# 概述

在数据库中join操作被称为连接，作用是能连接多个表的数据（通过连接条件），从多个表中获取数据合并在一起作为结果集返回给客户端。

连接两个表我们可以用两个关键字：on，using。on可以指定具体条件，using则指定相同名字和数据类型的列作为等值判断的条件，多个则通过逗号隔开。  
 如下：

on: select \* from A join B on A.id=B.id and B.name=''

using: select \* from A join B using(id,name) = select \* from A join B on A.id=B.id and A.name=B.name

# 分类

SQL标准中Join的类型包括5种：

内连接（INNER）

全外连接（FULL OUTER）

左外连接（LEFT OUTER）

右外连接（RIGHT OUTER）

交叉连接（CROSS）

为了测试不同join的使用场景，以下面表结构为例：

西天取经四人组：

|  |  |
| --- | --- |
| id | user\_name |
| 1 | 唐僧 |
| 2 | 猪八戒 |
| 3 | 孙悟空 |
| 4 | 沙僧 |

孙悟空的朋友们：

|  |  |
| --- | --- |
| id | user\_name |
| 1 | 孙悟空 |
| 2 | 牛魔王 |
| 3 | 蛟魔王 |
| 4 | 鹏魔王 |
| 5 | 狮驼王 |

## 内连接/inner join

**等值连接**：在连接条件中使用等于号(=)运算符比较被连接列的列值，其查询结果中列出被连接表中的所有列，包括其中的重复列。

**不等值连接**：在连接条件使用除等于运算符以外的其它比较运算符比较被连接的列的列值。这些运算符包括>、>=、<=、<、!>、!<和<>。

**自然连接**：在连接条件中使用等于(=)运算符比较被连接列的列值，但它使用选择列表指出查询结果集合中所包括的列，并删除连接表中的重复列。

### 内连接

内连接（join/inner join）基于连接谓词将两张表（如A和B）的列组合在一起，产生新的结果集（即A和B的公共部分）。



表现：**A和B满足连接条件记录的交集，如果没有连接条件，则是A和B的笛卡尔积**。

特点：**在MySQL中,cross join，inner join和join所实现的功能是一样的。因此在MySQL的官方文档中，指明了三者是等价的关系**。

举例SQL：

SELECT a.`user\_name`,`a.`over`,b.`over`

FROM user1 a

INNER JOIN user2 b

ON a.`user\_name`=`b.`user\_name`

;

### 隐式连接

语法：from A,B,C

表现：相当于无法使用on和using的join

特点：逗号是隐式连接运算符。隐式连接是SQL92中的标准内容，而在SQL99中显式连接才是标准，虽然很多人还在用隐私连接，但是它已经从标准中被移除。从使用的角度来说，还是推荐使用显示连接，这样可以更清楚的显示出多个表之间的连接关系和连接依赖的属性。

注：隐式连接属于内连接的一种。

### 交叉连接/cross join

语法：A join | inner join | cross join B

表现：A和B满足连接条件记录的**交集**，如果没有连接条件，则是A和B的笛卡尔积。

交叉连接（cross join），又称笛卡尔连接（cartesian join）或叉乘（Product），如果A和B是两个集合，它们的交叉连接就记为：A×B。比如A表4条记录，B表5条记录，则结果为20条记录。

**在MySQL中,cross join ，inner join和join所实现的功能是一样的。因此在MySQL的官方文档中，指明了三者是等价的关系**。

select \* from table1 a cross join table2 b ；

也可以写为 select \* from table1,table2;

举例SQL：

SELECT a.`user\_name`,a.`over`,b.`user\_name`,b.`over`

FROM user1 a

CROSS JOIN user2 b

;

注：CROSS JOIN是不需要查询条件关键字的。

## 外连接

### 左外连接/outer join

语法：A left join B

表现：**左表的数据全部保留，右表满足连接条件的记录展示，不满足的条件的记录则全是null**

左外连接（left outer join）又称左连接，比如A，B两表左连接，则结果以A表为依据，包含A的所有结果。

下面是左连接的两种应用方式：



图1表示包含A的结果集，图2含有where条件，排除属于A但是不属于B的情况（不属于B即为NULL）。

对于图2这种查询也可以使用NOT IN，但是不推荐使用NOT IN子查询，因为NOT IN不会使用索引。

举例SQL：

查询取经四人中哪些人不是悟空的结拜兄弟？

SELECT a.`user\_name`,a.`over`,b,`over`

FROM user` a

LEFT JOIN user2 b

ON a.`user\_name`=b.`user\_name`

WHERE b.user\_name IS NULL

;

### 右外连接/left join

**语法：**A right join B

**表现：右表的数据全部保留，左表满足连接条件的记录展示，不满足的条件的记录则全是null**

右外连接（right outer join）又称右连接，结果集包含右侧的表。



举例SQL：

查询悟空的结拜兄弟中哪些人没有去取经？

SELECT b.`user\_name`,b.`over`,a.`over`

FROM user1 a

RIGHT JOIN user2 b

ON a.`user\_name`=b.`user\_name`

WHERE a.user\_name IS NULL

;

### 全连接/full join

Full join其实就是左连接和右连接的结合，应用方式包括2种：



图1是全连接，图2含有where条件查询，去除A，B的公共部分。

**MySQL不支持全外连接，只支持左外连接和右外连接。如果要获取全连接的数据，要可以通过合并左右外连接的数据获取到。**

如：select \* from A left join B onA.name = B.name union select \* from A right join B on B.name = B.name;

这里union会自动去重，这样取到的就是全外连接的数据了。

MySQL本身是不直接支持full join的：

SELECT\*

FROMuser1 a

FULL JOIN

User2 b on a.user\_name=b.user\_name

;

客户端报错：

ERROR 1064（42000）：you have an error in your SQL syntax;

check the manual that corresponds to your MySQL server version for that right sytax to user near full join.

可以采用如下的方式解决（left和right join做union all）：

SELECT a.`user\_name`,a.`over`,b.`over`

FROM user1 a

LEFT JOIN user2 b ON a.`user\_name`=b.`user\_name`

**UNION ALL**

SELECT b.`user\_name`,b.`over`,a.`over`

FROM user1 a

RIGHT JOIN user2 b ON a.`user\_name`=b.`user\_name`

;

## 自然连接/natual join

语法：A natural join B ==== A natural left join B ==== A natural right join B

表现：**相当于不能指定连接条件的连接，MySQL会使用左右表内相同名字和类型的字段作为连接条件**。

特点：自然连接也分自然内连接，左外连接，右外连接，其表现和上面提到的一致，只是**连接条件由MySQL自动判定**。

## 其他链接

### SELF JOIN

SELF JOIN是同一个表的两个实例之间的JOIN操作。

对于层次结构问题，通常可以通过SELF JOIN来解决。

### NONEQUI JOIN

前面介绍的都是EQUAL JOIN（等值联接），即联接条件是基于“等于”运算符的联接操作。NONEQUI JOIN的联接条件包含“等于”运算符之外的运算符。

### SEMI JOIN

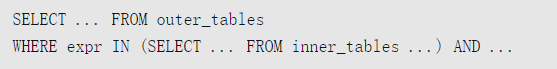
Semi-join并不是一种连接，它是**子查询的一种优化**。

SEMI JOIN是根据一个表中存在的相关记录找到另一个表中相关数据的联接。如果从左表返回记录，该联接被称为左半联接；如果从右表返回记录，该联接被称为右半联接。

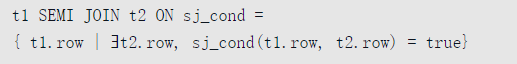
实现SEMI JOIN的方法有多种，如内部联接、子查询、集合操作等。在使用内部联接方式时，只从一个表中选择记录，然后应用DISTINCT。

对于用户来说，这一切都是透明的， MariaDB 5.3.3已经默认开始对于SEMI JOIN进行优化。

那什么是SEMI JOIN呢？其SQL语句的一般形式如下：



从严格的数学定义来说，SEMI JOIN的定义为：



由于目前Oracle和MySQL都将SEMI JOIN转换为了EXISTS语句，因此在执行效率上显得非常低。从理论上来说，SEMI JOIN应该只需要关心外部表中与子查询匹配的部分即可。这就是MariaDB要对SEMI JOIN进行的优化，在MariaDB中子查询变得实际可用得多，效率也得到了极大的提升。如果用户在实际环境中需要使用大量的SEMI JOIN子查询，那么MariaDB 5.3是最好的选择。从另一方面讲，如果用户能理解MariaDB对于子查询所做的优化，就能够将这些优化用在所编写的SQL语句中。

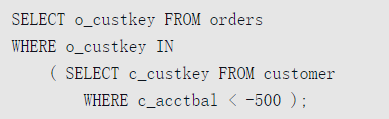
#### Table Pullout优化

##### 简介

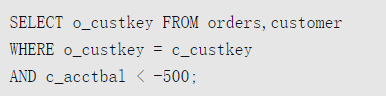
如果子查询字段为唯一索引，可以将IN子查询改写为关联查询。

##### 原理

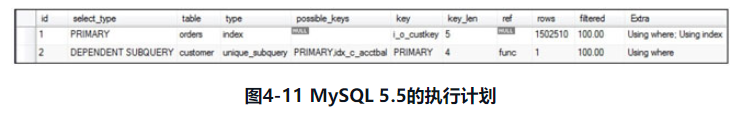
有些时候，一个子查询可以被重写为JOIN，例如：



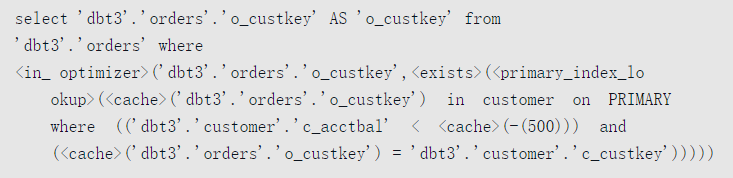
如果知道c\_custkey是唯一的，即主键或唯一索引，那么上述的SQL语句可以被重写为如下形式：



**Table Pullout的作用就是根据唯一索引将子查询重写为JOIN语句**。在MySQL 5.5中，上述SQL语句的执行计划如图4-11所示。



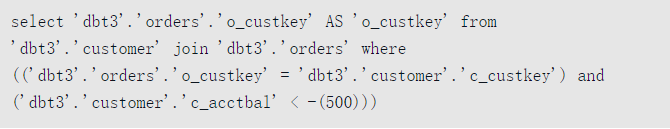
如果通过EXPLAIN EXTENDED和SHOW WARNINGS命令，可以看到如下的结果：



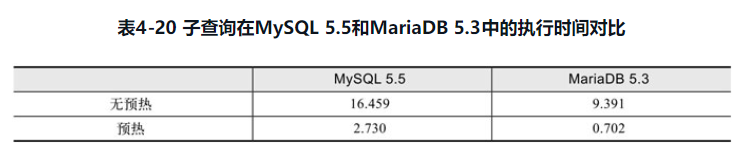
而在MariaDB 5.3中，优化器会对SQL语句进行重写，得到的执行计划如图4-12所示。



从图4-12可以发现，在MariaDB中，优化器没有将独立子查询重写为相关子查询。通过EXPLAINEXTENDED和SHOW WARNINGS命令，得到优化器的执行方式为：



很显然，优化器将上述子查询重写为JOIN语句，这就是Table Pullout优化。表4-20显示了上述子查询分别在MariaDB 5.3和MySQL 5.5中的执行时间。



上述SQL语句中选择的存储引擎为InnoDB。预热是指所要读取的表中的数据都已经在InnoDB存储引擎的缓冲池中，这时不涉及磁盘的读取。而无预热指的是数据库刚启动，缓冲池中没有数据，需要读取磁盘上的数据到缓冲池。

可以看到，在无预热的情况下，对于上述子查询，MariaDB比MySQL要快43%。对于数据已经预热的情况下，上述子查询在MariaDB中的执行时间接近MySQL中的四分之一。

#### Duplicate Weedout 优化

##### 简介

前一小节提到内部表查出的列是唯一的，因此MariaDB优化器会将子查询重写为JOIN语句，以提高SQL执行的效率。Duplicate Weedout优化是指**外部查询条件的列是唯一的，MariaDB优化器会先将子查询查出的结果进行去重**，这个步骤被称为Duplicate Weedout或者Duplicate Elimination。

**Duplicate weedout就是使用临时表对semi-join产生的结果集去重而达到目的**。如果使用了duplicate weedout，那么执行计划中可以看到Start temporary和End temporary。

注：Table pullout与Duplicate Weedout的区别就是是内表字段唯一还是外表的字段唯一。

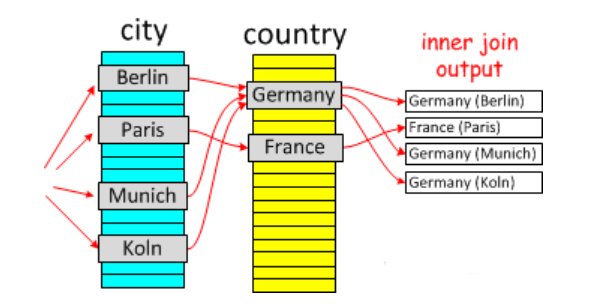
##### 原理

###### 分析一

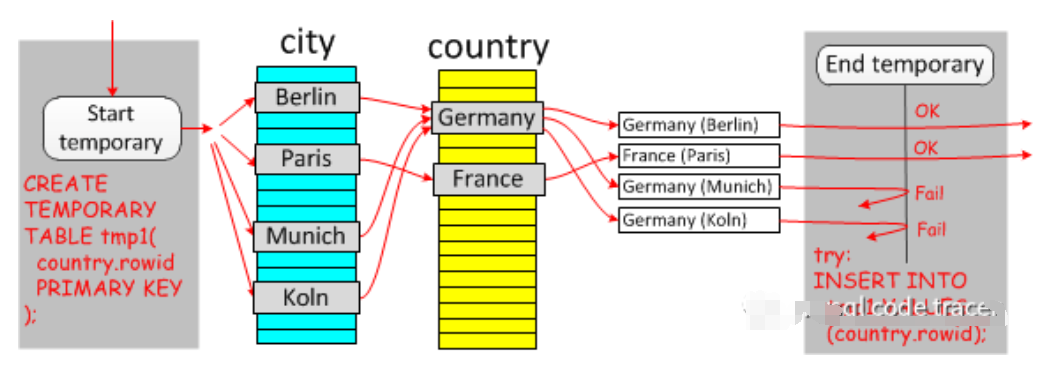
我们需要查询出拥有人口在1000万以上，并且人口比例超过国家1/3城市的国家，SQL如下：

select \*   
 from Country   
 where   
    Country.code IN (select City.Country  
                    from City   
                    where   
                      City.Population > 0.33 \* Country.Population and   
                      City.Population > 1\*1000\*1000);

通常city表（驱动表）和country表进行关联如下图所示：



看到结果集中Germany出现了3次，现在如果有了duplicate weedout，关联的过程图就像下面这样了：



1. 首先创建了一张临时表，表的字段为country表的rowid，且为主键
2. 将关联出的结果集中的country.rowid插入临时表
3. 如果成功就OK，如果有重复项，插入则失败

可以看到duplicate weedout就是使用临时表对semi-join产生的结果集去重而达到目的。如果使用了duplicate weedout，那么执行计划中可以看到Start temporary和End temporary。我们来分析一下上面的SQL执行计划

MariaDB [world]> explain select \* from Country where Country.code IN (select City.Country from City where City.Population > 0.33 \* Country.Population and City.Population > 1\*1000\*1000)\G  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
           id: 1  
  select\_type: PRIMARY  
        table: City  
         type: range  
possible\_keys: Population,Country  
          key: Population  
      key\_len: 4  
          ref: NULL  
         rows: 238  
        Extra: Using index condition; Start temporary  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
           id: 1  
  select\_type: PRIMARY  
        table: Country  
         type: eq\_ref  
possible\_keys: PRIMARY  
          key: PRIMARY  
      key\_len: 3  
          ref: world.City.Country  
         rows: 1  
        Extra: Using where; End temporary  
2 rows in set (0.00 sec)

从执行计划可以看出：

1. 将从City表扫描238行数据，并且将每一行数据代入到Country表去做基于主键的查找，所以总共需要扫描的行数为476行
2. 总共扫描了476行，那么你需要增加238次临时表查找，但是临时表是基于内存的，因此代价相对较低

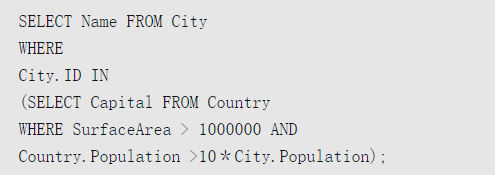
如果在MySQL查看执行计划的话，将会得到下面的执行计划：

mysql> explain select \* from Country where Country.code IN (select City.Country from City where City.Population > 0.33 \* Country.Population and City.Population > 1\*1000\*1000)  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
           id: 1  
  select\_type: PRIMARY  
        table: Country  
         type: ALL  
possible\_keys: NULL  
          key: NULL  
      key\_len: NULL  
          ref: NULL  
         rows: 239  
        Extra: Using where  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
           id: 2  
  select\_type: DEPENDENT SUBQUERY  
        table: City  
         type: index\_subquery  
possible\_keys: Population,Country  
          key: Country  
      key\_len: 3  
          ref: func  
         rows: 18  
        Extra: Using where  
2 rows in set (0.00 sec)

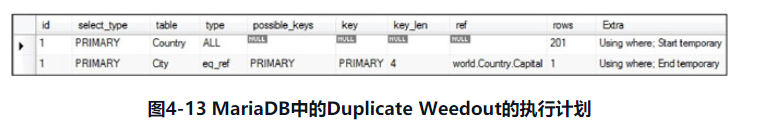
看这个执行计划，共需要扫描的行数是239+239\*18=4541行，这要比duplicate weedout代价要高得多。

###### 分析二

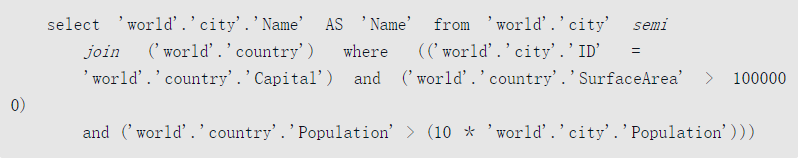
我们先来看下面的SQL语句：



因为City.ID是主键，所以应该对子查询得到的结果进行去重。在MariaDB数据库下，上述SQL语句的执行计划如图4-13所示。



Extra选项提示的Start temporary表示创建了一张去重的临时表，End temporary表示删除该临时表。而通过EXPLAIN EXTENDED和SHOW WARNINGS命令还可以发现：



通过上述的清单可以发现，**与Table Pullout不同的是，Duplicate Weedout显示的是SEMI JOIN而不是JOIN，其中原因在于多了一步去重的工作**。对于上述的执行计划，其扫描成本约为201+201\*1=402次。在无预热的情况下，执行所需时间为0.109秒。

而在MySQL 5.5下其执行计划如图4-14所示。



可以看到，在MySQL 5.5中该语句是相关子查询，扫描成本约为3868+3868\*246=1025020次。在无预热的情况下，执行该计划所需时间为2.293秒。

注：如果是在预热的情况下，可能就比较接近了。

##### 说明

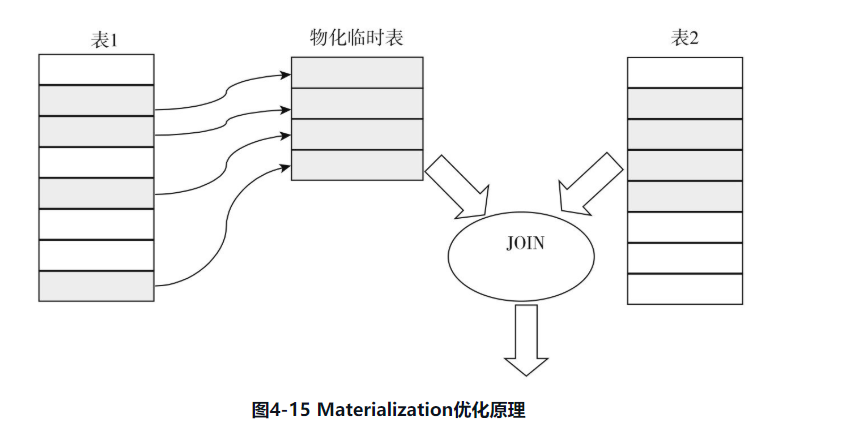
**其他说明：**

1. 如果出现DuplicateWeedout，可以在执行计划中看到Start temporary/End temporary
2. DuplicateWeedout策略可以用于**具有相关性的子查询，并且无所谓顺序**
3. **子查询不能有group by/聚合函数出现**，否则无法使用DuplicateWeedout
4. 可以通过@@optimizer\_switch='optimizer\_semijoin=off'来关闭DuplicateWeedout策略

#### Materialization优化

##### 简介

**如果子查询是独立子查询，则优化器可以选择将独立子查询产生的结果填充到单独一张物化临时表（materialized temporary table）中**，其实现原理如图4-15所示。



根据JOIN的顺序，Materialization优化可分为：

Materialization scan：JOIN是将物化临时表和表进行联接。

Materialization lookup：JOIN是将表和物化临时表进行联接。

注：

第一种方式涉及到物化表的全表扫描，我们也称之为"物化扫描"。

如果你用别的表去关联物化表，代价最低的方式就是用主键查找（没有重复数据）。因此，我们称之为"物化查找"。

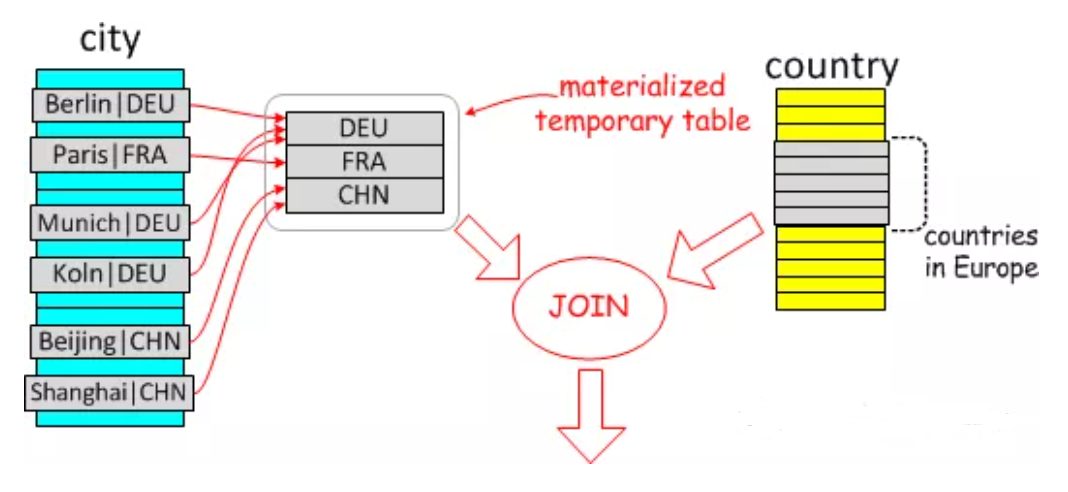
##### 原理

###### 分析一

假设我们用下面的SQL查询出有哪些欧洲国家拥有大城市：

select \* from Country   
 where Country.code IN (select City.Country   
                       from City   
                       where City.Population > 7\*1000\*1000)  
      and Country.continent='Europe'

子查询不是相关的，也就是说我们可以独立运行上层查询，半连接的思想就是将拥有大城市的City.Country写入到一个临时表，然后与欧洲的国家进行关联，如下图所示：



该连接根据关联顺序不同可以分为：

物化表关联欧洲国家

欧洲国家关联物化表

第一种方式涉及到物化表的全表扫描，我们也称之为"物化扫描"。

如果你用Countries表去关联物化表，代价最低的方式就是用主键查找（没有重复数据）。因此，我们称之为"物化查找"

Materialization-Scan

我们查找人口超过700万的城市，优化器会使用Materialization-Scan方式，EXPLAIN输出如下：

MariaDB [world]> explain select \* from Country where Country.code IN (select City.Country from City where  City.Population > 7\*1000\*1000);  
+----+--------------+-------------+--------+--------------------+------------+---------+--------------------+------+-----------------------+  
| id | select\_type  | table       | type   | possible\_keys      | key        | key\_len | ref                | rows | Extra                 |  
+----+--------------+-------------+--------+--------------------+------------+---------+--------------------+------+-----------------------+  
|  1 | PRIMARY      | <subquery2> | ALL    | distinct\_key       | NULL       | NULL    | NULL               |   15 |                       |  
|  1 | PRIMARY      | Country     | eq\_ref | PRIMARY            | PRIMARY    | 3       | world.City.Country |    1 |                       |  
|  2 | MATERIALIZED | City        | range  | Population,Country | Population | 4       | NULL               |   15 | Using index condition |  
+----+--------------+-------------+--------+--------------------+------------+---------+--------------------+------+-----------------------+  
3 rows in set (0.01 sec)

你可以看到

仍然有两个SELECT

第二个SELECT的select\_type=MATERIALIZED，也就是说它将执行的结果存储在一个有唯一键的临时表中。这个唯一键的作用就是防止产生重复数据。

第一个SELECT获得的表名是subquery2的结果集，这个结果集就是id=2的SELECT的结果物化得到的临时表

这是一个Materializetion-Scan策略的例子，优化器选择对物化表进行全表扫描。

关于执行代价估算，我们将从City表读取15条数据，然后写入物化表，然后再次读取它们（假设没有重复的情况下），然后需要做15次的eq\_ref访问Country表。所以总的来说，我们需要做45次读和15次写。

相比之下，如果你在MySQL将得到如下执行计划：

MySQL [world]> explain select \* from Country where Country.code IN (select City.Country from City where  City.Population > 7\*1000\*1000);  
+----+--------------------+---------+-------+--------------------+------------+---------+------+------+------------------------------------+  
| id | select\_type        | table   | type  | possible\_keys      | key        | key\_len | ref  | rows | Extra                              |  
+----+--------------------+---------+-------+--------------------+------------+---------+------+------+------------------------------------+  
|  1 | PRIMARY            | Country | ALL   | NULL               | NULL       | NULL    | NULL |  239 | Using where                        |  
|  2 | DEPENDENT SUBQUERY | City    | range | Population,Country | Population | 4       | NULL |   15 | Using index condition; Using where |  
+----+--------------------+---------+-------+--------------------+------------+---------+------+------+------------------------------------+

从执行计划来看，大约需要做（239+239\*15）=3824次读。

Materialization-Lookup

我们稍微修改一下查询条件，查找拥有人口超过100万城市的国家（不是700万）

MariaDB [world]> explain select \* from Country where Country.code IN (select City.Country from City where  City.Population > 1\*1000\*1000) ;  
+----+--------------+-------------+--------+--------------------+--------------+---------+------+------+-----------------------+  
| id | select\_type  | table       | type   | possible\_keys      | key          | key\_len | ref  | rows | Extra                 |  
+----+--------------+-------------+--------+--------------------+--------------+---------+------+------+-----------------------+  
|  1 | PRIMARY      | Country     | ALL    | PRIMARY            | NULL         | NULL    | NULL |  239 |                       |  
|  1 | PRIMARY      | <subquery2> | eq\_ref | distinct\_key       | distinct\_key | 3       | func |    1 |                       |  
|  2 | MATERIALIZED | City        | range  | Population,Country | Population   | 4       | NULL |  238 | Using index condition |  
+----+--------------+-------------+--------+--------------------+--------------+---------+------+------+-----------------------+  
3 rows in set (0.00 sec)

执行计划输出与Materialization-Scan类似：

<subquery2>表通过eq\_ref方式访问

通过索引distinct\_key来扫描

也就是说优化器将会进行索引扫描然后写入物化表。换句话说，我们将要使用Materialization-Lookup策略。

如果在MySQL（或者我们将optimizer\_swith='semijoin=off,materialization=off'），我们将会得到下面的执行计划：

MySQL [world]> explain select \* from Country where Country.code IN (select City.Country from City where  City.Population > 1\*1000\*1000) ;  
+----+--------------------+---------+----------------+--------------------+---------+---------+------+------+-------------+  
| id | select\_type        | table   | type           | possible\_keys      | key     | key\_len | ref  | rows | Extra       |  
+----+--------------------+---------+----------------+--------------------+---------+---------+------+------+-------------+  
|  1 | PRIMARY            | Country | ALL            | NULL               | NULL    | NULL    | NULL |  239 | Using where |  
|  2 | DEPENDENT SUBQUERY | City    | index\_subquery | Population,Country | Country | 3       | func |   18 | Using where |  
+----+--------------------+---------+----------------+--------------------+---------+---------+------+------+-------------+

两个执行计划都会对Country表进行全表扫描。而对于第二步，MariaDB将会填充物化表（从City表进行238次读取然后写入临时表），然后每读取一条Country表的记录都进行一次唯一性查找，因此就需要进行238次唯一键扫描。总的来说，第二步需要花费（239+238）=477次读和238次临时表写入。

而MySQL执行计划中的第二步，每读取一条Country表的记录，都要用到City.Country列上的索引读取18行。算下来总共需要18\*239=4302次读。如果子查询获取更少的数据，这个执行计划将会比物化性能更好。对于这个SQL，MariaDB也提供了一个选项（FirstMatch策略），但是这里优化器并没有选择FirstMatch。

带group的子查询

当子查询带有goup的情况下，MariaDB仍然能够使用物化策略（其他半连接策略不适用）

例如我们需要查找各大洲人口最多的城市

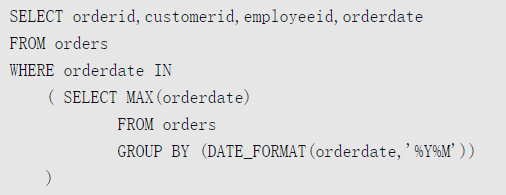
explain   
 select \* from City   
 where City.Population in (select max(City.Population) from City, Country   
                          where City.Country=Country.Code   
                          group by Continent)  
+------+--------------+-------------+------+---------------+------------+---------+----------------------------------+------+-----------------+  
| id   | select\_type  | table       | type | possible\_keys | key        | key\_len | ref                              | rows | Extra           |  
+------+--------------+-------------+------+---------------+------------+---------+----------------------------------+------+-----------------+  
|    1 | PRIMARY      | <subquery2> | ALL  | distinct\_key  | NULL       | NULL    | NULL                             |  239 |                 |  
|    1 | PRIMARY      | City        | ref  | Population    | Population | 4       | <subquery2>.max(City.Population) |    1 |                 |  
|    2 | MATERIALIZED | Country     | ALL  | PRIMARY       | NULL       | NULL    | NULL                             |  239 | Using temporary |  
|    2 | MATERIALIZED | City        | ref  | Country       | Country    | 3       | world.Country.Code               |   18 |                 |  
+------+--------------+-------------+------+---------------+------------+---------+----------------------------------+------+-----------------+  
4 rows in set (0.00 sec)

城市列表如下：

+------+-------------------+---------+------------+  
 | ID   | Name              | Country | Population |  
 +------+-------------------+---------+------------+  
 | 1024 | Mumbai (Bombay)   | IND     |   10500000 |  
 | 3580 | Moscow            | RUS     |    8389200 |  
 | 2454 | Macao             | MAC     |     437500 |  
 |  608 | Cairo             | EGY     |    6789479 |  
 | 2515 | Ciudad de México  | MEX     |    8591309 |  
 |  206 | São Paulo         | BRA     |    9968485 |  
 |  130 | Sydney            | AUS     |    3276207 |  
 +------+-------------------+---------+------------+

###### 分析二

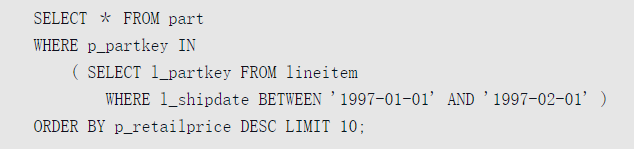
MariaDB SEMI JOIN的优化，其语句为：



在MariaDB 5.3中，上述SQL语句的执行计划如图4-16所示。



可以看到，在进行JOIN时（也就是id为1的步骤），先扫描的表是<subquery2>，然后是orders，因此这是Materialize scan优化。下面的子查询同样可以利用Materialization来进行优化。



其执行计划如图4-17所示。



可以看到上述的SQL语句和之前的SQL语句都可以通过Materialization来进行优化，但是在进行JOIN时顺序还是有所不同的。图4-17显示，这次SQL语句先扫描part表，然后再来联接<subquery2>这张物化的临时表，因此这时为Materialization lookup优化。

##### 说明

可以用于无关联的IN子查询。子查询也可以包含group或者聚合函数等

在执行计划中可以看到子查询的type=MATERIALIZED，父查询中可以看到一行table=<subqueryN>

同时打开optimizer\_switch中的materialization=on和semijoin=on才算启用semi-join materialization

materialization=on|off flag与Non-semijoin materialization共享

#### 优化器选项

optimizer\_switch（子查询相关）：

materialization=on

semijoin=on

loosescan=on

firstmatch=on

duplicateweedout=on

subquery\_materialization\_cost\_based=on

以上基于官方版本8.0.19

### ANTI SEMI JOIN

与SEMI JOIN相反的是ANTI SEMI JOIN，它根据一个表中不存在的记录而从另一个表中返回记录。使用OUTER JOIN并过滤外部行，可以实现ANTI SEMI JOIN。

# 原理

## 执行顺序

在连接过程中，MySQL各关键字执行的顺序如下：

from -> on|using ->where -> group by -> having ->select ->order by -> limit

可以看到，**连接的条件是先于where的**，也就是先连接获得结果集后，才对结果集进行where筛选，所以在使用join的时候，我们要尽可能提供连接的条件，而少用where的条件，这样才能提高查询性能。

**注：**在分布式数据库实践中，我们会把ON字段中非等值的部分放到WHERE中，这样就可以在ON关联条件中尽可能保证索引不失效，提升JOIN查询效率。

**Q：**如果想筛选驱动表的数据，例如左连接筛选左表的数据，该在连接条件还是where筛选？

**A：**要通过where筛选，连接条件只影响连接过程，不影响连接返回的结果数（某些情况下连接条件会影响连接返回的结果数，例如左连接中，右侧匹配的数据不唯一的时候）

## 参数

join语句是相对比较耗费性能，因为在执行join语句的时候必然要有一个比较的过程。

逐条比较两个表的语句是比较慢的，因此我们可以把两个表中数据依次读进一个**内存块**中（**子查询是建立临时表，这个要比内存块要耗时，因此子查询一般建议采用join优化**）, 以MySQL的InnoDB引擎为例，使用以下语句我们必然可以查到相关的内存区域：

show variables like '%buffer%'



**join\_buffer\_size**的大小将会影响我们join语句的执行性能。

大部分数据库中的数据最终要保存到硬盘上，并且以文件的形式进行存储。

以MySQL的InnoDB引擎为例：

InnoDB以页(page)为基本的IO单位，每个页的大小为16KB

InnoDB会为每个表创建用于存储数据的.ibd文件



验证：



这意味着我们有多少表要连接就需要读多少个文件（需要读取磁盘上的多个表对应的ibd文件），虽然可以利用索引，但还是免不了频繁的移动硬盘的磁头，频繁的移动磁头会影响性能。

在执行join操作前查看内存中的信息：



执行join操作查看内存信息：



为什么buff/cache占了那么多内存，可用内存即availlable还有1.1G？

为什么你可以通过两条命令来清理buff/cache占用的内存，而想要释放used只能通过结束进程来实现?

sync; echo 3 > /proc/sys/vm/drop\_caches就可以清理buff/cache了。

Linux会把内存当作是硬盘的高速缓存。

在join算法中，无索引的话,嵌套循环。有索引的话，则可以利用索引来提升性能。

在扫描过程中，数据库会选择一个表把他要返回以及需要进行和其他表进行比较的数据放进join\_buffer。

# JOIN算法

JOIN有三种算法，分别是Nested Loop Join，Hash join，Sort Merge Join。MySQL官方文档中提到，MySQL只支持Nested Loop Join这一种算法。

**无索引的话，嵌套循环。有索引的话，则可以利用索引来提升性能。**

表连接方式：嵌套循环连接(nested loops join)、哈希连接(hash join) 、排序合并连接(sort merge join)。

select

c.\*

from

hotel\_info\_original c

left join

hotel\_info\_collection h

on

c.hotel\_type=h.hotel\_type

and

c.hotel\_id =h.hotel\_id

where

h.hotel\_id is null

这个sql是用来查询出c表中有h表中无的记录，所以想到了用left join的特性（返回左边全部记录，右表不满足匹配条件的记录对应行返回null）来满足需求，不料这个查询非常慢。先来看查询计划：



rows代表这个步骤相对上一步结果的每一行需要扫描的行数，可以看到这个sql需要扫描的行数为35773\*8134，非常大的一个数字。

在EXPLAIN结果中，第一行出现的表就是驱动表。

**MySQL支持的Join算法很有限：**

Nested-Loop Join (NL Join)

Batched Key Access Join (BKA Join)

Block Nested-Loop Join（版本< 8.0.20）

Hash Join (版本>= 8.0.18)

如果Join两侧的任何一张表上join key列存在索引，那么MySQL通常会使用基于索引的BKA Join或NL Join，我们实际使用中的绝大多数情形都对应这种方式。如果Join两侧都没有索引可以用，那么MySQL只能退而求其次选择Block Nested-Loop Join或Hash Join（取决于MySQL版本）。

## Nested Loop Join

### 概述

NLJ算法即Nested Loop Join，就是扫描一个表（外表，也叫**驱动表**），每读到一条记录，就根据join字段上的索引去另一张表（内表）里查找。内表（一般是带索引的表）被外表（也叫**驱动表，一般为小表**，不仅相对其他表为小表，而且记录数的绝对值也小，不要求有索引）驱动，外表返回的每一行都要在内表中检索与其匹配的行，因此**整个返回的结果集不能太大（大于1万不适合）**。

**对于被连接的数据子集较小的情况，Nested Loop是个较好的选择**。Nested Loop就是扫描一个表（外表），每读到一条记录，就根据Join字段上的索引去另一张表（内表）里面查找，**若Join字段上没有索引查询优化器一般就不会选择 Nested Loop**。

**驱动表：**就是在嵌套循环连接和哈希连接中，用来**最先获得数据**，并以此表的数据为依据，逐步获得其他表的数据，直至最终查询到所有满足条件的数据的第一个表。驱动表不一定是表，有可能是数据集，即由某个表中满足条件的数据行，组成子集合后，再以此子集合作为连接其他表的数据来源。这个子集合，才是真正的驱动表，有时候为了简洁，直接将最先按照条件或得子集合的那张表叫做驱动表。我们常说，**驱动表一定是小表**，指的是根据条件获得的子集合一定要小，而不是说实体表本身一定要小，大表如果获得的子集合小，一样可以简称这个大表为驱动表。

**如果有三个及以上的表，则会先使用NLJ算法得到一、二个表的结果集，并将该结果集作为外层数据，遍历结果集到后第三个表中查询数据**。

一个简单的嵌套循环联接（NLJ）算法，循环从第一个表中依次读取行，取到每行再到联接的下一个表中循环匹配。这个过程会重复多次直到剩余的表都被联接了。假设表t1、t2、t3用下面的联接类型进行联接：

Table Join Type

t1 range

t2 ref

t3 ALL

如果使用的是简单NLJ算法，那么联接的过程像这样：

for each row in t1 matching range {

for each row in t2 matching reference key {

for each row in t3 {

if row satisfies join conditions,

send to client

}

}

}

因为NLJ算法是通过外循环的行去匹配内循环的行，所以内循环的表会被扫描多次。

由此可知道，on a.id = b.aid 代表着驱动表无法使用此索引，是给被驱动表用的。

Nested-Loop Join简单粗暴容易理解，就是通过双层循环比较数据来获得结果，但是这种算法显然太过于粗鲁，如果每个表有1万条数据，那么对数据比较的次数=1万\*1万=1亿次，很显然这种查询效率会非常慢。



当然mysql肯定不会这么粗暴的去进行表的连接，所以就出现了后面的两种对Nested-Loop Join优化算法，在执行join查询时mysql会根据情况选择后面的两种优join优化算法的一种进行join查询。

### 分类

具体来说Nested Loop Join又分三种细分的算法：

1、SNLJ

2、BNLJ

3、INLJ

对于连接语句select \* from A left join B on A.id=B.tid，这三种算法是怎么连接的。

#### Simple Nested Loop Join（SNLJ）

SNLJ是在**没有使用到索引**的情况下，通过**两层循环全量扫描**连接的两张表，得到符合条件的两条记录则输出。也就是让两张表做**笛卡尔积**进行扫描，是比较暴力的算法，会**比较耗时**。其过程如下：

for (a in A) {

for (b in B) {

if (a.id == b.tid) {

output <a, b>;

}

}

}

当然，MySQL即使在无索引可用，或者判断全表扫描可能比使用索引更快的情况下，还是不会选择使用过于粗暴的SNLJ算法，而是采用下面的算法。

#### Block Nested Loop Join（BNLJ）

BNLJ是MySQL**无法使用索引**的时候采用的join算法。会将外层循环的行分片存入join buffer, 内层循环的每一行与整个buffer中的记录做比较，从而减少内层循环的次数。

**Block 块，也就是说每次都会取一块数据到内存以减少I/O的开销**。当没有索引可以使用的时候，MySQL InnoDB就会使用这种算法。

注：BNLJ其实就是通过设置一个块大小，减少交互的IO次数。

具体逻辑如下:

for (blockA in A.blocks) {

for (b in B) {

if (b.tid in blockA.id) {

output <a, b>;

}

}

}

相比于SNLJ算法，BNLJ算法通过外层循环的结果集的分块，可以有效的减少内层循环的次数。

##### 原理

Block Nested-Loop Join其优化思路是减少外层表的循环次数，Block Nested-Loop Join通过一次性缓存多条数据，把参与查询的列缓存到join buffer 里，,然后拿join buffer里的数据批量与内层表的数据进行匹配，从而减少了外层循环的次数，**当我们不使用Index Nested-Loop Join的时候，默认使用的是Block Nested-Loop Join**。

BLJ算法即Block Nested-Loop Join（缓存块嵌套循环连接），是MySQL自己创建的方式。将指定的外层键对应的被驱动表缓存起来以提高性能。

Join操作使用内存(join\_buffer\_size)：应用程序经常会出现一些两表（或多表）Join的操作需求，MySQL在完成某些Join需求的时候（all/index join），为了减少参与Join的“被驱动表”的读取次数以提高性能，需要使用到Join Buffer来协助完成 Join操作。当Join Buffer太小，MySQL不会将该Buffer存入磁盘文件，而是先将Join Buffer中的结果集与需要Join的表进行Join操作，然后清空Join Buffer中的数据，继续将剩余的结果集写入此Buffer中，如此往复。这势必会造成被驱动表需要被多次读取，成倍增加IO访问，降低效率。

for each row in t1 matching range {

for each row in t2 matching reference key {

store used columns from t1, t2 in join buffer

if buffer is full {

for each row in t3 {

for each t1, t2 combination in join buffer {

if row satisfies join conditions,

send to client

}

}

empty buffer

}

}

}

if buffer is not empty {

for each row in t3 {

for each t1, t2 combination in join buffer {

if row satisfies join conditions,

send to client

}

}

}

对上面的过程解释如下：

1、将t1、t2的联接结果放到缓冲区，直到缓冲区满为止；

2、遍历t3，内部再循环缓冲区，并找到匹配的行，发送到客户端；

3、清空缓冲区；

4、重复上面步骤，直至缓冲区不满；

5、处理缓冲区中剩余的数据，重复步骤2。

设S是每次存储t1、t2组合的大小，C是组合的数量，则t3被扫描的次数为：

(S \* C)/join\_buffer\_size + 1

由此可见，随着join\_buffer\_size的增大，t3被扫描的次数会较少，如果join\_buffer\_size足够大，大到可以容纳所有t1和t2联接产生的数据，t3只会被扫描1次。

举例来说，外层循环的结果集是100行，使用SNLJ算法需要扫描内部表100次，如果使用BNLJ算法，假设每次分片的数量是10，则会先把对Outer Loop表(外部表)每次读取的10行记录放到join buffer,然后在InnerLoop表(内部表)中每次循环都直接匹配这10行数据，这样内层循环只需要10次，对内部表的扫描减少了9/10，所以BNLJ算法就能够显著减少内层循环表扫描的次数。

当然这里，不管SNLJ还是BNLJ算法，他们总的比较次数都是一样的，都是要拿外层循环的每一行与内层循环的每一行进行比较。

BNLJ算法减少的是总的扫描行数，SNLJ算法是外层循环要一行行扫描A表的数据，然后取A.id去表B一行行扫描看是否匹配。而BNLJ算法则是外层循环要一行行扫描A表的数据，然后放到内存分块里，然后去表B一行行扫描，扫描出来的B的一行数据与内存分块里的A的数据块进行比较。这里可以一次就是很多行A的数据与B的数据进行比较，而且是在内存中进行比较，速度更加快了。

##### 影响因素

这里BNLJ算法总的扫描行数是由外层循环的数据量N，和分块数量K还有内层循环的数据量M决定的。其中分块数量K与外层循环的数据量N又是息息相关的，我们可以表示为λN，其中λ取值为(0~1)。则总扫描次数C=N+λNM。

可以看出，在这个式子里，N和λ的大小都会影响扫描行数，但是λ才是影响扫描行数的关键因素，这个值越小越好（除非N和M的差值非常大，这时候N才会成为关键影响因素）。

那什么会影响λ的大小呢？那就是MySQL的join\_buffer\_size设置项的大小了。λ和join\_buffer\_size成倒数关系，join\_buffer\_size越大，分块越大，λ越小，分块数量也就越少，也就是外层循环的次数也越少。所以在使用不上索引的时候，我们要优先考虑扩大join\_buffer\_size的大小，这样优化效果会更明显。而在能使用上索引的时候，MySQL会使用以下算法来进行join。

Block块，也就是说每次都会取一块数据到内存以减少I/O的开销。**当没有索引可以使用的时候，MySQL InnoDB就会使用这种算法**。

##### 案例

如SQL：select \* from user tb1 left join level tb2 on tb1.id=tb2.user\_id

当level表的user\_id不为索引的时候执行过程会如下图：



**注意：**

1、使用Block Nested-Loop Join算法需要开启优化器管理配置的optimizer\_switch的设置block\_nested\_loop为on默认为开启，如果关闭则使用Simple Nested-Loop Join算法；

通过指令：Show variables like 'optimizer\_switc%'; 查看配置

2、设置join buffer的大小

通过join\_buffer\_size参数可设置join buffer的大小

指令：Show variables like 'join\_buffer\_size%';

#### Index Nested Loop Join（INLJ）

##### 背景

Nested-Loop Join是最简单的Join形式，可以看作一个两层For循环。对于外表（也称为驱动表）中的每一行，循环检查内表（也称为被驱动表）的每一行，如果满足Join条件则作为Join结果输出。如果Join Key在内侧表有索引可用，那么内表的循环可以大大简化——只要查索引即可拿到可以Join的行，无需遍历整个表。我们也将这种带索引的NL Join称为Index Nested-Loop Join。

# Nested-Loop Join

for outer\_row in outer\_table:

for inner\_row in inner\_table:

if join\_condition is True:

output (outer\_row, inner\_row)

# Index Nested-Loop Join

for outer\_row in outer\_table:

for inner\_row in inner\_index.lookup(outer\_join\_key):

if join\_condition is True:

output (outer\_row, inner\_row)

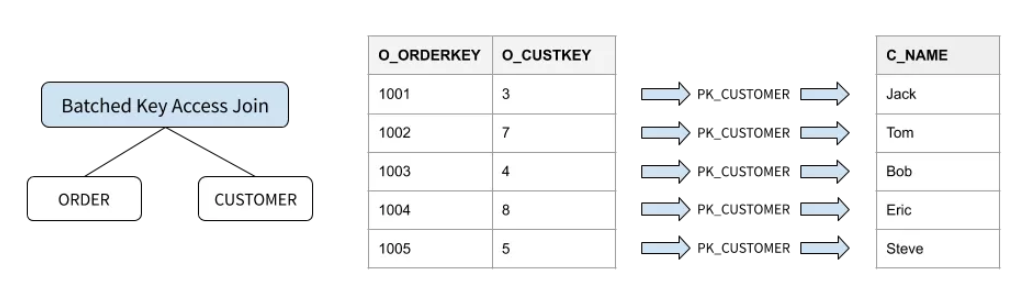
下面的例子中，orders表通过customer表的主键c\_custkey与之进行Join，MySQL会使用Index NL Join算法完成Join。

/\* Query 1 \*/

SELECT o\_orderkey, o\_custkey, c\_name

FROM orders JOIN customer ON o\_custkey = c\_custkey

WHERE o\_orderkey BETWEEN 1001 AND 1005



##### 原理

Index Nested-Loop Join（索引嵌套循环连接）其优化的思路主要是为了减少内层表数据的匹配次数，简单来说**Index Nested-Loop Join就是通过外层表匹配条件直接与内层表索引进行匹配，避免和内层表的每条记录去进行比较**，这样极大的**减少了对内层表的匹配次数，从原来的匹配次数=外层表行数\*内层表行数,变成了外层表的行数\*内层表索引的高度**，极大的提升了join的性能。

注：这种情况内表不需要全表扫描，会使用到索引。

假设A表的数据行为10，B表的数据行为100，且B.tid建立了索引，则对于select \* from A left join B on A.id=B.tid，MySQL会采用Index Nested Loop Join。其过程如下：

for (a in A) {

if (a.id in B.tid.Index) {

output <a, tid.Index所在行>;

}

}

总共需要循环10次A，每次循环的时候通过索引查询一次B的数据。而如果我们反过来是B left join A的话，总共要循环100次B，由此可见如果使用join的话，需要让小表做驱动表，这样才能有效减少循环次数。但是需要注意的是，这个结论的前提是可以使用被驱动表的索引。

**INLJ内层循环读取的是索引，可以减少内存循环的次数，提高join效率，但是也有缺点的，就是如果扫描的索引是非聚簇索引，并且需要访问非索引的数据，会产生一个回表读取数据的操作，这就多了一次随机的I/O操作**。例如上面在索引里匹配到了tid，还要去找tid所在的行在磁盘所在的位置。

##### 案例

如SQL：select \* from user tb1 left join level tb2 on tb1.id=tb2.user\_id

当level表的user\_id为索引的时候执行过程会如下图：



**注意：**使用Index Nested-Loop Join算法的前提是匹配的字段必须建立了索引。

#### BKA Join

BKA Join可以看作一个性能优化版的Index Nested-Loop Join。之所以称为Batched，是因为它的实现使用了存储引擎提供的MRR（Multi-Range Read）接口批量进行索引查询，并**通过PK排序的方法，将随机索引回表转化为顺序回表，一定程度上加速了查索引的磁盘IO**。

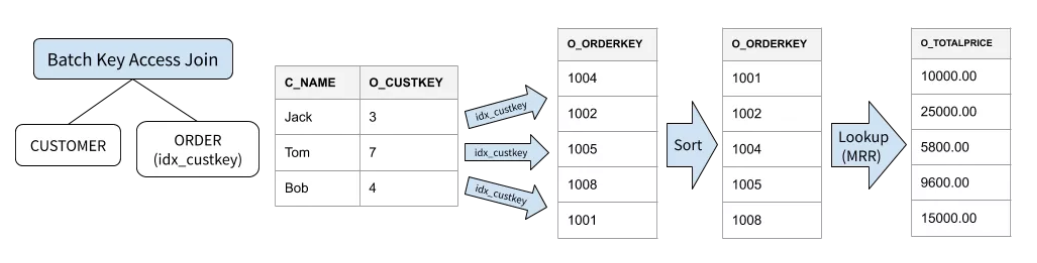
下面的例子中，Join Key命中的是二级索引，并且SELECT的列包含二级索引中所不包含的列，因此需要进行索引回表得到完整的Join结果。

/\* Query 2 \*/

SELECT c\_name, c\_custkey, o\_orderkey, o\_totalprice

FROM customer JOIN orders ON c\_cutkey = o\_custkey

WHERE c\_custkey BETWEEN 13 AND 15



通常OLTP查询中Join驱动侧的数据量不大，并且Join往往都有能匹配的索引。这种情况下，NL Join、BKA Join的代价与驱动侧的数据量呈线性相关，可以迅速计算出结果。

### 总结

不论是Index Nested-Loop Join还是Block Nested-Loop Join都是在Simple Nested-Loop Join的算法的基础上减少嵌套的循环次数，不同的是Index Nested-Loop Join是通过索引的机制减少内层表的循环次数，Block Nested-Loop Join是通过一次缓存多条数据批量匹配的方式来减少外层表的循环次数，通过 理解join的算法原理我们可以得出以下表连接查询的优化思路。

**在项目开发中如果需要使用join语句，如何优化提升性能?**

分为两种情况，数据规模小的，数据规模大的：

1. 数据规模较小，全部放进内存；
2. 数据规模较大。
3. 可以通过增加索引来优化join语句的执行速度(减少内层表的循环次数)。 可以通过冗余信息来减少join的次数，尽量减少表连接的次数，一个SQL语句表连接的次数不要超过5次。对被驱动表的join字段上建立索引。
4. 当被驱动表的join字段上无法建立索引的时候，设置足够的Join Buffer Size（一次缓存的数据越多，那么外层表循环的次数就越少）。
5. 减少不必要的字段查询（字段越少，join buffer所缓存的数据就越多，外层表的循环次数就越少）
6. 永远用小结果集驱动大结果集(其本质就是减少外层循环的数据数量)，尽量减少join语句中的Nested Loop的循环总次数。
7. 优先优化Nested Loop的内层循环，因为内层循环是循环中执行次数最多的，每次循环提升很小的性能都能在整个循环中提升很大的性能。

## Hash Join

Hash Join是做**大数据集连接时的常用方式**，优化器使用两个表中较小（相对较小）的表利用Join Key在内存中建立散列表，然后扫描较大的表并探测散列表，找出与Hash表匹配的行。

这种方式**适用于较小的表完全可以放于内存中的情况**，这样总成本就是访问两个表的成本之和。但是在表很大的情况下并不能完全放入内存，这时优化器会将它分割成若干不同的分区，不能放入内存的部分就把该分区写入磁盘的临时段，此时要求有较大的临时段从而尽量提高I/O的性能。它能够很好的工作于没有索引的大表和并行查询的环境中，并提供最好的性能。大多数人都说它是Join的重型升降机。**Hash Join只能应用于等值连接(如WHERE A.COL3 = B.COL4)，这是由Hash的特点决定的**。

## Sort Merge Join

通常情况下Hash Join的效果都比排序合并连接要好，然而如果**两表已经被排过序，在执行排序合并连接时不需要再排序了，这时Merge Join的性能会优于Hash Join**。Merge join的操作通常分三步：

1、对连接的每个表做table access full;

2、对table access full的结果进行排序。

3、进行merge join对排序结果进行合并。

在全表扫描比索引范围扫描再进行表访问更可取的情况下，Merge Join会比Nested Loop性能更佳。当表特别小或特别巨大的时候，实行全表访问可能会比索引范围扫描更有效。Merge Join的性能开销几乎都在前两步（Merge Join的主要开销是排序开销，如果能通过建立聚簇索引（如果Query必须显示排序），可以极大提高Merge Join的性能）。**Merge Join可适于于非等值Join（>，<，>=，<=，但是不包含!=，也即<>）**。

## 区别

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | Nested Loop | Hash Join | Merge Join |
| 使用条件 | 任何条件 | 等值连接（=） | 等值或非等值连接(>，<，=，>=，<=)，‘<>’除外 |
| 相关资源 | CPU、磁盘I/O | 内存、临时空间 | 内存、临时空间 |
| 特点 | 当有高选择性索引或进行限制性搜索时效率比较高，能够快速返回第一次的搜索结果。 | 当缺乏索引或者索引条件模糊时，Hash Join比Nested Loop有效。  通常比Merge Join快。在数据仓库环境下，如果表的纪录数多，效率高。 | 当缺乏索引或者索引条件模糊时，Merge Join比Nested Loop有效。  非等值连接时，Merge Join比Hash Join更有效 |
| 缺点 | 当索引丢失或者查询条件限制不够时，效率很低；当表的纪录数多时，效率低。 | 为建立哈希表，需要大量内存。第一次的结果返回较慢。 | 所有的表都需要排序。它为最优化的吞吐量而设计，并且在结果没有全部找到前不返回数据。 |

# SQL开发技巧

1、尽量增加连接条件，减少join后数据集的大小

2、用小结果集驱动大结果集，将筛选结果小的表首先连接，再去连接结果集比较大的表

3、被驱动表的被join的字段要建立索引，且使用上索引。使用上索引包括使用该字段，且不会有索引失效的情况出现

4、设置足够大的join\_buffer\_size

## 使用join更新表

如何更新使用过滤条件中包含自身的表？

情景：

把同时存在于取经四人组和悟空兄弟表中的记录的人在取经四人组表中的over字段更新为“齐天大圣”。

考虑使用这样的更新SQL：

UPDATE user1

SET over=”齐天大圣”

WHERE user1.`user\_name` IN(

SELECT b.`user\_name`

FROM user1 a INNER JOIN user2 b ON

a.`user\_name`=b.`user\_name`)

;

该语句在oracle和SQL Serve中是支持的，但是在MySQL中客户端报错：ERROR 1093（HY000）：you can`t sepecify target table in from clause.

既然在from子句中不允许这种操作，那么可以这样修改SQL：

UPDATE user1 a JOIN(

SELECT b.`user\_name`

FROM user1 a INNER JOIN user2 b ON

a.`user\_name`=b.`user\_name`

)b ON a.`user\_name`=b.`user\_name`

SET a.`over` = “齐天大圣”

;

**总结：**IN子查询的方式可以通过JOIN转换为带有ON字段的查询（这一就可以使用索引了）。

## join优化子查询

SELECT a.`user\_name`

,a.`over`

,(SELECT over FROM user2 b

WHERE a.`user\_name`=b.`user\_name`) AS

OVER2

FROM user1 a

;

执行时间 ：



使用join优化子查询：

SELECT a.`user\_name`,a.`over`,b.`over` AS OVER2

FROM user1 a

LEFT JOIN user2 b ON

a.`user\_name`=b.`user\_name`

;

执行时间：



**总结：**在查询列表中的子查询可以使用多表JOIN方式实现。

## join优化聚合子查询



问题：如何查询出四人组中打怪最多的日期？

SELECT a.`user\_name`,b.`timestr`,b.`kills`

FROM user1 a JOIN user\_kills b

ON a.id = b.user\_id

WHERE b.kills = (SELECT MAX(c.kills) FROM user\_kills c WHERE

c.user\_id = b.user\_id)

;

使用join优化：

SELECT a.`user\_name`,b.`timestr`,b.`kills`

FROM user1 a

JOIN user\_kills b ON a.id = b.user\_id

GROUP BY a.user\_name,b.timestr,b.kills

HAVING b.kills = MAX(c.kills)

;

总结：=聚合函数的形式可以转换为GROUP BY+HAVING。

## 小表驱动大表（MULTI\_STEP）

注：在分布式数据库GoldenDB中采用添加hint信息MULTI\_STEP(t1)的形式来进行优化（括号内填写小表）。

# 分组选择数据

场景：

SELECT a.user\_name,b.timestr,b.kills

FROM user1 a JOIN user\_kills b

ON a.id = b.user\_id

WHERE user\_name = “孙悟空”

ORDER BY b.`kills` DESC

LIMIT 2

；

我们可以对每个人分别执行上面的查询。

问题：

1. 如果分类或是用户很多的情况下则需要多次执行同一查询
2. 增加应用程序同数据库的交互次数
3. 增加了数据库执行查询的次数，不符合批处理的原则
4. 增加了网络流量

我们的场景：

SELECT d.user\_name,c.timestr,kills

FROM (

SELECT user\_id,timestr,kills

,(SELECT COUNT(\*) FROM user\_kills b WHERE

b.user\_id=a.user\_id AND a.kills <= b.kills) AS cnt

FROM user\_kills a

GROUP BY user\_id,timestr,kills

) c JOIN user1 d ON c.user\_id=d.id

WHERE cnt<=2

;

# 使用规范

数据规模较小，全部放进内存。数据规模较大，可以通过增加索引来优化join语句的执行速度，可以通过冗余信息来减少join的次数，尽量减少表连接的次数，一个SQL语句表连接的次数不要超过5次。

这个规则超过三张表禁止join ,由于数据量太大的时候，mysql根本查询不出来，导致阿里出了这样一个规定。(其实如果表数据量少，10张表也不成问题)而我们公司支付系统朝着大规模高并发目标设计的，所以，遵循这个规定。

在业务层面来讲，写简单sql，把更多逻辑放到应用层，我的需求我会更了解，在应用层实现特定的join也容易得多。

参考：

<https://mp.weixin.qq.com/s/vZDw-WdErzYkc1I1Z0Q-sg>

# 分布式数据库JOIN

## PolarDB

PolarDB-X的架构与MySQL有很大的不同，它的架构可以分为SQL层和存储层，SQL层的计算节点需要计算数据所在的分片，然后从多个DN节点（数据节点）拉取所需的数据。

对于Join查询，如果恰好Join Key和拆分键一致，那么可以将其下推到DN执行。否则，就需要在CN节点执行Join。

注：PolarDB-X的这种分布式Join实现算法与GoldenDB基本一致。

PolarDB-X支持多种Join算法，包括Lookup Join、Nested-Loop Join、Hash Join、Sort-Merge Join等多种执行方式。在OLTP查询中最常用的就是类似MySQL BKA Join的Lookup Join。

除了Join本身的功能需求，PolarDB-X的Lookup Join的设计中还要考虑以下两个性能需求：

1. **批量**。在分布式数据库中，CN到DN的每一次查询都会经过网络RPC，其延迟相比MySQL的本地调用要大几个数量级，因此批量处理显得更为重要。
2. **并发**。由于数据可能分布在多个DN节点上，如果依次遍历则会引入大量不必要的等待，最好的做法是并发地对所有DN进行查询，这样每批数据仅需一次网络round-trip。

### Lookup Join

**Lookup Join的执行过程如下（非索引回表情形）：**

从驱动侧拉取一批数据。通常情况下数据量不会很多，如果数据较多，那么每个批的大小受到lookup端的分片数量以及是否可以进行分片裁剪限制。批大小的选择会直接影响查询性能，如果批特别小会导致RPC次数太高，批太大则会导致内存中暂存的数据量膨胀，高并发情况下可能导致OOM。默认情况下我们尽可能让每个分片平均查询50个值、最多不超过300个值。

计算batch内每行数据所在分片，由于lookup侧是一个分区表，驱动表的每行数据要lookup的数据位于不同的分区中。只有包含数据的分片才需要参与 Join，如果没有任何值被路由到某个分片上，那么这个分片也无需被Lookup。

并发请求所有需要lookup的分片，并将查到的数据行以Join Key为Key构建成哈希表，缓存在内存中。

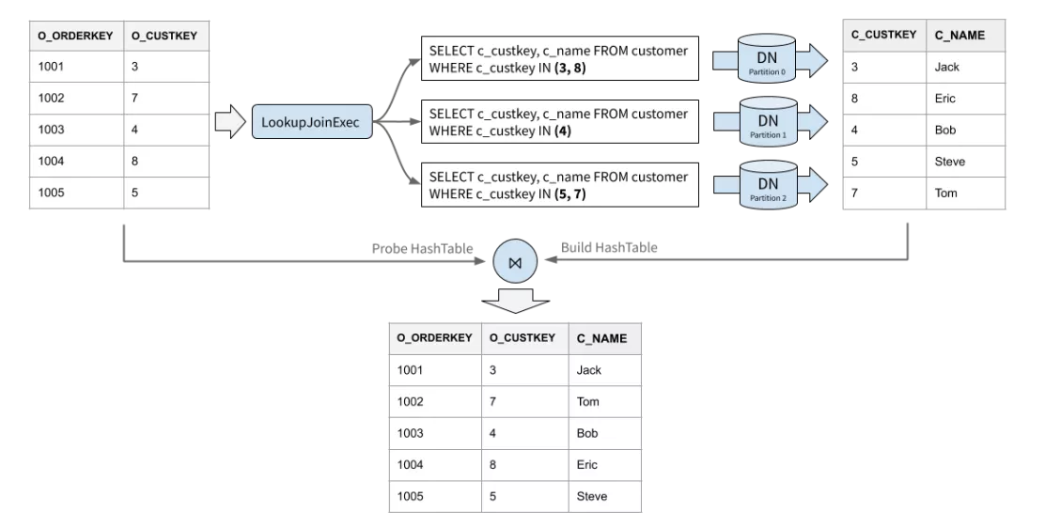
类似于Hash Join，利用哈希表为驱动侧的每行找到与其Join的行，取决于Join类型，可能Join出0行、1行或多行。

/\* Query 1 \*/

SELECT o\_orderkey, o\_custkey, c\_name

FROM orders JOIN customer ON o\_cutkey = c\_custkey

WHERE o\_orderkey BETWEEN 1001 AND 1005



这个过程中有一些有趣的细节，例如，当要lookup的列不止一列（例如 X = A AND Y = B）时如何处理？这时可以通过row-expression组成多列的IN条件。如果多列IN条件中出现NULL如何处理？对于Anti-Join如何处理？这些就不在这里展开了，有兴趣的同学可以在评论交流。

对于绝大多数TP查询，Lookup Join都可以通过一次lookup完成Join，将延迟降到了最低。

### 全局索引与Lookup Join

PolarDB-X还支持全局索引，用户可以为分区表创建全局索引表，加快对索引键的查询。和本地索引一样，如果查询中包含索引未覆盖的列，全局索引也需要进行回表。回表的做法和Lookup Join过程是完全一致的，不难理解——索引回表可以看作一种特殊的1:1的Join。

依赖全局索引的Join则更为复杂一些，回忆下MySQL的BKA Join，需要进行两次lookup：

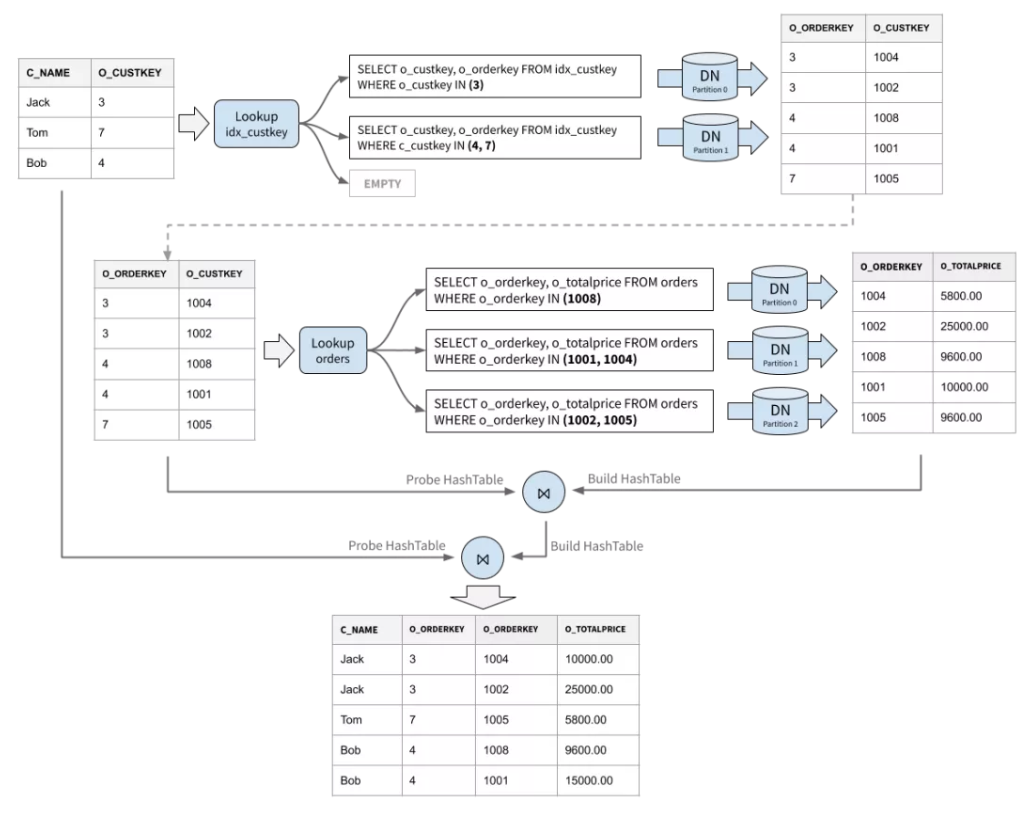
1. 第一次用Join key查询全局索引表（用于Join）
2. 第二次用全局索引表中的主键查询主表（用于索引回表）
3. 将回表结果以PK为key构建哈希表，与2中的查询结果Join，得到完整的 Join 右侧数据
4. 将完整的Join右侧数据以Join Key为key构建哈希表，与1的数据Join，得到最终Join结果

/\* Query 2 \*/

SELECT c\_name, c\_custkey, o\_orderkey, o\_totalprice

FROM customer JOIN orders ON c\_cutkey = o\_custkey

WHERE c\_custkey BETWEEN 13 AND 15



绝大多数OLTP查询数据量都能通过单个batch完成，完成查询的总延迟为3次round-trip。不难证明在分布式情况下这是最优的。

此外，PolarDB-X也允许用户手动将更多的列加入全局索引的覆盖列，牺牲部分写入性能换取更好的读取性能，如果所有列均被覆盖则无需进行回表，只需两轮round-trip即可。更进一步，PolarDB-X鼓励用户通过合理设计拆分键尽可能将Join下推。

## TDSQL

业务逻辑中，经常需要查询两个或多个表中的列之间的关系（JOIN），这在关系型实例（单机架构）上的简单操作，在分布式实例中就比较复杂。

由于需要JOIN的数据可能分别存储在多个物理节点中，导致JOIN过程需要大量网络交互，这导致某些分布式数据库处理JOIN请求时，无法提供数据一致性和性能兼得的方案。TDSQL业内少数几个支持分布式JOIN，且在大规模业务中验证过的产品。一般来说，分布式JOIN分为可下推和不可下推：

### 可下推Join

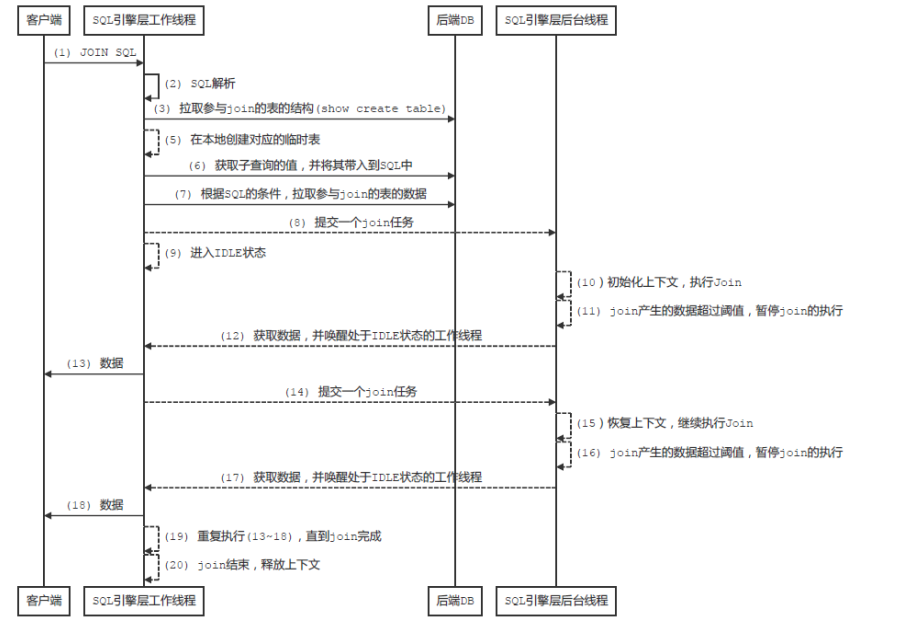
可下推JOIN，是指可在存储层直接JOIN的情况，通常包括：  
 ⚫ 同纬度（拆分建）的JOIN：两张表采用相同的拆分键。

例如：

SELECT \* FROM user JOIN user\_order ON user.user\_id=user\_order.user\_id;由于user与user\_order均已user\_id为拆分键，因此同一用户（user\_id）的记录位于同一分片上，JOIN直接由底层出错了完成。此时性能最高。  
 ⚫ 分表与广播表的JOIN：由于所有分片中都存在一个完整的广播表副本，因此分表与广播表的JOIN也可下推到存储层执行。

### 不可下推Join

不可下推的JOIN，是指需要由存储层和SQL Engine共同完成的JOIN，通常包括：单表与分表的JOIN，分表与分表且不同字段的JOIN等。腾讯云优化不可下推的分布式JOIN，并采用如下的过程执行（如下图）。



另外，分布式实例也支持子查询、函数等复杂语句。

## OceanBase

## GoldenDB

### 可下推Join

通过ON关联字段，尽量将关联的语句下发到一个分片（即下推），减少在proxy的计算（即减少节点之间的流量）。

### 不可下推Join

通过设置关联字段是否为索引，以及是否存在排序，确定是使用Nested Loop Join还是Sort-Merge Join或Hash Join。

### 全局索引

支持全局索引功能，防止群发分片，准确定位分片，减少不必要的网络开销。