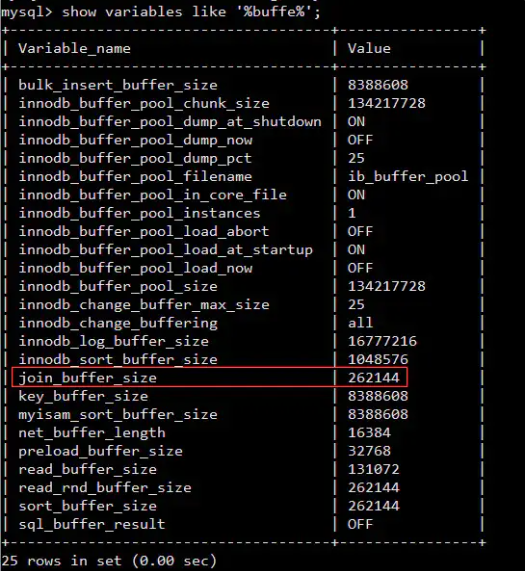
# 原理

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, 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。我们有多少表要连接就需要读多少个文件，虽然可以利用索引，但还是免不了频繁的移动硬盘的磁头。也就是说频繁的移动磁头会影响性能。

# 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

### 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`

;

### 交叉连接/cross join

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。

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。

### 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。

## 外连接

### 左外连接/outer join

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。

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**

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

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



**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**

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

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。

# 原理

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, 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

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。

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。

## Nested Loop Join

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。

### Index Nested-Loop Join

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。

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。

### Block nested loop

INLJ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，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**

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**。

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。

**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%';

### 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”

;

## 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：



## 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)

;

# 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>