### [高效sql性能优化极简教程](http://yangactive.iteye.com/blog/2407211)

**博客分类：**

* [互联网](http://yangactive.iteye.com/category/220093)

# 一，sql性能优化基础方法论

对于功能，我们可能知道必须改进什么；但对于性能问题，有时我们可能无从下手。其实，任何计算机应用系统最终队可以归结为：

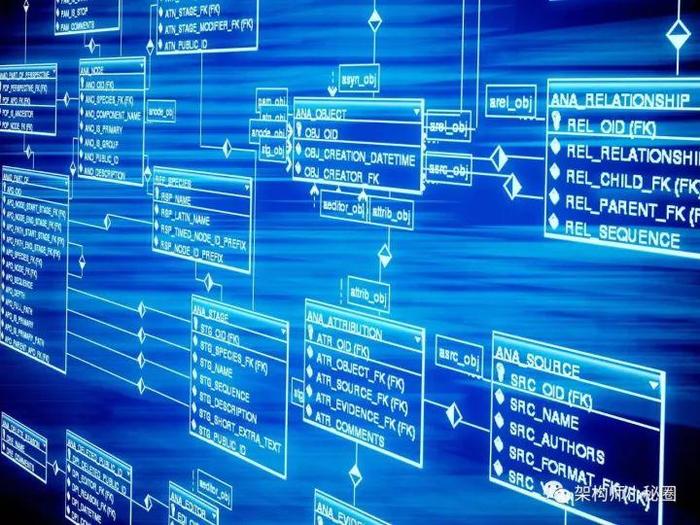
cpu消耗

内存使用

对磁盘，网络或其他I/O设备的输入/输出(I/O)操作。

阅读本文，建议大家已经掌握了扎实的互联网架构技术，可参考：[互联网架构技术清单](https://link.jianshu.com/?t=http://www.3xmq.com/domain/wenzhang)

但我们遇到性能问题时，要判断的第一点就是“在这三种资源中，是否有哪一种资源达到了有问题的程度”，因为这一点能指导我们搞清楚“需要优化重构什么”和“如何优化重构它”



# 二，sql调优领域

**应用程序级调优**

sql语句调优

管理变化调优

**示例级调优**

内存

数据结构

实例配置

**操作系统交互**

I/O

swap

Parameters

# 三，sql优化方法

优化业务数据

优化数据设计

优化流程设计

优化sql语句

优化物理结构

优化内存分配

优化I/O

优化内存竞争

优化操作系统

# 四，sql优化过程

定位有问题的语句

检查执行计划

检查执行计划中优化器的统计信息

分析相关表的记录数、索引情况

改写sql语句、使用HINT、调整索引、表分析

有些sql语句不具备优化的可能，需要优化处理方式

达到最佳执行计划

# 五，什么是好的sql语句

尽量简单，模块化

易读，易维护

节省资源

内存

cpu

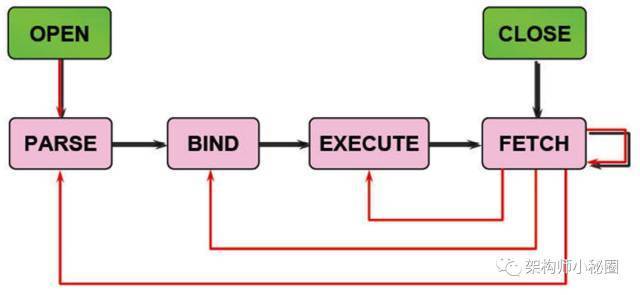
扫描的数据块要少

少排序

不造成死锁

# 六，sql语句的处理过程

sql语句的四个处理阶段：



**解析（PARSE）:**

检查语法

检查语义和相关的权限

在共享池中查找sql语句

合并（MERGE）视图定义和子查询

确定执行计划

**绑定（BIND）**

在语句中查找绑定变量

赋值（或重新赋值）

**执行（EXECUTE）**

应用执行计划

执行必要的I/O和排序操作

**提取（FETCH）**

从查询结果中返回记录

必要时进行排序

使用ARRAY FETCH机制

# 七，sql表的基本连接方式

**表连接有几种?**

sql表连接分成**外连接**、**内连接**和**交叉连接。**

新建两张表：

表1：student  截图如下：



表2：course  截图如下：



（此时这样建表只是为了演示连接SQL语句，当然实际开发中我们不会这样建表，实际开发中这两个表会有自己不同的主键。）

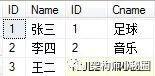
**一、外连接**

外连接可分为：左连接、右连接、完全外连接。

1、左连接  left join 或 left outer join

SQL语句：select \* from student left join course on student.ID=course.ID

执行结果：



左外连接包含left join左表所有行，如果左表中某行在右表没有匹配，则结果中对应行右表的部分全部为空(NULL).

注：此时我们不能说结果的行数等于左表数据的行数。当然此处查询结果的行数等于左表数据的行数，因为左右两表此时为一对一关系。

2、右连接  right join 或 right outer join

SQL语句：select \* from student right join course on student.ID=course.ID

执行结果：



右外连接包含right join右表所有行，如果左表中某行在右表没有匹配，则结果中对应左表的部分全部为空(NULL)。

注：同样此时我们不能说结果的行数等于右表的行数。当然此处查询结果的行数等于左表数据的行数，因为左右两表此时为一对一关系。

3、完全外连接  full join 或 full outer join

SQL语句：select \* from student full join course on student.ID=course.ID

执行结果：



完全外连接包含full join左右两表中所有的行，如果右表中某行在左表中没有匹配，则结果中对应行右表的部分全部为空(NULL)，如果左表中某行在右表中没有匹配，则结果中对应行左表的部分全部为空(NULL)。

**二、内连接  join 或 inner join**

SQL语句：select \* from student inner join course on student.ID=course.ID

执行结果：

IMG_263

inner join 是比较运算符，只返回符合条件的行。

此时相当于：select \* from student,course where student.ID=course.ID

**三、交叉连接 cross join**

1.概念：没有 WHERE 子句的交叉联接将产生连接所涉及的表的笛卡尔积。第一个表的行数乘以第二个表的行数等于笛卡尔积结果集的大小。

SQL语句：select \* from student cross join course

执行结果：



如果我们在此时给这条SQL加上WHERE子句的时候比如SQL:select \* from student cross join course where student.ID=course.ID

此时将返回符合条件的结果集，结果和inner join所示执行结果一样。

# 八，sql优化最佳实践

1，选择最有效率的表连接顺序

首先要明白一点就是SQL 的语法顺序和执行顺序是不一致的

SQL的语法顺序：

    select   【distinct】 ....from ....【xxx  join】【on】....where....group by ....having....【union】....order by......

SQL的执行顺序：

   from ....【xxx  join】【on】....where....group by ....avg()、sum()....having....select   【distinct】....order by......

from 子句--执行顺序为从后往前、从右到左

表名(最后面的那个表名为驱动表，执行顺序为从后往前, 所以数据量较少的表尽量放后）

where子句--执行顺序为自下而上、从右到左

将可以过滤掉大量数据的条件写在where的子句的末尾性能最优

group by 和order by 子句执行顺序都为从左到右

select子句--少用\*号，尽量取字段名称。 使用列名意味着将减少消耗时间。

2，避免产生笛卡尔积

含有多表的sql语句，必须指明各表的连接条件，以避免产生笛卡尔积。N个表连接需要N-1个连接条件。

3，避免使用\*

当你想在select子句中列出所有的列时，使用动态sql列引用“\*”是一个方便的方法，不幸的是，是一种非常低效的方法。sql解析过程中，还需要把“\*”依次转换为所有的列名，这个工作需要查询数据字典完成！

4，用where子句替换having子句

where子句搜索条件在进行分组操作之前应用；而having自己条件在进行分组操作之后应用。避免使用having子句，having子句只会在检索出所有纪录之后才对结果集进行过滤，这个处理需要排序，总计等操作。如果能通过where子句限制记录的数目，那就能减少这方面的开销。

5，用exists、not exists和in、not in相互替代

原则是哪个的子查询产生的结果集小，就选哪个

select \* from t1 where x in (select y from t2)

select \* from t1 where exists (select null from t2 where y =x)

IN适合于外表大而内表小的情况；exists适合于外表小而内表大的情况

6，使用exists替代distinct

当提交一个包含一对多表信息（比如部门表和雇员表）的查询时，避免在select子句中使用distinct，一般可以考虑使用exists代替，exists使查询更为迅速，因为子查询的条件一旦满足，立马返回结果。

低效写法：

select distinct dept\_no,dept\_name from dept d,emp e where d.dept\_no=e.dept\_no

高效写法：

select dept\_no,dept\_name from dept d where  exists (select 'x' from emp e where e.dept\_no=d.dept\_no)

备注：其中x的意思是：因为exists只是看子查询是否有结果返回，而不关心返回的什么内容，因此建议写一个常量，性能较高！

用exists的确可以替代distinct，不过以上方案仅适用dept\_no为唯一主键的情况，如果要去掉重复记录，需要参照以下写法：

select \* from emp  where dept\_no exists (select Max(dept\_no)) from dept d, emp e where e.dept\_no=d.dept\_no group by d.dept\_no)

7，避免隐式数据类型转换

隐式数据类型转换不能适用索引，导致全表扫描！t\_tablename表的phonenumber字段为varchar类型

以下代码不符合规范：

select column1 into i\_l\_variable1 from t\_tablename where phonenumber=18519722169;

应编写如下：

select column1 into i\_lvariable1 from t\_tablename where phonenumber='18519722169';

8，使用索引来避免排序操作

在执行频度高，又含有排序操作的sql语句，建议适用索引来避免排序。排序是一种昂贵的操作，在一秒钟执行成千上万次的sql语句中，如果带有排序操作，往往会消耗大量的系统资源，性能低下。索引是一种有序结果，如果order by后面的字段上建有索引，将会大大提升效率！

9，尽量使用前端匹配的模糊查询

例如，column1 like 'ABC%'方式，可以对column1字段进行索引范围扫描；而column1 kike '%ABC%'方式，即使column1字段上存在索引，也无法使用该索引，只能走全表扫描。

10，不要在选择性较低的字段建立索引

在选择性较低的字段使用索引，不但不会降低逻辑I/O，相反，往往会增加大量逻辑I/O降低性能。比如，性别列，男和女！

11，避免对列的操作

不要在where条件中对字段进行数学表达式运算，任何对列的操作都可能导致全表扫描，这里所谓的操作，包括数据库函数，计算表达式等等，查询时要尽可能将操作移到等式的右边，甚至去掉函数。

例如：下列sql条件语句中的列都建有恰当的索引，但几十万条数据下已经执行非常慢了：

select \* from record where amount/30<1000 (执行时间11s)

由于where子句中对列的任何操作结果都是在sql运行时逐行计算得到，因此它不得不进行全表扫描，而没有使用上面的索引；如果这些结果在查询编译时就能得到，那么就可以被sql优化器优化，使用索引，避免全表扫描，因此sql重写如下：

select \* from record where amount<1000\*30 (执行时间不到1秒)

12，尽量去掉"IN","OR"

含有"IN"、"OR"的where子句常会使用工作表，使索引失效，如果不产生大量重复值，可以考虑把子句拆开；拆开的子句中应该包含索引；

select count(\*) from stuff where id\_no in('0','1')

可以拆开为：

select count(\*) from stuff where id\_no='0'

select count(\*) from stuff where id\_no='1'

然后在做一个简单的加法

13,尽量去掉"<>"

尽量去掉"<>"，避免全表扫描，如果数据是枚举值，且取值范围固定，可以使用"or"方式

update serviceinfo set state=0 where state<>0;

以上语句由于其中包含了"<>"，执行计划中用了全表扫描（Table access full），没有用到state字段上的索引，实际应用中，由于业务逻辑的限制，字段state智能是枚举值，例如0,1或2，因此可以去掉"<>" 利用索引来提高效率。

update serviceinfo set state=0 where state =1 or state =2

14,避免在索引列上使用IS NULL或者NOT

避免在索引中使用任何可以为空的列，导致无法使用索引

15，批量提交sql

如果你需要在一个在线的网站上去执行一个大的DELETE或INSERT查询，你需要非常小心，要避免你的操作让你的整个网站停止相应。因为这两个操作是会锁表的，表一锁住了，别的操作都进不来了。

Apache会有很多的子进程或线程。所以，其工作起来相当有效率，而我们的服务器也不希望有太多的子进程，线程和数据库链接，这是极大的占服务器资源的事情，尤其是内存。

如果你把你的表锁上一段时间，比如30秒钟，那么对于一个有很高访问量的站点来说，这30秒所积累的访问进程或线程，数据库链接，打开的文件数，可能不仅仅会让你的WEB服务崩溃，还可能会让你的整台服务器马上挂了。所以，如果你有一个大的处理，你一定把其拆分。