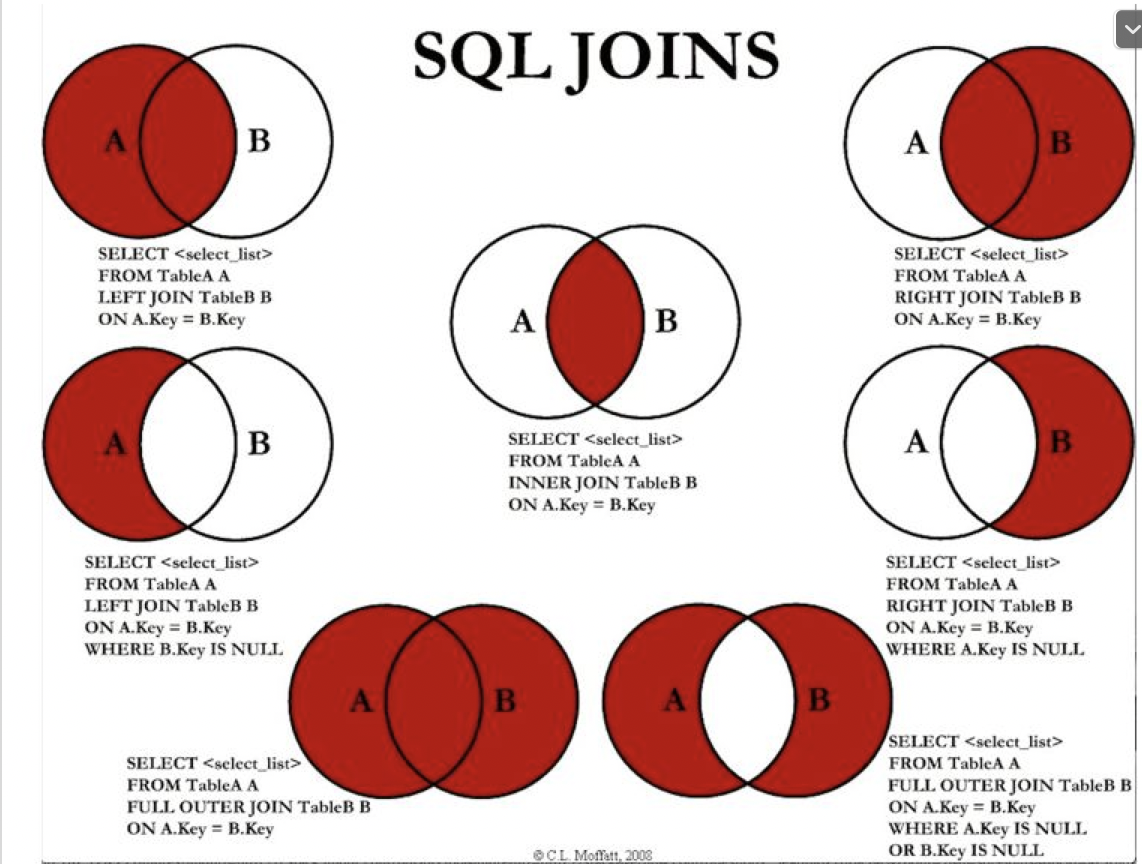
*本页文档资料来源:* [*https://blog.csdn.net/qq\_42449963/article/details/109067181*](https://blog.csdn.net/qq_42449963/article/details/109067181)

*todo SQL性能下降原因:*1. 查询语句写的太差;  
2. 索引失效, 建立了索引, 但是查询语句中并没有使用到;  
3. 关联查询太多(JOIN), 设计缺陷或不得已的需求;  
4. 服务器调优以及各个参数的配置(缓冲/线程数)等没有配置好;



索引的定义:

1. INDEX用于帮助Mysql高效获取数据, 其本质是排序好的快速查找数据结构;
2. 主要帮助mysql查找和排序, 是除数据本身外维护的另外一个满足特定查找算法的数据结构;
3. 它本身一很大, 不可能全部存储在数据库中, 往往以索引文件的形式存储在磁盘上;

索引的优势:

1. 查找: 类似图书馆的书目索引, 提高数据库的检索效率, 降低数据库的IO成本;
2. 排序: 通过索引对数据进行排序, 降低排序的成本, 减少对CPU的消耗;

索引的劣势:

1. 索引实际也是数据库维护的一张表, 保存了主键和索引记录, 所以它实际也要占用磁盘空间;
2. 索引虽然提高了查询速度,但是也降低了表的更新速度, 如频繁的对表进行insert/update/delete, MySQL不光要更新表数据, 还要更新与之对应的索引数据;
3. 索引只是提高效率的一个因素, 但数据库如果有大量的表, 就需要花时间研究建立最佳的索引;

索引分类:

1. 单值索引: 一个索引只包含一个列, 一个表可以建立多个单值索引;
2. 唯一索引: 索引列的值必须是唯一, 但是允许空值;
3. 复合索引: 一个索引可以包含多个列;
4. 建议一张表不要超过5个索引;

哪些情况要建立索引:

1. 主键自动建立索引(唯一 + 非空);
2. 频繁作为查询条件的字段应该创建索引;
3. 查询中与其他表关联的字段, 外键关系建立索引;
4. 查询中排序的字段, 通过索引访问将加速排序速度;
5. 查询中统计或分组字段(group by);

哪些情况不建立索引:

1. 记录太少的表;
2. 经常增删改的表;
3. 频繁更新的字段;
4. Where用不到的字段;
5. 对于字段的值变化不大的字段, 比如枚举: 只有两个值的情况下, 各自占50%, 索引不会太大的提高查询效率;

性能分析: explain

explain字段:

1. id: 表的读取顺序
2. select\_type: 数据读取操作的类型;
3. possible\_keys: 哪些索引可以使用;
4. key: 哪些索引实际被使用;
5. ref: 表之间的引用;
6. rows: 每张表有多少行被优化器查询;

id:

1. 相同, 由上往下顺序执行;
2. 不同, 如果是子查询, id序号会递增, id值越大优先级越高, 越先被执行;

select\_type:

1. simple: 简单的select查询, 查询中不包含子查询或union
2. primary: 查询中如果包含任何子查询部分, 那么最外层则被标记为primary
3. subquery: 在select或者where语句中包含子查询
4. derived: 在from子句包含的子查询被标记为derived(衍生), msyql会递归执行这些子查询, 把结果放入临时表中
5. union: 如果第二个select出现union后, 则被标记为union, 若union包含在from子句查询中, 外层select将被标记为derived

type:

从最好到最差依次是system-const-eq\_ref-ref-range-index-all, 除了all没用到索引, 其他级别都用到索引了

1. system: 表只有一行记录(等于系统表), 它是const的一个特例;
2. const: 表示通过一次索引就找到了, const用于比较primary key或者unique索引. 因为只匹配一行数据, 所以很快. 如将主键置于where后面, msyql将该查询转为一个常量
3. eq\_ref: 唯一性索引扫描, 读取本表和关联表中的每行组成的一行, 查出来只有一行记录
4. ref: 非唯一性索引扫描, 返回本表与关联表某个值匹配的所有行, 查出来多条记录
5. range: 只检索给定范围的行, 一般就是在where语句后出现between <> in等的查询. 这种范围扫描比全表扫描要好
6. index: 全索引扫描, 和all的区别在于, index只遍历索引树. 也就是说index从索引读取, 而all是从磁盘读取
7. all: 全表扫描, 最差

possible\_keys: 显示表中可能用到的索引, 但不一定被实际用到

key: 查询实际用到的索引.

Key\_len: 表示索引中使用的字节数, 显示的是最大可能长度, 并非实际使用长度. 在不损失精度的情况下, 长度越短越好

ref: 显示哪些列或常量被用于查找索引列上的值, 比如where xx=’a’, 此时ref就是const

rows: 根据表统计信息及索引选用情况, 大致估算出找到所需记录的行数

extra:

1. using filesort: 说明会对数据使用一个外部的索引排序, 而不是按照表内的索引顺序进行读取.
2. Using temporary: 使用临时表保存中间结果, 常见于order by和 group by, 对系统的性能消耗很大
3. Using index: 表示相应的select操作使用了覆盖索引, 避免了访问数据行. 如果同时出现using where, 表示索引被用来执行索引键值查找, 如果没有出现, 则表明索引用来读取数据而非执行查找操作
4. Using where: 表明使用了where过滤
5. Using join buffer: 使用了连接缓存
6. Impossible where: where子句总是false, 不能用于获取任何元组
7. Select tables optimized away: 在没有groupby子句的情况下, 基于索引优化的max/min/count操作, 不必等到执行阶段再计算
8. Distinct: 在找到第一匹配的元组后即停止找同样值的工作

索引优化:

1. 联合查询的时候, 左连尽量添加索引到右表, 右连尽量添加索引到左表;
2. 尽可能减少join语句中的nestedloop(嵌套循环)的总次数, 永远小表驱动大表;
3. 保证join语句中被驱动表上的join条件字段被使用;
4. 当无法保证被驱动表的join条件被索引且内存充足的情况下, 不要吝惜join buffer的使用;

索引失效:

1. 最好是全值匹配
2. 最佳左前缀法则: 如果索引中用得是复合索引, 那么查询从索引最左列开始并且不跳过中间字段. 一旦中间字段被跳过, 那么后续的索引都将失效
3. 不在索引列上做任何操作(计算/函数), 会导致索引失效而转向全表扫描
4. 索引范围条件右边的字段会全部失效
5. 尽量使用覆盖索引(只访问索引列的查询), 减少select \*的使用
6. 尽量不要在使用不等于<> !=, 会导致索引失效而进行全盘扫描
7. Is null, is not null也无法使用索引
8. Like以通配符开头的%xxx, 索引失效而进行全盘扫描
9. 字符串不加单引号, 索引失效. 虽然有时不加的时候同样能查出结果, 但那是mysql自己发生的强制转换, 但查询出的效率会完全不一样. 比如name=’2000’和name=2000, 前者用到索引且type是ref, key\_len也短. 后者也使用到索引且type是index, ken\_len也长. 且用到了using where进行键值查找
10. 少用or, 用它来来连接时会索引失效

口诀：

* 带头大哥不能死。
* 中间兄弟不能断。
* 索引列上不计算。
* 范围之后全失效。
* 覆盖索引尽量用。
* 不等有时会失效。
* like百分加右边。
* 字符要加单引号。
* 一般SQL少用or。

查询优化:

Order by:

1. 尽量使用index方式来排序, 避免使用到file sort
2. 尽可能在索引列上完成排序, 遵照最佳左前缀原则;
3. 如果不在索引列上, file sort会采用两种算法: 双路/单路排序
   1. 双路排序: 读取行指针和orderby列, 对他们进行排序. 然后扫描已排序好的列表, 按照列表中的值重新从列表中读取对应的数据传输. (从磁盘上取排序字段, 然后在buffer上排序, 然后再从磁盘上其他字段) – 二次磁盘io操作
   2. 单路排序: 从磁盘查询需要排序的所有列, 按照orderby在buffer上进行排序, 然后扫描排序后的列表输出. 此操作虽然只有一次io操作, 但更占用内存空间. 如果取出的数据超过sort\_buffer容量, 造成每次只能去sort\_buffer大小的数据进行排序(创建tmp文件, 多路合并), 排完取同样大小的容量. 再排序… 多次io, 最终得不偿失

Group by:

1. 实则先排序后分组, 参考索引优化原则.
2. Where高于having, 能用where限定条件时就不要使用having

两者优化策略:

1. 增大sort\_buffer\_size参数的设置
2. 增大max\_length\_for\_sort\_data参数的设置
3. 提高order by的速度, 参考索引优化原则

慢查询日志:

Mysql提供一种日志记录, 可以自己通过slow\_query\_log来开启, 默认记录值是10s, 当执行的操作超过10s, 那么它回收集该记录, 然后可以通过explain来分析. 可以通过slow\_query\_log\_file来检查日志记录位置. 也可通过msyql提供的日志分析工具mysqldumpslow来分析, 如: msyqldumpslow -s -c -t 10 /var/lib/mysql/slow.log

Show Profile:

用于分析当前会话中语句执行的资源消耗情况,记录最近执行的语句的详细流程,默认情况下是关闭的.

可以通过set profiling = 1来开启. 当在执行一系列sql后, 通过show profiles, 就能看到最近执行的sql语句及query id. 通过show profile cpu, block io for query ‘query id’即可诊断sql

以下为show profile各个参数, 可以根据时间情况自行添加

Show Profile查询参数备注：

* ALL：显示所有的开销信息。
* BLOCK IO：显示块IO相关开销（通用）。
* CONTEXT SWITCHES：上下文切换相关开销。
* CPU：显示CPU相关开销信息（通用）。
* IPC：显示发送和接收相关开销信息。
* MEMORY：显示内存相关开销信息。
* PAGE FAULTS：显示页面错误相关开销信息。
* SOURCE：显示和Source\_function。
* SWAPS：显示交换次数相关开销的信息。

通过show profile xxx命令, 我们可以看到查询执行的详细流程, 以及各流程的耗时:

+-------------------------+--------+--------+----------+------------+-------------+

|Status |Duration|CPU\_user|CPU\_system|Block\_ops\_in|Block\_ops\_out|

+-------------------------+--------+--------+----------+------------+-------------+

|starting |0.000115|0.000107|0.000000 |0 |0 |

|checking permissions |0.000012|0.000010|0.000000 |0 |0 |

|Opening tables |0.000025|0.000025|0.000000 |0 |0 |

|init |0.000044|0.000044|0.000000 |0 |0 |

|System lock |0.000010|0.000009|0.000000 |0 |0 |

|optimizing |0.000006|0.000006|0.000000 |0 |0 |

|statistics |0.000031|0.000031|0.000000 |0 |0 |

|preparing |0.000016|0.000016|0.000000 |0 |0 |

|Creating tmp table |0.000048|0.000048|0.000000 |0 |0 |

|Sorting result |0.000006|0.000005|0.000000 |0 |0 |

|executing |0.000003|0.000003|0.000000 |0 |0 |

|Sending data |0.116506|0.115016|0.000000 |0 |0 |

|converting HEAP to ondisk|0.119648|0.119455|0.000000 |0 |0 |

|Sending data |1.984473|1.915144|0.000000 |0 |6272 |

|Creating sort index |2.371744|1.345900|0.000000 |0 |434680 |

|end |0.000023|0.000017|0.000000 |0 |0 |

|query end |0.000010|0.000009|0.000000 |0 |0 |

|removing tmp table |0.001171|0.001172|0.000000 |0 |0 |

|query end |0.000010|0.000008|0.000000 |0 |0 |

|closing tables |0.000008|0.000008|0.000000 |0 |0 |

|freeing items |0.000090|0.000089|0.000000 |0 |0 |

|logging slow query |0.000034|0.000034|0.000000 |0 |8 |

|cleaning up |0.000013|0.000012|0.000000 |0 |0 |

+-------------------------+--------+--------+----------+------------+-------------+

着重需要注意下面几个流程的执行时间:

* converting HEAP to MyISAM：查询结果太大，内存都不够用了，往磁盘上搬了。
* Creating tmp table：创建临时表（拷贝数据到临时表，用完再删除），非常耗费数据库性能。
* Copying to tmp table on disk：把内存中的临时表复制到磁盘，危险！！！
* locked：死锁。

表锁:

表锁偏向于myisam引擎, 开销小/加锁快/无死锁/锁定粒度大 发生锁冲突概率高, 并发度低;

加锁: lock tables ‘表名’write/read, ‘表名’write/read

解锁: unlock tables

读锁: 加了读锁的表, 当前会话能读此表, 不能修改; 其他会话能读此表, 但修改也会被堵塞(直到解锁或会话关闭)

写锁: 加了写锁的表, 当前会话能读能写此表, 不能读写其他表; 其他会话读写此表都会被堵塞(直到解锁或会话关闭)

* 対MyISAM表的读操作（加读锁），不会阻塞其他线程対同一表的读操作，但是会阻塞其他线程対同一表的写操作。只有当读锁释放之后，才会执行其他线程的写操作。
* 対MyISAM表的写操作（加写锁），会阻塞其他线程対同一表的读和写操作，只有当写锁释放之后，才会执行其他线程的读写操作。

表锁分析: show status like ‘table%’

# +---------------------+  
# |Variable\_name |  
# +---------------------+  
# |Table\_locks\_immediate| 产生表级锁定的次数, 每一次调用show status like 'table%'都会加1  
# |Table\_locks\_waited | 出现表级锁定而发生等待的次数, 此值越高, 说明严重存在表级锁争用的情况  
# +---------------------+  
# myisam引擎的读写锁调度是写优先, 所以它不适合用作主表的引擎. 一旦开启写锁, 大量的写操作会使得其他查询很难得到锁. 从而造成堵塞

行锁: 偏向innodb, 开销大, 加锁慢. 锁定粒度小, 并发度高, 发生冲突概率低. 打开行锁只需要关闭autocommit即可 set autocommit = 0

*当关闭autocommit之后, 在未手动commit之前, 本会话修改的内容, 本会话可以查询, 其他会话无法查询.*# *本会话commit之后, 其他会话在同样关闭autocommit之后且未手动commit之前, 也无法查询修改内容(手动的commit相当于获取最新的数据)*

*本会话在更改操作在未手动commit之前, 其他会话操作同一条数据会被阻塞. 两个会话操作不同的数据不会被阻塞*

*当where后面的varchar带入数字时, 导致索引失效, 会导致行锁变表锁. 其他会话对该表的写操作都被阻塞*

*间隙锁. 当使用范围进行写操作时, 不管在范围内的数据是否存在. 在操作提交之前, 其他会话对该范围内的数据进行写操作, 都会被阻塞*

*锁定一行. 其他会话无法对该行数剧修改*

行锁分析:

SHOW STATUS LIKE 'innodb\_row\_lock%';  
# +-----------------------------+  
# |Variable\_name |  
# +-----------------------------+  
# |Innodb\_row\_lock\_current\_waits| 获取当前正在等待锁的数量  
# |Innodb\_row\_lock\_time | 从系统启动到现在锁定总时长  
# |Innodb\_row\_lock\_time\_avg |  
# |Innodb\_row\_lock\_time\_max |  
# |Innodb\_row\_lock\_waits | 系统启动到现在总共等待的次数  
# +-----------------------------+