Flink学习（八） Flink SQL & Table 编程和案例

Flink Table & SQL 概述

背景

我们在前面的课时中讲过 Flink 的分层模型，Flink 自身提供了不同级别的抽象来支持我们开发流式或者批量处理程序，下图描述了 Flink 支持的 4 种不同级别的抽象。

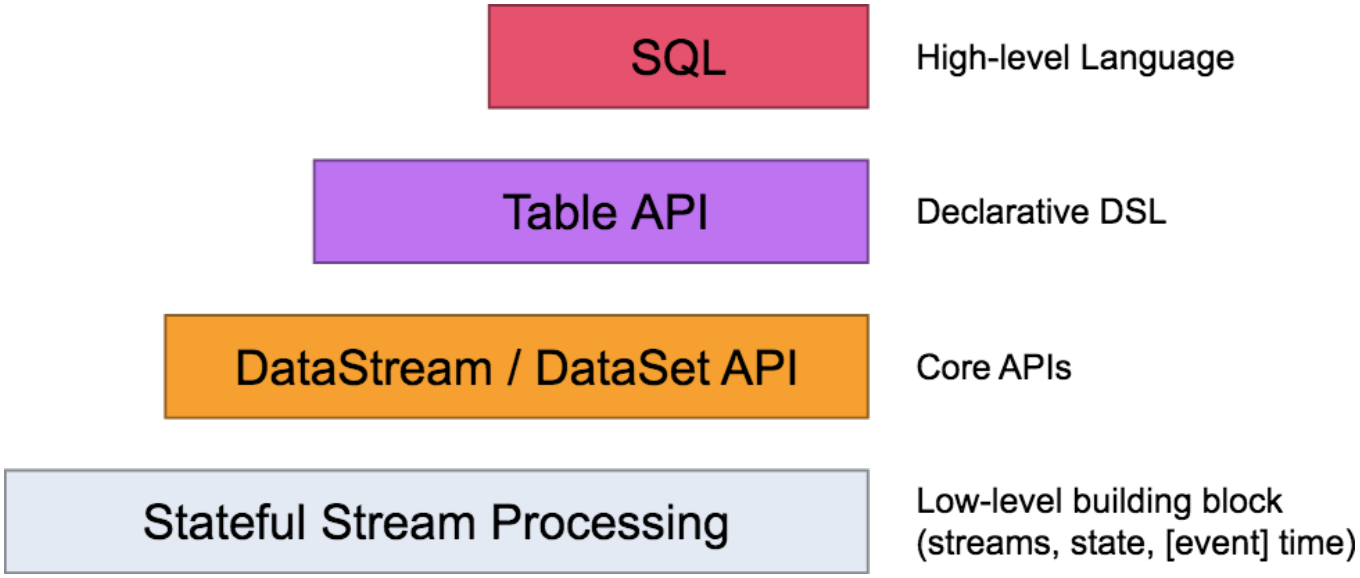


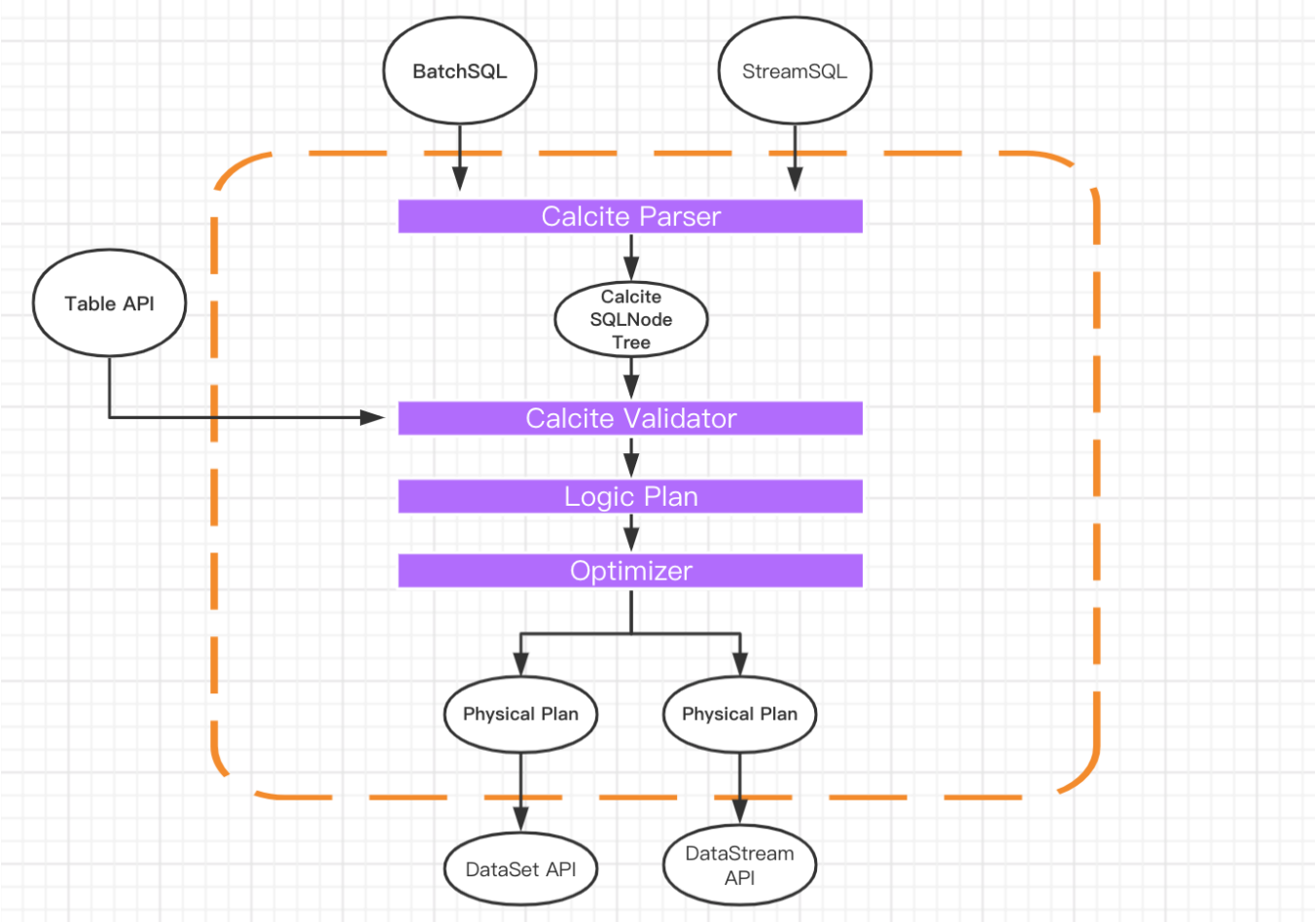
Table API 和 SQL 处于最顶端，是 Flink 提供的高级 API 操作。Flink SQL 是 Flink 实时计算为简化计算模型，降低用户使用实时计算门槛而设计的一套符合标准 SQL 语义的开发语言。

Flink 在编程模型上提供了 DataStream 和 DataSet 两套 API，并没有做到事实上的批流统一，因为用户和开发者还是开发了两套代码。正是因为 Flink Table & SQL 的加入，可以说 Flink 在某种程度上做到了事实上的批流一体。

原理

你之前可能都了解过 Hive，在离线计算场景下 Hive 几乎扛起了离线数据处理的半壁江山。它的底层对 SQL 的解析用到了 Apache Calcite，Flink 同样把 SQL 的解析、优化和执行教给了 Calcite。

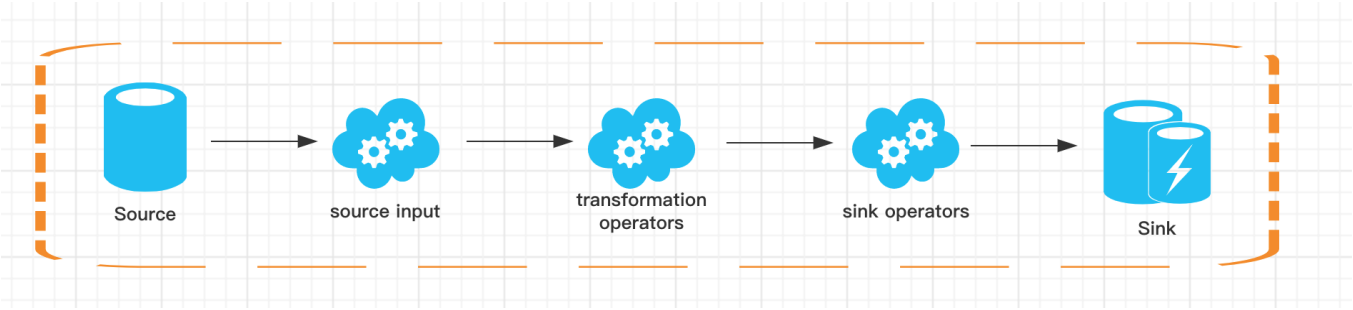
下图是一张经典的 Flink Table & SQL 实现原理图，可以看到 Calcite 在整个架构中处于绝对核心地位。



从图中可以看到无论是批查询 SQL 还是流式查询 SQL，都会经过对应的转换器 Parser 转换成为节点树 SQLNode tree，然后生成逻辑执行计划 Logical Plan，逻辑执行计划在经过优化后生成真正可以执行的物理执行计划，交给 DataSet 或者 DataStream 的 API 去执行。

在这里不对 Calcite 的原理过度展开，有兴趣的可以直接在官网上学习。

一个完整的 Flink Table & SQL Job 也是由 Source、Transformation、Sink 构成：



Source 部分来源于外部数据源，我们经常用的有 Kafka、MySQL 等；
Transformation 部分则是 Flink Table & SQL 支持的常用 SQL 算子，比如简单的 Select、

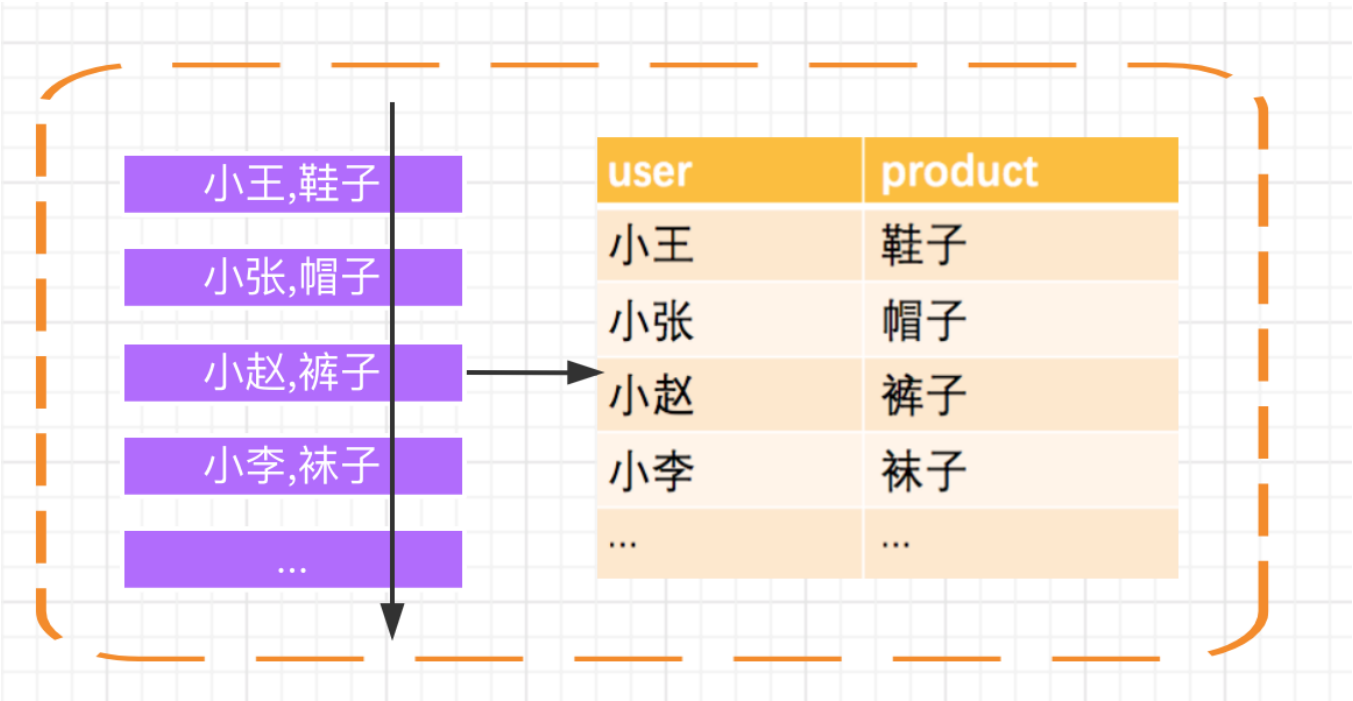
Groupby 等，当然在这里也有更为复杂的多流 Join、流与维表的 Join 等；
Sink 部分是指的结果存储比如 MySQL、HBase 或 Kakfa 等。

动态表

与传统的表 SQL 查询相比，Flink Table & SQL 在处理流数据时会时时刻刻处于动态的数据变化中，所以便有了一个动态表的概念。

动态表的查询与静态表一样，但是，在查询动态表的时候，SQL 会做连续查询，不会终止。

我们举个简单的例子，Flink 程序接受一个 Kafka 流作为输入，Kafka 中为用户的购买记录：



首先，Kafka 的消息会被源源不断的解析成一张不断增长的动态表，我们在动态表上执行的 SQL 会不断生成新的动态表作为结果表。

Flink Table & SQL 算子和内置函数

我们在讲解 Flink Table & SQL 所支持的常用算子前，需要说明一点，Flink 自从 0.9 版本开始支持 Table & SQL 功能一直处于完善开发中，且在不断进行迭代。

我们在官网中也可以看到这样的提示：

Please note that the Table API and SQL are not yet feature complete and are being actively developed. Not all operations are supported by every combination of [Table API, SQL] and [stream, batch] input.

Flink Table & SQL 的开发一直在进行中，并没有支持所有场景下的计算逻辑。从我个人实践角度来讲，在使用原生的 Flink Table & SQL 时，务必查询官网当前版本对 Table & SQL 的支持程

度，尽量选择场景明确，逻辑不是极其复杂的场景。

常用算子

目前 Flink SQL 支持的语法主要如下：

```
query:
  values
   | {
      select
        | selectWithoutFrom
       | query UNION [ ALL ] query
       | query EXCEPT query
       | query INTERSECT query
    }
    [ ORDER BY orderItem [, orderItem ]* ]
    [ LIMIT { count | ALL } ]
     [ OFFSET start { ROW | ROWS } ]
     [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY]

orderItem:
  expression [ ASC | DESC ]

select:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }
  FROM tableExpression
  [ WHERE booleanExpression ]
   [ GROUP BY { groupItem [, groupItem ]* } ]
  [ HAVING booleanExpression ]
   [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]* ]

selectWithoutFrom:
  SELECT [ ALL | DISTINCT ]
  { * | projectItem [, projectItem ]* }

projectItem:
  expression [ [ AS ] columnAlias ]
  | tableAlias . *

tableExpression:
  tableReference [, tableReference ]*
   | tableExpression [ NATURAL ] [ LEFT | RIGHT | FULL ] JOIN tableExpression [ joinCon
dition ]

joinCondition:
  ON booleanExpression
  | USING '(' column [, column ]* ')'
```

```
tableReference:
  tablePrimary
  [ matchRecognize ]
  [ [ AS ] alias [ '(' columnAlias [, columnAlias ]* ')' ] ]

tablePrimary:
  [ TABLE ] [ [ catalogName . ] schemaName . ] tableName
  | LATERAL TABLE '(' functionName '(' expression [, expression ]* ')' ')'
  | UNNEST '(' expression ')'

values:
  VALUES expression [, expression ]*

groupItem:
  expression
  | '(' ')'
  | '(' expression [, expression ]* ')'
  | CUBE '(' expression [, expression ]* ')'
  | ROLLUP '(' expression [, expression ]* ')'
  | GROUPING SETS '(' groupItem [, groupItem ]* ')'

windowRef:
    windowName
  | windowSpec

windowSpec:
    [ windowName ]
    '('
    [ ORDER BY orderItem [, orderItem ]* ]
    [ PARTITION BY expression [, expression ]* ]
    [
        RANGE numericOrIntervalExpression {PRECEDING}
      | ROWS numericExpression {PRECEDING}
    ]
    ')'
...
```

可以看到 Flink SQL 和传统的 SQL 一样，支持了包含查询、连接、聚合等场景，另外还支持了包括窗口、排序等场景。下面我就以最常用的算子来做详细的讲解。

SELECT/AS/WHERE

SELECT、WHERE 和传统 SQL 用法一样，用于筛选和过滤数据，同时适用于 DataStream 和 DataSet。

```
SELECT * FROM Table;
SELECT name, age FROM Table;
```

GROUP BY / DISTINCT/HAVING

GROUP BY 用于进行分组操作，DISTINCT 用于结果去重。
HAVING 和传统 SQL 一样，可以用来在聚合函数之后进行筛选。

```
SELECT DISTINCT name FROM Table;
SELECT name, SUM(score) as TotalScore FROM Table GROUP BY name;
SELECT name, SUM(score) as TotalScore FROM Table GROUP BY name HAVING
SUM(score) > 300;
```

JOIN

JOIN 可以用于把来自两个表的数据联合起来形成结果表，目前 Flink 的 Join 只支持等值连接。
Flink 支持的 JOIN 类型包括：

```
JOIN - INNER JOIN
LEFT JOIN - LEFT OUTER JOIN
RIGHT JOIN - RIGHT OUTER JOIN
FULL JOIN - FULL OUTER JOIN
```

例如，用用户表和商品表进行关联：

```
SELECT *
FROM User LEFT JOIN Product ON User.name = Product.buyer

SELECT *
FROM User RIGHT JOIN Product ON User.name = Product.buyer

SELECT *
FROM User FULL OUTER JOIN Product ON User.name = Product.buyer
```

LEFT JOIN、RIGHT JOIN 、FULL JOIN 相与我们传统 SQL 中含义一样。

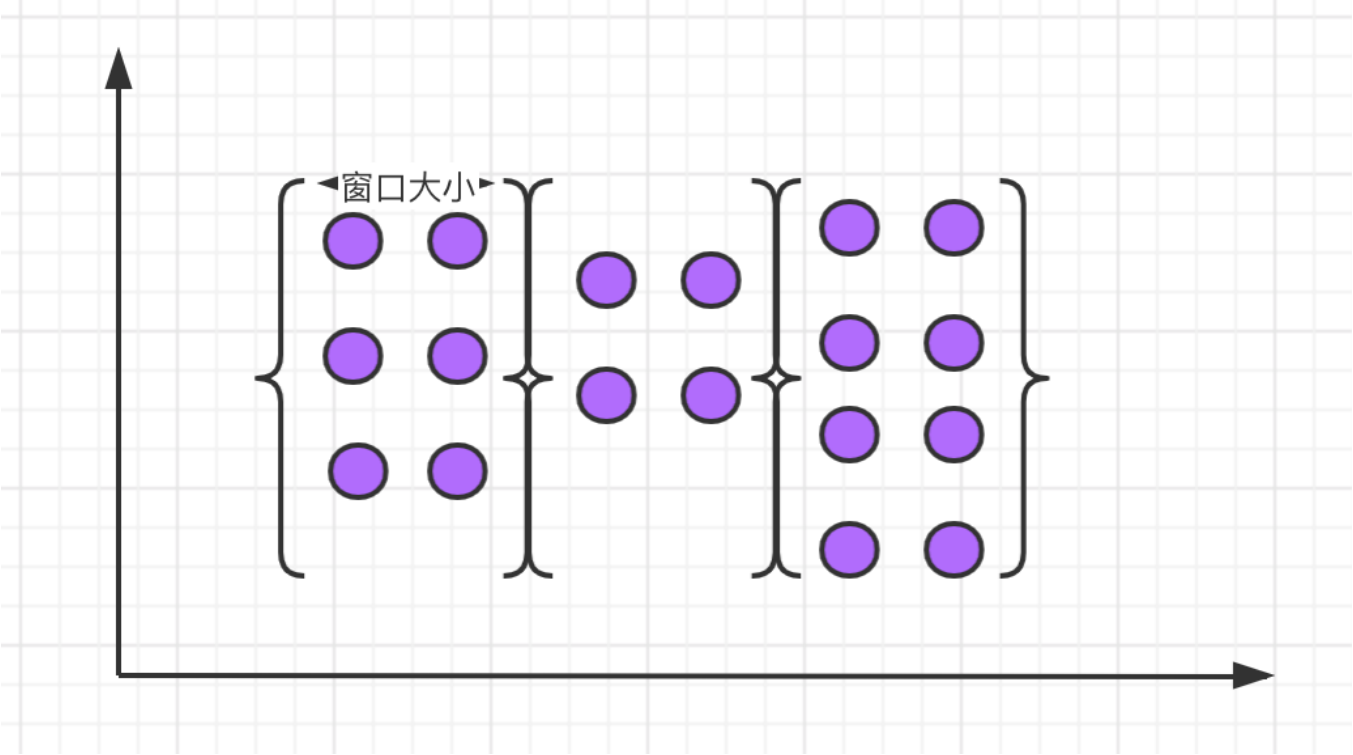
WINDOW

根据窗口数据划分的不同，目前 Apache Flink 有如下 3 种：

滚动窗口，窗口数据有固定的大小，窗口中的数据不会叠加；
滑动窗口，窗口数据有固定大小，并且有生成间隔；
会话窗口，窗口数据没有固定的大小，根据用户传入的参数进行划分，窗口数据无叠加；

滚动窗口

滚动窗口的特点是：有固定大小、窗口中的数据不会重叠，如下图所示：



滚动窗口的语法：

```
SELECT
     [gk],
    [TUMBLE_START(timeCol, size)],
    [TUMBLE_END(timeCol, size)],
    agg1(col1),
    ...
    aggn(colN)
FROM Tab1
GROUP BY [gk], TUMBLE(timeCol, size)
```

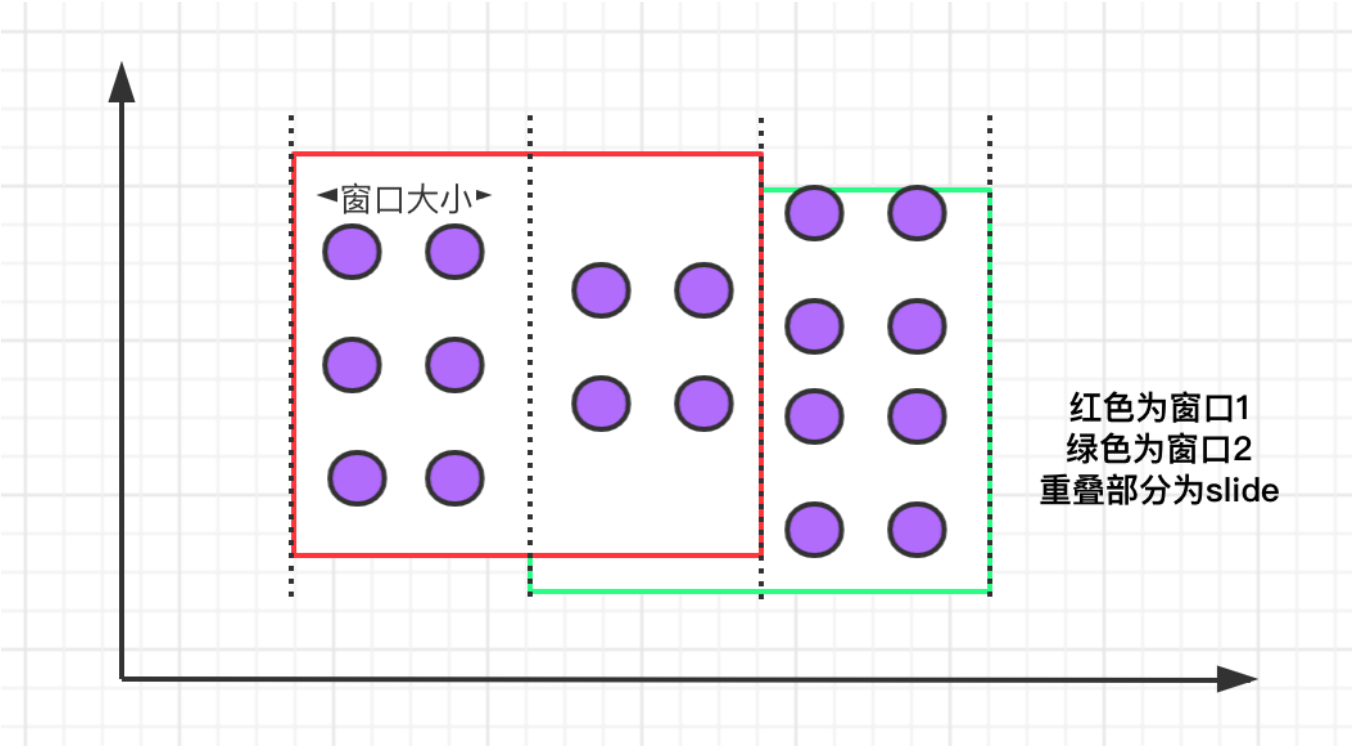
举例说明，我们需要计算每个用户每天的订单数量：

```
SELECT user, TUMBLE_START(timeLine, INTERVAL '1' DAY) as winStart, SUM(amount) FR
OM Orders GROUP BY TUMBLE(timeLine, INTERVAL '1' DAY), user;
```

其中，TUMBLE_START 和 TUMBLE_END 代表窗口的开始时间和窗口的结束时间，TUMBLE (timeLine, INTERVAL '1' DAY) 中的 timeLine 代表时间字段所在的列，INTERVAL '1' DAY 表示时间间隔为一天。

滑动窗口

滑动窗口有固定的大小，与滚动窗口不同的是滑动窗口可以通过 slide 参数控制滑动窗口的创建频率。需要注意的是，多个滑动窗口可能会发生数据重叠，具体语义如下：



滑动窗口的语法与滚动窗口相比，只多了一个 slide 参数：

```
SELECT
     [gk],
    [HOP_START(timeCol, slide, size)] ,
    [HOP_END(timeCol, slide, size)],
    agg1(col1),
    ...
    aggN(colN)
FROM Tab1
GROUP BY [gk], HOP(timeCol, slide, size)
```

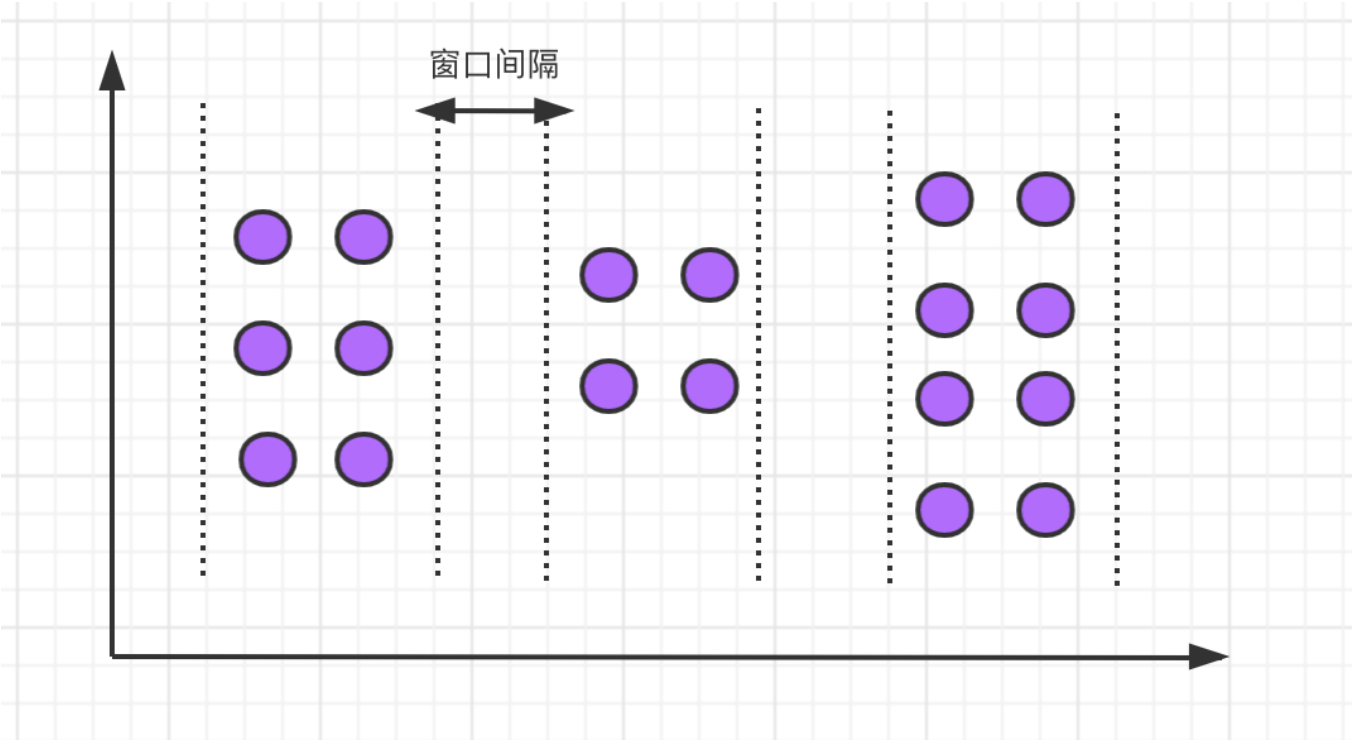
例如，我们要每间隔一小时计算一次过去 24 小时内每个商品的销量：

```
SELECT product, SUM(amount) FROM Orders GROUP BY HOP(rowtime, INTERVAL '1' HOUR, INTERVAL '1' DAY), product
```

上述案例中的 INTERVAL '1' HOUR 代表滑动窗口生成的时间间隔。

会话窗口

会话窗口定义了一个非活动时间，假如在指定的时间间隔内没有出现事件或消息，则会话窗口关闭。



会话窗口的语法如下：

```
SELECT
     [gk],
    SESSION_START(timeCol, gap) AS winStart,
    SESSION_END(timeCol, gap) AS winEnd,
    agg1(col1),
     ...
    aggn(colN)
FROM Tab1
GROUP BY [gk], SESSION(timeCol, gap)
```

举例，我们需要计算每个用户过去 1 小时内的订单量：

```
SELECT user, SESSION_START(rowtime, INTERVAL '1' HOUR) AS sStart, SESSION_ROWTIME(rowtime, INTERVAL '1' HOUR) AS sEnd, SUM(amount) FROM Orders GROUP BY SESSION(rowtime, INTERVAL '1' HOUR), user
```

内置函数

Flink 中还有大量的内置函数，我们可以直接使用，将内置函数分类如下：

比较函数

函数	逻辑描述
value1=value2	如果 value1 等于 value2，则返回 TRUE ; 如果 value1 或 value2 为 NULL，则返回 UNKNOWN
value1<>value2	如果 value1 不等于 value2，则返回 TRUE ; 如果 value1 或 value2 为 NULL，则返回 UNKNOWN
value1>value2	如果 value1 大于 value2，则返回 TRUE ; 如果 value1 或 value2 为 NULL，则返回 UNKNOWN
value1 <value2	如果 value1 小于 value2，则返回 TRUE ; 如果 value1 或 value2 为 NULL，则返回 UNKNOWN
value IS NULL	如果 value 为 NULL，则返回 TRUE
value IS NOT NULL	如果 value 不为 NULL，则返回 TRUE
string1 LIKE string2	如果 string1 匹配模式 string2，则返回 TRUE ; 如果 string1 或 string2 为 NULL，则返回 UNKNOWN
value1 IN (value2, value3…)	如果给定列表中存在 value1 （value2，value3，…），则返回 TRUE 。当（value2，value3，…）包含 NULL，如果可以找到该数据元则返回 TRUE，否则返回 UNKNOWN；如果 value1 为 NULL，则始终返回 UNKNOWN

逻辑函数

函数	逻辑描述
A OR B	如果 A 为 TRUE 或 B 为 TRUE，则返回 TRUE
A AND B	如果 A 和 B 都为 TRUE，则返回 TRUE
NOT boolean	如果 boolean 为 FALSE，则返回 TRUE，否则返回 TRUE如果 boolean 为 TRUE，则返回 FALSE
A IS TRUE 或 FALSE	判断 A 是否为真

算术函数

函数	逻辑描述
numeric1 ±*/ numeric2	分别代表两个数值加减乘除
ABS(numeric)	返回 numeric 的绝对值
POWER(numeric1, numeric2)	返回 numeric1 上升到 numeric2 的幂

字符串处理函数

函数	逻辑描述
UPPER/LOWER	以大写 / 小写形式返回字符串
LTRIM(string)	返回一个字符串，从去除左空格的字符串 类似还有 RTRIM
CONCAT(string1, string2,…)	返回连接 string1、string2、… 的字符串

时间函数

函数	逻辑描述
DATE string	返回以“yyyy-MM-dd”形式从字符串解析的 SQL 日期
TIMESTAMP string	返回以字符串形式解析的 SQL 时间戳，格式为“yyyy-MM-dd HH：mm：ss [.SSS]”
CURRENT_DATE	返回 UTC 时区中的当前 SQL 日期
DATE_FORMAT(timestamp, string)	返回使用指定格式字符串格式化时间戳的字符串

Flink Table & SQL 案例

上面分别介绍了 Flink Table & SQL 的原理和支持的算子，我们模拟一个实时的数据流，然后讲解 SQL JOIN 的用法。

之前利用 Flink 提供的自定义 Source 功能来实现一个自定义的实时数据源，具体实现如下：

```
package wyh.tableApi;

import org.apache.flink.streaming.api.functions.source.SourceFunction;
import wyh.datastreamingApi.Item;

import java.util.ArrayList;
import java.util.Random;

class MyStreamingSourceTable implements SourceFunction<Item> {

    private boolean isRunning = true;

    /**
     * 重写run方法产生一个源源不断的数据发送源
     * @param ctx
     * @throws Exception
     */
    @Override
    public void run(SourceContext<Item> ctx) throws Exception {
        while(isRunning){
```

```
            Item item = generateItem();
            ctx.collect(item);

            //每秒产生一条数据
            Thread.sleep(1000);
        }
    }

    @Override
    public void cancel() {
        isRunning = false;
    }

    //随机产生一条商品数据
    private Item generateItem(){
        int i = new Random().nextInt(1000);
        ArrayList<String> list = new ArrayList<>();
        list.add("HAT");
        list.add("TIE");
        list.add("SHOE");

        Item item = new Item();
        item.setName(list.get(new Random().nextInt(3)));
        item.setId(i);


        return item;
    }


}
```

我们把实时的商品数据流进行分流，分成 even 和 odd 两个流进行 JOIN，条件是名称相同，最后，把两个流的 JOIN 结果输出。

```
package wyh.tableApi;


import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.typeinfo.TypeHint;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.java.tuple.Tuple4;
import org.apache.flink.streaming.api.collector.selector.OutputSelector;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.datastream.SplitStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.java.StreamTableEnvironment;
import wyh.datastreamingApi.Item;

import java.util.ArrayList;
```

```
public class StreamingDemoTable {
    public static void main(String[] args) throws Exception{
        //BlinkPlanner 对SQl进行了一些优化 比如去重，取TopN
        EnvironmentSettings bsSettings = EnvironmentSettings.newInstance().useBlinkPlanner().in
StreamingMode().build();
        //流处理环境
        StreamExecutionEnvironment bsEnv = StreamExecutionEnvironment.getExecutionEnvironment()
;
        //支持table sql环境
        StreamTableEnvironment bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings);

        SingleOutputStreamOperator<Item> source = bsEnv.addSource(new MyStreamingSourceTable())
.map(new MapFunction<Item, Item>() {
            @Override
            public Item map(Item item) throws Exception {
                return item;
            }
        });

        DataStream<Item> splitAll = source.split(new OutputSelector<Item>() {
            @Override
            public Iterable<String> select(Item item) {
                //将实时商品流分成even和odd两个流
                ArrayList<String> output = new ArrayList<>();

                if (item.getId() % 2 == 0) {
                    output.add("even");
                } else {
                    output.add("odd");
                }
                return output;
            }
        });

        //将两个流筛选出来
        DataStream<Item> evenSelect = ((SplitStream<Item>) splitAll).select("even");
        DataStream<Item> oddSelect = ((SplitStream<Item>) splitAll).select("odd");

        //把这两个流在我们的Flink环境中注册为临时表
        bsTableEnv.createTemporaryView("evenTable",evenSelect,"name,id");
        bsTableEnv.createTemporaryView("oddTable",oddSelect,"name,id");

        Table quertTable = bsTableEnv.sqlQuery("select a.id,a.name,b.id,b.name from evenTable a
s a join oddTable as b on a.name=b.name");

        quertTable.printSchema();

        bsTableEnv.toRetractStream(quertTable,TypeInformation.of(new TypeHint<Tuple4<Integer,St
ring,Integer,String>>() {})).print();


        bsEnv.execute("streaming sql job");
    }
}
```

直接右键运行，在控制台可以看到输出：