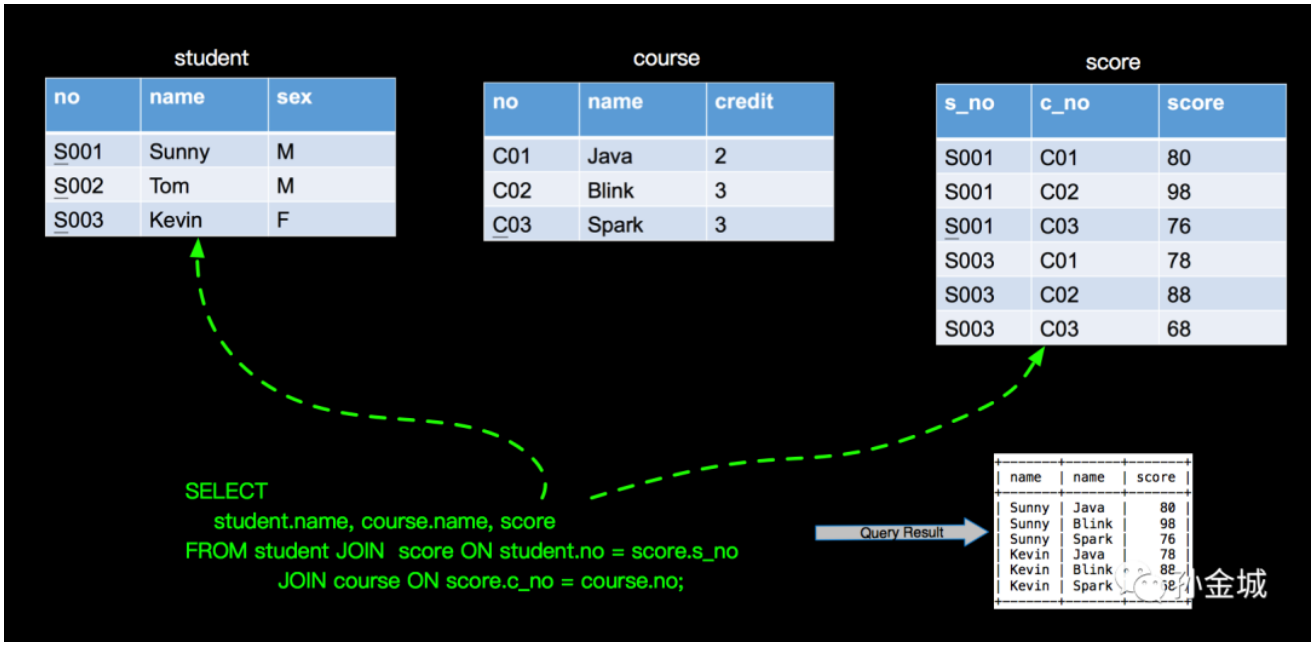
Apache Flink 漫谈系列 - 双流JOIN

什么是JOIN

JOIN的本质是分别从N(N>=1)张表中获取不同的字段，进而得到最完整的记录行。比如我们有一个查询需求：在学生表(学号，姓名，性别)，课程表(课程号，课程名，学分)和成绩表(学号，课程号，分数)中查询所有学生的姓名，课程名和考试分数。如下：



为啥需要JOIN

JOIN的本质是数据拼接，那么如果我们将所有数据列存储在一张大表中，是不是就不需要JOIN了呢？如果真的能将所需的数据都在一张表存储，我想就真的不需要JOIN的算子了，但现实业务中真的能做到将所需数据放到同一张大表里面吗？答案是否定的，核心原因有2个：

- 产生数据的源头可能不是一个系统；
- 产生数据的源头是同一个系统，但是数据冗余的沉重代价，迫使我们会遵循数据库范式，进行表的设计。简说NF如下：
    - 1NF - 列不可再分
    - 2NF - 符合1NF，并且非主键属性全部依赖于主键属性
    - 3NF - 符合2NF，并且传递依赖，即：即任何字段不能由其他字段派生出来
    - BCNF - 符合3NF，并且主键属性之间无依赖关系

JOIN的种类

- CROSS JOIN - 交叉连接，计算笛卡儿积
- INNER JOIN - 内连接，返回满足条件的记录
- OUTER JOIN
  - LEFT - 返回左表所有行，右表不存在补NULL；
  - RIGHT - 返回右表所有行，左边不存在补NULL；
  - FULL - 返回左表和右表的并集，不存在一边补NULL;
- SELF JOIN - 自连接，将表查询时候命名不同的别名；

JOIN语法

JOIN 在SQL89和SQL92中有不同的语法，以INNER JOIN为例说明：

- SQL89 - 表之间用“，”逗号分割，链接条件和过滤条件都在Where子句指定

```
SELECT
  a.colA,
  b.colA
FROM
  tab1 AS a , tab2 AS b
WHERE a.id = b.id and a.other > b.other
```

- SQL92

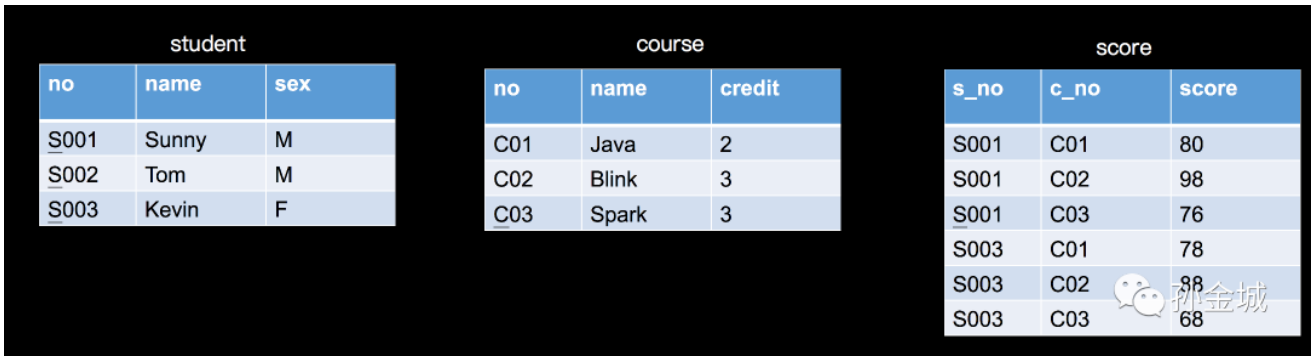
```
SELECT
  a.colA,
  b.colA
FROM
  tab1 AS a JOIN tab2 AS b ON a.id = b.id
WHERE
  a.other > b.other
```

SQL92将链接条件在ON子句指定，过滤条件在WHERE子句指定，逻辑更为清晰，本篇中的后续示例将应用SQL92语法进行SQL的编写。

```
tableExpression [ LEFT|RIGHT|FULL|INNER|SELF ] JOIN tableExpression [ ON jo
```

语义示例说明

我们以开篇示例中的三张表学生表(学号，姓名，性别)，课程表(课程号，课程名，学分)和成绩表(学号，课程号，分数)来介绍各种JOIN的语义。



student

no	name	sex
S001	Sunny	M
S002	Tom	M
S003	Kevin	F

course

no	name	credit
C01	Java	2
C02	Blink	3
C03	Spark	3

score

s_no	c_no	score
S001	C01	80
S001	C02	98
S001	C03	76
S003	C01	78
S003	C02	88
S003	C03	68

CROSS JOIN

交叉连接会对两个表进行笛卡尔积，也就是LEFT表的每一行和RIGHT表的所有行进行联接，因此生成结果表的行数是两个表行数的乘积，如student和course表的CROSS JOIN结果如下：

```
mysql> SELECT * FROM student JOIN course;
+------+-------+------+-----+-------+--------+
| no   | name  | sex  | no  | name  | credit |
+------+-------+------+-----+-------+--------+
| S001 | Sunny | M    | C01 | Java  |      2 |
| S002 | Tom   | F    | C01 | Java  |      2 |
| S003 | Kevin | M    | C01 | Java  |      2 |
| S001 | Sunny | M    | C02 | Blink |      3 |
| S002 | Tom   | F    | C02 | Blink |      3 |
| S003 | Kevin | M    | C02 | Blink |      3 |
| S001 | Sunny | M    | C03 | Spark |      3 |
| S002 | Tom   | F    | C03 | Spark |      3 |
| S003 | Kevin | M    | C03 | Spark |      3 |
+------+-------+------+-----+-------+--------+
9 rows in set (0.00 sec)
```

如上结果我们得到9行=student(3) x course(3)。交叉联接一般会消耗较大的资源，也被很多用户质疑交叉联接存在的意义？(任何时候我们都有质疑的权利，同时也建议我们养成自己质疑自己“质疑”的习惯，就像小时候不理解父母的“废话”一样)。我们以开篇的示例说明交叉联接的巧妙之处。

开篇中我们的查询需求是：在学生表(学号，姓名，性别)，课程表(课程号，课程名，学分)和成绩表(学号，课程号，分数)中查询所有学生的姓名，课程名和考试分数。开篇中的SQL语句得到的结果如下：

```
mysql> SELECT
    ->   student.name, course.name, score
    -> FROM student JOIN  score ON student.no = score.s_no
    ->              JOIN course ON score.c_no = course.no;
+-------+-------+-------+
| name  | name  | score |
+-------+-------+-------+
| Sunny | Java  |    80 |
| Sunny | Blink |    98 |
| Sunny | Spark |    76 |
| Kevin | Java  |    78 |
| Kevin | Blink |    88 |
| Kevin | Spark |    68 |
+-------+-------+-------+
6 rows in set (0.00 sec)
```

如上INNER JOIN的结果我们发现少了Tom同学的成绩，原因是Tom同学没有参加考试，在score表中没有Tom的成绩，但是我们可能希望虽然Tom没有参加考试但仍然希望Tom的成绩能够在查询结果中显示(成绩 0 分)，面对这样的需求，我们怎么处理呢？交叉联接可以帮助我们：

- 第一步 student和course 进行交叉联接：

```
mysql> SELECT
    ->   stu.no, c.no, stu.name, c.name
    -> FROM student stu JOIN course c  笛卡尔积
    -> ORDER BY stu.no; -- 排序只是方便大家查看:)
+------+-----+-------+-------+
| no   | no  | name  | name  |
+------+-----+-------+-------+
| S001 | C03 | Sunny | Spark |
| S001 | C01 | Sunny | Java  |
| S001 | C02 | Sunny | Blink |
| S002 | C03 | Tom   | Spark |
| S002 | C01 | Tom   | Java  |
| S002 | C02 | Tom   | Blink |
| S003 | C02 | Kevin | Blink |
| S003 | C03 | Kevin | Spark |
| S003 | C01 | Kevin | Java  |
+------+-----+-------+-------+
9 rows in set (0.00 sec)
```

- 第二步 将交叉联接的结果与score表进行左外联接，如下：

```
mysql> SELECT
    ->   stu.no, c.no, stu.name, c.name,
    ->    CASE
    ->     WHEN s.score IS NULL THEN 0
    ->     ELSE s.score
    ->   END AS score
    -> FROM student stu JOIN course c  -- 迪卡尔积
    -> LEFT JOIN score s ON stu.no = s.s_no and c.no = s.c_no -- LEFT OUTER
    -> ORDER BY stu.no; -- 排序只是为了大家好看一点:)
+------+-----+-------+-------+-------+
| no   | no  | name  | name  | score |
+------+-----+-------+-------+-------+
| S001 | C03 | Sunny | Spark |    76 |
| S001 | C01 | Sunny | Java  |    80 |
| S001 | C02 | Sunny | Blink |    98 |
| S002 | C02 | Tom   | Blink |     0 | -- TOM 虽然没有参加考试，但是仍然看到他的信
| S002 | C03 | Tom   | Spark |     0 |
| S002 | C01 | Tom   | Java  |     0 |
| S003 | C02 | Kevin | Blink |    88 |
| S003 | C03 | Kevin | Spark |    68 |
| S003 | C01 | Kevin | Java  |    78 |
+------+-----+-------+-------+-------+
9 rows in set (0.00 sec)
```

INNER JOIN

内联接在SQL92中 ON 表示联接添加，可选的WHERE子句表示过滤条件，如开篇的示例就是一个多表的内联接，我们在看一个简单的示例: 查询成绩大于80分的学生学号，学生姓名和成绩:

```
mysql> SELECT
    ->   stu.no, stu.name , s.score
    -> FROM student stu JOIN score s ON  stu.no = s.s_no
    -> WHERE s.score > 80;
+------+-------+-------+
| no   | name  | score |
+------+-------+-------+
| S001 | Sunny |    98 |
| S003 | Kevin |    88 |
+------+-------+-------+
2 rows in set (0.00 sec)
```

上面按语义的逻辑是:

- 第一步：先进行student和score的内连接，如下：

```
mysql> SELECT
    ->   stu.no, stu.name , s.score
    -> FROM student stu JOIN score s ON  stu.no = s.s_no ;
+------+-------+-------+
| no   | name  | score |
+------+-------+-------+
| S001 | Sunny |    80 |
| S001 | Sunny |    98 |
| S001 | Sunny |    76 |
| S003 | Kevin |    78 |
| S003 | Kevin |    88 |
| S003 | Kevin |    68 |
+------+-------+-------+
6 rows in set (0.00 sec)
```

- 第二步：对内联结果进行过滤， score > 80 得到，如下最终结果：

```
    -> WHERE s.score > 80;
+------+-------+-------+
| no   | name  | score |
+------+-------+-------+
| S001 | Sunny |    98 |
| S003 | Kevin |    88 |
+------+-------+-------+
2 rows in set (0.00 sec)
```

上面的查询过程符合语义，但是如果在filter条件能过滤很多数据的时候，先进行数据的过滤，在进行内联接会获取更好的性能，比如我们手工写一下：

```
mysql> SELECT
    ->   no, name , score
    -> FROM student stu JOIN ( SELECT s_no, score FROM score s WHERE s.scor
+------+-------+-------+
| no   | name  | score |
+------+-------+-------+
| S001 | Sunny |    98 |
| S003 | Kevin |    88 |
+------+-------+-------+
2 rows in set (0.00 sec)
```

上面写法语义和第一种写法语义一致，得到相同的查询结果，上面查询过程是：

- 第一步：执行过滤子查询

```
mysql> SELECT s_no, score FROM score s WHERE s.score >80;
+------+-------+
| s_no | score |
+------+-------+
| S001 |    98 |
| S003 |    88 |
+------+-------+
2 rows in set (0.00 sec)
```

- 第二步：执行内连接

```
    -> ON no = s_no;
+------+-------+-------+
| no   | name  | score |
+------+-------+-------+
| S001 | Sunny |    98 |
| S003 | Kevin |    88 |
+------+-------+-------+
2 rows in set (0.00 sec)
```

如上两种写法在语义上一致，但在查询性能在数量很大的情况下会有很大差距。上面为了和大家演示相同的查询语义，可以有不同的查询方式，不同的执行计划。实际上数据库本身的优化器会我们做查询优化，在内联接中ON的联接条件和WHERE的过滤条件具有相同的优先级，具体的执行顺序可以由数据库的优化器根据性能消耗决定。也就是说物理执行计划可以先执行过滤条件进行查询优化，如果细心的读者可能发现，在第二个写法中，子查询我们不但有行的过滤，也进行了列的裁剪(去除了对查询结果没有用的c_no列)，这两个变化实际上对应了数据库中两个优化规则：

- filter push down
- project push down

OUTER JOIN

LEFT OUTER JOIN

左外联接语义是返回左表所有行，右表不存在补NULL，为了演示作用，我们查询没有参加考试的所有学生的成绩单：

```
mysql> SELECT
    ->   no, name , s.c_no, s.score
    -> FROM student stu LEFT JOIN score s ON stu.no = s.s_no
    -> WHERE s.score is NULL;
+------+------+------+-------+
| no   | name | c_no | score |
+------+------+------+-------+
| S002 | Tom  | NULL |  NULL |
+------+------+------+-------+
1 row in set (0.00 sec)
```

上面查询的执行逻辑上也是分成两步：

- 第一步：左外联接查询

```
mysql> SELECT
    ->   no, name , s.c_no, s.score
    -> FROM student stu LEFT JOIN score s ON stu.no = s.s_no;
+------+-------+------+-------+
| no   | name  | c_no | score |
+------+-------+------+-------+
| S001 | Sunny | C01  |    80 |
| S001 | Sunny | C02  |    98 |
| S001 | Sunny | C03  |    76 |
| S002 | Tom   | NULL |  NULL | -- 右表不存在的补NULL
| S003 | Kevin | C01  |    78 |
| S003 | Kevin | C02  |    88 |
| S003 | Kevin | C03  |    68 |
+------+-------+------+-------+
7 rows in set (0.00 sec)
```

- 第二步：过滤查询

```
mysql> SELECT
    ->   no, name , s.c_no, s.score
    -> FROM student stu LEFT JOIN score s ON stu.no = s.s_no
    -> WHERE s.score is NULL;
+------+------+------+-------+
| no   | name | c_no | score |
+------+------+------+-------+
| S002 | Tom  | NULL |  NULL |
+------+------+------+-------+
1 row in set (0.00 sec)
```

这个两个过程和上面分析的INNER JOIN一样，但是这时候能否利用上面说的 filter push down 的优化呢？根据LEFT OUTER JOIN的语义来讲，答案是否定的。我们手工操作看一下：

第一步：先进行过滤查询(获得一个空表)

```
mysql> SELECT * FROM score s WHERE s.score is NULL;
Empty set (0.00 sec)
```

第二步：进行左外链接

```
mysql> SELECT
    ->   no, name , s.c_no, s.score
    -> FROM student stu LEFT JOIN (SELECT * FROM score s WHERE s.score is N
+------+-------+------+-------+
| no   | name  | c_no | score |
+------+-------+------+-------+
| S001 | Sunny | NULL |  NULL |
| S002 | Tom   | NULL |  NULL |
| S003 | Kevin | NULL |  NULL |
+------+-------+------+-------+
3 rows in set (0.00 sec)
```

我们发现两种写法的结果不一致，第一种写法只返回Tom没有参加考试，是我们预期的。第二种写法返回了Sunny，Tom和Kevin三名同学都没有参加考试，这明显是非预期的查询结果。所有LEFT OUTER JOIN不能利用INNER JOIN的 filter push down优化。

RIGHT OUTER JOIN

右外链接语义是返回右表所有行，左边不存在补NULL，如下：

```
mysql> SELECT
    ->   s.c_no, s.score, no, name
    -> FROM score s RIGHT JOIN student stu ON stu.no = s.s_no;
+------+-------+------+-------+
| c_no | score | no   | name  |
+------+-------+------+-------+
| C01  |    80 | S001 | Sunny |
| C02  |    98 | S001 | Sunny |
| C03  |    76 | S001 | Sunny |
| NULL |  NULL | S002 | Tom   | -- 左边没有的进行补 NULL
| C01  |    78 | S003 | Kevin |
| C02  |    88 | S003 | Kevin |
| C03  |    68 | S003 | Kevin |
+------+-------+------+-------+
7 rows in set (0.00 sec)
```

上面右外链接我只是将上面左外链接查询的左右表交换了一下:)。

FULL OUTER JOIN

全外链接语义返回左表和右表的并集，不存在一边补NULL,用于演示的MySql数据库不支持

FULL OUTER JOIN。这里不做演示了。

SELF JOIN

上面介绍的INNER JOIN、OUTER JOIN都是不同表之间的联接查询，自联接是一张表以不同的别名做为左右两个表，可以进行如上的INNER JOIN和OUTER JOIN. 如下看一个INNER 自联接：

```
mysql> SELECT * FROM student l JOIN student r where l.no = r.no;
+------+-------+------+------+-------+------+
| no   | name  | sex  | no   | name  | sex  |
+------+-------+------+------+-------+------+
| S001 | Sunny | M    | S001 | Sunny | M    |
| S002 | Tom   | F    | S002 | Tom   | F    |
| S003 | Kevin | M    | S003 | Kevin | M    |
+------+-------+------+------+-------+------+
3 rows in set (0.00 sec)
```

不等值联接

这里说的不等值联接是SQL92语法里面的ON子句里面只有等值联接，比如：

```
mysql> SELECT
    ->   s.c_no, s.score, no, name
    -> FROM score s RIGHT JOIN student stu ON stu.no != s.c_no;
+------+-------+------+-------+
| c_no | score | no   | name  |
+------+-------+------+-------+
| C01  |    80 | S001 | Sunny |
| C01  |    80 | S002 | Tom   |
| C01  |    80 | S003 | Kevin |
| C02  |    98 | S001 | Sunny |
| C02  |    98 | S002 | Tom   |
| C02  |    98 | S003 | Kevin |
| C03  |    76 | S001 | Sunny |
| C03  |    76 | S002 | Tom   |
| C03  |    76 | S003 | Kevin |
| C01  |    78 | S001 | Sunny |
| C01  |    78 | S002 | Tom   |
| C01  |    78 | S003 | Kevin |
| C02  |    88 | S001 | Sunny |
| C02  |    88 | S002 | Tom   |
| C02  |    88 | S003 | Kevin |
| C03  |    68 | S001 | Sunny |
| C03  |    68 | S002 | Tom   |
| C03  |    68 | S003 | Kevin |
+------+-------+------+-------+
18 rows in set (0.00 sec)
```

上面这示例，其实没有什么实际业务价值，在实际的使用场景中，不等值联接往往是结合等值联接，将不等值条件在WHERE子句指定，即，带有WHERE子句的等值联接。

Flink双流JOIN的支持

	CROSS JOIN	INNER JOIN	OUTER JOIN	SELF JOIN	ON(condition)	WHERE
Flink	N	Y	Y	Y	必选	可选

Flink目前支持INNER JOIN和LEFT OUTER JOIN（SELF 可以转换为普通的INNER和OUTER)。在语义上面Flink严格遵守标准SQL的语义，与上面演示的语义一致。下面我重点介绍Flink中JOIN的实现原理。

双流JOIN与传统数据库表JOIN的区别

传统数据库表的JOIN是静态两张静态表的数据联接，在流上面是 动态表，双流JOIN的数据不断流入与传统数据库表的JOIN有如下3个核心区别：

- 左右两边的数据集合无穷 - 传统数据库左右两个表的数据集合是有限的，双流JOIN的数据会源源不断的流入；
- JOIN的结果不断产生/更新 - 传统数据库表JOIN是一次执行产生最终结果后退出，双流JOIN会持续不断的产生新的结果。
- 查询计算的双边驱动 - 双流JOIN由于左右两边的流的速度不一样，会导致左边数据到来的时候右边数据还没有到来，或者右边数据到来的时候左边数据没有到来，所以在实现中要将左右两边的流数据进行保存，以保证JOIN的语义。在Flink中会以State的方式进行数据的存储。

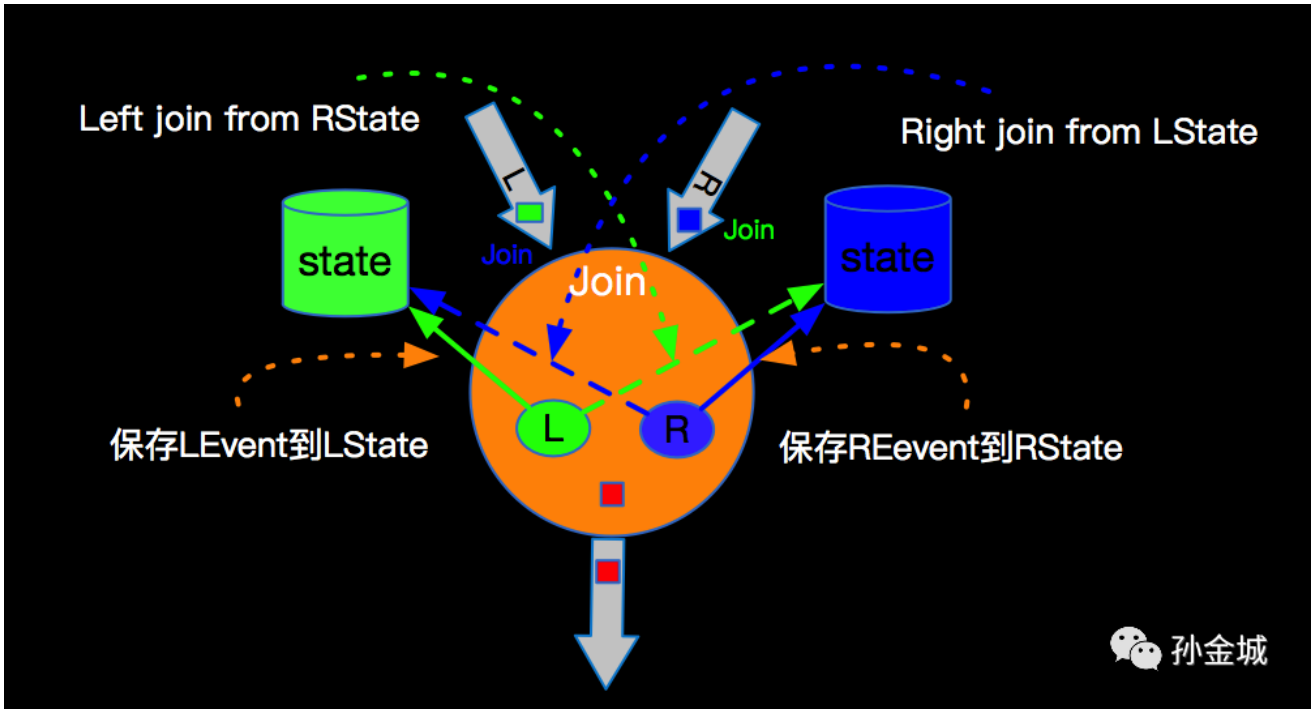
数据Shuffle

分布式流计算所有数据会进行Shuffle，怎么才能保障左右两边流的要JOIN的数据会在相同的节点进行处理呢？在双流JOIN的场景，我们会利用JOIN中ON的联接key进行partition，确保两个流相同的联接key会在同一个节点处理。

数据的保存

不论是INNER JOIN还是OUTER JOIN 都需要对左右两边的流的数据进行保存，JOIN算子会开辟左右两个State进行数据存储，左右两边的数据到来时候，进行如下操作：

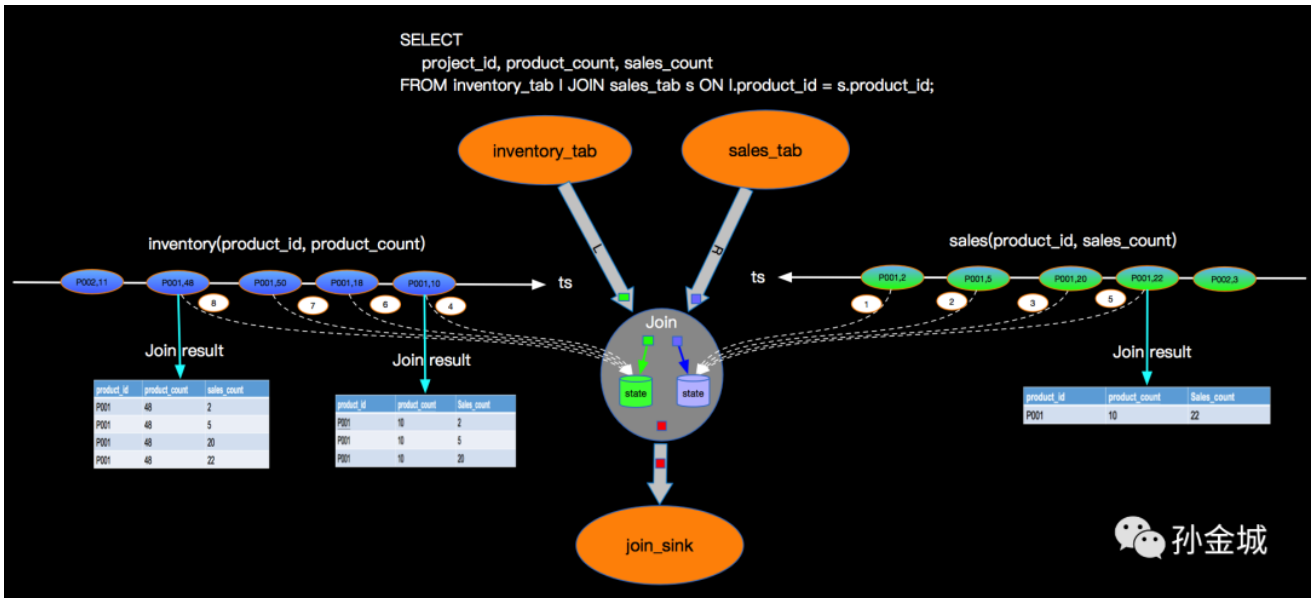
- LeftEvent到来存储到LState，RightEvent到来的时候存储到RState；
- LeftEvent会去RightState进行JOIN，并发出所有JOIN之后的Event到下游；
- RightEvent会去LeftState进行JOIN，并发出所有JOIN之后的Event到下游



简单场景介绍实现原理

INNER JOIN 实现

JOIN有很多复杂的场景，我们先以最简单的场景进行实现原理的介绍，比如：最直接的两个进行INNER JOIN，比如查询产品库存和订单数量，库存变化事件流和订单事件流进行INNER JOIN，JION条件是产品ID，具体如下：

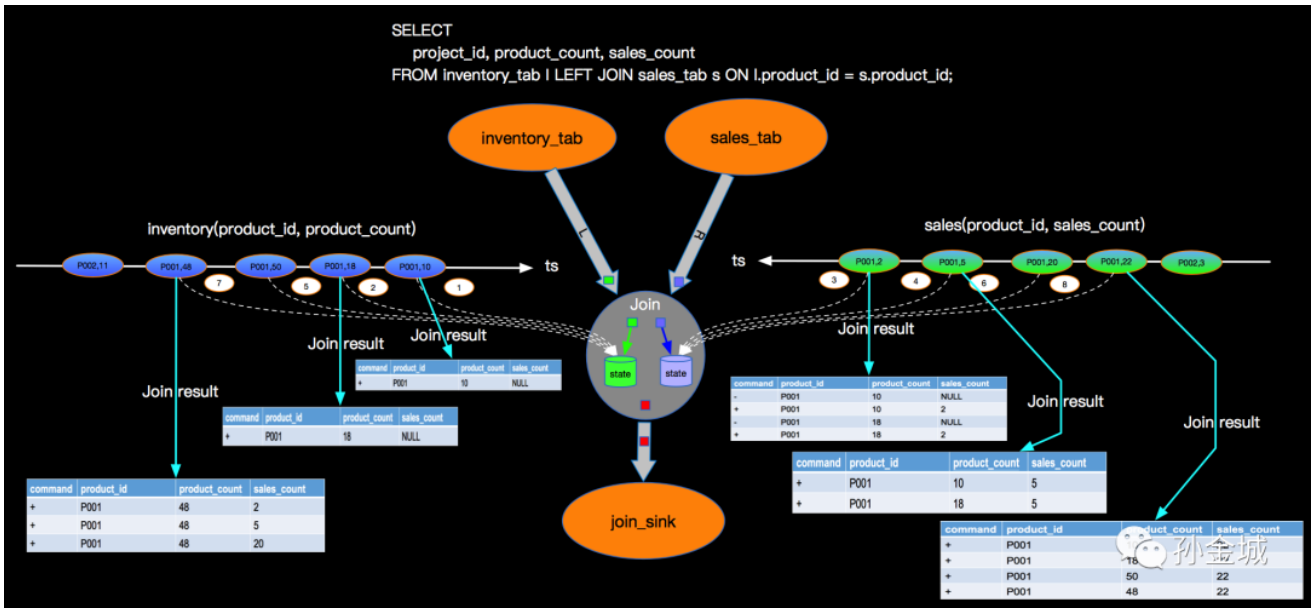


双流JOIN两边事件都会存储到State里面，如上，事件流按照标号先后流入到join节点，我们假设右边流比较快，先流入了3个事件，3个事件会存储到state中，但因为左边还没有数据，所有右边前3个事件流入时候，没有join结果流出，当左边第一个事件序号为4的流入时候，先存储左边state，再与右边已经流入的3个事件进行join，join的结果如图 三行结果会流入到下游节点sink。当第5号事件流入时候，也会和左边第4号事件进行join，流出一条jion结果到下游节点。这里关于INNER JOIN的语义和大家强调两点：

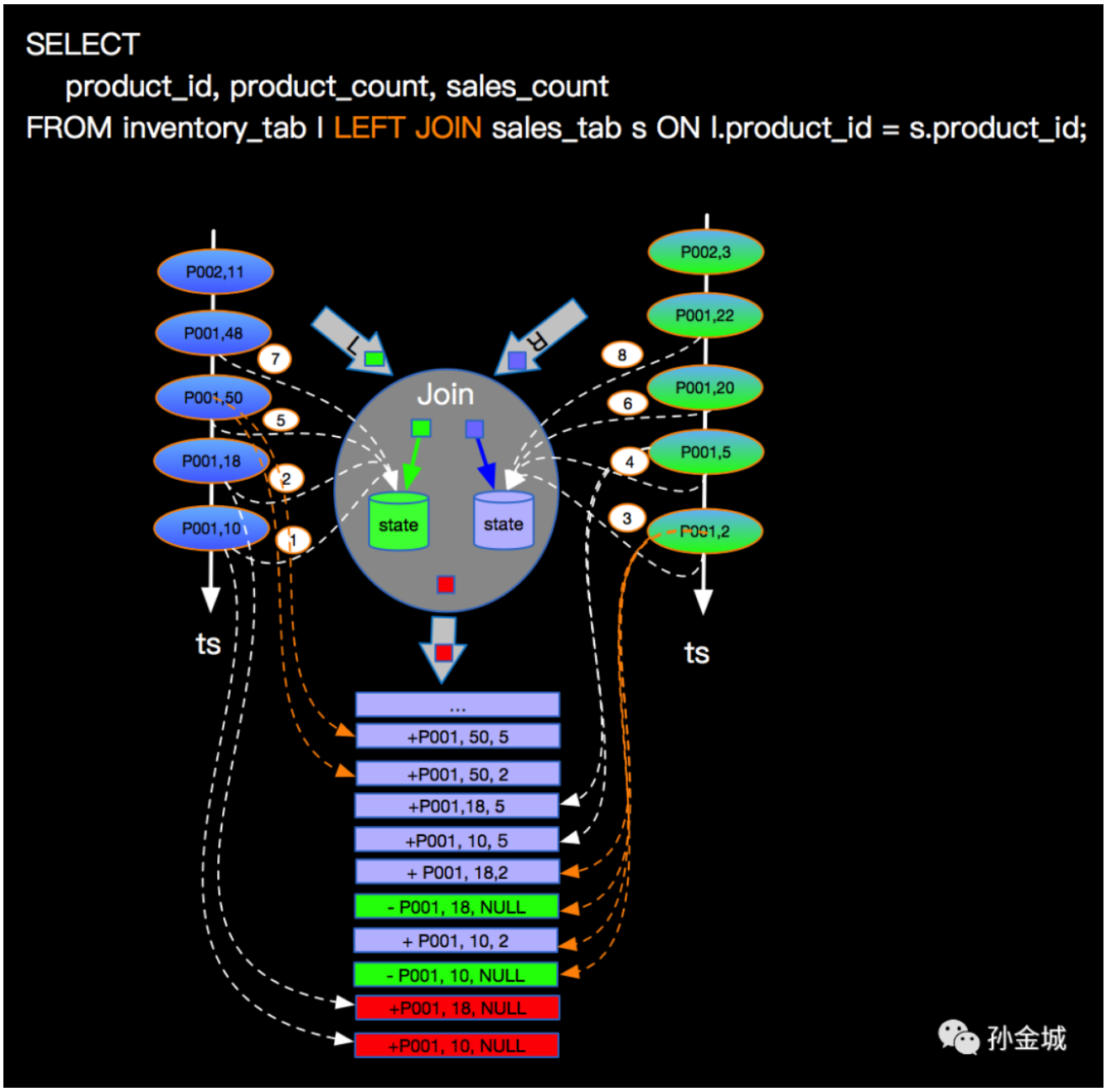
- INNER JOIN只有符合JOIN条件时候才会有JOIN结果流出到下游，比如右边最先来的1，2，3个事件，流入时候没有任何输出，因为左边还没有可以JOIN的事件；
- INNER JOIN两边的数据不论如何乱序，都能够保证和传统数据库语义一致，因为我们保存了左右两个流的所有事件到state中。

LEFT OUTER JOIN 实现

LEFT OUTER JOIN 可以简写 LEFT JOIN，语义上和INNER JOIN的区别是不论右流是否有JOIN的事件，左流的事件都需要流入下游节点，但右流没有可以JION的事件时候，右边的事件补NULL。同样我们以最简单的场景说明LEFT JOIN的实现，比如查询产品库存和订单数量，库存变化事件流和订单事件流进行LEFT JOIN，JION条件是产品ID，具体如下：



下图也是表达LEFT JOIN的语义，只是展现方式不同：



上图大主要关注点是当左边先流入1，2事件时候，右边没有可以join的事件时候会向下游发送

左边事件并补NULL向下游发出，当右边第一个相同的Join key到来的时候会将左边先来的事件发出的带有NULL的事件撤回（对应上面command的-记录，+代表正向记录，-代表撤回记录）。这里强调三点：

- 左流的事件当右边没有JOIN的事件时候，将右边事件列补NULL后流向下游；
- 当右边事件流入发现左边已经有可以JOIN的key的时候，并且是第一个可以JOIN上的右边事件（比如上面的3事件是第一个可以和左边JOIN key P001进行JOIN的事件）需要撤回左边下发的NULL记录，并下发JOIN完整（带有右边事件列）的事件到下游。后续来的4，5，6，8等待后续P001的事件是不会产生撤回记录的。
- 在Flink系统内部事件类型分为正向事件标记为“+”和撤回事件标记为“-”。

RIGHT OUTER JOIN 和 FULL OUTER JOIN

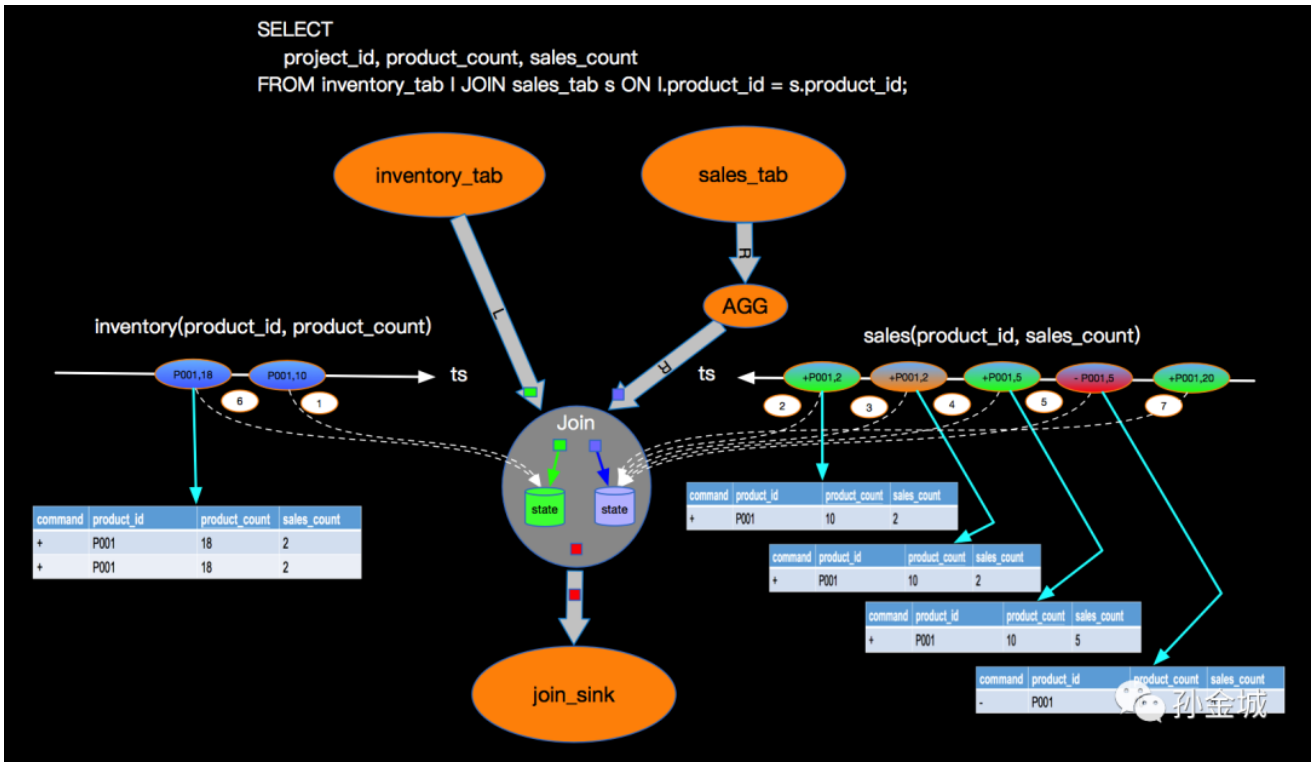
RIGHT JOIN内部实现与LEFT JOIN类似， FULL JOIN和LEFT JOIN的区别是左右两边都会产生补NULL和撤回的操作。对于State的使用都是相似的，这里不再重复说明了。

复杂场景介绍State结构

上面我们介绍了双流JOIN会使用State记录左右两边流的事件，同时我们示例数据的场景也是比较简单，比如流上没有更新事件（没有撤回事件），同时流上没有重复行事件。那么我们尝试思考下面的事件流在双流JOIN时候是怎么处理的?

上图示例是连续产生了2笔销售数量一样的订单，同时在产生一笔销售数量为5的订单之后，有将该订单取消了（或者退货了），这样在事件流上面就会是上图的示意，这种情况Flink内部如何支撑呢?

根据JOIN的语义以INNER JOIN为例，右边有两条相同的订单流入，我们就应该想下游输出两条JOIN结果，当有撤回的事件流入时候，我们也需要将已经下发下游的JOIN事件撤回，如下：



上面的场景以及LEFT JOIN部分介绍的撤回情况，需要Flink内部需要处理几个核心点：

- 记录重复记录（完整记录重复记录或者记录相同记录的个数）
- 记录正向记录和撤回记录（完整记录正向和撤回记录或者记录个数）
- 记录那一条事件是第一个可以与左边事件进行JOIN的事件

双流JOIN的State数据结构

在Flink内部对不同的场景有特殊的数据结构优化，本篇我们只针对上面说的情况（通用设计）介绍一下双流JOIN的State的数据结构和用途。

数据结构

- Map<JoinKey, Map<rowData, count>
  - 第一级MAP的key是Join key，比如示例中的P001， value是流上面的所有完整事件
  - 第二级MAP的key是行数据，比如示例中的P001, 2， value是相同事件值的个数

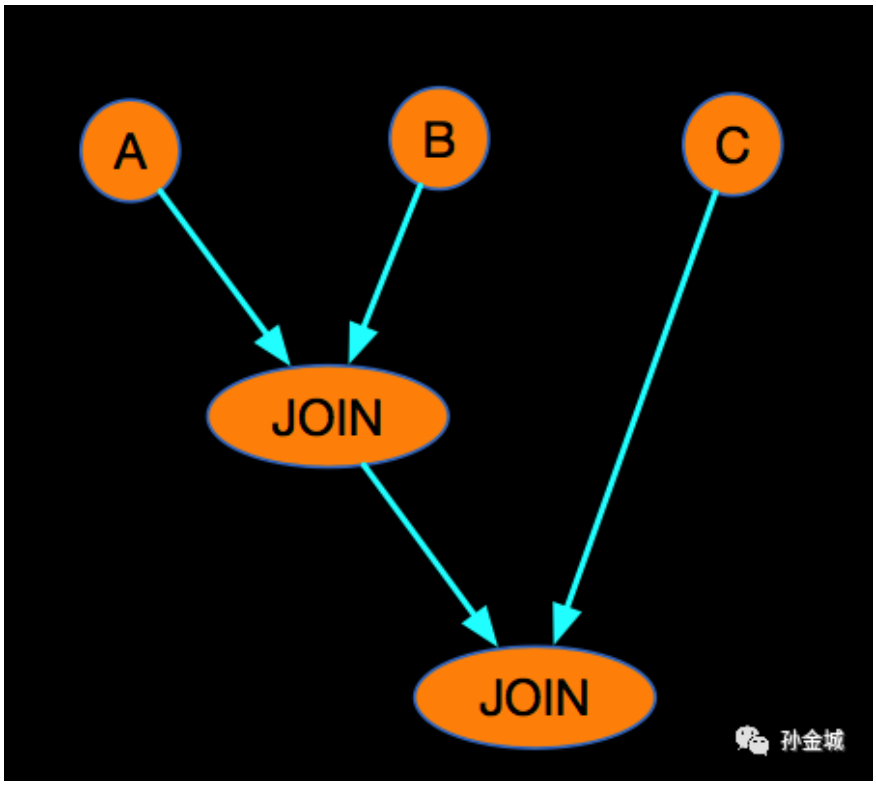
数据结构的利用

- 记录重复记录 - 利用第二级MAP的value记录重复记录的个数，这样大大减少存储和读取
- 正向记录和撤回记录 - 利用第二级MAP的value记录，当count=0时候删除改元素
- 判断右边是否产生撤回记录 - 根据第一级MAP的value的size来判断是否产生撤回，只有size由0变成1的时候（第一条和左可以JOIN的事件）才产生撤回

双流JOIN的应用优化

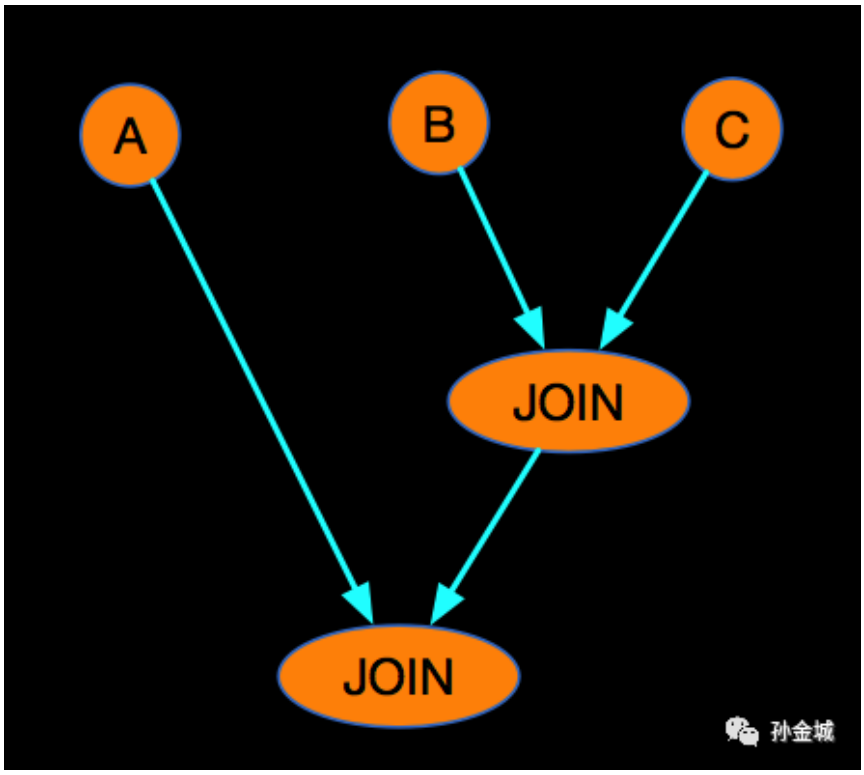
NULL造成的热点

比如我们有A LEFT JOIN B ON A.aCol = B.bCol LEFT JOIN C ON B.cCol = C.cCol 的业务，JOB的DAG如下：



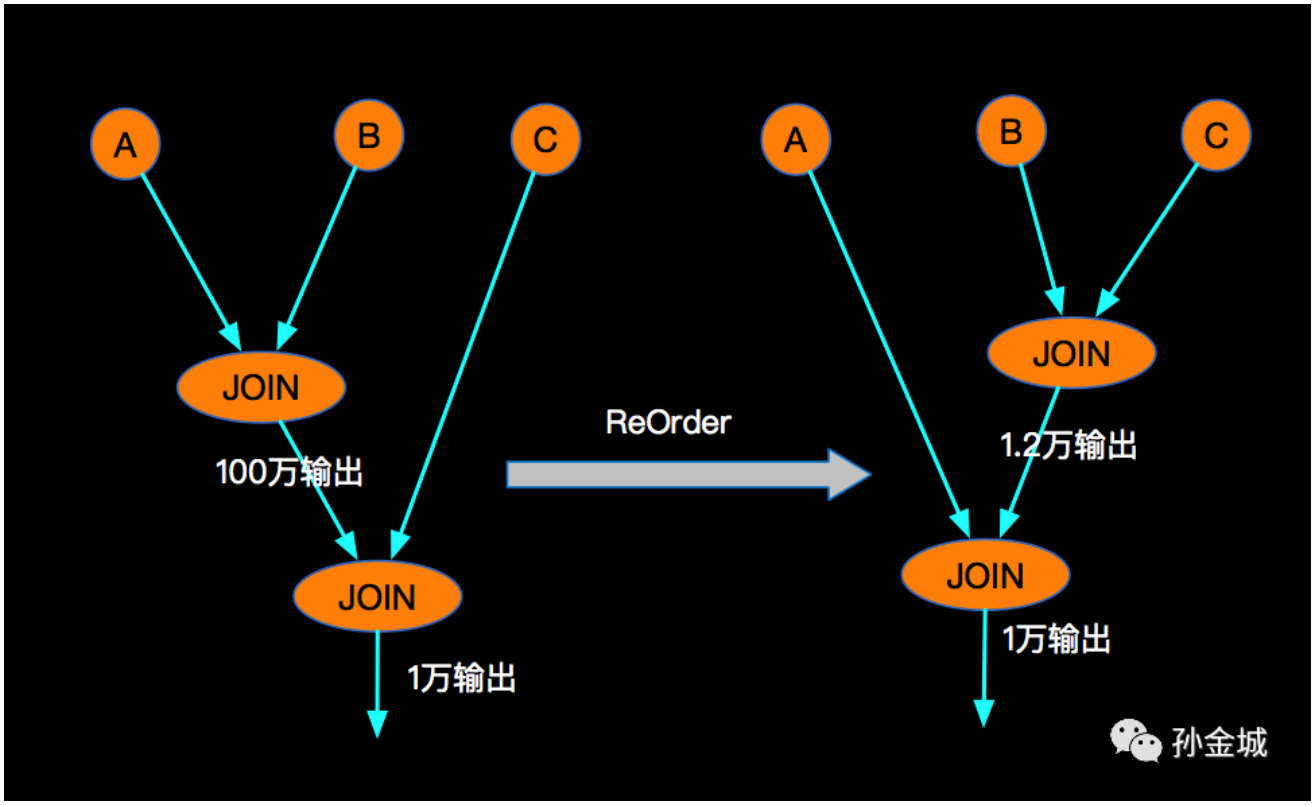
假设在实际业务中有这这样的特点，大部分时候当A事件流入的时候，B还没有可以JION的数据，但是B来的时候，A已经有可以JOIN的数据了，这特点就会导致，A LEFT JOIN B 会产生大量的 (A, NULL),其中包括B里面的 cCol 列也是NULL，这时候当与C进行LEFT JOIN的时候，首先Flink内部会利用cCol对AB的JOIN产生的事件流进行Shuffle， cCol是NULL进而是下游节点大量的NULL事件流入，造成热点。那么这问题如何解决呢?

我们可以改变JOIN的先后顺序，来保证A LEFT JOIN B 不会产生NULL的热点问题，如下：



JOIN ReOrder

对于JOIN算子的实现我们知道左右两边的事件都会存储到State中，在流入事件时候在从另一边读取所有事件进行JOIN计算，这样的实现逻辑在数据量很大的场景会有一定的state操作瓶颈，我们某些场景可以通过业务角度调整JOIN的顺序，来消除性能呢瓶颈，比如：A JOIN B ON A.acol = B.bcol JOIN C ON B.bcol = C.ccol. 这样的场景，如果 A与B进行JOIN产生数据量很大，但是B与C进行JOIN产生的数据量很小，那么我们可以强制调整JOIN的联接顺序，B JOIN C ON b.bcol = c.ccol JOIN A ON a.acol = b.bcol. 如下示意图：



小结

本篇初步向大家介绍传统数据库的JOIN的类型，语义和可以使用的查询优化，再以实际的例子介绍Flink上面的双流JOIN的实现和State数据结构设计，最后向大家介绍双流JION的使用优化。本篇只介绍了等值JOIN(ON 子句只有等值条件)，Flink目前也支持非等值联接条件和等值联接条件相结合使用，本质是相当于添加了WHERE子句。