从文件系统中读取或者向文件系统中写入行时，需要指定相应的 format。
- 代码示例

```

○ CREATE TABLE MyUserTable (
    column_name1 INT,
    column_name2 STRING,
    ...
    part_name1 INT,
    part_name2 STRING
) PARTITIONED BY (part_name1, part_name2) WITH (
    'connector' = 'filesystem',           -- 必选：指定连接器类型
    'path' = 'file:///path/to/whatever',   -- 必选：指定路径
    'format' = '...',                   -- 必选：文件系统连接器指定 format
                                         -- 有关更多详情，请参考 Table

Formats
    'partition.default-name' = '...',    -- 可选：默认的分区名，动态分区模式下
                                         分区字段值是 null 或空字符串

                                         -- 可选：该属性开启了在 sink 阶段通过动态分区字段来 shuffle 数据，该功能可以大大
                                         减少文件系统 sink 的文件数，但是可能会导致数据倾斜，默认值是 false
    'sink.shuffle-by-partition.enable' = '...', 
    ...
)
```

- 分区文件

- Flink 的文件系统连接器支持分区，使用了标准的 hive。但是，不需要预先注册分区到 table catalog，而是基于目录结构自动做了分区发现。
- 例如，根据下面的目录结构，分区表将被推断包含 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
```

- 文件格式 File Formats

- CSV: RFC-4180。是非压缩的。
- JSON: 注意，文件系统连接器的 JSON format 与传统的标准的 JSON file 的不同，而是非压缩的。换行符分割的 JSON。
- Avro: Apache Avro。通过配置 avro.codec 属性支持压缩。
- Parquet: Apache Parquet。兼容 hive。
- Orc: Apache Orc。兼容 hive。

- Debezium-JSON: debezium-json.
- Canal-JSON: canal-json.
- Raw: raw.

- 配置参数

◦ Options	Description
path	文件路径
source.monitor-interval	源检查新文件的时间间隔。
sink.rolling-policy.file-size	滚动前的最大文件大小。
sink.rolling-policy.rollover-interval	文件在滚动前可以保持打开的最长时间（默认为30分钟，以避免出现许多小文件）
sink.rolling-policy.check-interval	检查基于时间的滚动策略的间隔。
auto-compaction=false	是否在流接收器中启用自动压缩。
compaction.file-size	压缩目标文件大小，默认值为滚动文件大小。和rolling file size一致
sink.partition-commit.trigger	process-time、partition-time 提交分区的时间
sink.partition-commit.delay	延迟提交分区的时间。如果是日分区，应是'1d'，小时分区，应是'1 h'，默认为'0 s'
sink.partition-commit.policy.kind = success-file,metastore	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。
sink.parallelism	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

- Source

- 文件系统连接器可用于将单个文件或整个目录的数据读取到单个表中。当使用目录作为source路径时，对目录中的文件进行**无序的读取**。
- 当运行模式为流模式时，文件系统连接器会自动监控输入目录。可以使用以下属性修改监控时间间隔。
  - 设置新文件的监控时间间隔，并且必须设置 > 0 的值。每个文件都由其路径唯一标识，一旦发现新文件，就会处理一次。
  - 已处理的文件在 source 的整个生命周期内存储在 state 中，因此，source 的 state 在 checkpoint 和 savepoint 时进行保存。
  - 更短的时间间隔意味着文件被更快地发现，但也意味着更频繁地遍历文件系统/对象存储。
  - 如果未设置此配置选项，则提供的路径仅被扫描一次，因此源将是有界的。
- 可用的元数据

键	数据类型	描述
file.path	STRING NOT NULL	输入文件的完整路径。
file.name	STRING NOT NULL	文件名，即距离文件根路径最远的元素。
file.size	BIGINT NOT NULL	文件的字节数。
file.modification-time	TIMESTAMP_LTZ(3) NOT NULL	文件的修改时间。

## 4. FlinkSQL Schema

### 4.1. physical column

- 物理字段：源自于“外部存储”系统本身 schema 中的字段
  - kafka 消息的 key、value (json 格式) 中的字段；
  - mysql 表中的字段；
  - hive 表中的字段；
  - parquet 文件中的字段.....

### 4.2. computed column

- 表达式字段（逻辑字段）：在物理字段上施加一个 sql 表达式，并将表达式结果定义为一个字段
- Java代码
  - Schema.newBuilder()  
.columnByExpression("age\_exp", "age+10") // 声明表达式字段 age\_exp，它来源于物理字段 age+10
- SQL代码
  - CREATE TABLE MyTable (
 `user\_id` BIGINT,
 `price` DOUBLE,
 `quantity` DOUBLE,
 `cost` AS price \* quantity, -- cost 来源于: price\*quantity
 ) WITH (
 'connector' = 'kafka'
 ...
 );

### 4.3. metadata column

- 元数据字段：来源于 connector 从外部存储系统中获取到的“外部系统元信息”
- kafka 的消息，通常意义上的数据内容是在 record 的 key 和 value 中的，而实质上（底层角度来看），kafka 中的每一条 record，不光带了 key 和 value 数据内容，还带了这条 record 所属的 topic，所属的 partition，所在的 offset，以及 record 的 timestamp 和 timestamp 类型等“元信息”。而 flink 的 connector 可以获取并暴露这些元信息，并允许用户将这些信息定义成 flinksq 表中的字段；
- Java代码
  - Schema.newBuilder()  
.columnByMetadata("topic", DataTypes.STRING())

- SQL代码

- ```
CREATE TABLE MyTable (
    `user_id` BIGINT,
    `name` STRING,
    `record_time` TIMESTAMP_LTZ(3) METADATA FROM 'timestamp' -- 元数据字段,
来源于 kafka record 的 timestamp
) WITH (
    'connector' = 'kafka'
    ...
);
```

## 4.4. 主键约束

- 单字段主键约束语法:

- ```
id INT PRIMARY KEY NOT ENFORCED,
name STRING
```

- 多字段主键约束语法:

- ```
id,
name,
PRIMARY KEY(id, name) NOT ENFORCED
```

- 注意

- The Kafka table 'default\_catalog.default\_database.t\_dept' with 'csv' format doesn't support defining

## 4.5. 代码示例

- Java代码

- ```
// {"id":4,"name":"zs","nick":"tiedan","age":18,"gender":"male"}
tenv.createTable("t_person",
    TableDescriptor
        .forConnector("kafka")
        .schema(Schema.newBuilder()
            .column("id", DataTypes.INT()) // column是声明
物理字段到表结构中来
            .column("name", DataTypes.STRING()) // column
是声明物理字段到表结构中来
            .column("nick", DataTypes.STRING()) // column
是声明物理字段到表结构中来
            .column("age", DataTypes.INT()) // column是声
明物理字段到表结构中来
            .column("gender", DataTypes.STRING()) // column是声明物理字段到表结构中来
            .columnByExpression("guid","id") // 声明表达式字
段
/*
.columnByExpression("big_age",$( "age").plus(10)) */ // 声明表达式字
段
            .columnByExpression("big_age","age + 10") // 声明表达式字段
        )
    )
);
```

// isvirtual 是表示：当这个表被sink表时，该字段是否出现在schema中

```
.columnByMetadata("offs",DataTypes.BIGINT(),"offset",true) // 声明元数据  
字段  
  
.columnByMetadata("ts",DataTypes.TIMESTAMP_LTZ(3),"timestamp",true) //  
声明元数据字段  
    /*.primaryKey("id","name")*/  
    .build()  
    .format("json")  
    .option("topic","mytopic")  
    .option("properties.bootstrap.servers","hdp01:9092")  
    .option("properties.group.id","g1")  
    .option("scan.startup.mode","earliest-offset")  
    .option("json.fail-on-missing-field","false")  
    .option("json.ignore-parse-errors","true")  
    .build()  
);  
  
tenv.executeSql("select * from t_person").print();
```

- SQL代码

- // {"id":4,"name":"zs","nick":"tiedan","age":18,"gender":"male"}  
tenv.executeSql(  
 "create table t\_person  
"  
 + " ("  
"  
 + " id int ,  
" // -- 物理字段  
 + " name string,  
" // -- 物理字段  
 + " nick string,  
"  
 + " age int ,  
"  
 + " gender string ,  
"  
 + " guid as id,  
" // -- 表达式字段（逻辑字段）  
 + " big\_age as age + 10 ,  
" // -- 表达式字段（逻辑字段）  
 + " offs bigint metadata from 'offset' ,  
" // -- 元数据字段  
 + " ts TIMESTAMP\_LTZ(3) metadata from 'timestamp' ,  
" // -- 元数据字段  
 /\*+ " PRIMARY KEY(id,name) NOT ENFORCED  
\*/ // -- 主键约束  
 + " )  
"  
 + " WITH (  
"  
 + " 'connector' = 'kafka',  
"

```

        + "    'topic' = 'mytopic',
        "
        + "    'properties.bootstrap.servers' = 'hdp01:9092',    "
        + "    'properties.group.id' = 'g1',
        "
        + "    'scan.startup.mode' = 'earliest-offset',
        "
        + "    'format' = 'json',
        "
        + "    'json.fail-on-missing-field' = 'false',
        "
        + "    'json.ignore-parse-errors' = 'true'
        "
        + " )
        "
    );
}

tenv.executeSql("desc t_person").print();
tenv.executeSql("select * from t_person where id>2").print();

```

## 5. FlinkSQL Format

- connector 连接器：对接外部存储时，根据外部存储中的数据格式不同，需要用到不同的 format 组件；
- format 组件：作用就是告诉连接器，如何解析外部存储中的数据及映射到表 schema；
- format 组件的使用要点
  - 导入 format 组件的 jar 包依赖
  - 指定 format 组件的名称
  - 设置 format 组件所需的参数（不同 format 组件有不同的参数配置需求）
- FlinkSQL目前支持的 Format
  - [https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/connectors/table/format\\_s/overview/](https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/connectors/table/format_s/overview/)

Formats	Supported Connectors
CSV	Apache Kafka, Upsert Kafka, Amazon Kinesis Data Streams, Filesystem
JSON	Apache Kafka, Upsert Kafka, Amazon Kinesis Data Streams, Filesystem, Elasticsearch
Apache Avro	Apache Kafka, Upsert Kafka, Amazon Kinesis Data Streams, Filesystem
Confluent Avro	Apache Kafka, Upsert Kafka
Debezium CDC	Apache Kafka, Filesystem
Canal CDC	Apache Kafka, Filesystem
Maxwell CDC	Apache Kafka, Filesystem
OGG CDC	Apache Kafka, Filesystem
Apache Parquet	Filesystem
Apache ORC	Filesystem
Raw	Apache Kafka, Upsert Kafka, Amazon Kinesis Data Streams, Filesystem

## 5.1. CSV Format

- <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/connectors/table/formats/csv/>
- maven依赖

- ```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-csv</artifactId>
    <version>1.15.2</version>
</dependency>
```

- 代码实现

- ```
CREATE TABLE user_behavior (
    user_id BIGINT,
    item_id BIGINT,
    category_id BIGINT,
    behavior STRING,
    ts TIMESTAMP(3)
) WITH (
    'connector' = 'kafka',
    'topic' = 'user_behavior',
    'properties.bootstrap.servers' = 'localhost:9092',
    'properties.group.id' = 'testGroup',
    'format' = 'csv',
    'csv.ignore-parse-errors' = 'true',
    'csv.allow-comments' = 'true'
)
```

- 参数配置

- | 参数                          | 是否必选 | 默认值    | 类型      | 描述  |
|-----------------------------|------|--------|---------|---|
| format                      | 必选   | (none) | String  | 指定要使用的格式，这里应该是 'csv'。   |
| csv.field-delimiter         | 可选   | ,      | String  | 字段分隔符（默认 ','），必须为单字符。你可以使用反斜杠字符指定一些特殊字符，例如 '\t' 代表制表符。你也可以通过 unicode 编码在纯 SQL 文本中指定一些特殊字符，例如 'csv.field-delimiter' = U&'0001' 代表 0x01 字符。 |
| csv.disable-quote-character | 可选   | false  | Boolean | 是否禁止对引用的值使用引号（默认是 false）。如果禁止，选项 'csv.quote-character' 不能设置。  |
| csv.quote-character         | 可选   | "      | String  | 用于围住字段值的引号字符（默认 "）。   |
| csv.allow-comments          | 可选   | false  | Boolean | 是否允许忽略注释行（默认不允许），注释行以 '#' 作为起始字符。如果允许注释行，请确保 csv.ignore-parse-errors 也开启了从而允许空行。  |
| csv.ignore-parse-errors     | 可选   | false  | Boolean | 当解析异常时，是跳过当前字段或行，还是抛出错误失败（默认为 false，即抛出错误失败）。如果忽略字段的解析异常，则会将该字段值设置为 null。   |
| csv.array-element-delimiter | 可选   | ;      | String  | 分隔数组和行元素的字符串（默认 ';'）。   |
| csv.escape-character        | 可选   | (none) | String  | 转义字符（默认关闭）。   |
| csv.null-literal            | 可选   | (none) | String  | 是否将 "null" 字符串转化为 null 值。   |

- 数据类型映射

- 目前 CSV 的 schema 都是从 table schema 推断而来的。显式地定义 CSV schema 暂不支持。
- Flink 的 CSV Format 数据使用 jackson databind API 去解析 CSV 字符串。

| Flink SQL 类型            | CSV 类型                        |
|-------------------------|-------------------------------|
| CHAR / VARCHAR / STRING | string                        |
| BOOLEAN                 | boolean                       |
| BINARY / VARBINARY      | string with encoding: base64  |
| DECIMAL                 | number                        |
| TINYINT                 | number                        |
| SMALLINT                | number                        |
| INT                     | number                        |
| ○ BIGINT                | number                        |
| FLOAT                   | number                        |
| DOUBLE                  | number                        |
| DATE                    | string with format: date      |
| TIME                    | string with format: time      |
| TIMESTAMP               | string with format: date-time |
| INTERVAL                | number                        |
| ARRAY                   | array                         |
| ROW                     | object                        |

## 5.2. Json Format

- <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/connectors/table/formats/json/>
- maven 依赖
  - ```
<dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-json</artifactId>
            <version>1.15.2</version>
          </dependency>
```
- 代码实现
  - ```
CREATE TABLE user_behavior (
```

```

    user_id BIGINT,
    item_id BIGINT,
    category_id BIGINT,
    behavior STRING,
    ts TIMESTAMP(3)
) WITH (
  'connector' = 'kafka',
  'topic' = 'user_behavior',
  'properties.bootstrap.servers' = 'localhost:9092',
  'properties.group.id' = 'testGroup',
  'format' = 'json',
  'json.fail-on-missing-field' = 'false',
  'json.ignore-parse-errors' = 'true'
)

```

- 参数配置

| 参数                               | 是否必须 | 默认值    | 类型      | 描述   |
|----------------------------------|------|--------|---------|--|
| format                           | 必选   | (none) | String  | 声明使用的格式，这里应为'json'。  |
| json.fail-on-missing-field       | 可选   | false  | Boolean | 当解析字段缺失时，是跳过当前字段或行，还是抛出错误失败（默认为 false，即抛出错误失败）。  |
| json.ignore-parse-errors         | 可选   | false  | Boolean | 当解析异常时，是跳过当前字段或行，还是抛出错误失败（默认为 false，即抛出错误失败）。如果忽略字段的解析异常，则会将该字段值设置为null。   |
| ○ json.timestamp-format.standard | 可选   | 'SQL'  | String  | <p>声明输入和输出的 TIMESTAMP 和 TIMESTAMP_LTZ 的格式。当前支持的格式为'sql' 以及 'ISO-8601'：</p> <ul style="list-style-type: none"> <li>可选参数 'SQL' 将会以 "yyyy-MM-dd HH:mm:ss.s{precision}" 的格式解析 TIMESTAMP，例如 "2020-12-30 12:13:14.123"，以 "yyyy-MM-dd HH:mm:ss.s{precision}Z" 的格式解析 TIMESTAMP_LTZ，例如 "2020-12-30 12:13:14.123Z" 且会以相同的格式输出。</li> <li>可选参数 'ISO-8601' 将会以 "yyyy-MM-ddTHH:mm:ss.s{precision}" 的格式解析输入 TIMESTAMP，例如 "2020-12-30T12:13:14.123"，以 "yyyy-MM-ddTHH:mm:ss.s{precision}Z" 的格式解析 TIMESTAMP_LTZ，例如 "2020-12-30T12:13:14.123Z" 且会以相同的格式输出。</li> </ul> |
| json.map-null-key.mode           | 选填   | 'FAIL' | String  | <p>指定处理 Map 中 key 值为空的方法。当前支持的值有 'FAIL', 'DROP' 和 'LITERAL'：</p> <ul style="list-style-type: none"> <li>Option 'FAIL' 将抛出异常，如果遇到 Map 中 key 值为空的数据。</li> <li>Option 'DROP' 将丢弃 Map 中 key 值为空的数据项。</li> <li>Option 'LITERAL' 将使用字符串常量来替换 Map 中的空 key 值。字符串常量的值由 'json.map-null-key.literal' 定义。</li> </ul>   |

- 数据类型映射

- JSON schema 将会自动从 table schema 之中自动推导得到。不支持显式地定义 JSON schema。
- 在 Flink 中，JSON Format 使用 jackson databind API 去解析和生成 JSON。

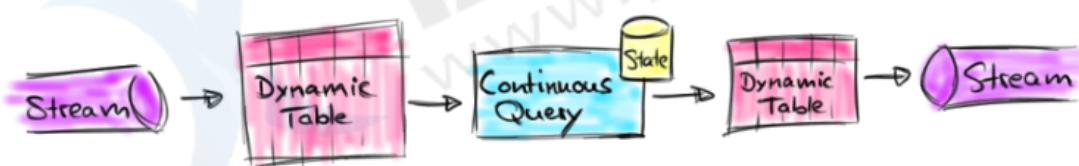
| Flink SQL 类型                   | JSON 类型  |
|--------------------------------|--|
| CHAR / VARCHAR / STRING        | string   |
| BOOLEAN                        | boolean  |
| BINARY / VARBINARY             | string with encoding: base64                       |
| DECIMAL                        | number   |
| TINYINT                        | number   |
| SMALLINT                       | number   |
| INT                            | number   |
| BIGINT                         | number   |
| ○ FLOAT                        | number   |
| DOUBLE                         | number   |
| DATE                           | string with format: date                           |
| TIME                           | string with format: time                           |
| TIMESTAMP                      | string with format: date-time                      |
| TIMESTAMP_WITH_LOCAL_TIME_ZONE | string with format: date-time (with UTC time zone) |
| INTERVAL                       | number   |
| ARRAY                          | array  |
| MAP / MULTISET                 | object   |
| ROW                            | object   |

## 6. FlinkSQL WaterMark

### 6.1. 时间语义

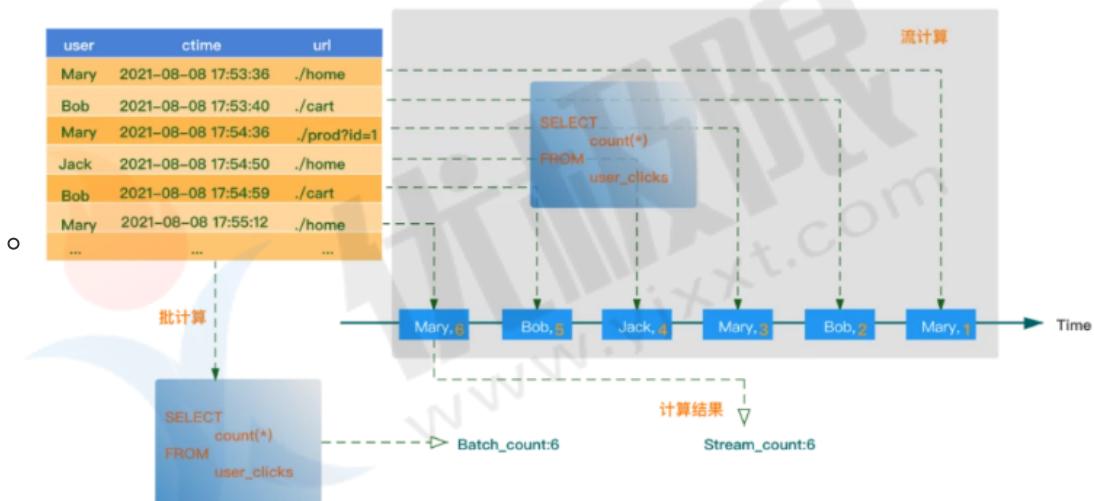
#### 6.1.1. 动态表

- 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 (Continuous Query)。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。
- 

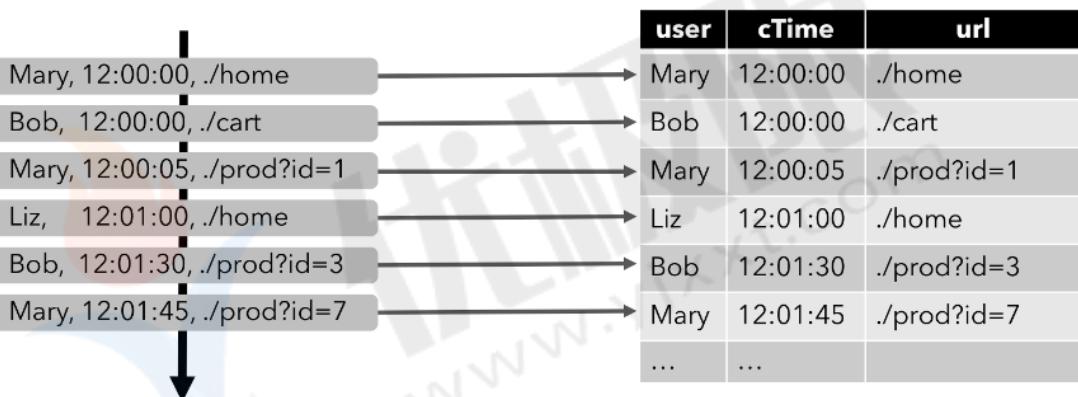


- 与 spark、hive 等组件中的“表”的最大不同之处：flinksql 中的表是动态表！

- flink 对数据的核心抽象是“无界（或有界）的数据流”
- 对数据处理过程的核心抽象是“流式持续处理”

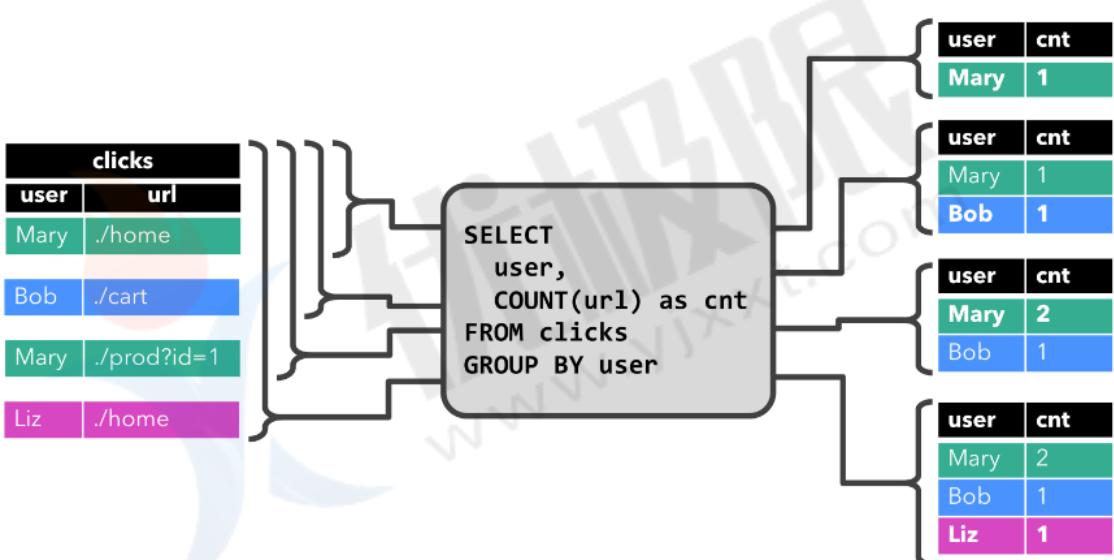


- flinksql 对“源表（动态表） ”的计算及输出结果（结果表），也是流式、动态、持续的；
  - 数据源的数据是持续输入
  - 查询过程是持续计算
  - 查询结果是持续输出

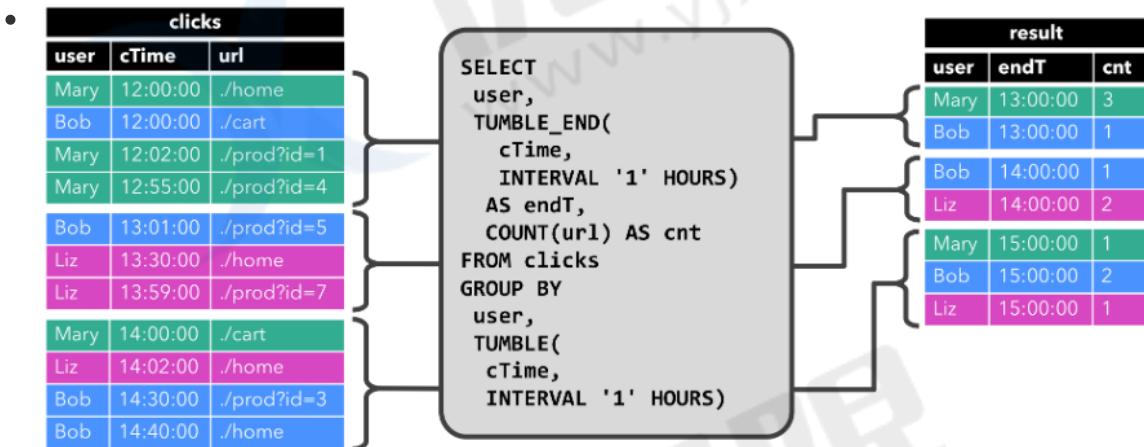


### 6.1.2. 连续查询

- 在动态表上计算一个连续查询，并生成一个新的动态表。与批处理查询不同，连续查询从不终止，并根据其输入表上的更新更新其结果表。在任何时候，连续查询的结果在语义上与以批处理模式在输入表快照上执行的相同查询的结果相同。



- 第一个查询是一个简单的 GROUP-BY COUNT 聚合查询。它基于 user 字段对 clicks 表进行分组，并统计访问的 URL 的数量。
- 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。
  - 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]。
  - 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。



- 第一个查询是一个简单的 GROUP-BY COUNT 聚合查询。它基于 user 字段对 clicks 表进行分组，并统计访问的 URL 的数量。
- 除此之外还将 clicks 分组至每小时滚动窗口中
  - 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。
  - 对于时间戳在 12:00:00 和 12:59:59 之间的窗口。clicks 表包含四行数据，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。
  - 对于时间戳在 13:00:00 和 13:59:59 之间的窗口。clicks 表包含三行数据，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。

## 6.2. 事件时间

### 6.2.1. SQL中定义时间

- 创建表的 DDL (CREATE TABLE 语句) 中，可以增加一个字段，通过 WATERMARK语句来定义事件时间属性。
- WATERMARK 语句主要用来定义水位线 (watermark) 的生成表达式，这个表达式会将带有事件时间戳的字段标记为事件时间属性，并在它基础上给出水位线的延迟时间。
- 定义方式如下：

```

CREATE TABLE EventTable(
    user STRING,
    url STRING,
    ts TIMESTAMP(3), // 单位是毫秒
    WATERMARK FOR ts AS ts - INTERVAL '5' SECOND // 水位线延迟5秒
) WITH (
    ...
);
  
```

- 这里我们把 ts 字段定义为事件时间属性，而且基于 ts 设置了 5 秒的水位线延迟。
- 格式是 INTERVAL <数值> <时间单位>: INTERVAL '5' SECOND

## 6.2.2. DataStream 定义时间

- 事件时间属性也可以在将 DataStream 转换为表的时候来定义。
- 调用 fromDataStream() 方法创建表时，可以追加参数来定义表中的字段结构；这时可以给某个字段加上 rowtime() 后缀，就表示将当前字段指定为事件时间属性。
  - 字段可以是数据中本不存在、额外追加上去的“逻辑字段”，就像之前 DDL 中定义的第二种情况；
  - 字段可以是本身固有的字段，那么这个字段就会被事件时间属性所覆盖，类型也会被转换为 TIMESTAMP。
  - 不论那种方式，时间属性字段中保存的都是事件的时间戳（TIMESTAMP 类型）
- 需要注意的是，这种方式只负责指定时间属性，而时间戳的提取和水位线的生成应该之前就在 DataStream 上定义好了。
- 由于 DataStream 中没有时区概念，因此 Flink 会将事件时间属性解析成不带时区的 TIMESTAMP 类型，所有的时间值都被当作 UTC 标准时间

```
// 1. 创建一个表执行环境
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
env.setParallelism(1);

SingleOutputStreamOperator<Event> streamOperator = env.addSource(new
clickSource())
    // 乱序流的WaterMark生成
    .assignTimestampsAndWatermarks(WatermarkStrategy
        .<Event>forBoundedOutOfOrder(Duration.ofSeconds(2)) ///
延迟2秒保证数据正确
    .withTimestampAssigner(new
SerializableTimestampAssigner<Event>() {
        @Override // 时间戳的提取器
        public long extractTimestamp(Event event, long l) {
            return event.getTimestamp();
        }
    })
;

tableEnv.fromDataStream(streamOperator, $("user_name"), $("url"), $("timestamp"
).as ("ts")
,$("et").rowtime()));
```

## 6.3. 处理时间

- 定义处理时间属性时，必须要额外声明一个字段，专门用来保存当前的处理时间

### 6.3.1. SQL 中定义时间

- 在创建表的 DDL (CREATE TABLE 语句) 中，可以增加一个额外的字段，通过调用系统内置的 PROCTIME() 函数来指定当前的处理时间属性，返回的类型是 TIMESTAMP\_LTZ
- 代码实现

```
o CREATE TABLE EventTable(
    user STRING,
    url STRING,
    ts AS PROCTIME()
) WITH (
    ...
);
```

- 时间属性，其实是以“计算列” (computed column) 的形式定义出来的
- 所谓的计算列是 Flink SQL 中引入的特殊概念，可以用一个 AS 语句来在表中产生数据中不存在的列，并且可以利用原有的列、各种运算符及内置函数。

### 6.3.2. DataStream定义时间

- 处理时间属性同样可以在将DataStream转换为表的时候来定义。我们调用fromDataStream()方法创建表时，可以用.proctime()后缀来指定处理时间属性字段。
- 由于处理时间是系统时间，原始数据中并没有这个字段，所以处理时间属性一定不能定义在一个已有字段上，只能定义在表结构所有字段的最后，作为额外的逻辑字段出现
- 代码实现

```
o DataStream<Tuple2<String, String>> stream = ...;
// 声明一个额外的字段作为处理时间属性字段
Table table = tEnv.fromDataStream(stream, $("user"), $("url"),
$("ts").proctime());
```

## 7. FlinkSQL 窗口TVF

- <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/dev/table/sql/queries/window-tvf/>
- TVF[Windowing table-valued functions] 窗口化表值函数
- 目前 Flink 提供了以下几个窗口 TVF:
  - 滚动窗口 (Tumbling Windows)
  - 滑动窗口 (Hop Windows, 跳跃窗口)
  - 累积窗口 (Cumulate Windows)
  - 会话窗口 (Session Windows, 目前尚未完全支持)
- 在窗口 TVF 的返回值中，除去原始表中的所有列，还增加了用来描述窗口的额外 3 个列：
  - “窗口起始点” (window\_start)
  - “窗口结束点” (window\_end)
  - “窗口时间” (window\_time)
- 起始点和结束点比较好理解，这里的“窗口时间”指的是窗口中的时间属性，它的值等于 window\_end - 1ms，所以相当于是窗口中能够包含数据的最大时间戳

### 7.1. TUMBLE

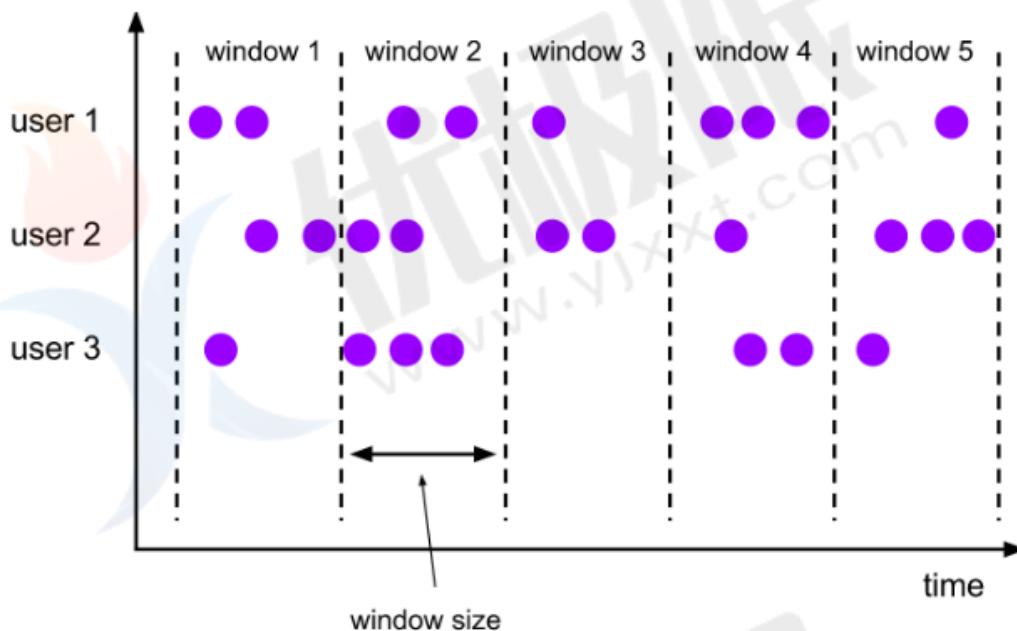
- 滚动窗口在 SQL 中的概念与 DataStream API 中的定义完全一样，是长度固定、时间对齐、无重叠的窗口，一般用于周期性的统计计算
- TUMBLE函数有三个必需的参数:
  - TUMBLE(TABLE data, DESCRIPTOR(timecol), size [, offset ])

- data: 表参数, 此表需要包含有一个时间属性列 【 time attribute column】

- timecol: 一个列描述符, 指示数据的哪个时间属性列应该映射到滚动的窗口

- size: 指定滚动窗口的大小

- 



- 案例:

- 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 |

## 7.2. HOP

- Hopping windows 也被称为 “sliding windows”.
- HOP函数分配的窗口覆盖大小间隔内的行, 并根据时间属性列移动每个窗口
- 【assigns windows that cover rows within the interval of size and shifting every slide based on a time attribute column】
- HOP函数有三个必需的参数:

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

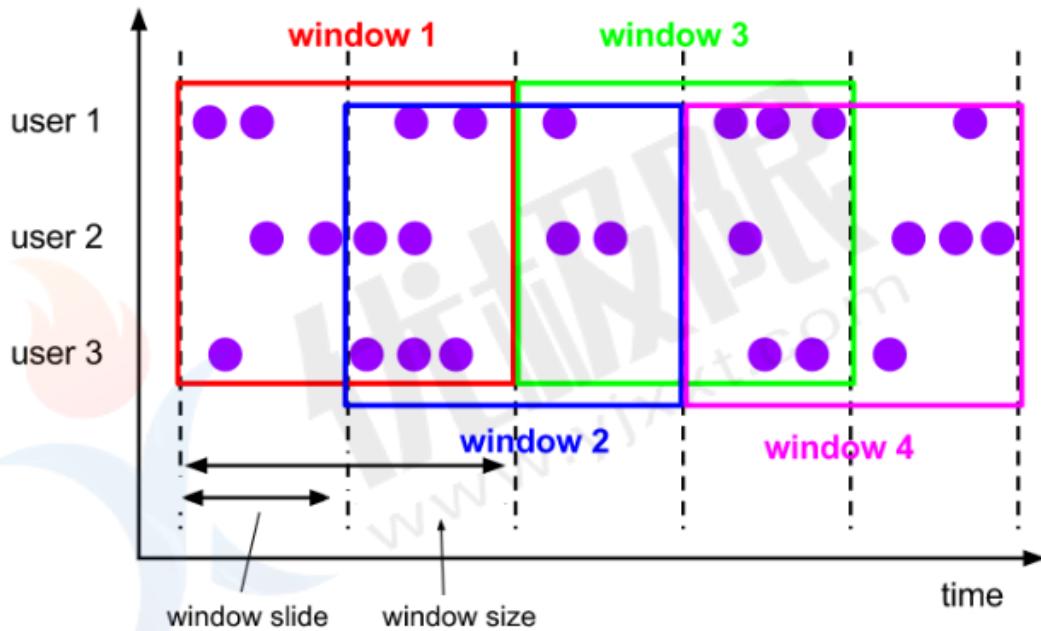
- data: 表参数, 此表需要包含有一个时间属性列 【 time attribute column】

- timecol: 一个列描述符, 指示数据的哪个时间属性列应该映射到滑动的窗口

- slide: 指定顺序hopping 窗口开始之间的持续时间

- size: 指定hopping 窗口宽度的持续时间, size必须是slide的整数倍。

-



- 案例：

```

    -- 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 |
    +-----+-----+-----+
  
```

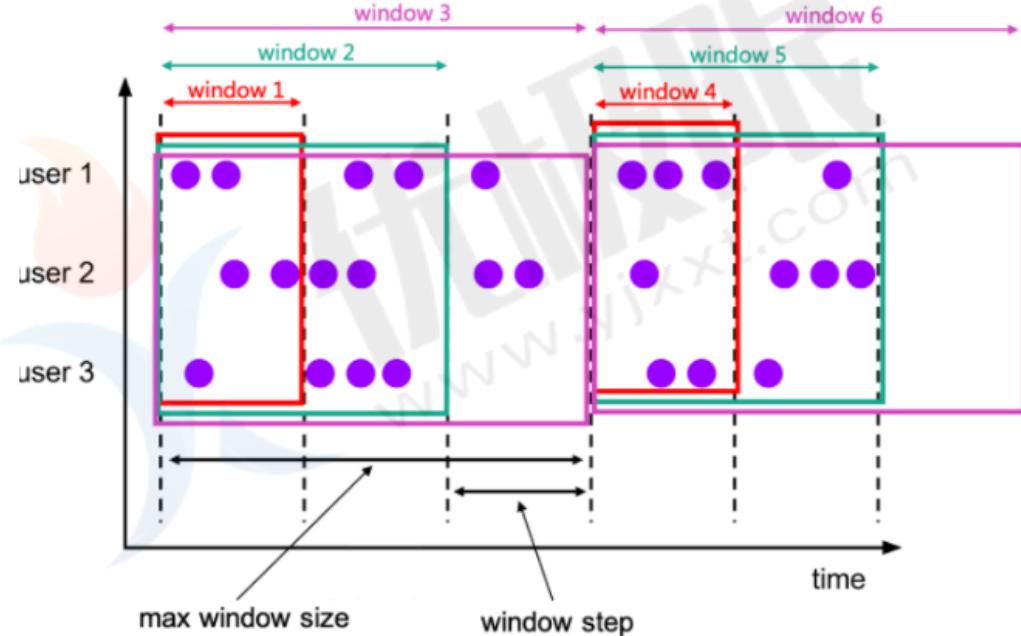
### 7.3. CUMULATE

- Cumulating windows 【累积窗口】在某些场景中非常有用。
  - 例如每日仪表板从00:00到每分钟绘制累积UV数，10:00的UV线代表从00:00到10:00的UV总数，这可以通过累积窗口轻松有效地实现
- CUMULATE函数将元素分配给覆盖在初始步长间隔内的行，并将每一步扩展为多一个步长(保持window start固定)，直到最大窗口大小。
- 可以把cumulative函数看作应用TUMBLE窗口，首先使用最大窗口大小，然后将每个滚动窗口分割成几个具有相同窗口开始和窗口结束步长差异的窗口。
- 因此，累积窗口确实是重叠的，而且没有固定的大小。
- CUMULATE 函数有三个必需的参数：

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

- data: 表参数，此表需要包含有一个时间属性列 【 time attribute column】
- timecol: 一个列描述符，指示数据的哪个时间属性列应该映射到滑动的窗口

- step: 指定连续累积窗口结束之间增加的窗口大小的持续时间
- size: 指定累积窗口的最大宽度的持续时间。大小必须是步长的整数倍
- 



- 示例：

```

◦ -- apply aggregation on the hopping 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 |
+-----+-----+-----+

```

## 8. FlinkSQL 聚合查询

### 8.1. 分组聚合

- SQL 中一般所说的聚合我们都很熟悉，主要是通过内置的一些聚合函数来实现的，比如SUM()、MAX()、MIN()、AVG()以及COUNT()。
- 它们的特点是对多条输入数据进行计算，得到一个唯一的值，属于“多对一”的转换。比如我们可以通过下面的代码计算输入数据的个数：
- 更多的情况下，我们可以通过 GROUP BY 子句来指定分组的键（key），从而对数据按照某个字段做一个分组统计。

```

    o  tableEnvironment.sqlQuery("SELECT pid, sum(num) AS total\n" +
        "FROM (VALUES\n" +
        "  ('省1','市1','县1',100),\n" +
        "  ('省1','市2','县2',101),\n" +
        "  ('省1','市2','县1',102),\n" +
        "  ('省2','市1','县4',103),\n" +
        "  ('省2','市2','县1',104),\n" +
        "  ('省2','市2','县1',105),\n" +
        "  ('省3','市1','县1',106),\n" +
        "  ('省3','市2','县1',107),\n" +
        "  ('省3','市2','县2',108),\n" +
        "  ('省4','市1','县1',109),\n" +
        "  ('省4','市2','县1',110))\n" +
        "AS t_person_num(pid, cid, xid, num)\n" +
        "GROUP BY pid;").execute().print();

```

|  | pid | total |
|--|-----|-------|
|  | 省1  | 303   |
|  | 省3  | 321   |
|  | 省2  | 312   |
|  | 省4  | 219   |

### 8.1.1. Group Set

- 在一个GROUP BY 查询中，根据不同的维度组合进行聚合。GROUPING SETS就是一种将多个 GROUP BY逻辑UNION在一起。GROUPING SETS会把在单个GROUP BY逻辑中没有参与GROUP BY的那一列置为NULL值。空分组集意味着所有行都聚合到一个组中

```

    o  tableEnvironment.sqlQuery("SELECT pid, cid, xid, sum(num) AS total\n" +
        "FROM (VALUES\n" +
        "  ('省1','市1','县1',100),\n" +
        "  ('省1','市2','县2',101),\n" +
        "  ('省1','市2','县1',102),\n" +
        "  ('省2','市1','县4',103),\n" +
        "  ('省2','市2','县1',104),\n" +
        "  ('省2','市2','县1',105),\n" +
        "  ('省3','市1','县1',106),\n" +
        "  ('省3','市2','县1',107),\n" +
        "  ('省3','市2','县2',108),\n" +
        "  ('省4','市1','县1',109),\n" +
        "  ('省4','市2','县1',110))\n" +
        "AS t_person_num(pid, cid, xid, num)\n" +
        "GROUP BY GROUPING SETS ((pid, cid, xid),(pid, cid),(pid),\n
        ()").execute().print();

```

|     | pid   | cid |
|-----|-------|-----|
| xid | total |     |
| 省1  |       | 市2  |
| 县2  | 101   |     |
|     | 省2    | 市2  |
| 县1  | 209   |     |

|  |               |  |        |
|--|---------------|--|--------|
|  | 省4            |  | 市1     |
|  | <NULL>   109  |  |        |
|  | 省1            |  | 市2     |
|  | <NULL>   203  |  |        |
|  | 省2            |  | 市1     |
|  | 县4   103      |  |        |
|  | 省3            |  | <NULL> |
|  | <NULL>   321  |  |        |
|  | 省3            |  | 市2     |
|  | <NULL>   215  |  |        |
|  | 省1            |  | 市1     |
|  | 县1   100      |  |        |
|  | 省1            |  | 市1     |
|  | <NULL>   100  |  |        |
|  | <NULL>        |  | <NULL> |
|  | <NULL>   1155 |  |        |
|  | 省1            |  | 市2     |
|  | 县1   102      |  |        |
|  | 省2            |  | 市1     |
|  | <NULL>   103  |  |        |
|  | 省3            |  | 市1     |
|  | <NULL>   106  |  |        |
|  | 省4            |  | 市1     |
|  | 县1   109      |  |        |
|  | 省4            |  | <NULL> |
|  | <NULL>   219  |  |        |
|  | 省3            |  | 市2     |
|  | 县1   107      |  |        |
|  | 省4            |  | 市2     |
|  | 县1   110      |  |        |
|  | 省1            |  | <NULL> |
|  | <NULL>   303  |  |        |
|  | 省3            |  | 市1     |
|  | 县1   106      |  |        |
|  | 省4            |  | 市2     |
|  | <NULL>   110  |  |        |
|  | 省2            |  | <NULL> |
|  | <NULL>   312  |  |        |
|  | 省2            |  | 市2     |
|  | <NULL>   209  |  |        |
|  | 省3            |  | 市2     |
|  | 县2   108      |  |        |

+-----+-----+-----+

23 rows in set

### 8.1.2. ROLLUP

- `ROLLUP` is a shorthand notation for specifying a common type of grouping set.
- It represents the given list of expressions and all prefixes of the list, including the empty list.
- ```
tableEnvironment.sqlQuery("SELECT pid, cid, xid, sum(num) AS total\n" +\n    "FROM (VALUES\n" +\n    " ('省1','市1','县1',100),\n    " ('省1','市2','县2',101)\n" +\n    " )T(pid,cid,xid,num)\n    GROUP BY ROLLUP (pid, cid, xid, pid, cid)\n    ORDER BY pid, cid, xid, num")
```

```

    " ('省1','市2','县1',102),\n" +
    " ('省2','市1','县4',103),\n" +
    " ('省2','市2','县1',104),\n" +
    " ('省2','市2','县1',105),\n" +
    " ('省3','市1','县1',106),\n" +
    " ('省3','市2','县1',107),\n" +
    " ('省3','市2','县2',108),\n" +
    " ('省4','市1','县1',109),\n" +
    " ('省4','市2','县1',110))\n" +
"AS t_person_num(pid, cid, xid, num)\n" +
"GROUP BY ROLLUP(pid, cid, xid)").execute().print();

```

	pid	xid	total	cid
		省1		市2
		县2	101	
				市2
		省2		
		县1	209	
				市1
		省4		
		<NULL>	109	
				市2
		省1		
		<NULL>	203	
				市1
		省2		
		县4	103	
				市1
		省3		<NULL>
		<NULL>	321	
				市2
		省3		
		<NULL>	215	
				市1
		省1		
		县1	100	
				市1
		省1		
		<NULL>	100	
				<NULL>
		<NULL>	1155	
		省1		市2
		县1	102	
				市1
		省2		
		<NULL>	103	
				市1
		省3		
		<NULL>	106	
				市1
		省4		
		县1	109	
				市1
		省4		<NULL>
		<NULL>	219	
		省3		市2
		县1	107	
				市2
		省4		
		县1	110	
		省1		<NULL>
		<NULL>	303	
				市1
		省3		
		县1	106	
				市2
		省4		
		<NULL>	110	

	省2	<NULL>	<NULL>
	<NULL>	312	
	省2		市2
	<NULL>	209	
	省3		市2
	县2	108	
-----+-----+-----+-----			
-----+-----+-----+-----			
23 rows in set			

### 8.1.3. CUBE

- CUBE is a shorthand notation for specifying a common type of grouping set.
- It represents the given list and all of its possible subsets - the power set.

```

• 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 ),
    (           )
  )
)

```

## 8.2. 开窗聚合

- 在标准 SQL 中还有另外一类比较特殊的聚合方式，可以针对每一行计算一个聚合值。
- 比如说，我们可以以每一行数据为基准，计算它之前 1 小时内所有数据的平均值；也可以计算它之前 10 个数的平均值。
- 就好像是在每一行上打开了一扇窗户、收集数据进行统计一样，这就是所谓的“开窗函数”。
- 开窗函数的聚合与之前两种聚合有本质的不同：
  - 分组聚合、窗口 TVF 聚合都是“多对一”的关系，将数据分组之后每组只会得到一个聚合结果；

- 而开窗函数是对每行都要做一次开窗聚合，因此聚合之后表中的行数不会有任何减少，是一个“多对多”的关系
- 与标准 SQL 中一致，Flink SQL 中的开窗函数也是通过 OVER 子句来实现的，所以有时开窗聚合也叫作“OVER 聚合”（Over Aggregation）。
- 基本语法如下：

```

○ SELECT
  <聚合函数> OVER (
    [PARTITION BY <字段 1>[, <字段 2>, ...]]
    ORDER BY <时间属性字段>
    <开窗范围>),
    ...
  FROM ...

```

- OVER 关键字前面是一个聚合函数，它会应用在后面 OVER 定义的窗口上。
- PARTITION BY (可选)
  - 用来指定分区的键（key），类似于 GROUP BY 的分组，这部分是可选的
- ORDER BY
  - OVER 窗口是基于当前行扩展出的一段数据范围，选择的标准可以基于时间也可以基于数量。
  - 数据都应该是以某种顺序排列好的；而表中的数据本身是无序的。
  - 在 Flink 的流处理中，目前只支持按照时间属性的升序排列，所以这里 ORDER BY 后面的字段必须是定义好的时间属性
- 开窗范围
  - 对于开窗函数而言，还有一个必须要指定的就是开窗的范围，也就是到底要扩展多少行来做聚合。
  - 这个范围是由 BETWEEN <下界> AND <上界> 来定义的，也就是“从下界到上界”的范围。
  - 目前支持的上界只能是 CURRENT ROW，也就是定义一个“从之前某一行到当前行”的范围
  - 开窗选择的范围可以基于时间，也可以基于数据的数量。所以开窗范围还应该在两种模式之间做出选择：
    - 范围间隔（RANGE intervals）
    - 行间隔（ROW intervals）
  - 范围间隔
    - 范围间隔以 RANGE 为前缀，就是基于 ORDER BY 指定的时间字段去选取一个范围，一般就是当前行时间戳之前的一段时间。
    - 例如开窗范围选择当前行之前 1 小时的数据：
    - RANGE BETWEEN INTERVAL '1' HOUR PRECEDING AND CURRENT ROW
  - 行间隔
    - 行间隔以 ROWS 为前缀，就是直接确定要选多少行，由当前行出发向前选取就可以了。
    - 例如开窗范围选择当前行之前的 5 行数据
    - ROWS BETWEEN 5 PRECEDING AND CURRENT ROW

```

○ public static void main(String[] args) {
  //执行环境
  StreamExecutionEnvironment environment =
  StreamExecutionEnvironment.getExecutionEnvironment();
  environment.setParallelism(1);

```

```

StreamTableEnvironment tableEnvironment =
StreamTableEnvironment.create(environment);

//执行SQL
tableEnvironment.executeSql("CREATE TABLE t_goods (\n" +
    "  gid STRING,\n" +
    "  type INT,\n" +
    "  price INT,\n" +
    "  ts AS localtimestamp,\n" +
    "  WATERMARK FOR ts AS ts - INTERVAL '5'
SECOND\n" +
") WITH (\n" +
"  'connector' = 'datagen',\n" +
"  'rows-per-second'='1',\n" +
"  'fields.gid.length'='10',\n" +
"  'fields.type.min'='1',\n" +
"  'fields.type.max'='5',\n" +
"  'fields.price.min'='1',\n" +
"  'fields.price.max'='9'\n" +
")");

// tableEnvironment.sqlQuery("select * from
t_goods").execute().print();

//开窗聚合计算--时间范围
// tableEnvironment.sqlQuery("select t.* ,avg(price) OVER(" +
//     "PARTITION BY type " +
//     "ORDER BY ts " +
//     "RANGE BETWEEN INTERVAL '10' SECONDS PRECEDING AND
CURRENT ROW)" +
//     " from t_goods t").execute().print();

//开窗聚合计算--计数范围
tableEnvironment.sqlQuery("select t.* ,avg(price) OVER(" +
    "PARTITION BY type " +
    "ORDER BY ts " +
    "ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
)" +
    " from t_goods t").execute().print();
}

```

## 8.3. TopN

### 8.3.1. 普通TopN

- 在 Flink SQL 中，是通过 OVER 聚合和一个条件筛选来实现 Top N 的。
- 基本语法如下：

- ```

SELECT ...
FROM (
    SELECT ...,
    ROW_NUMBER() OVER (
        [PARTITION BY <字段 1>[, <字段 1>...]]
        ORDER BY <排序字段 1> [asc|desc][, <排序字段 2> [asc|desc]...]
    ) AS row_num
    FROM ...
WHERE row_num <= N [AND <其它条件>]

```

- 利用 ROW\_NUMBER()函数为每一行数据聚合得到一个排序之后的行号。
- 行号重命名为 row\_num，并在外层的查询中以row\_num <= N 作为条件进行筛选，就可以得到根据排序字段统计的 Top N 结果了
- Flink SQL专门用 OVER 聚合做了优化实现。所以只有在 Top N 的应用场景中，OVER 窗口 ORDER BY后才可以指定其它排序字段；而要想实现 Top N，就必须按照上面的格式进行定义，否则 Flink SQL 的优化器将无法正常解析。而且，目前 Table API 中并不支持 ROW\_NUMBER()函数，所以也只有 SQL 中这一种通用的 Top N 实现方式

- ```

public static void main(String[] args) {
    //执行环境
    StreamExecutionEnvironment environment =
    StreamExecutionEnvironment.getExecutionEnvironment();
    environment.setParallelism(1);
    StreamTableEnvironment tableEnvironment =
    StreamTableEnvironment.create(environment);

    //执行SQL
    tableEnvironment.executeSql("CREATE TABLE t_goods (\n" +
        "    gid STRING,\n" +
        "    type INT,\n" +
        "    price INT,\n" +
        "    ts AS localtimestamp,\n" +
        "    WATERMARK FOR ts AS ts - INTERVAL '5'
SECOND\n" +
        ") WITH (\n" +
        "    'connector' = 'datagen',\n" +
        "    'rows-per-second'='1',\n" +
        "    'fields.gid.length'='10',\n" +
        "    'fields.type.min'='1',\n" +
        "    'fields.type.max'='1',\n" +
        "    'fields.price.min'='100',\n" +
        "    'fields.price.max'='999'\n" +
        ")");
    // tableEnvironment.sqlQuery("select * from
    t_goods").execute().print();

    //排序开窗函数--所有数据的排序
    tableEnvironment.sqlQuery("select * from (" +
        "    select *, ROW_NUMBER() OVER (" +
        "        PARTITION BY type " +
        "        ORDER BY price desc " +
        "    ) AS rownum from t_goods" +
        ") WHERE rownum <= 3
") .execute().print();

```

```
}
```

### 8.3.2. 窗口Top N

```
• public static void main(String[] args) throws Exception {
    // 1. 创建一个表执行环境
    StreamExecutionEnvironment env =
    StreamExecutionEnvironment.getExecutionEnvironment();
    StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

    env.setParallelism(1);
    env.getConfig().setAutoWatermarkInterval(100); // 100毫秒生成一次水位线

    singleOutputStreamOperator<Event> streamOperator = env.addSource(new
clickSource())
        // 乱序流的WaterMark生成
        .assignTimestampsAndWatermarks(watermarkStrategy
            <Event>forBoundedOutOfOrderliness(Duration.ofSeconds(2)) // 延迟2秒保证数据正确
            .withTimestampAssigner(new
SerializableTimestampAssigner<Event>() {
                @Override // 时间戳的提取器
                public long
extractTimestamp(Event event, long l) {
                    return event.getTimestamp();
                }
            });
    }

    Table clickTable = tableEnv.fromDataStream(streamOperator, $("user"),
$("url"), $("timestamp").as("ts"))
        , $("et").rowtime());
    // 将表注册到表环境中
    tableEnv.createTemporaryView("clickTable",clickTable);

    /**
     * 第一步 根据每个用户分组求出其访问量 并且设置窗口 A = select user,
     count(url) as cnt, window_start, window_end from table( tumble(table
     clickTable, DESCRIPTOR(et), INTERVAL '10' SECOND) ) group by user,
     window_start, window_end;
     * 第二步 row_number排名 并根据窗口分组 B = select *, row_number() over
     ( partition by window_start, window_end order by cnt DESC) as rank_num from
     ( A );
     * 第三步 where过滤 select user, cnt, rank_num from B where rank_num
     <= 3;
     */
}

String subQuery = "select user, count(url) as cnt, window_start,
window_end " +
    "from table( tumble(table clickTable, DESCRIPTOR(et), INTERVAL '10'
SECOND) ) " +
    "group by user, window_start, window_end";

Table table = tableEnv.sqlQuery("select user, cnt, rank_num from ( " +
```

```

        "select *, row_number() over ( partition
        by window_start, window_end order by cnt DESC) as rank_num " +
        "from ( " + subQuery + " ) ) " +
        "where rank_num <= 3");

    tableEnv.toChangelogStream(table).print();

    env.execute();
}

```

## 8.4. Join

- 与标准 SQL 一致, Flink SQL 的常规联结也可以分为内联结 (INNER JOIN) 和外联结 (OUTER JOIN), 区别在于结果中是否包含不符合联结条件的行。
- 目前仅支持“等值条件”作为联结条件, 也就是关键字 ON 后面必须是判断两表中字段相等的逻辑表达式

### 8.4.1. 等值内联结

- 内联结用 INNER JOIN 来定义, 会返回两表中符合联接条件的所有行的组合, 也就是所谓的笛卡尔积 (Cartesian product)。
- 目前仅支持等值联结条件

```

• SELECT *
  FROM Order
  INNER JOIN Product
  ON Order.product_id = Product.id

```

### 8.4.2. 等值外联结

- 与内联结类似, 外联结也会返回符合联结条件的所有行的笛卡尔积; 另外, 还可以将某一侧表中找不到任何匹配的行也单独返回。
- Flink SQL 支持左外 (LEFT JOIN)、右外 (RIGHT JOIN) 和全外 (FULL OUTER JOIN)

### 8.4.3. 间隔联结查询

- 两条流的 Join 就对应着 SQL 中两个表的 Join, 这是流处理中特有的联结方式。
- 目前 Flink SQL 还不支持窗口联结, 而间隔联结则已经实现
- 间隔联结 (Interval Join) 返回的, 同样是符合约束条件的两条中数据的笛卡尔积。只不过这里的“约束条件”除了常规的联结条件外, 还多了一个时间间隔的限制。
- 具体语法有以下要点:
  - 间隔联结不需要用 JOIN 关键字, 直接在 FROM 后将要联结的两表列出来就可以, 用逗号分隔。这与标准 SQL 中的语法一致, 表示一个“交叉联结” (Cross Join), 会返回两表中所有行的笛卡尔积
  - 联结条件用 WHERE 子句来定义, 用一个等值表达式描述。交叉联结之后再用 WHERE 进行条件筛选, 效果跟内联结 INNER JOIN ... ON ... 非常类似, 我们可以在 WHERE 子句中, 联结条件后用 AND 追加一个时间间隔的限制条件;
  - 做法是提取左右两侧表中的时间字段, 然后用一个表达式来指明两者需要满足的间隔限制。
- 具体定义方式有下面三种, 这里分别用 ltime 和 rtime 表示左右表中的时间字段

- ltime = rtime
- ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE
- ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND
- 例如，我们现在除了订单表 Order 外，还有一个“发货表”Shipment，要求在收到订单后四个小时内发货。那么我们就可以用一个间隔联结查询，把所有订单与它对应的发货信息连接合并在一起返回

- ```
SELECT *
FROM Order o, Shipment s
WHERE o.id = s.order_id
AND o.order_time BETWEEN s.ship_time - INTERVAL '4' HOUR AND
s.ship_time
```

## 9. FlinkSQL Client

### 9.1. 基本介绍

- Flink提供了SQL client，有了它我们可以像Hive的beeline一样直接在控制台编写SQL并提交作业。
- Flink SQL client支持运行在standalone集群和Yarn集群上。提交任务的命令有所不同。
- standalone集群

- ```
//启动集群
./start-cluster.sh

//启动客户端
./sql-client.sh embedded
```

-

```
[root@node01 ~]# sql-client.sh embedded

BETA
Flink SQL> [REDACTED]
Welcome! Enter 'HELP;' to list all available commands. 'QUIT;' to exit.
Command history file path: /root/.flink-sql-history
Flink SQL> [REDACTED]
Flink SQL> help;
The following commands are available:
○
HELP Prints the available commands.
QUIT/EXIT Quits the SQL CLI client.
CLEAR Clears the current terminal.
SET Sets a session configuration property. Syntax: "SET '<key>=<value>';". Use "SET;" for listing all properties.
RESET Resets a session configuration property. Syntax: "RESET '<key>';". Use "RESET;" for reset all session properties.
INSERT INTO Inserts the results of a SQL SELECT query into a declared table sink.
INSERT OVERWRITE Inserts the results of a SQL SELECT query into a declared table sink and overwrite existing data.
SELECT Executes a SQL SELECT query on the Flink cluster.
EXPLAIN Describes the execution plan of a query or table with the given name.
BEGIN STATEMENT SET Begins a statement set. Syntax: "BEGIN STATEMENT SET;".
END Ends a statement set. Syntax: "END;".
ADD JAR Adds the specified jar file to the submitted jobs' classloader. Syntax: "ADD JAR '<path_to_filename>.jar'".
REMOVE JAR Removes the specified jar file from the submitted jobs' classloader. Syntax: "REMOVE JAR '<path_to_filename>.jar'".
SHOW JARS Shows the list of user-specified jar dependencies. This list is impacted by the --jar and --library startup options as well as the ADD/REMOVE JAR commands.

Hint: Make sure that a statement ends with ";" for finalizing (multi-line) statements.
You can also type any Flink SQL statement, please visit https://nightlies.apache.org/flink/flink-docs-stable/docs/dev/table/sql/overview/ for more details.
```

- Yarn集群

- Flink每次启动 `yarn-session`，都会创建一个 `/tmp/.yarn-properties-root` 文件(root为用户名，源码位于 `FlinkYarnSessionCli`)
- INFO org.apache.flink.yarn.cli.FlinkYarnSessionCli [] - Found Yarn properties file under `/tmp/.yarn-properties-root`.
- 记录了最近一次提交的yarn session对应的application ID。
- 额外注意，启动Yarn session和SQL client必须使用相同的用户。

```
//启动YarnSession模式
yarn-session.sh -n 3 -jm 1024 -tm 1024

//启动客户端
./sql-client.sh embedded -s yarn-session
```

## 9.2. 安装依赖

- 需要将项目中用到的依赖放入Flink安装位置的lib目录下。
  - 例如：
    - `flink-csv-1.13.2.jar`: CSV格式支持

- flink-connector-jdbc\_2.11-1.13.2.jar: 读写JDBC支持
- flink-connector-kafka\_2.11-1.13.2.jar: 读写Kafka支持
- kafka-clients-2.4.1.jar: Kafka客户端
- mysql-connector-java-8.0.29.jar: MySQL JDBC驱动
- 程序员根据需求酌情添加jars...

- 配置Hadoop路径

- `export HADOOP_CLASSPATH=`hadoop classpath``

- 配置数据格式

- ```
# 在专门的界面展示, 使用分页table格式。可按照界面下方说明, 使用快捷键前后翻页和退出到
SQL命令行
SET sql-client.execution.result-mode = table;

# changelog格式展示, 可展示数据增(I)删(D)改(U)
SET sql-client.execution.result-mode = changelog;

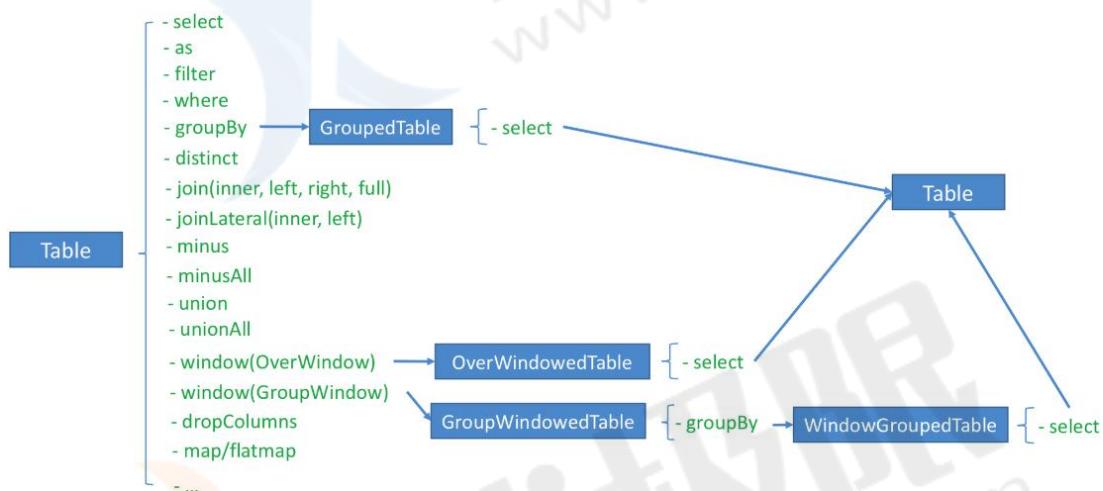
# 接近传统数据库的展示方式, 不使用专门界面
SET sql-client.execution.result-mode = tableau;
```

## 10. FlinkSQL 官方文档

本章节要求: 《常见会用, 陌生会查》

### 10.1. TableApi

#### Table API operations



- <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/dev/table/tableapi/>
- Table API 是批处理和流处理的统一的关系型 API。Table API 的查询不需要修改代码就可以采用批输入或流输入来运行。Table API 是 SQL 语言的超集，并且是针对 Apache Flink 专门设计的。Table API 集成了 Scala, Java 和 Python 语言的 API。Table API 的查询是使用 Java, Scala 或 Python 语言嵌入的风格定义的，有诸如自动补全和语法校验的 IDE 支持，而不是像普通 SQL 一样使用字符串类型的值来指定查询。

[概述 & 示例](#)

[Operations](#)

[Scan, Projection, and Filter](#)

[列操作](#)

[Aggregations](#)

[Joins](#)

[Set Operations](#)

[OrderBy, Offset & Fetch](#)

[Insert](#)

[Group Windows](#)

[Over Windows](#)

[Row-based Operations](#)

## 10.2. SQL

- <https://nightlies.apache.org/flink/flink-docs-release-1.15/zh/docs/dev/table/sql/overview/>
- 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。
- 目前 Flink SQL 所支持的所有语句：
  - [SELECT \(Queries\)](#)
  - [CREATE TABLE, CATALOG, DATABASE, VIEW, FUNCTION](#)
  - [DROP TABLE, DATABASE, VIEW, FUNCTION](#)
  - [ALTER TABLE, DATABASE, FUNCTION](#)
  - [ANALYZE TABLE](#)
  - [INSERT](#)
- - [SQL HINTS](#)
  - [DESCRIBE](#)
  - [EXPLAIN](#)
  - [USE](#)
  - [SHOW](#)
  - [LOAD](#)
  - [UNLOAD](#)
- [Queries\[目前 Flink SQL 所支持的所有查询语句：\]](#)

# Operations

- [WITH clause](#)
- [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](#)

## 11. FlinkSQL 函数

- 在 SQL 中，我们可以把一些数据的转换操作包装起来，嵌入到 SQL 查询中统一调用，这就是“函数”(functions)。
- Flink 的 Table API 和 SQL 同样提供了函数的功能。两者在调用时略有不同：
  - Table API 中的函数是通过数据对象的方法调用来实现的；
  - SQL 则是直接引用函数名称，传入数据作为参数
- 由于 Table API 是内嵌在 Java 语言中的，很多方法需要在类中额外添加，因此扩展功能比较麻烦，目前支持的函数比较少；而且 Table API 也不如 SQL 的通用性强，所以一般情况下较少使用。

### 11.1. 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。
  - 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

- 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`。
  - 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`。

## 11.2. 系统函数

- Flink Table API & SQL 为用户提供了一组内置的数据转换函数。
- 系统函数 (System Functions) 也叫内置函数 (Built-in Functions)，是在系统中预先实现好的功能模块。可以通过固定的函数名直接调用，实现想要的转换操作。
- Flink SQL 提供了大量的系统函数，几乎支持所有的标准 SQL 中的操作，这为我们使用 SQL 编写流处理程序提供了极大的方便。
- Flink SQL 中的系统函数又主要可以分为两大类：标量函数 (Scalar Functions) 和聚合函数 (Aggregate Functions)
- 函数分类
  - 标量函数
    - 比较函数
    - 逻辑函数
    - 算术函数
    - 字符串函数
    - 时间函数
    - 条件函数
    - 类型转换函数
    - 集合函数
    - JSON Functions
    - 值构建函数
    - 值获取函数
    - 分组函数
    - 哈希函数
    - 辅助函数
  - 聚合函数
  - 时间间隔单位和时间点单位标识符
  - 列函数

### 11.2.1. 标量函数

- 标量函数将零、一个或多个值作为输入并返回单个值作为结果。
- <https://nightlies.apache.org/flink/flink-docs-release-1.16/zh/docs/dev/table/functions/systemfunctions/#%E6%A0%87%E9%87%8F%E5%87%BD%E6%95%B0>

### 11.2.2. 聚合函数

- 聚合函数将所有的行作为输入，并返回单个聚合值作为结果。
- 聚合函数是以表中多个行作为输入，提取字段进行聚合操作的函数，会将唯一的聚合值作为结果返回。

- 聚合函数应用非常广泛，不论分组聚合、窗口聚合还是开窗（Over）聚合，对数据的聚合操作都可以用相同的函数来定义。
- <https://nightlies.apache.org/flink/flink-docs-release-1.16/zh/docs/dev/table/functions/systemfunctions/#%E8%81%9A%E5%90%88%E5%87%BD%E6%95%B0>

## 11.3. 自定义函数

- 系统函数尽管庞大，也不可能涵盖所有的功能；如果有系统函数不支持的需求，我们就需要用自定义函数（User Defined Functions, UDF）来实现了
- Flink 的 Table API 和 SQL 提供了多种自定义函数的接口，以抽象类的形式定义。
- 当前 UDF主要有以下几类：
  - 标量函数（Scalar Functions）：将输入的标量值转换成一个新的标量值
  - 表函数（Table Functions）：将标量值转换成一个或多个新的行数据，也就是扩展成一个表
  - 聚合函数（Aggregate Functions）：将多行数据里的标量值转换成一个新的标量值
  - 表聚合函数（Table Aggregate Functions）：将多行数据里的标量值转换成一个或多个新的行数据

### 11.3.1. UDF标量函数

- 从输入和输出表中行数据的对应关系看，标量函数是“一对一”的转换
- 自定义标量函数可以把 0 个、1 个或多个标量值转换成一个标量值，它对应的输入是一行数据中的字段，输出则是唯一的值。
- 自定义方式：
  - 需要自定义一个类来继承抽象类 ScalarFunction，并实现叫作 eval() 的求值方法。
  - 标量函数的行为就取决于求值方法的定义，它必须是公有的（public），而且名字必须是 eval。
  - 求值方法 eval 可以重载多次，任何数据类型都可作为求值方法的参数和返回值类型，写完后将类注册到表环境就可以直接在SQL 中调用了
- 代码实现：

```

o import org.apache.flink.table.annotation.InputGroup;
import org.apache.flink.table.api.*;
import org.apache.flink.table.functions.scalarFunction;
import static org.apache.flink.table.api.Expressions.*;

public static class HashFunction extends ScalarFunction {

    // 接受任意类型输入，返回 INT 型输出
    public int eval(@DataTypeHint(inputGroup = InputGroup.ANY) object o)
    {
        return o.hashCode();
    }
}

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(HashFunction.class, $("myField")));

// 注册函数

```

```

env.createTemporarySystemFunction("HashFunction", HashFunction.class);

// 在 Table API 里调用注册好的函数
env.from("MyTable").select(call("HashFunction", $("myField")));

// 在 SQL 里调用注册好的函数
env.sqlQuery("SELECT HashFunction(myField) FROM MyTable");

```

### 11.3.2. UDF表值函数

- 使用表函数，可以对一行数据拆分得到一个表，和 Hive 中的 UDTF 非常相似。
- 自定义方式：
  - 要实现自定义的表函数，需要自定义类来继承抽象类 TableFunction，内部必须要实现的也是一个名为 eval 的求值方法。
  - 与标量函数不同的是，TableFunction 类本身是有一个泛型参数T的，这就是表函数返回数据的类型；
  - 而 eval()方法没有返回类型，内部也没有 return语句，是通过调用 collect()方法来发送想要输出的行数据的
- 代码实现：

```

import org.apache.flink.table.annotation.DataTypeHint;
import org.apache.flink.table.annotation.FunctionHint;
import org.apache.flink.table.api.*;
import org.apache.flink.table.functions.TableFunction;
import org.apache.flink.types.Row;
import static org.apache.flink.table.api.Expressions.*;

@FunctionHint(output = @DataTypeHint("ROW<word STRING, length INT>"))
public static class SplitFunction extends TableFunction<Row> {

    public void eval(String str) {
        for (String s : str.split(" ")) {
            // use collect(...) to emit a row
            collect(Row.of(s, s.length()));
        }
    }
}

TableEnvironment env = TableEnvironment.create(...);

// 在 Table API 里不经注册直接“内联”调用函数
env
    .from("MyTable")
    .joinLateral(call(SplitFunction.class, $("myField")))
    .select($("myField"), $("word"), $("length"));
env
    .from("MyTable")
    .leftOuterJoinLateral(call(splitFunction.class, $("myField")))
    .select($("myField"), $("word"), $("length"));

// 在 Table API 里重命名函数字段
env
    .from("MyTable")
    .leftOuterJoinLateral(call(splitFunction.class,
        $("myField")).as("newword", "newLength"))
    .select($("myField"), $("newword"), $("newLength"));

```

```

// 注册函数
env.createTemporarySystemFunction("SplitFunction",
SplitFunction.class);

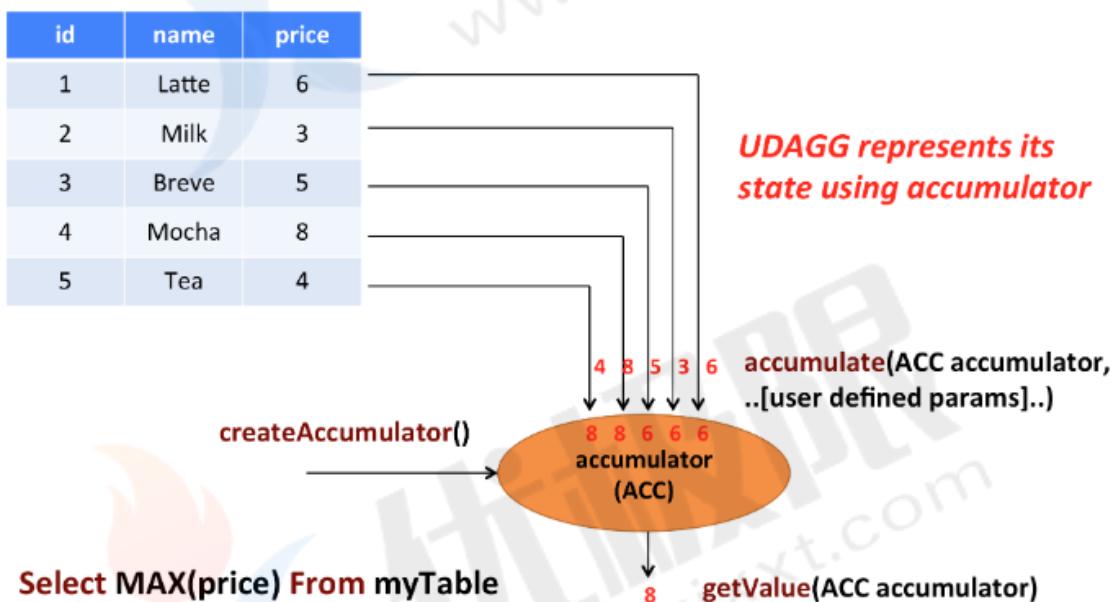
// 在 Table API 里调用注册好的函数
env
    .from("MyTable")
    .joinLateral(call("SplitFunction", ${"myField"}))
    .select(${"myField"}, ${"word"}, ${"length"});
env
    .from("MyTable")
    .leftOuterJoinLateral(call("SplitFunction", ${"myField"}))
    .select(${"myField"}, ${"word"}, ${"length"});

// 在 SQL 里调用注册好的函数
env.sqlQuery(
    "SELECT myField, word, length " +
    "FROM MyTable, LATERAL TABLE(SplitFunction(myField))");
env.sqlQuery(
    "SELECT myField, word, length " +
    "FROM MyTable " +
    "LEFT JOIN LATERAL TABLE(SplitFunction(myField)) ON TRUE");

// 在 SQL 里重命名函数字段
env.sqlQuery(
    "SELECT myField, newword, newLength " +
    "FROM MyTable " +
    "LEFT JOIN LATERAL TABLE(SplitFunction(myField)) AS T(newword,
newLength) ON TRUE");

```

### 11.3.3. UDF聚合函数



- 用户自定义聚合函数 (User Defined AGGRegate function, UDAGG) 会把一行或多行数据 (也就是一个表) 聚合成一个标量值。这是一个标准的“多对一”的转换
- 自定义方式:
  - 自定义聚合函数需要继承抽象类 AggregateFunction。

- AggregateFunction 有两个泛型参数<T, ACC>, T 表示聚合输出的结果类型, ACC 则表示聚合的中间状态类型
- 每个 AggregateFunction 都 必须 实现以下几个方法:
  - createAccumulator()
    - 这是创建累加器的方法。没有输入参数, 返回类型为累加器类型 ACC
  - accumulate()
    - 这是进行聚合计算的核心方法, 每来一行数据都会调用。它的第一个参数是确定的, 就是当前的累加器, 类型为 ACC, 表示当前聚合的中间状态;
    - 后面的参数则是聚合函数调用时传入的参数, 可以有多个, 类型也可以不同。这个方法主要是更新聚合状态, 所以没有返回类型
  - getValue()
    - 这是得到最终返回结果的方法。输入参数是 ACC 类型的累加器, 输出类型为 T。在遇到复杂类型时, Flink 的类型推导可能会无法得到正确的结果。
    - 所以AggregateFunction也可以专门对累加器和返回结果的类型进行声明, 这是通过 getAccumulatorType()和 getResultType()两个方法来指定的
- 代码实现

```

◦ import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.table.annotation.DataTypeHint;
import org.apache.flink.table.annotation.FunctionHint;
import org.apache.flink.table.functions.AggregateFunction;

/**
 * @Description :
 * @School:优极限学堂
 * @official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class AggregateFunction4Order2WeightPrice extends
AggregateFunction<Double, Tuple2<Integer, Integer>> {

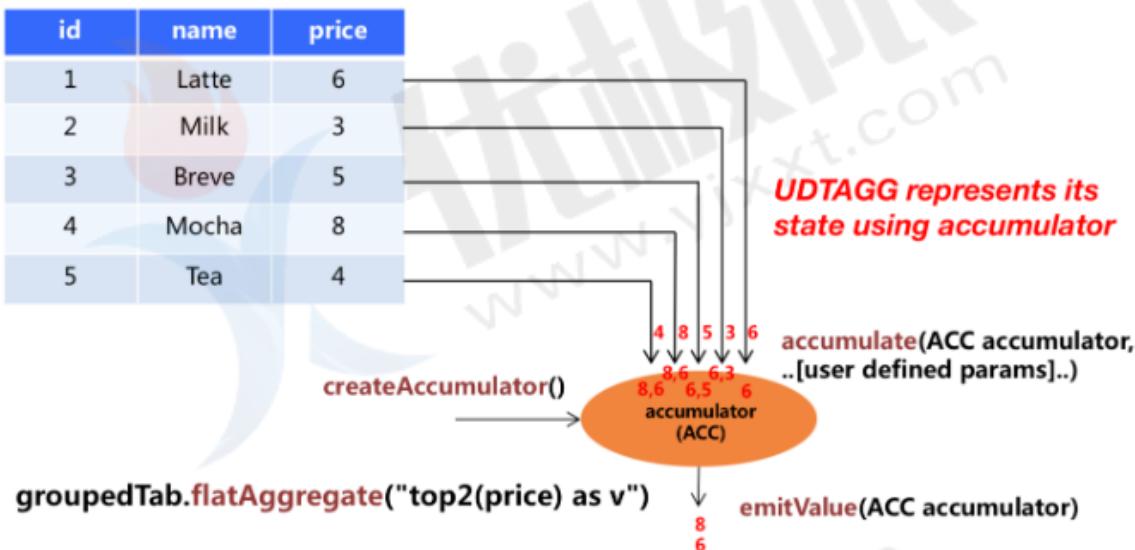
    @Override
    public Tuple2<Integer, Integer> createAccumulator() {
        //Tuple2.of(总销售额, 总重量)
        return Tuple2.of(0, 0);
    }

    @FunctionHint(
        input = {@DataTypeHint("INT"), @DataTypeHint("INT")})
    public void accumulate(Tuple2<Integer, Integer> acc, int weight,
int price) {
        acc.f0 += weight * price;
        acc.f1 += weight;
    }

    @Override
    public Double getValue(Tuple2<Integer, Integer> accumulator) {
        if (accumulator.f1 == 0) {
            return 0.0;
        }
        return accumulator.f0 * 1.0 / accumulator.f1;
    }
}

```

### 11.3.4. UDF表值聚合函数



- 用户自定义表聚合函数 (UDTAGG) 可以把一行或多行数据 (也就是一个表) 聚合成另一张表，结果表中可以有多行多列。
- 自定义方式：
  - 自定义表聚合函数需要继承抽象类 TableAggregateFunction。TableAggregateFunction 的结构和原理与 AggregateFunction 非常类似，同样有两个泛型参数<T, ACC>，用一个 ACC 类型的累加器 (accumulator) 来存储聚合的中间结果。聚合函数中必须实现的三个方法，在 TableAggregateFunction 中也必须对应实现：
    - createAccumulator()
      - 创建累加器的方法，与 AggregateFunction 中用法相同
    - accumulate()
      - 聚合计算的核心方法，与 AggregateFunction 中用法相同
    - emitValue()
      - 所有输入行处理完成后，输出最终计算结果的方法。
      - 这个方法对应着 AggregateFunction 中的 getValue() 方法；区别在于 emitValue 没有输出类型，而输入参数有两个：
        - 第一个是 ACC类型的累加器
        - 第二个则是用于输出数据的“收集器”out，它的类型为 Collect。
      - 所以很明显，表聚合函数输出数据不是直接 return，而是调用 out.collect()方法，调用多次就可以输出多行数据了；这一点与表函数非常相似。另外， emitValue() 在抽象类中也没有定义，无法 override，必须手动实现
- 代码实现

```

○ /**
 * Accumulator for Top2.
 */
public class Top2Accum {
    public Integer first;
    public Integer second;
}
  
```

```
/**  
 * 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;  
        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));  
        }  
    }  
}  
  
// 注册函数  
StreamTableEnvironment tEnv = ...  
tEnv.registerFunction("top2", new Top2());  
  
// 初始化表  
Table tab = ...;  
  
// 使用函数  
tab.groupBy("key")  
.flatAggregate("top2(a) as (v, rank)")  
.select("key, v, rank");
```

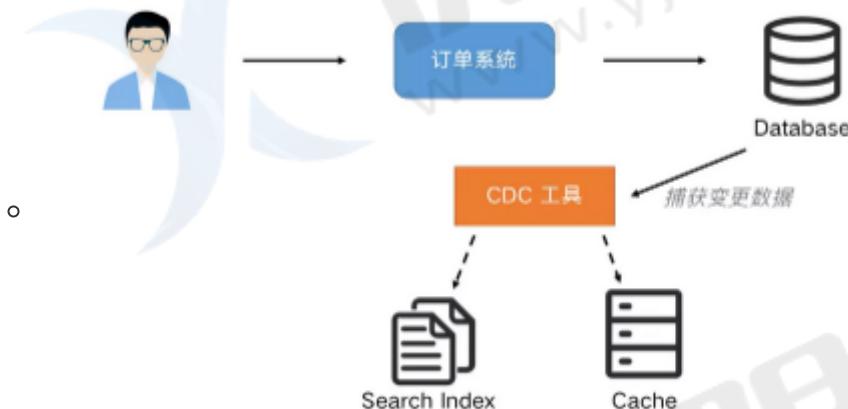
## 12. FlinkSQL CDC

### 12.1. CDC框架

- CDC, Change Data Capture, 变动数据获取的简称，使用CDC咱们能够从数据库中获取已提交的更改并将这些更改发送到下游，供下游使用。
- 业务场景
  - 业务系统经常会遇到需要更新数据到多个存储的需求。例如：
  - 一个订单系统刚刚开始只需要写入数据库即可完成业务使用。某天 BI 团队期望对数据库做全文索引，于是我们同时要写多一份数据到 ES 中，改造后一段时间，又有需求需要写入到 Redis 缓存中。



- 很明显这种模式是不可持续发展的，这种双写到各个数据存储系统中可能导致不可维护和扩展，数据一致性问题等，需要引入分布式事务，成本和复杂度也随之增加。我们可以通过 CDC (Change Data Capture) 工具进行解除耦合，同步到下游需要同步的存储系统。



- 实现方式
  - 业界主要有基于查询的 CDC 和基于日志的 CDC
  -

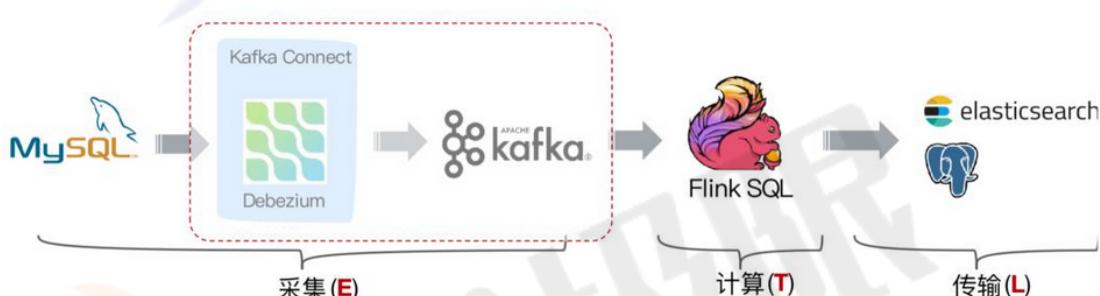
|                       | 基于查询的 CDC                                     | 基于日志的 CDC                                 |
|-----------------------|---|---|
| 概念                    | 每次捕获变更发起 Select 查询<br>进行全表扫描，过滤出查询之间<br>变更的数据 | 读取数据存储系统的 log，例如<br>MySQL 里面的 binlog 持续监控 |
| 开源产品                  | Sqoop, Kafka JDBC Source                      | Canal, Maxwell, Debezium                  |
| 执行模式                  | Batch   | Streaming                                 |
| 捕获所有数据的变化             | ✗   | ✓   |
| 低延迟，不增加数据库负载          | ✗   | ✓   |
| 不侵入业务 (LastUpdated字段) | ✗   | ✓   |
| 捕获删除事件和旧记录的状态         | ✗   | ✓   |
| 捕获旧记录的状态              | ✗   | ✓   |

- 基于日志 CDC 有以下这几种优势：

- 能够捕获所有数据的变化，捕获完整的变更记录。在异地容灾，数据备份等场景中得到广泛应用，如果是基于查询的 CDC 有可能导致两次查询的中间一部分数据丢失
- 每次 DML 操作均有记录无需像查询 CDC 这样发起全表扫描进行过滤，拥有更高的效率和性能，具有低延迟，不增加数据库负载的优势
- 无需入侵业务，业务解耦，无需更改业务模型
- 捕获删除事件和捕获旧记录的状态，在查询 CDC 中，周期的查询无法感知中间数据是否删除

## 12.2. Flink CDC

- 在以前的数据同步中，比如我们想实时获取数据库的数据，一般采用的架构就是采用第三方工具，比如canal、debezium等，实时采集数据库的变更日志，然后将数据发送到kafka等消息队列。然后再通过其他的组件，比如flink、spark等等来消费kafka的数据，计算之后发送到下游系统。



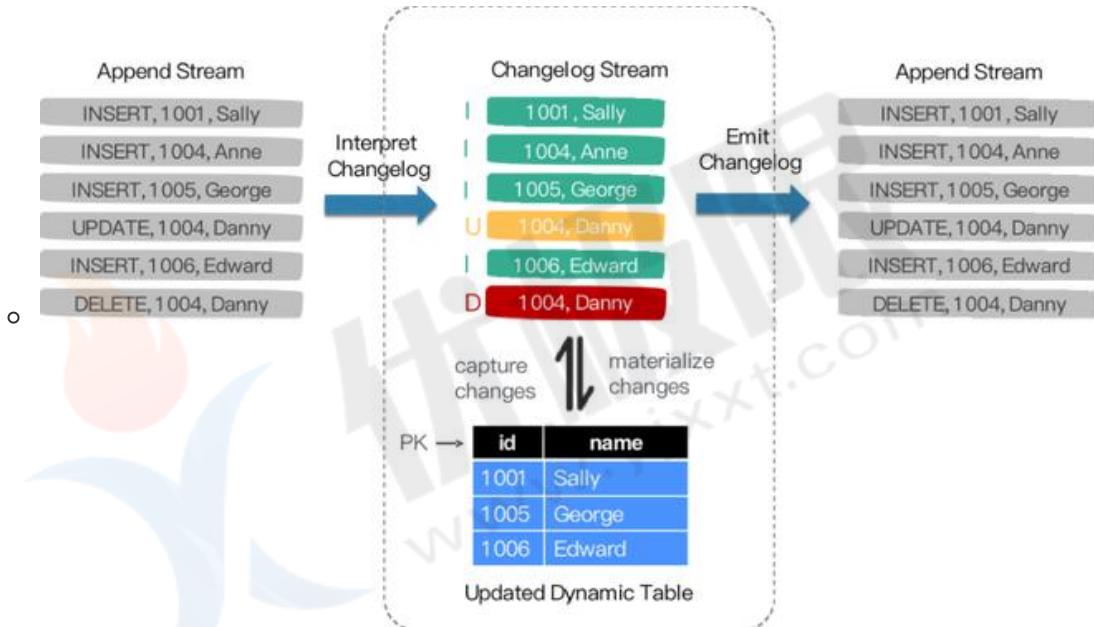
- 以前架构，我们需要部署canal (debezium) + kafka，然后flink再从kafka消费数据，这种架构下我们需要部署多个组件并且数据也需要落地到kafka
- 于是Flink提供了 cdc connector
-



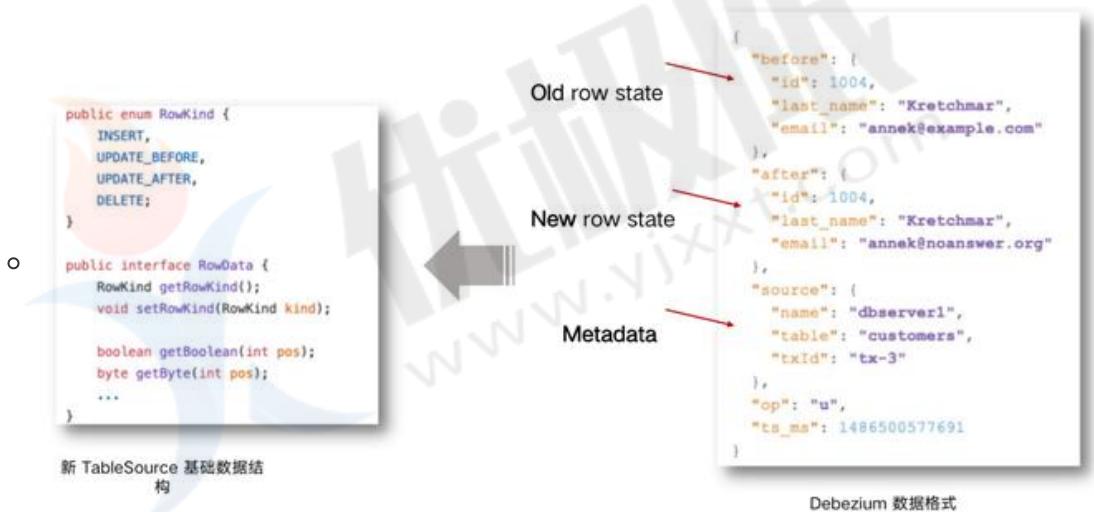
- 新架构下flink直接消费数据库的增量日志，替代了原来的数据采集层，然后直接对数据进行计算，最后将计算结果发送到下游。
- 工作原理
  - 启动MySQL CDC源时，它将获取一个全局读取锁（FLUSH TABLES WITH READ LOCK），该锁将阻止其他数据库的写入。然后，它读取当前binlog位置以及数据库和表的schema之后，将释放全局读取锁。然后，它扫描数据库表并从先前记录的位置读取binlog。Flink将定期执行checkpoints以记录binlog位置。如果发生故障，作业将重新启动并从checkpoint完成的binlog位置恢复。因此，它保证了仅一次的语义。
  - 如果未授予MySQL用户RELOAD权限，则MySQL CDC源将改为使用表级锁，并使用此方法执行快照。**这会阻止写入更长的时间。**
  - 全局读取锁在读取binlog位置和schema期间保持。这可能需要**几秒钟**，具体取决于表的数量。全局读取锁定会**阻止写入**，因此它仍然**可能影响在线业务**。
  - **如果要跳过读取锁，并且可以容忍至少一次语义**，则可以添加'dbezahlum.snapshot.locking.mode' = 'none'选项以跳过锁。
  - 每个用于读取binlog的MySQL数据库客户端都应具有唯一的ID，称为server id。MySQL服务器将使用此ID维护网络连接和binlog位置。如果不同的作业共享相同的server id，则可能导致从错误的binlog位置进行读取。
  - 提示：默认情况下，启动TaskManager时，server id是随机的。如果TaskManager失败，则再次启动时，它可能具有不同的server id。但这不应该经常发生（作业异常不会重新启动TaskManager），也不会对MySQL服务器造成太大影响。
  - Mysql的binlog可以说是针对库级别，所以相同的server id去拉一个库里的不同表或者相同表可能会造成数据丢失。所以建议设置server id。
- flink-cdc-connectors 可以用来替换 Debezium+Kafka 的数据采集模块，从而实现 Flink SQL 采集+计算+传输（ETL）一体化，优点如下：
  - 开箱即用，简单易上手
  - 减少维护的组件，简化实时链路，减轻部署成本
  - 减小端到端延迟
  - Flink 自身支持 Exactly Once 的读取和计算
  - 数据不落地，减少存储成本
  - 支持全量和增量流式读取
  - binlog 采集位点可回溯

### 12.3. ChangeLog

- 在 Flink 1.11 里面重构了 TableSource 接口，Flink SQL 内部支持了完整的 changelog 机制，所以 Flink 对接 CDC 数据只需要把CDC 数据转换成 Flink 认识的数据，以便更好支持和集成 CDC。



- 重构后的 TableSource 输出的都是 RowData 数据结构，代表了一行的数据。在 RowData 上面会有一个元数据的信息，我们称为 RowKind。
- RowKind 里面包括了插入、更新前、更新后、删除，这样和数据库里面的 binlog 概念十分类似。
- 通过 Debezium 采集的 JSON 格式，包含了旧数据和新数据行以及原数据信息，对接 Debezium JSON 的数据，其实就是将这种原始的 JSON 数据转换成 Flink 认识的 RowData。



## 12.4. Mysql CDC

- 代码实现
  - <https://github.com/ververica/flink-cdc-connectors>
  -

## Supported (Tested) Databases

| Connector     | Database   | Driver                  |
|---------------|--|-------------------------|
| mongodb-cdc   | • MongoDB: 3.6, 4.x, 5.0   | MongoDB Driver: 4.3.1   |
| mysql-cdc     | • MySQL: 5.6, 5.7, 8.0.x<br>• RDS MySQL: 5.6, 5.7, 8.0.x<br>• PolarDB MySQL: 5.6, 5.7, 8.0.x<br>• Aurora MySQL: 5.6, 5.7, 8.0.x<br>• MariaDB: 10.x<br>• PolarDB X: 2.0.1 | JDBC Driver: 8.0.27     |
| oceanbase-cdc | • OceanBase CE: 3.1.x<br>• OceanBase EE (MySQL mode): 2.x, 3.x   | JDBC Driver: 5.1.4x     |
| oracle-cdc    | • Oracle: 11, 12, 19   | Oracle Driver: 19.3.0.0 |
| postgres-cdc  | • PostgreSQL: 9.6, 10, 11, 12  | JDBC Driver: 42.2.12    |
| sqlserver-cdc | • Sqlserver: 2012, 2014, 2016, 2017, 2019  | JDBC Driver: 7.2.2.jre8 |
| tidb-cdc      | • TiDB: 5.1.x, 5.2.x, 5.3.x, 5.4.x, 6.0.0  | JDBC Driver: 8.0.27     |
| Db2-cdc       | • Db2: 11.5  | Db2 Driver: 11.5.0.0    |

- Mysql修改配置信息

  - [root@node01 ~]# vim /etc/my.cnf

    - # 服务器ID  
`server_id=12345`  
`log_bin=/var/lib/mysql/mysql-bin`  
`expire_logs_days=7`  
# 必须为ROW  
`binlog_format=ROW`  
`binlog_cache_size=16M`  
`max_binlog_size=100M`  
`max_binlog_cache_size=256M`  
`relay_log_recovery=1`  
# 必须为FULL, MySQL-5.7后才有该参数  
`binlog_row_image=FULL`  
`expire_logs_days=30`  
`binlog_do_db=scott`

  - [root@node01 ~]# systemctl restart mysqld.service

    - mysql> show variables like 'log\_%';

```
+-----+-----+
|           variable_name          |           value
|                               |
+-----+-----+
| log_bin                                | ON
|                               |
| log_bin_basename                         | /var/lib/mysql/mysql-bin
|                               |
| log_bin_index                           | /var/lib/mysql/mysql-bin.index
```

- 创建数据表

- ```

DROP TABLE IF EXISTS `dept`;
CREATE TABLE `dept` (
    `deptno` int(11) NOT NULL,
    `dname` varchar(255) DEFAULT NULL,
    `loc` varchar(255) DEFAULT NULL,
    PRIMARY KEY (`deptno`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

--代码运行之后再开始插入数据
INSERT INTO `dept` VALUES ('10', 'ACCOUNTING', 'NEW YORK');
INSERT INTO `dept` VALUES ('20', 'RESEARCH', 'DALLAS');
INSERT INTO `dept` VALUES ('30', 'SALES', 'CHICAGO');
INSERT INTO `dept` VALUES ('40', 'OPERATIONS', 'BOSTON');

```

- o pom.xml

- ```

<!-- Flink CDC 的依赖 -->
<dependency>
    <groupId>com.ververica</groupId>
    <artifactId>flink-connector-mysql-cdc</artifactId>
    <version>2.3.0</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.27</version>
</dependency>

```

- o 代码实现

- ```

import org.apache.flink.configuration.Configuration
import
org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.table.api.EnvironmentSettings
import
org.apache.flink.table.api.bridge.scala.StreamTableEnvironment

object Hello01TablesSql {

    def main(args: Array[String]): Unit = {
        //配置信息
        val envSetting = EnvironmentSettings.newInstance()
            .useBlinkPlanner()
            .inStreamingMode()
            .build()
        //执行环境
        val executionEnvironment =
StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new
Configuration())
        //创建Flink Table
        val tableEnv =
StreamTableEnvironment.create(executionEnvironment, envSetting)
        //建表语句
        val sourceDDL =
"CREATE TABLE flink_cdc_dept (" +

```

```

    "      deptno INT NOT NULL," +
    "      dname STRING," +
    "      loc STRING" +
  ") WITH (" +
  "'connector' = 'mysql-cdc'," +
  "'hostname' = '192.168.88.101'," +
  "'port' = '3306'," +
  "'username' = 'root'," +
  "'password' = '123456'," +
  "'database-name' = 'scott'," +
  "'table-name' = 'dept'" +
  ")"
}

tableEnv.executeSql(sourceDDL)
tableEnv.executeSql("select * from flink_cdc_dept").print()

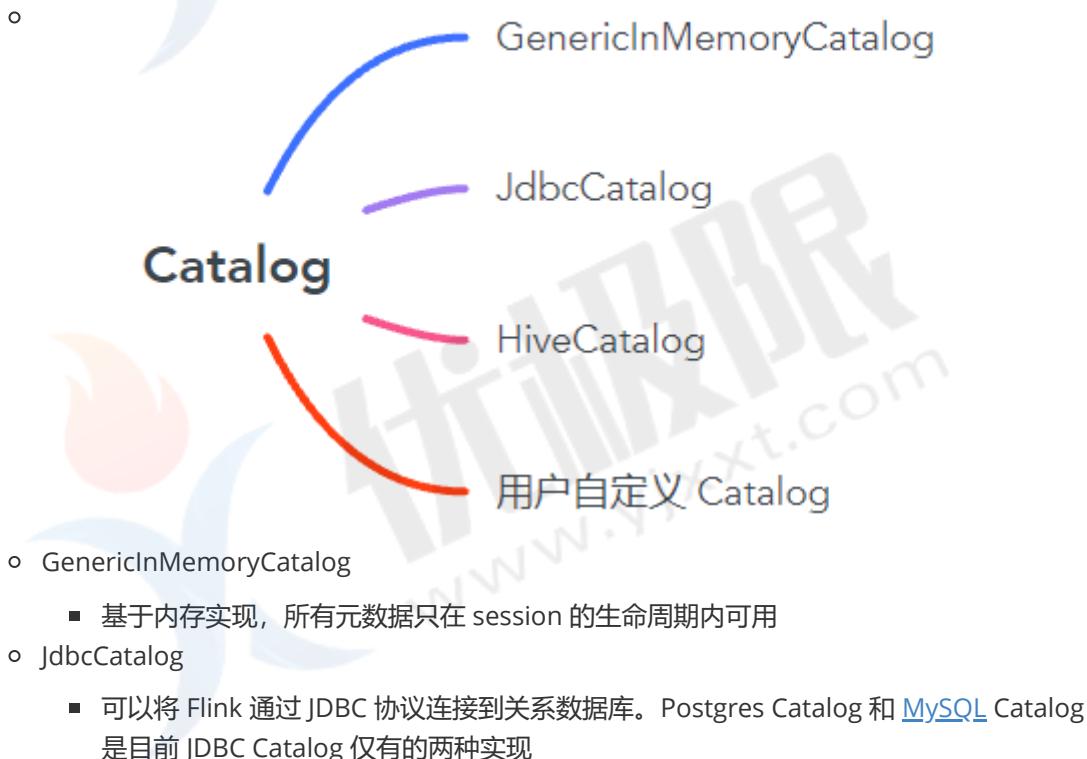
}
}

```

## 13. FlinkSQL on Hive

### 13.1. Catalog

- 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
  - JdbcCatalog
  - HiveCatalog
  - 用户自定义 Catalog



- HiveCatalog
  - 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
- 用户自定义 Catalog
  - 用户可以通过实现 Catalog 接口来开发自定义 Catalog，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

## 13.2. 版本支持

- 使用Hive构建数据仓库已经成为了比较普遍的一种解决方案，不同版本的Flink对于Hive的集成有所差异
- Flink 与 Hive 的集成主要体现在以下两个方面：
  - 持久化元数据：
    - Flink利用 Hive 的 MetaStore 作为持久化的 Catalog，我们可通过 HiveCatalog 将不同会话中的 Flink 元数据存储到 Hive Metastore 中。例如，我们可以使用 HiveCatalog 将其 Kafka 的数据源表存储在 Hive Metastore 中这样该表的元数据信息会被持久化到 Hive 的 MetaStore 对应的元数据库中
  - 读写 Hive 表数据
    - Flink打通了与Hive的集成，如同使用SparkSQL或者Impala操作Hive中的数据一样，我们可以使用Flink直接读写Hive中的表。
- 版本支持
  - 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

- o 3.1

- 3.1.0
- 3.1.1
- 3.1.2

Metastore version	Maven dependency	SQL Client JAR
1.0.0 - 1.2.2	flink-sql-connector-hive-1.2.2	<a href="#">Download</a>
2.0.0 - 2.2.0	flink-sql-connector-hive-2.2.0	<a href="#">Download</a>
2.3.0 - 2.3.6	flink-sql-connector-hive-2.3.6	<a href="#">Download</a>
3.0.0 - 3.1.2	flink-sql-connector-hive-3.1.2	<a href="#">Download</a>

- o <!-- Flink On Hive-->

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-hive_2.12</artifactId>
    <version>1.15.2</version>
</dependency>
<dependency>
    <groupId>org.apache.hive</groupId>
    <artifactId>hive-exec</artifactId>
    <version>3.1.2</version>
    <exclusions>
        <exclusion>
            <groupId>org.apache.calcite.avatica</groupId>
            <artifactId>avatica</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.calcite</groupId>
            <artifactId>*</artifactId>
        </exclusion>
        <exclusion>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>*</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

### 13.3. 连接Hive集群

- 通过TableEnvironment或者YAML配置，使用Catalog接口和HiveCatalog连接到现有的Hive集群。

- ```

EnvironmentSettings settings = EnvironmentSettings.inStreamingMode();
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");

```

| 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 <b>不要手动设置</b> 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 <b>HADOOP_CONF_DIR</b> 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。 |

## 14. FlinkSQL 相关概念

### 14.1. 执行计划

- able API 提供了一种机制来解释计算 Table 的逻辑和优化查询计划。
  - Table.explain() 返回一个 Table 的计划。
  - StatementSet.explain() 返回多 sink 计划。
- 返回的计划包括
  - 关系查询的抽象语法树 (the Abstract Syntax Tree) , 即未优化的逻辑查询计划,

- 优化的逻辑查询计划
  - 物理执行计划。
  - 使用 `Table.explain()` 方法的相应输出:
- ```

○ StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
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());

```

- 使用 `StatementSet.explain()` 的多 sink 计划的相应输出:

- ```

EnvironmentSettings settings = EnvironmentSettings.inStreamingMode();
TableEnvironment tEnv = TableEnvironment.create(settings);

final Schema schema = Schema.newBuilder()
    .column("count", DataTypes.INT())
    .column("word", DataTypes.STRING())
    .build();

tEnv.createTemporaryTable("MySource1",
    TableDescriptor.forConnector("filesystem")
        .schema(schema)
        .option("path", "/source/path1")
        .format("csv")
        .build());
tEnv.createTemporaryTable("MySource2",
    TableDescriptor.forConnector("filesystem")
        .schema(schema)
        .option("path", "/source/path2")
        .format("csv")
        .build());
tEnv.createTemporaryTable("MySink1",
    TableDescriptor.forConnector("filesystem")
        .schema(schema)
        .option("path", "/sink/path1")
        .format("csv")
        .build());
tEnv.createTemporaryTable("MySink2",
    TableDescriptor.forConnector("filesystem")
        .schema(schema)
        .option("path", "/sink/path2")
        .format("csv")
        .build());

```

```

StatementSet stmtSet = tEnv.createStatementSet();

Table table1 = tEnv.from("MySource1").where($"word").like("F%"));
stmtSet.add(table1.insertInto("MySink1"));

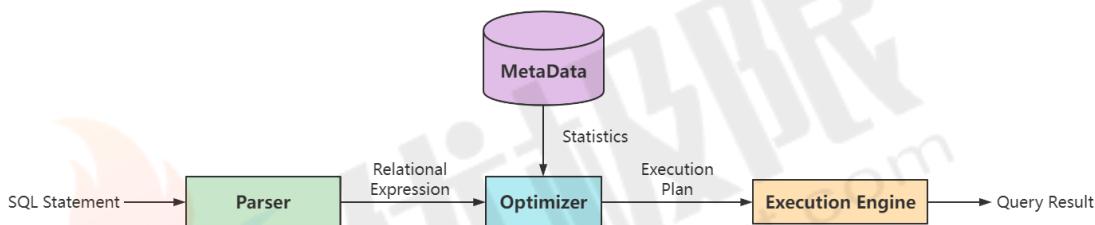
Table table2 = table1.unionAll(tEnv.from("MySource2"));
stmtSet.add(table2.insertInto("MySink2"));

String explanation = stmtSet.explain();
System.out.println(explanation);

```

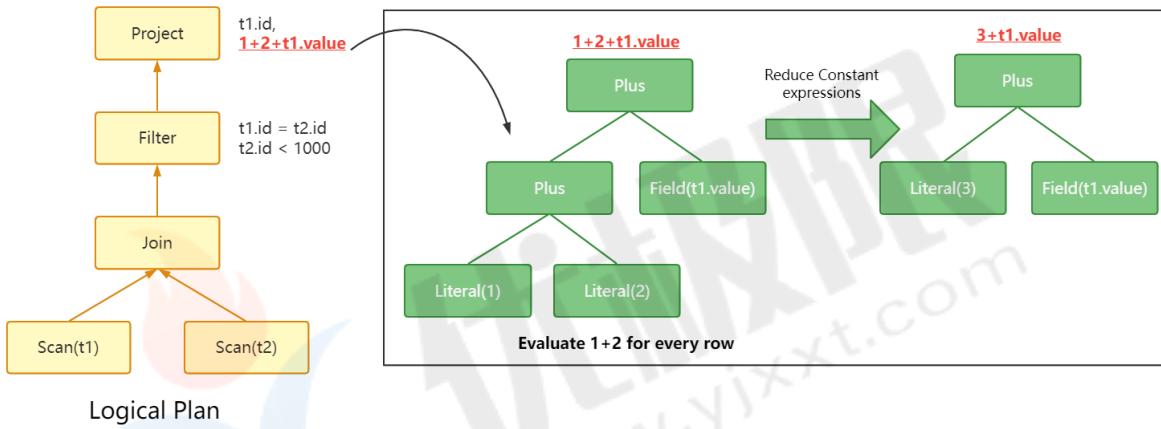
## 14.2. 查询优化

- Apache Flink 使用并扩展了 Apache Calcite 来执行复杂的查询优化。其中包括两种优化器：
  - RBO (基于规则的优化器)
  - CBO (基于成本的优化器)
- 优化方案：
  - 基于 Apache Calcite 的子查询解相关
  - 投影下推 (Projection Pushdown)
  - 分区剪裁 (Partition Prune)
  - 谓词下推 (Predicate Pushdown)
  - 常量折叠 (Constant Folding)
  - 子计划消除重复数据以避免重复计算
  - 特殊子查询重写，包括两部分：
    - 将 IN 和 EXISTS 转换为 left semi-joins
    - 将 NOT IN 和 NOT EXISTS 转换为 left anti-join
  - 可选 join 重新排序
    - 通过 `table.optimizer.join-reorder-enabled` 启用
- 注意：**当前仅在子查询重写的结合条件下支持 IN / EXISTS / NOT IN / NOT EXISTS。
- 优化器不仅基于计划，而且还基于可从数据源获得的丰富统计信息以及每个算子（例如 io, cpu, 网络和内存）的细粒度成本来做出明智的决策。
- 高级用户可以通过 `CalciteConfig` 对象提供自定义优化，可以通过调用 `TableEnvironment#getConfig#setPlannerConfig` 将其提供给 `TableEnvironment`。
- 



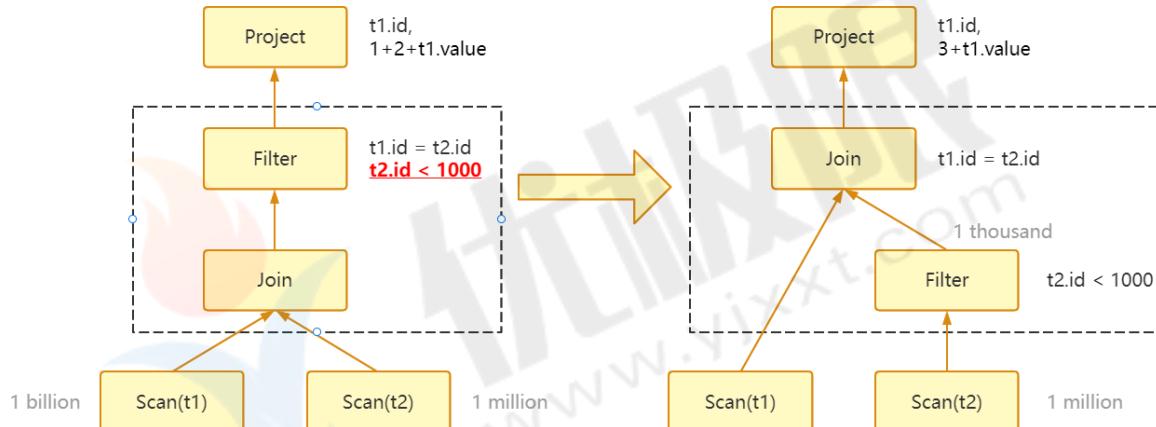
### 14.2.1. 常量折叠

- 常量折叠：**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：
- 折叠前： $1+2+t1.value$ ； 折叠后： $3+t1.value$



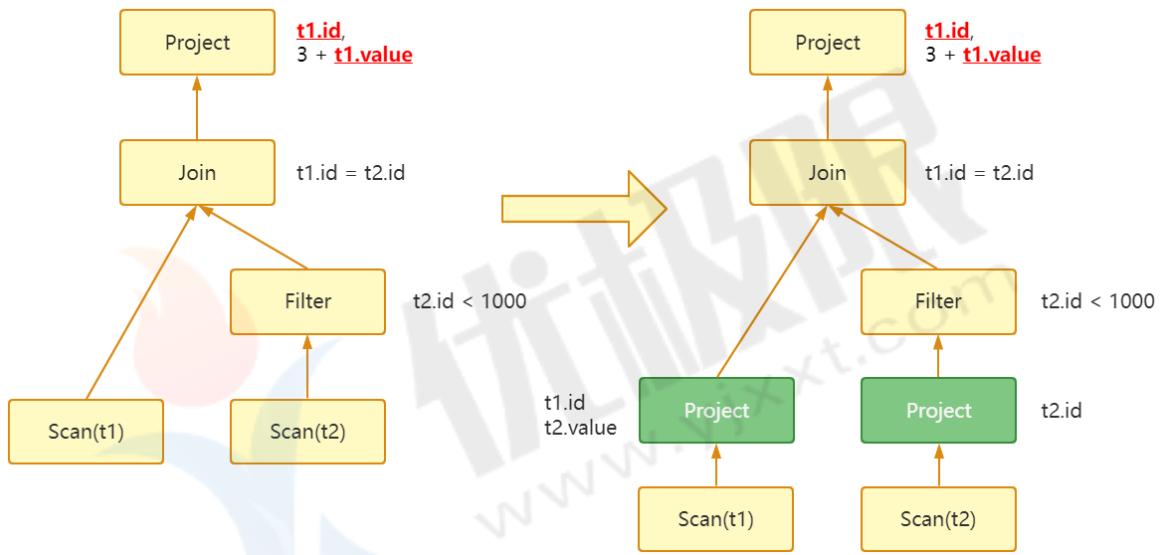
### 14.2.2. 谓词下推

- **谓词推执行**: 这里就是把 $t2.id < 1000$ ,下推到扫描 t2表的时候。
- 执行过程是: 全量数据扫描, 执行join操作, 然后才进行filter, 这明显很浪费, id大于1000的不需要执行join操作, 将filter操作下推到join之前执行, 减少了join的数据量, 大大提升性能。



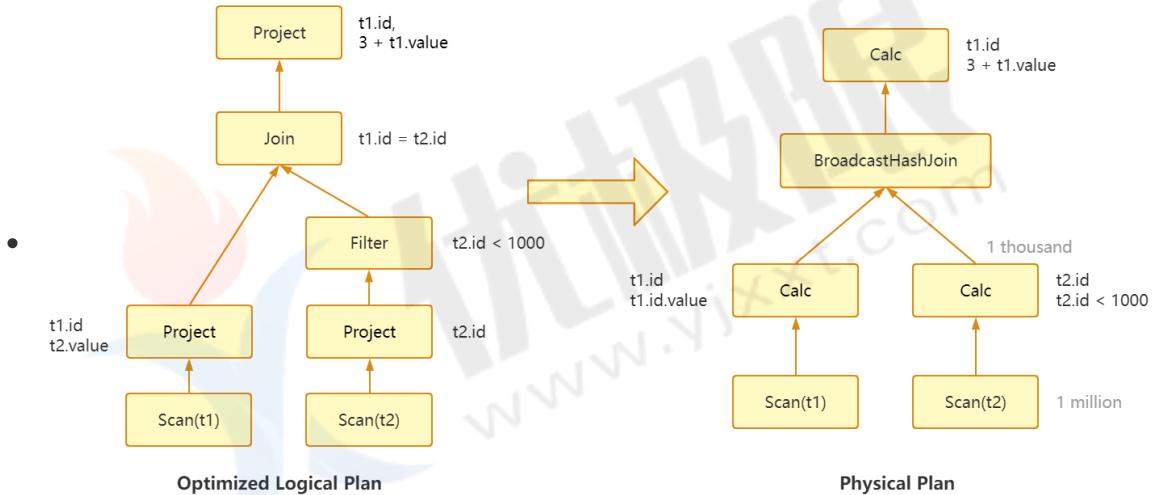
### 14.2.3. 投影下推

- **投影下推**: 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。

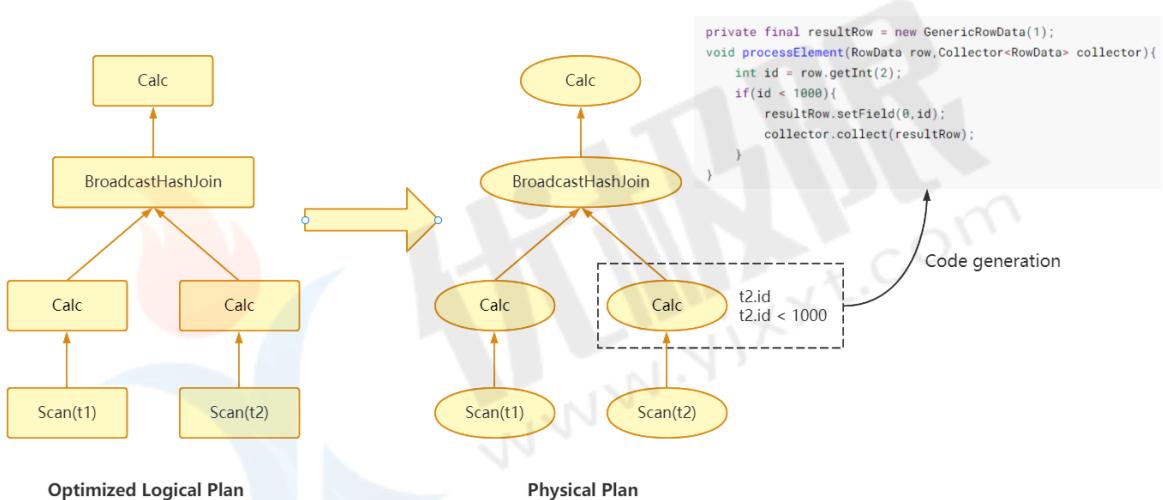


#### 14.2.4. Hash Join

- 根据代价 cost 选择批处理 join 有方式(sortmergejoin, hashjoin, broadcasthashjoin)。
- 比如前面例子，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。



#### 14.2.5. Transformation Tree



## 14.3. 性能调整

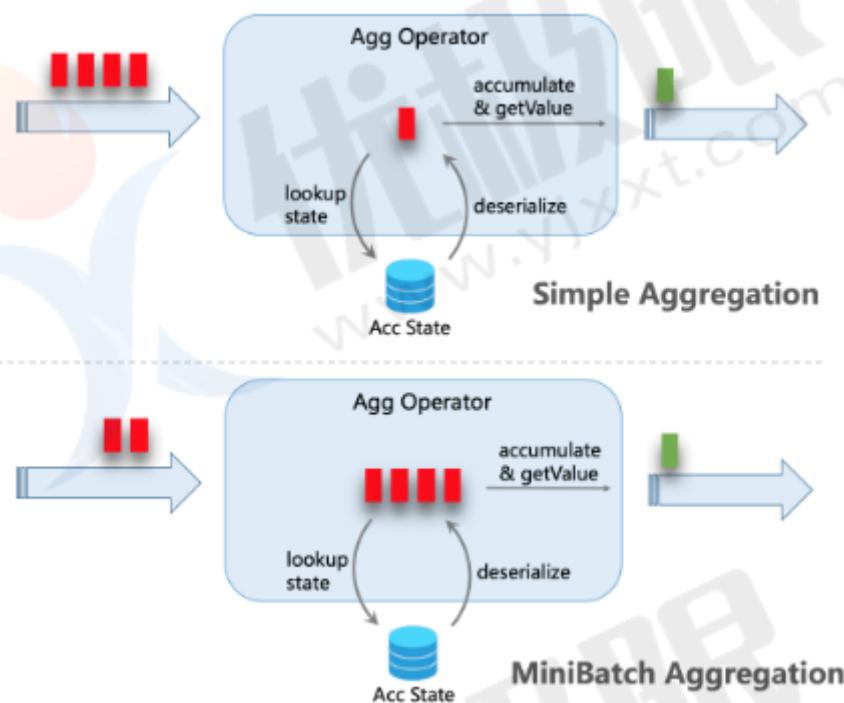
- SQL 是数据分析中使用最广泛的语言。Flink Table API 和 SQL 使用用户能够以更少的时间和精力定义高效的流分析应用程序。
- 此外，Flink Table API 和 SQL 是高效优化过的，它集成了许多查询优化和算子优化。
- 但并不是所有的优化都是默认开启的，因此对于某些工作负载，可以通过打开某些选项来提高性能。

### 14.3.1. MiniBatch 聚合

- 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
  - (5) 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。
- 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。

- 



- 代码实现

```

    o // instantiate table environment
    TableEnvironment tEnv = ...;

    // access flink configuration
    TableConfig configuration = tEnv.getConfig();

    // enable mini-batch optimization
    configuration.set("table.exec.mini-batch.enabled", "true");
    // use 5 seconds to buffer input records
    configuration.set("table.exec.mini-batch.allow-latency", "5 s");
    // the maximum number of records can be buffered by each aggregate
    operator task
    configuration.set("table.exec.mini-batch.size", "5000");

```

### 14.3.2. Local-Global 聚合

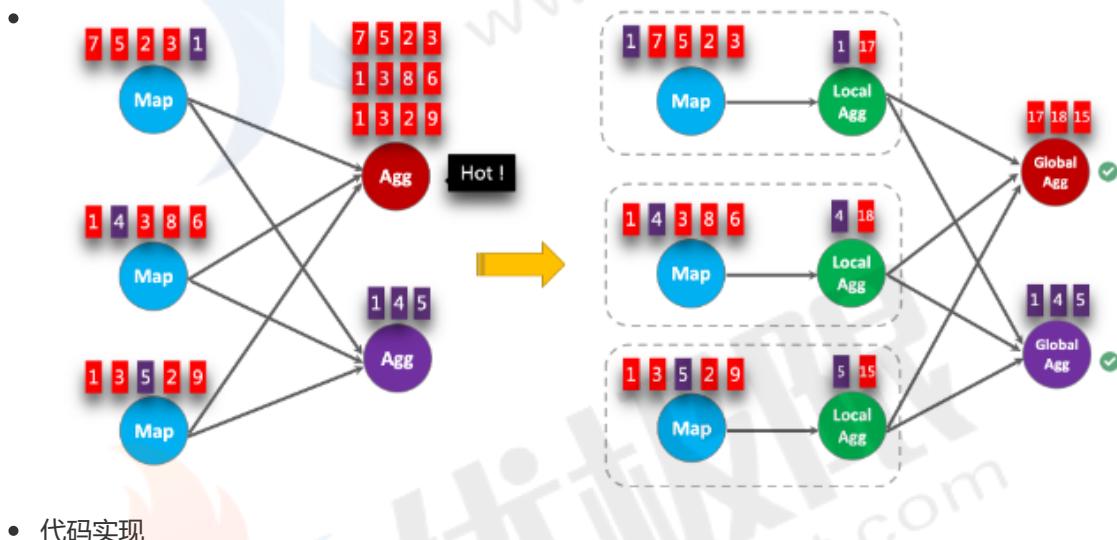
- Local-Global 聚合是为解决数据倾斜问题提出的，通过将一组聚合分为两个阶段，首先在上游进行本地聚合，然后在下游进行全局聚合，类似于 MapReduce 中的 Combine + Reduce 模式。
- 例如

```

    o 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。



- ```
// instantiate table environment
TableEnvironment tEnv = ...;

// access flink configuration
Configuration configuration = tEnv.getConfig().getConfiguration();

// local-global aggregation depends on mini-batch is enabled
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");
// enable two-phase, i.e. local-global aggregation
configuration.setString("table.optimizer.agg-phase-strategy",
"TWO_PHASE");
```

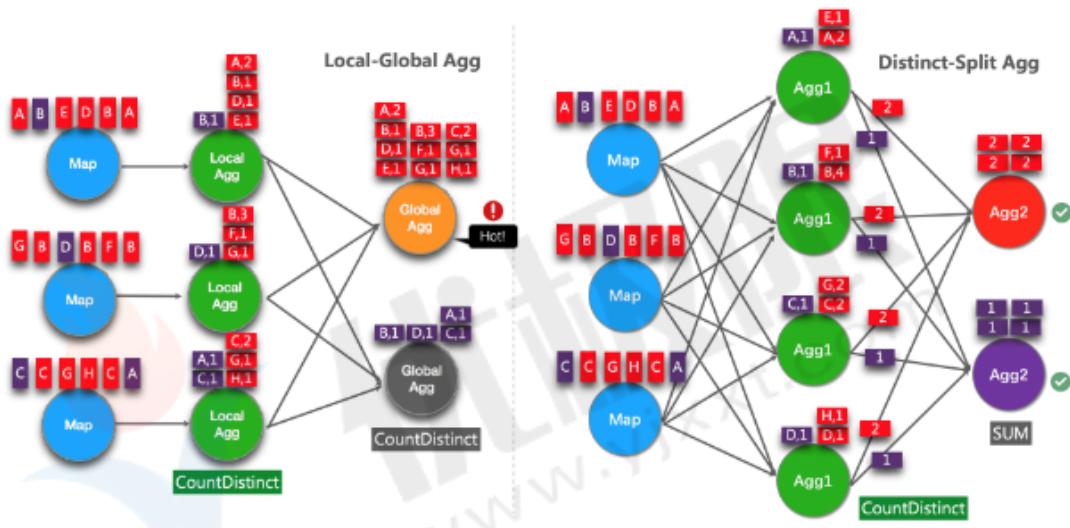
### 14.3.3. 拆分 distinct 聚合

- 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
```



- 代码实现

```
// instantiate table environment
TableEnvironment tEnv = ...;

tEnv.getConfig()
    .set("table.optimizer.distinct-agg.split.enabled", "true"); // enable distinct agg split
```

#### 14.3.4. distinct 聚合过滤

- 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

```
o 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
```

- 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

```
o 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
```

## 15. 附录

## 15.1. 时间转换

```
flinksq1里面最常用的事情就是时间格式转换，比如各种时间格式转换成TIMESTAMP(3)。  
now() bigint-- CAST(TO_TIMESTAMP(log_time) as TIMESTAMP(3)) ,log_time=now()  
localtimestamp timestamp(3)  
  
timestamp - 不带括号数字表示timestamp(6)  
now() 1403006911000 bigint - 毫秒时间戳数值 1528257600000  
localtimestamp 1636272032500 timestamp(3) - 毫秒时间戳  
timestamp(3) 1636272032500 - 毫秒时间戳  
timestamp(9)  
timestamp(6)  
  
TIMESTAMP(9) TO_TIMESTAMP(BIGINT time)  
TIMESTAMP(9) TO_TIMESTAMP(STRING time)  
TIMESTAMP(9) TO_TIMESTAMP(STRING time, STRING format)  
BIGINT TIMESTAMP_TO_MS(TIMESTAMP time)  
BIGINT TIMESTAMP_TO_MS(STRING time, STRING format)  
  
TO_DATE(CAST(LOCALTIMESTAMP AS VARCHAR))  
  
FROM_UNIXTIME(TIMESTAMP_TO_MS(localtimestamp)/1000, 'yyyy-MM-dd HH:mm:ss')  
  
event_time 6点到6点  
time_pt as cast(to_timestamp(eventTime - 6 * 3600 * 1000) as TIMESTAMP(3)) - 偏移  
6小时
```