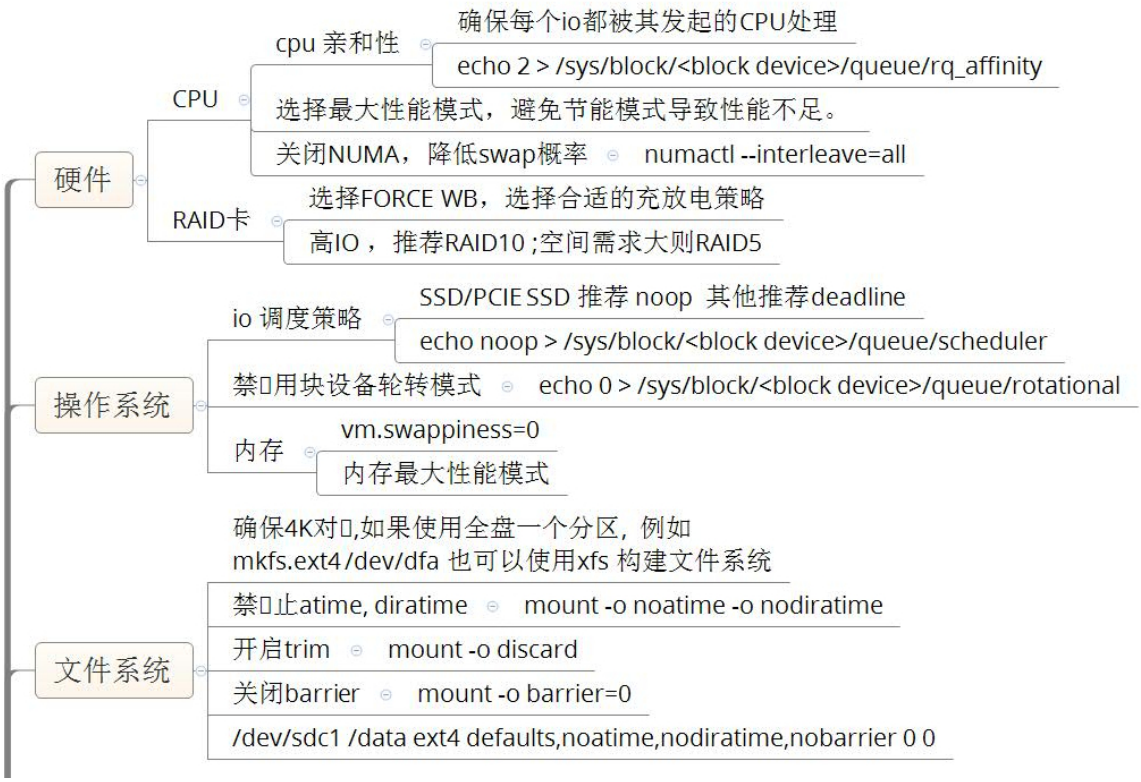
MySQL数据库优化

可以从几个方面进行数据库优化

1、硬件、操作系统、文件系统



- 操作系统：增加TCP支持的队列数
- 网络：带宽和传输协议
- 磁盘：seek、read、write

- CPU： 核心数多并且主频高的
- 内存：增大内存

2、配置优化

> - ==mysql配置文件优化==：Innodb缓存池设置(innodb_buffer_pool_size，推荐总内存的75%)和缓存池的个数（innodb_buffer_pool_instances）

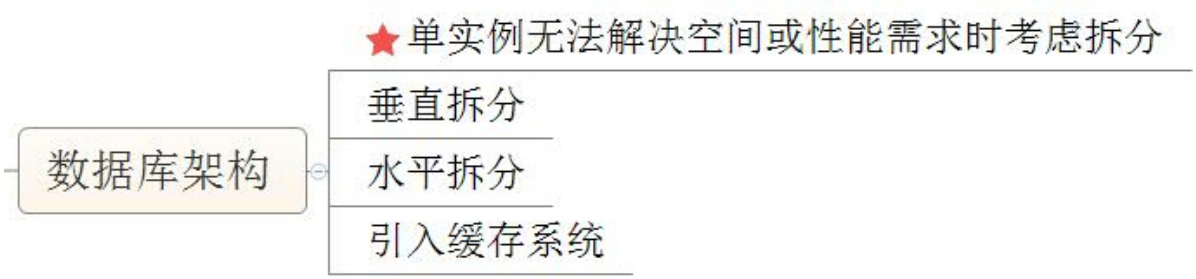
> - IO处理的常用参数
> - 最大连接数设置
> - 缓存使用参数的设置
> - 慢日志的参数的设置
> - innodb相关参数的设置
> - 存在主从关系：设置主从同步的相关参数

- **下面是MySQL的相关系统配置**

```
[mysqld]
skip-name-resolve,server-id = 1,bind-address = 0.0.0.0  port = 3306,datadir =
/home/mysql,tmpdir = /tmp,default_storage_engine = InnoDB
character_set_server = utf8,innodb_file_per_table = 1,innodb_log_file_size =
512M,innodb_log_files_in_group = 4,innodb_rollback_on_timeout =
1,slow_query_log = 1,slow_query_log_file =/var/log/mysql/mysql-
slow.log,long_query_time = 1
#log-queries-not-using-indexes#这个参数不安全，说是记录没有用到索引的语句，其实记录的全部的日志，占用大量的IO，建议不要打开
#relay_log_recovery=1#这个参数在丛库上一定要加上，query_cache_type =
off,query_cache_size = 0
```

```
#这两项是禁用缓存，这个使服务器用途而定：写比较多的数据库最好禁用，因为没写一次他要修改缓存中的数据，给数据库带来额外的开销，读比较的可以开启，可以提高查询效率 #一下4个参数是mysql5.6上的新特性
innodb_buffer_pool_dump_at_shutdown = 1 #解释：在关闭时把热数据dump到本地磁盘。
innodb_buffer_pool_dump_now = 1 #解释：采用手工方式把热数据dump到本地磁盘。
innodb_buffer_pool_load_at_startup = 1 #解释：在启动时把热数据加载到内存。
innodb_buffer_pool_load_now = 1 #解释：采用手工方式把热数据加载到内存。
read_buffer_size = 2M,sort_buffer_size = 2M,join_buffer_size =
1M,key_buffer_size = 2G ,thread_cache_size = 2048,open_files_limit=65535
innodb_open_files = 8192
max_allowed_packet = 64M
thread_stack = 512k,max_length_for_sort_data = 16k,tmp_table_size =
256M,max_heap_table_size = 256M,max_connections = 4000
,max_connect_errors = 30000,innodb_read_io_threads = 8,innodb_write_io_threads
= 16,innodb_flush_method = O_DIRECT
innodb_io_capacity =20000#根据硬盘的情况修改，stat的用100，sas的200，sas做riad10的为400fision-io的可以设置为20000
innodb_buffer_pool_size = 72G#内存的80%
innodb_buffer_pool_instances=18,thread_concurrency=0,innodb_thread_conc,rrency
= 0,innodb_log_buffer_size = 16M,innodb_flush_log_at_trx_commit =
2,innodb_lock_wait_timeout =
60,innodb_old_blocks_time=1000,innodb_use_native_aio =
1,innodb_purge_threads=1,innodb_change_buffering=inserts
```

3、数据库表结构



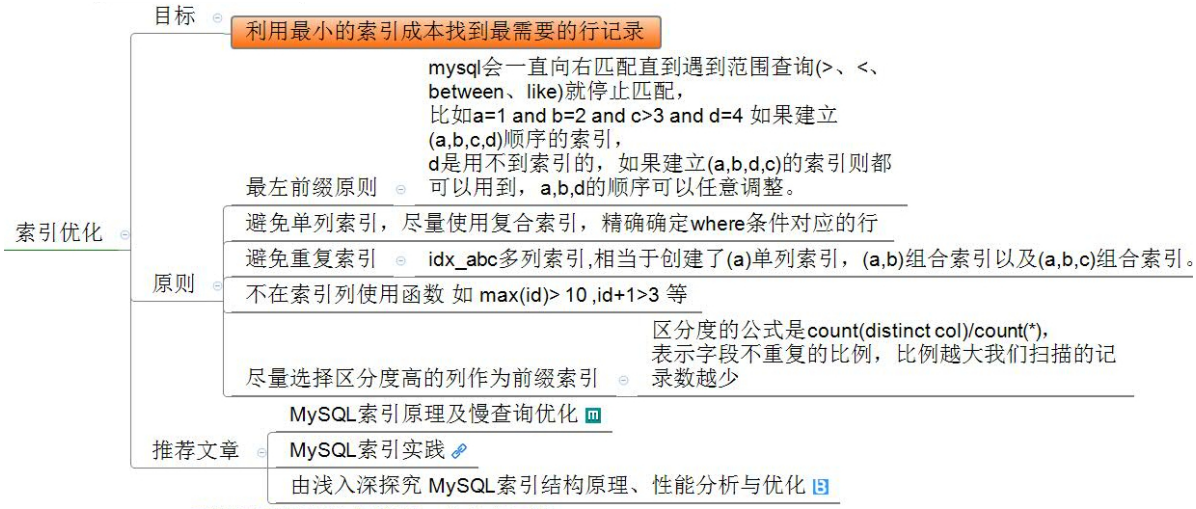
- 表数据压缩优化
- 范式.索引：一般情况下，表的设计应该++遵循三大范式++
- ==表结构设计优化之表的垂直拆分==：把含有多个列的表拆分成多个表，解决表宽度问题，好处，后业务清晰，拆分规则明确、系统之间整合或扩展容易、数据维护简单，下面是方法：
  - 把不常用的字段单独放在同一个表中；
  - 把大字段独立放入一个表中；
  - 把经常使用的字段放在一起；
- ==表结构设计优化之表的水平拆分==：表的水平拆分用于解决数据表中数据过大的问题，水平拆分每一个表的结构都是完全一致的。一般地，将数据平分到N张表中的常用方法包括以下两种：
  - 对ID进行hash运算，如果要拆分成5个表，mod(id,5)取出0~4个值；
  - 针对不同的hashID将数据存入不同的表中；

- ==表结构设计优化之水平拆分表的优缺点==：
    - 优点：
        - 表分割后可以降低在查询时++需要读的数据和索引的页数++，同时也降低了索引的层数，提高查询速度；
        - 表中的数据本来就有独立性，例如表中分别记录各个地区的数据或不同时期的数据，特别是有些数据常用，而另外一些数据不常用。
        - 需要把数据存放到多个数据库中，提高系统的总体可用性(分库，鸡蛋不能放在同一个篮子里)。
    - 缺点：
        - 跨分区表的数据查询
        - 统计及后台报表的操作等问题

- ==表结构设计优化之选择合适数据类型==
- 使用较小、较简单的数据类型解决问题
- 尽可能的使用not null 定义字段；
- 尽量避免使用text类型，非用不可时最好考虑分表；
- 表数据逻辑分布策略优化
    - 应用层优化
    - 连接池并发度优化
    - 数据压缩
    - 缓存相关的优化
- 库级优化
    - 主从结构
    - 使用长链接
    - 表数量
    - 查询缓存
- 存储引擎优化
    - InnoDB
        - 日志文件和日志缓存
        - 事务管理选择
        - 数据压缩
        - 单表统计数据优化
        - 单表容量优化
    - MyISAM

4、索引优化

==建议在经常作查询选择的字段、经常作表连接的字段以及经常出现在order by、group by、distinct 后面的字段中建立索引==



- 优化索引(重要必看)
- MySQL索引原理及慢查询优化
- [译] MYSQL索引最佳实践
- 下面的情况索引可能失效
  - 以“%(表示任意0个或多个字符)”开头的LIKE语句，模糊匹配；
  - OR语句前后没有同时使用索引；
  - 数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型）
  - 对于多列索引，必须满足最左匹配原则(eg,多列索引col1、col2和col3，则 索引生效的情形包括col1或col1，col2或col1，col2，col3)。
  - 在索引列上使用IS NULL 或IS NOT NULL操作。索引是不索引空值的，所以这样的操作不能使用索引，可以用其他的办法处理
  - 在索引字段上使用not，<>，!=，eg<> 操作符（不等于）：不等于操作符是永远不会用到索引的，因此对它的处理只会产生全表扫描。
  - 对索引字段进行计算操作
  - 在索引字段上使用函数

5、SQL语句优化方案

SQL语句的优化主要包括三个问题，==即如何发现有问题的SQL、如何分析SQL的执行计划、以及如何优化SQL==

SQL优化顺序

- **怎么发现有问题的SQL?**（通过MySQL慢查询日志对有效率问题的SQL进行监控)
  - 慢查询日志是MySQL的一种日志记录，==记录在MySQL中响应时间超过阀值的语句==，即运行时间超过long_query_time值的SQL，记录到慢查询日志中。
  - long_query_time的默认值为10s。
  - 查询出执行的++次数多占用时间长的SQ++L、++通过pt_query_disgest(一种mysql慢日志分析工具)分析Rows examine(MySQL执行器需要检查的行数)项++==去找出IO大的SQL以及发现未命中索引的SQL==，这些SQL，是我们优化的对象。
- **通过explain查询和分析SQL的执行计划**
  - explain 关键字可以知道MySQL是如何处理SQL语句的，++以此来分析查询语句、是表结构的性能瓶颈。++
  - ++通过explain命令可以得到表的读取顺序、数据读取操作的操作类型、哪些索引可以使用、哪些索引被实际使用、表之间的引用、每张表有多少行被优化器查询等问题。++
  - 扩展列extra出现**Using filesort**和**Using temporay**，则往往表示SQL需要优化了。

- **==SQL语句的优化==**
  - **explain**: [ɪk'splen] 说明；解释。
    - 尽量对数据库中的每一条SQL进行explain，收集他们的执行计划。
    - 大多都需要去发掘，++需要进行大量的explain操作收集执行计划，并判断是否需要进行优化++。
    - 优化SQL，需要做到心中有数，知道SQL的执行计划才能判断是否有优化余地，才能判断是否存在执行计划问题。
  - **少计算**:
    - Mysql的作用是用来存取数据的，不是做计算的。
    - 做计算的话可以用其他方法去实现，mysql做计算是很耗资源的。
  - **少排序**:

    - 排序会消耗较多 CPU 资源，所以减少排序可以在缓存命中率高、IO 能力足够的场景下会影响 SQL 的响应时间。
    - MySQL减少排序有多种办法：
        - 通过利用索引来排序的方式进行优化
        - 减少参与排序的记录条数
        - 非必要不对数据进行排序
- **少用or**：
    - 当 where 子句中存在多个条件以“或”并存的时候，MySQL 的优化器并没有很好的解决其执行计划优化问题，
    - 再加上MySQL 特有的 SQL 与 Storage 分层架构方式，造成了其性能比较低下，
    - ++使用 union all 或者是union(必要的时候)的方式来代替“or”会得到更好的效果。++
- **少用join**：
    - 对于复杂的多表 Join，第一是优化器受限，第二在Join这方面性能表现离Oracle还有一定距离。
    - MySQL的优势在于简单，但这在某些方面其实也是其劣势。
    - MySQL 优化器效率高，但是由于其统计信息的量有限，优化器工作过程出现偏差的可能性也就更多。
    - 但如果是简单的单表查询，这一差距就会极小甚至在有些场景下要优于这些数据库前辈。
- **尽量用join代替子查询**：
    - ++虽然 Join 性能并不佳，但和子查询相比有非常大的性能优势++。
    - ++MySQL的子查询执行计划一直存在较大的问题++，虽然这个问题已经存在多年，不过到目前为止已经发布的所有稳定版本中一直没有太大改善。
    - 官方也在很早就承认这一问题，并且承诺尽快解决，但是至少到目前为止我们还没有看到哪一个版本较好的解决了这一问题。
- **用 union all 代替 union**：
    - union需要将两个(或者多个)结果集合并后再进行++唯一性过滤操作++，这就会涉及到排序，增加大量的CPU运算，加大资源消耗及延迟。
    - 所以当我们可以确认不可能出现==重复结果集或者不在乎重复结果集==的时候，尽量使用 union all代替union。
- **尽量早过滤**：
    - 该优化策略最常见于索引的优化设计中(==将过滤性更好的字段放得更靠前==)。
    - 在 SQL 编写中使用这一原则来优化一些 Join 的 SQL。
    - 比如在多个表进行分页数据查询时，最好是能够在一个表上先过滤好数据并分好页
    - 然后再用分好页的结果集与另外的表 Join，这样可以尽可能多的减少不必要的 IO 操作，大大节省 IO 操作所消耗的时间。
- **避免类型转换**：
    - 这里的“类型转换”是指 where子句中出现column字段的类型和传入的参数类型不一致而发生的转换：分两种情况。
        - ==人为在column_name上使用转换函数==：直接导致MySQL无法使用索引(实际上其他数据库也有同样的问题)。++如果非要转换，应该在传入的参数上进行转换++。
        - ==由数据库自己进行转换==：如果传入的数据和字段两者类型不一致，同时又没有做任何类型转换处理，++MySQL 可能会自己对数据进行类型转换操作，也可能

不进行处理而交由存储引擎去处理++，++这样会导致索引无法使用++而造成执行计划问题。
    - 以上两种情况在开发的时候经常会发生，导致索引无法使用，结果造成很严重的开发事故。
- **建立索引**：
    - 对查询进行优化，++应尽量避免全表扫描++，首先应考虑在 where 及 order by 涉及的++列上建立索引++。
    - 避免在建立的索引的数据列字段上有下列操作:
        - 计算
        - 使用not，<>，!=
        - 使用IS NULL和IS NOT NULL
        - 数据类型转换
        - 使用函数
        - 索引字段中不要有null。
- **查询中有些索引无效**
    - SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引
    - 比如：一表中有字段 sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。
- **适量的索引**：
    - 怎样建索引需要视具体情况而定，索引可以提高 select 的效率，但同时会降低 insert 及 update 的效率，因为这两个操作有可能会重建索引。
    - 一个表的索引数最好不要超过6个，尽量只在常用的列上建立索引。
- **避免更新 clustered索引数据列**:
    - 因为 ==clustered（['klʌstəd]聚集）索引数据列的顺序就是表记录的物理存储顺序==，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。
    - 如果应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。
- **尽量使用数字类型字段**：
    - 若只含数值信息的字段尽量不要设计为字符型，++这会降低查询和连接的性能++，并会增加存储开销。
    - 这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。
- **尽量用 varchar/nvarchar 代替 char/nchar**：
    - ++变长字段存储空间灵活不固定++。
    - 其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。
- **用具体的字段列表代替通配符**：
    - ++ 任何地方都不要使用select * from table_naem++,用什么字段取什么字段，减少不必要的资源浪费,不要返回用不到的任何字段。
- **避免使用临时表**：
    - 除非却有需要，否则应尽量避免使用临时表，相反，++可以使用表变量代替++;
    - 大多数时候(99%)，表变量驻扎在内存中，因此速度比临时表更快，临时表驻扎在TempDb数据库中，因此临时表上的操作需要跨数据库通信，速度自然慢。

- 可以使用联合(UNION)来代替手动创建的临时表：

```
  1、UNION 查询，它可以把需要使用临时表的两条或更多的 SELECT 查询合并的一个查询中
  2、在客户端的查询会话结束的时候，临时表会被自动删除，从而保证数据库整齐、高效，
  3、使用 UNION 来创建查询的时候，我们只需要用UNION作为关键字把多个SELECT语句连接起来就可以了，要注意的是所>有 SELECT 语句中的字段数目要想同
  ---------------------------------------------------------------
  SELECT Name, Phone FROM client UNION SELECT Name, BirthDate FROM author
  UNION
  SELECT Name, Supplier FROM product
```

- **用表变量代替临时表**:
  - 如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。
  - ==临时表并不是不可使用==，++有时候它可以使某些例程更有效++
    - 例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。
    - 在新建临时表时，如果一次性插入数据量很大，==可以使用 select into 代替 create table==，避免造成大量log，以提高速度；
    - 如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。
  - 如果使用到了临时表，==在存储过程的最后务必将所有的临时表显式删除==，++先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定++。
  - **避免频繁创建和删除临时表**:以减少系统表资源的消耗。
- **尽量少使用游标**：
  - 游标是一种能从包括多条数据记录的结果集中每次提取一条记录的数据处理手段或者说机制 ，是指向查询结果集的一个指针。
  - 因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写（游标是一个集合，使用存储过程需要使用游标）。
  - ++基于集的方法通常更有效，因此使用基于游标的方法或临时表方法之前++，应先寻找基于集的解决方案来解决问题。
  - ==与临时表一样，游标并不是不可使用=：
    - 对小型数据集使用 FAST_FORWARD游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。
    - 在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。
- **优先优化高并发SQL**：
- 而不是执行频率低某些“大”SQL
- SQL优化充分考虑系统中所有的SQL，尤其是在通过调整索引优化SQL的执行计划的时候，千万不能顾此失彼，因小失大。
- **模糊查询**：
- 不能前置%,否则会导致全表扫描；

- 若要提高效率，可以考虑全文检索：`select id from t where name like '%c%' //相当于精确查找`
- **尽量避免大数据量、大事务**
- 尽量避免大事务操作，提高系统并发能力。
- 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。
- **关于存储过程和触发器**：
- 在所有的存储过程和触发器的开始处设置 set nocount on，在结束时设置 set nocount on。
- 无需在执行存储过程和触发器的每个语句后向客户端发送 done_in_proc 。
- **尽量少做重复的工作**：
- 控制同一语句的多次执行，特别是一些基础数据的多次执行。
- 减少多次的数据转换。
- 杜绝不必要的子查询和连接表，子查询在执行计划一般解释成外连接，多余的连接表带来额外的开销。
- update操作不要拆成 delete + insert 的形式，虽然功能相同，但是性能差别是很大的。
- 不要写一些没有意义的查询：`比如：SELECT * FROM EMPLOYEE WHERE 1=2`
- 优化insert语句：一次插入多值；
- 合并对同一表同一条件的多次update。
- **尽量避免在where句子中出现以下情况，否则会放弃索引进行全表扫描**
- in 和 not in 也要慎用；
- 使用!=、<、>等操作符；
- 对字段进行null值判断；
- 使用or来连接条件；
- 使用参数；
- 对字段进行表达式操作；
- 对字段进行函数操作；
- 在"="左边进行函数、算术运算或其他表达式运算，无法正确使用索引；

```
in 和 not in 也要慎用，很多时候用 exists 代替 in，not exists代表not in。
select id from t where num in(1,2,3)
------------------------------------------------------------
对于连续的数值，能用 between 就不要用 in 了
select id from t where num between 1 and 3
------------------------------------------------------------
很多时候用 exists 代替 in 是一个好的选择：
select num from a where num in(select num from b)
替换为：
select num from a where exists(select 1 from b where num=a.num)
```

对字段进行null值判断；

```
select id from t where num is null
```

可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

```
select id from t where num=0
```

```
select id from t where num=10 or num=20
```

可以这样查询：

```
select id from t where num=10 union all select id from t where num=20
```

如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。

然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫：

```
select id from t where num=@num
```

可以改为强制查询使用索引：

```
select id from t with(index(索引名)) where num=@num
```

应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描

```
select id from t where num/2=100
```

应改为：

```
select id from t where num=100*2
```

应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描

```
select id from t where substring(name,1,3)='abc'–name以abc开头的id
select id from t where datediff(day,createdate,'2005-11-30')=0–'2005-11-30'生成的id
```

应改为：

```
select id from t where name like 'abc%'
select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'
```