join

join策略

Nested-Loop Join

> 嵌套循环连接算法，双层for循环，通过驱动表（外层循环）的行数据，逐个与内层表的所有行数据进行比较来获取结果

比如这条join的sql

```
select * from A left join B on A.id=B.id;
```

转为伪代码就为

```
for(Row row:A){
    for(Row row:B){
        if(A.id==B.id){
            return *;
        }
    }
}
```

- 次数=外层行数*内层行数

Index Nested-LoopJoin

> 基于索引进行连接的算法，索引基于内层表，通过外层表的匹配条件直接与内标索引进行匹配，避免了与内层表的所有记录进行匹配，通过索引查询减少内表的匹配次数，提高性能（减少内层表的匹配次数）

- 前提：
  - 内层表join的列要有索引
  - 如果是辅助索引，且返回数据要有内层表其他数据的，则要做回表查询，增加IO操作

```
for(Row row:A){
    for(Index index:B){
        if(A.id==index.id){
            if(contains(otherData)){
                回表查询();
            }
        }
    }
}
```

- 次数= 外层行数*内层表索引高度

Block Nested-Loop Join

> 缓存块嵌套循环连接，通过一次性缓存多条数据，将参与查询的列缓存到join buffer中，然后拿join buffer里的数据批量与内层表数据进行匹配，减少内层循环的次数（循环一次内层表，就可以批量匹配一次Join Buffer里的外层数据） 减少内层表数据的循环次数

- 前提：
  - 当不使用index Nested-Loop Join，默认使用Block Nested-Loop Join
- JoinBuffer
  - 缓存所有参与查询的列
  - 可调整缓存大小

```
for each row in t1 matching range {
   for each row in t2 matching reference key {

    // 存储join Buffer信息
    store used columns from t1, t2 in join buffer
    // 如果满了，则开始做匹配
    if buffer is full {
      // 对t3表做join
      for each row in t3 {
         // 对join Buffer做join
         for each t1, t2 combination in join buffer {
          // 如果满足，则返回行数据
          if row satisfies join conditions,
          send to client
        }
       }
      empty buffer
    }
  }
}

// 如果还不为空
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
     }
  }
}
```

提高join速度

- 小表驱动大表，若两个表均为索引列，mysql优化时（对于内连接场景）会优化为通过小表驱动大表，∵索引查询成本固定，通过缩小外表循环次数，继而提高join的速度，因为对于Index Nested Loop-Join来说，次数=外表循环次数*内表索引树高度
- 为匹配条件增加索引：争取使用INLJ，减少内表的循环次数
- 增大join buffer size，减少外层循环次数
- 减少不必要的字段查询
    - 减少join Buffer缓存数据，外层循环次数减少

left join(Outer Join)

> 外连接：当遇到了内关联表不存在的记录时，以null进行填充，其他逻辑差不多

```
select
    *
from
    temp L
left join
    temp2 R
on
    L.id=R.id;
where
    R.id is null
```

```
for(Row row1:L){
    find=false;
    for(Row row2:R){
        // 如果有匹配的被驱动表记录，则合并输出
        if(find(row2)){
            combine(row1,row2);
            sendToClient();
            find=true;
        }
    }
    if(!find){
        // 补上null后输出
        combine(row1,null);
        sendToClient();
    }
}

// 提出R为null的行记录
for(Row row:temp){
    if(row.R_id is null){
        filter(row);
    }
}
```

on和where

对于外连接而言，on是对被驱动表做限制，而where是对连接后产生的结果集进行筛选，所以判断条件放到on和where后面是完全不同的结果。

- on
  - 生成临时表的使用的条件，不管on条件结果怎么样，都会返回左表的记录，对于右表不存在的，则以null进行填充
- where
  - 对上面的临时表进行筛选，这个时候就与left join没什么关系，就只是正常的where筛选了。

> 放在不同的地方就会有不同的结果，比如说 `on R.id between 10 and 100` 那么就过滤了10和100的id，右表的记录最多就为90条，但如果这个是放在了where语句中，就变成了只要右表10和100的数据，意义是不同的

关于in的子查询

> Mysql5.6中对于in子查询优化为semi-join策略，识别需要子查询语句，消除来自内表的重复项，其中有四种策略用于去重

简单来说，就是避免O（n*m)的情况，通过最大化减少内表、被驱动表的数量，减少循环次数，内表只需要提取足以对外表outer_tables记录进行筛选的信息即可

DuplicatWeedout

> 使用临时表对semi-join产生结果集进行去重，临时表存储in的子查询信息，去重处理，然后将结果集视为一个连接，与外表join查出相关结果集 -----**缩小了内表达标**

First-Match

> 只选用内表的第一条与外表匹配的记录，只有当无法匹配时，才会回到内表进行全局匹配

LooseScan

> 把内表基于索引排序，取每组的第一条数据进行匹配

MeterializeLookup

> 内表去重变为物化临时表，遍历外表，在固化表上寻找匹配

理解为外表join物化临时表

MeterializeScan

> 内表去重变为临时表，遍历固化表，在外表上寻找匹配

理解为物化临时表驱动内表