



MySQL性能优化

陈 飞

Jan., 2019

CONTENTS

目 录

- 1 数据库优化法则
- 2 MySQL性能优化
- 3 SQL执行分析
- 4 SQL语句优化
- 5 MySQL服务器及配置优化



数据库优化法则

-
- 数据库优化法则
 - 存储引擎对比与选择



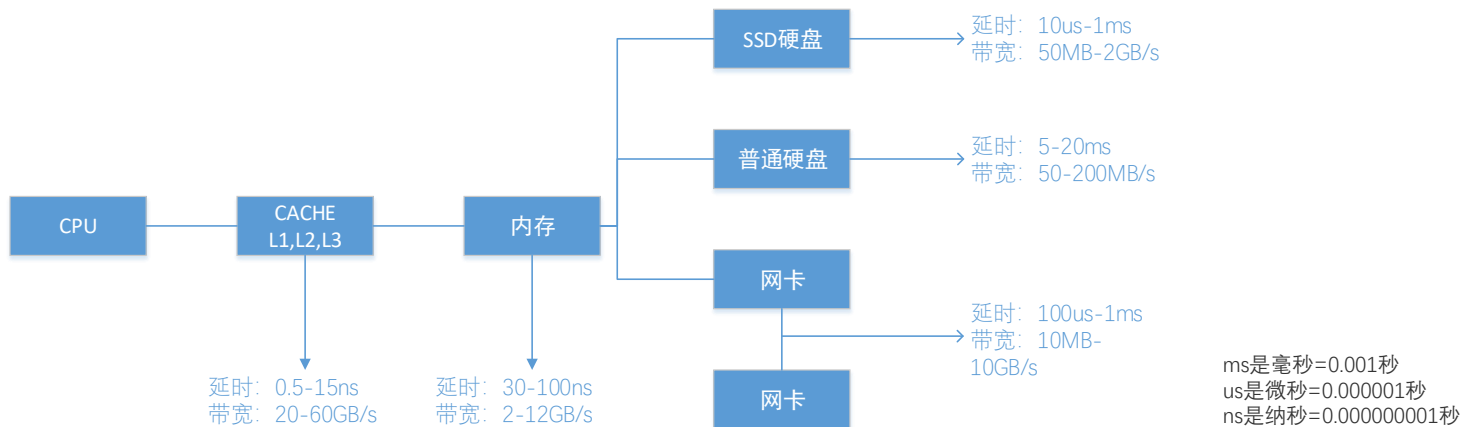
一、优化法则

- 要正确的优化SQL，我们需要快速定位性能的瓶颈点，也就是说快速找到我们SQL主要的开销在哪里？
- 而大多数情况性能最慢的设备会是瓶颈点，如下载时网络速度可能会是瓶颈点，本地复制文件时硬盘可能会是瓶颈点，为什么这些一般的工作我们能快速确认瓶颈点呢，因为我们对这些慢速设备的性能数据有一些基本的认识，如网络带宽是2Mbps，硬盘是每分钟7200转等等。
- 因此，为了快速找到SQL的性能瓶颈点，我们也需要了解我们计算机系统的硬件基本性能指标。

一、优化法则

下图展示当前主流计算机性能指标数据：

IO各层次性能汇总



图中每种设备都有两个指标：

- 延时（响应时间）：表示硬件的突发处理能力；
- 带宽（吞吐量）：代表硬件持续处理能力。

一、优化法则

从上图可以看出，计算机系统硬件性能从高到低依次为：

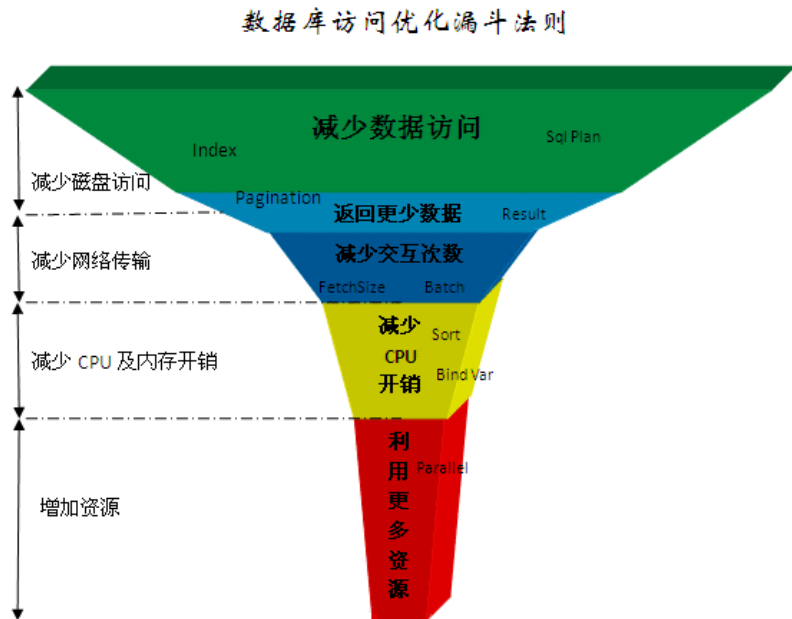
CPU > Cache(L1-L2-L3) > 内存 > SSD硬盘 > 网络 > 硬盘

每种硬件主要的工作内容：

- CPU及内存：缓存数据访问、比较、排序、事务检测、SQL解析、函数或逻辑运算；
- 网络：结果数据传输、SQL请求、远程数据库访问（dblink）；
- 硬盘：数据访问、数据写入、日志记录、大数据量排序、大表连接。

一、优化法则

性能基本优化法则：



这个优化法则归纳为5个层次：

- 1、减少数据访问（减少磁盘访问）
- 2、返回更少数据（减少网络传输或磁盘访问）
- 3、减少交互次数（减少网络传输）
- 4、减少服务器CPU开销（减少CPU及内存开销）
- 5、利用更多资源（增加资源）

一、优化法则

每个优化法则层级对应优化效果及成本经验参考：

优化法则	性能提升效果	优化成本
减少数据访问	1~1000	低
返回更少数据	1~100	低
减少交互次数	1~20	低
减少服务器CPU开销	1~5	低
利用更多资源	@~10	高



MySQL性能优化

-
- MySQL存储引擎对比与选择
 - MySQL表结构
 - MySQL索引优化



一、存储引擎对比

功 能	MYISAM	Memory	InnoDB	Archive	NDB
存储限制	256TB	RAM	64TB	None	3TB
支持事务	No	No	Yes	No	Yes
锁机制	Table	Table	Row		Row
支持全文索引	Yes	No	No	No	No
支持B-tree索引	Yes	Yes	Yes	Yes	Yes
支持哈希索引	No	Yes	No	No	Yes
支持数据缓存	No	N/A	Yes	No	Yes
支持外键	No	No	Yes	No	Yes
内存使用	Low	Middle	High		High
批量插入速度	High	High	Low	High	High

一、存储引擎选择

- 如果要提供提交、回滚、崩溃恢复能力的事物安全（ACID兼容）能力，并要求实现并发控制，InnoDB是一个好的选择。
- 如果数据表主要用来插入和查询记录，则MyISAM引擎能提供较高的处理效率。
- 如果只是临时存放数据，数据量不大，并且不需要较高的数据安全性，可以选择将数据保存在内存中的Memory引擎，MySQL中使用该引擎作为临时表，存放查询的中间结果。
- 如果只有INSERT和SELECT操作，可以选择Archive，Archive支持高并发的插入操作，但是本身不是事务安全的。Archive非常适合存储归档数据，如记录日志信息可以使用Archive。
- 使用哪一种引擎需要灵活选择，一个数据库中多个表可以使用不同引擎以满足各种性能和实际需求，使用合适的存储引擎，将会提高整个数据库的性能。

二、表结构——概述

别再将 数据库设计范式 作为数据库表结构设计“圣经”！！！！

现代的数据存储和量级决定，N年前被奉为“圣经”的数据库设计3范式早已不完全适用了。

因此，基于性能的考虑，我们常常需要有意的故意的违反三范式，适度地做冗余，以达到提高查询效率的目的。

二、表结构——概述

- 由于MySQL数据库是基于行（Row）存储的数据库，而数据库操作 IO 的时候是以 page（block）的方式，也就是说，如果我们每条记录所占用的空间量减小，就会使每个page中可存放的数据行数增大，那么每次 IO 可访问的行数也就增多了。
- 反过来说，处理相同行数的数据，需要访问的 page 就会减少，也就是 IO 操作次数降低，直接提升性能。
- 此外，由于我们的内存是有限的，增加每个page中存放的数据行数，就等于增加每个内存块的缓存数据量，同时还会提升内存换中数据命中的几率，也就是缓存命中率。

二、表结构——概述

- 数据库操作中最为耗时的操作就是 IO 处理，大部分数据库操作 90% 以上的时间都花在了 IO 读写上。所以尽可能减少 IO 读写量，可以在很大程度上提高数据库操作的性能。
- 我们无法改变数据库中需要存储的数据，但我们可以改变这些数据的存储方式！
- 因此，数据库表字段类型的选取，在数据量较大的场景下会带来意外的惊喜，尽管精细化的数据类型设置可能带来维护成本的提高。

二、表结构——字段类型的选取

- 原则：保小不保大。
- 解释：能用占用字节少的字段就不用大字段。空间就是效率！
- 更小的字段类型占用的内存就更少，占用的磁盘空间和磁盘I/O也会更少，而且还会占用更少的带宽。

二、表结构——数值类型

- 数值类型，MySQL中主要整型有五种：tinyint、smallint、mediumint、int、bigint。
- 请注意它们的占用内存空间和允许范围（最大最小值）。
- 常见的数据存储举例：
 - 1) 手机号的字段类型；
 - 2) IP地址的字段类型；

二、表结构——数值类型

1) 手机号的字段类型；

常规会选择varchar类型，但是占用空间大，影响查询性能。

因此推荐： bigint类型。

是否可以使用char(11)? 亦可，但考虑到占用空间的问题，一般字符集gbk或utf8，gbk占2字节，utf8占3字节，那么char(11)占 $11 \times 3 = 33$ 个字节，而bigint(20)宽度为20，只占用8字节。从性能考虑，bigint优于char。

2) IP地址的字段类型；

常规来说，我们IP地址的存储一般都用varchar（通病...），好一点使用char(15)。

这里IP地址也可以采用int整型（无符号）存储。MySQL提供了很好用的函数：INET_ATON()负责把IP转换为数字和INET_NTOA()负责将数字转换为IP。

示例：

```
Select INET_NTON(ip) from sys.ipaaddrs;
```

```
Select INET_NTON(ip) from sys.ipaaddrs where ip >= INET_ATON('192.168');
```

二、表结构——数值类型

3) 其他

a) 年龄采用什么类型最佳？

一般来说，年龄大都在1-120岁之间，长度只有3，用int就不合适了，可用tinyint代替。

b) 用户在线状态呢？

用户在线状态，0:离线、1:在线、2:离开、3:忙碌、4:隐身...用tinyint即可满足需求。Int占4个字节，tinyint只占1个字节。

思考：是否可以用enum枚举类型代替tinyint？它也只占用1字节。

如果采用enum枚举类型，那么就会存在扩展的问题，上述例子中，枚举值为：0、1、2、3、4。如果插入的值不在这个范围中，则会报错。

因此，假如要新增：5:请勿打扰、6:开会中、7:隐身对好友可见几个新状态，此时如果使用枚举类型，那就只能修改字段类型了...

二、表结构——数值类型

关于FLOAT、DOUBLE和DECIMAL

- FLOAT 数值类型用于表示单精度浮点数值，而 DOUBLE 数值类型用于表示双精度浮点数值。
- 与整数一样，这些类型也带有附加参数：一个显示宽度指示器和一个小数点指示器(必须要带有指示器,要不然会查不到结果,并且**宽度指示器和XXint类型的宽度指示器不同,这里是有实际限制宽度的**)。比如语句 `FLOAT(7,3)` 规定显示的值不会超过 7 位数字(包括小数位)，小数点后面带有 3 位数字。对于小数点后面的位数超过允许范围的值，MySQL 会自动将它四舍五入为最接近它的值，再插入它。
- DECIMAL 数据类型用于精度要求非常高的计算中，这种类型允许指定数值的精度和计数方法作为选择参数。精度在这里指为这个值保存的有效数字的总个数，而计数方法表示小数点后数字的位数。比如语句 `DECIMAL(7,3)` 规定了存储的值不会超过 7 位数字，并且小数点后不超过 3 位。
- 总结：对于单精度浮点数Float: 当数据范围在 ± 131072 (65536×2) 以内的时候，float数据精度是正确的，但是超出这个范围的数据就不稳定。
Double类型存在类似情况。

二、表结构——数值类型

- 关于FLOAT和DOUBLE精度的问题

例子：

```
1 CREATE TABLE f1 (i INT, f1 FLOAT(32,3), d1 DOUBLE(32,3));
2 INSERT INTO f1 VALUES (1, 131072.32, 131072.32);
3 INSERT INTO f1 VALUES (1, 131072.68, 131072.68);
4 INSERT INTO f1 VALUES (1, 13107200000000000000.66, 13107200000000000000.66);
5 SELECT * FROM f1;
```

f1 (3×3)

i	f1	d1
1	131,072.312	131,072.320
1	131,072.688	131,072.680
1	1,310,720,035,725,538,000,000.000	1,310,720,000,000,000,000.000

二、表结构——数值类型

- 关于FLOAT和DOUBLE精度的问题

例子：

```
1 CREATE TABLE t1 (i INT, d1 DOUBLE, d2 DOUBLE);
2 INSERT INTO t1 VALUES (1, 101.40, 21.40), (1, -80.00, 0.00),
3   (2, 0.00, 0.00), (2, -13.20, 0.00), (2, 59.60, 46.40),
4   (2, 30.40, 30.40), (3, 37.00, 7.40), (3, -29.60, 0.00),
5   (4, 60.00, 15.40), (4, -10.60, 0.00), (4, -34.00, 0.00),
6   (5, 33.00, 0.00), (5, -25.80, 0.00), (5, 0.00, 7.20),
7   (6, 0.00, 0.00), (6, -51.40, 0.00);
8 SELECT * FROM t1;
9
```

t1 (3×16)		
i	d1	d2
1	101.4	21.4
1	-80	0
2	0	0
2	-13.2	0
2	59.6	46.4
2	30.4	30.4
3	37	7.4
3	-29.6	0
4	60	15.4
4	-10.6	0
4	-34	0
5	33	0
5	-25.8	0
5	0	7.2
6	0	0
6	-51.4	0

二、表结构——数值类型

- 上述执行:

SELECT i, SUM(d1) AS a, SUM(d2) AS b FROM t3 GROUP BY i HAVING a <> b;

- 结果是什么?

```
1  mysql> CREATE TABLE t1 (i INT, d1 DOUBLE, d2 DOUBLE);
2  mysql> INSERT INTO t1 VALUES (1, 101.40, 21.40), (1, -80.00, 0.00),
3      -> (2, 0.00, 0.00), (2, -13.20, 0.00), (2, 59.60, 46.40),
4      -> (2, 30.40, 30.40), (3, 37.00, 7.40), (3, -29.60, 0.00),
5      -> (4, 60.00, 15.40), (4, -10.60, 0.00), (4, -34.00, 0.00),
6      -> (5, 33.00, 0.00), (5, -25.80, 0.00), (5, 0.00, 7.20),
7      -> (6, 0.00, 0.00), (6, -51.40, 0.00);
8
9  mysql> SELECT i, SUM(d1) AS a, SUM(d2) AS b
10     -> FROM t1 GROUP BY i HAVING a <> b;
11
12  +-----+-----+-----+
13  | i      | a      | b      |
14  +-----+-----+-----+
15  | 1      | 21.4   | 21.4   |
16  | 2      | 76.8   | 76.8   |
17  | 3      | 7.4    | 7.4    |
18  | 4      | 15.4   | 15.4   |
19  | 5      | 7.2    | 7.2    |
20  | 6      | -51.4  | 0       |
21  +-----+-----+-----+
```

```
10 SELECT i, SUM(d1) AS a, SUM(d2) AS b FROM t1 GROUP BY i HAVING a <> b;
```

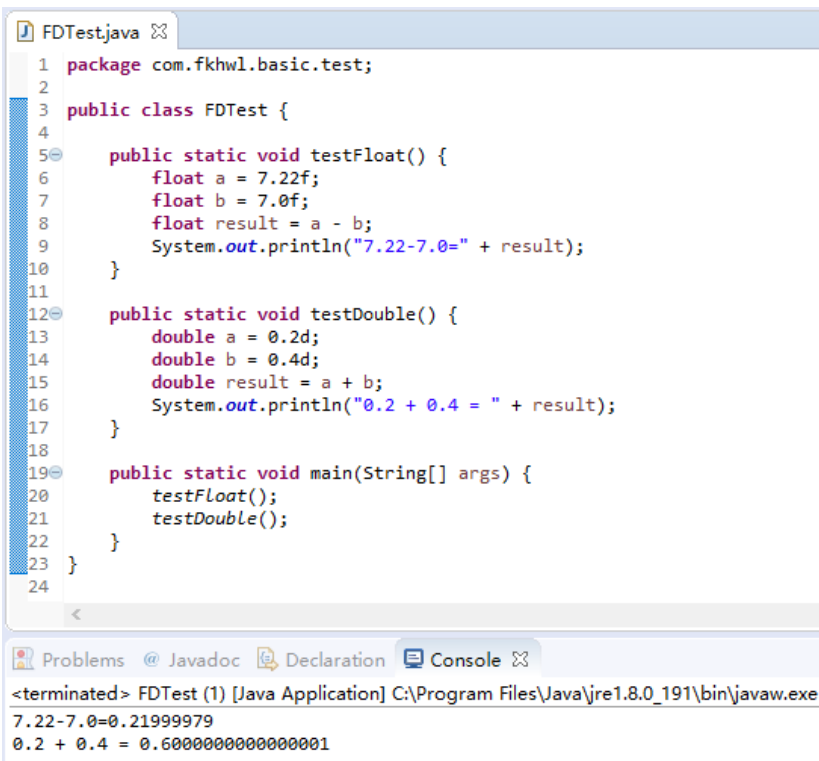
t1 (3x16)		
i	a	b
1	21.400000000000006	21.4
2	76.800000000000001	76.8
3	7.399999999999999	7.4
4	15.399999999999999	15.4
5	7.199999999999999	7.2
6	-51.4	0

此结果是错误的，尽管前5条的a和b的值看起来是不满足 $a \neq b$ 条件的。

此现象取决于各种因素，如计算机架构、编译器版本或者优化级别等。例如，不同的CPU评估的浮点数不同。

二、表结构——数值类型

- 其他平台是否存在类似的问题呢？拿我们最熟悉的Java来试试..



```
FDTest.java
1 package com.fkhwl.basic.test;
2
3 public class FDTest {
4
5     public static void testFloat() {
6         float a = 7.22f;
7         float b = 7.0f;
8         float result = a - b;
9         System.out.println("7.22-7.0=" + result);
10    }
11
12    public static void testDouble() {
13        double a = 0.2d;
14        double b = 0.4d;
15        double result = a + b;
16        System.out.println("0.2 + 0.4 = " + result);
17    }
18
19    public static void main(String[] args) {
20        testFloat();
21        testDouble();
22    }
23 }
24
```

Problems @ Javadoc Declaration Console

```
<terminated> FDTest (1) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe
7.22-7.0=0.21999979
0.2 + 0.4 = 0.6000000000000001
```

testFloat: 我们可能会想当然地认为输出结果应该是0.22，实际结果却是 0.21999979！

testDouble: 简单的0.2+0.4，但是返回的结果不是0.6！

由此可见，Float和Double类型不只是在MySQL中存在精度错误的问题，在Oracle、Java等平台同样存在此问题。

二、表结构——数值类型

- 结论：
 1. 浮点数存在误差问题；
 2. 如果想要确切的存储小数（例如，金额等），建议使用DECIMAL类型，而不是DOUBLE、float类型。对货币等对精度敏感的数据，应该用定点数表示或存储；
 3. 编程中，如果用到浮点数，要特别注意误差问题，并尽量避免做浮点数比较；
 4. 要注意浮点数中一些特殊值的处理。
- 其他：
 1. 对于整数的存储，在数据量较大的情况下，建议区分开 TINYINT、INT、BIGINT 的选择。
 2. 能确定不会使用负数的字段，建议添加unsigned定义。
 3. 永远为每张表设置一个 ID。该主键应为 INT /BIGINT或 UNSIGNED 类型，并设置上自动增加的 AUTO_INCREMENT 标志。

二、表结构——字符类型

- char和varchar是日常使用最多的字符类型。
- char(N): 用于保存固定长度的字符串, 长度255, 比指定长度大的值将被截短, 比指定长度小的值将会以空格填补。
- varchar(N): 用于保存可变长度的字符串, 长度最大为65535, 只存储字符串实际需要的长度, varchar会使用额外的1-2个字节来存储值的长度, 如果列的长度 ≤ 255 , 则使用1字节, 否则2字节。
- 与字符编码的关系: latin1占用1个字节, gbk占用2个字节, utf8占用3个字节。
- 思考:
 - 1) 采用gbk字符集, 字段可设置的最大长度? utf8呢?
 - 2) 表 (id int, name varchar(20), phone bigint, address varchar(N)) ,求N最大值?

二、表结构——字符类型

- 什么情况下使用char和varchar?

固定长度的值，使用char，经常变化的值或长度不固定的值使用varchar。

- 思考：

1) char(10)和varchar(10)效果一样吗？

2) varchar(20)或varchar(100)存储'abcd'效果一样吗？

- 解答：

问题1：效果不一样，varchar会使用额外1字节存储值的长度。

问题2：不一样，很多人都觉得把值定义大些便于以后扩展，虽然两者实际存储的空间一样，但是两者性能完全不一样。MySQL需要现在内存中分配固定的空间来保存值，值定义得大，内存浪费也就越大，而且对表的排序或使用临时表尤其不好。因此，合理分配空间非常重要！

二、表结构——字符类型

- 总结:

1. 非万不得已不要使用 TEXT 数据类型，其处理方式决定了他的性能要低于char或者是varchar类型的处理。
2. 定长字段，建议使用 CHAR 类型，不定长字段尽量使用 VARCHAR，且仅仅设定适当的最大长度。
3. 根据需要定义合适的编码集，亦可统一编码集。

二、表结构——时间类型

- MySQL中五种时间类型：DATE、TIME、DATETIME、TIMESTAMP和YEAR。
- DATETIME和TIMESTAMP都可以精确到秒，但DATETIME占8个字节，TIMESTAMP只占4个字节。
- 日常建表时建议：优先选择TIMESTAMP类型（具有自动更新时间功能，MySQL5.6版本后有突破，后续会阐述）。

数据类型	值	存储的字节
DATE	'0000-00-00'	3字节
TIME	'00:00:00'	3字节
DATETIME	'0000-00-00 00:00:00'	8字节
TIMESTAMP	'0000-00-00 00:00:00'	4字节
YEAR	'0000'	1字节

二、表结构——时间类型

- 在MySQL5.6+版本中，时间类型TIMESTAMP和DATETIME有重大改变。
- MySQL5.5或更老版本，TIMESTAMP类型，一个表只允许一个字段既拥有自动插入时间，又拥有自动更新时间。同时，不支持多个字段设置为TIMESTAMP类型。
- MySQL5.6+版本后，一个表可以有多个字段拥有自动插入时间和自动更新时间。另外，DATETIME类型也拥有TIMESTAMP类似的功能。支持多个字段设置为TIMESTAMP类型。
- MySQL5.6+版本后，对YEAR(2)不再识别，会自动转换为YEAR(4)，如果插入一条记录18，会自动转换为2018.

二、表结构——时间类型

- 总结:

1. 根据实际需要选择能够满足应用的最小存储日期类型;
2. 尽量使用TIMESTAMP类型, 因为其存储空间只需要 DATETIME 类型的一半;
3. 对于只需要精确到某一天的数据类型, 建议使用DATE类型, 因为他的存储空间只需要3个字节, 比TIMESTAMP还少;
4. 如果记录年月日时分秒, 并且记录年份比较久远, 最好使用DATETIME, 不要使用TIMESTAMP;
5. 如果记录的日期需要让不同时区的用户使用, 最好使用TIMESTAMP, 因为日期类型中只有它能够和实际时区相对应;
6. 不建议通过INT类型类存储一个UNIX TIMESTAMP的值, 因为这太不直观, 会给维护带来不必要的麻烦, 同时还不会带来任何好处。

三、索引优化

大家都知道索引对于数据访问的性能有非常关键的作用，都知道索引可以提高数据访问效率。请思考：

1. 为什么索引能提高数据访问性能？
2. 他会不会有“副作用”？
3. 是不是索引创建越多，性能就越好？
4. 到底该如何设计索引，才能最大限度的发挥其效能？

下面将会通过一个非常典型的例子来讲解上述问题。

三、索引优化

- 问题1：为什么索引能提高数据访问性能？

假如我们让唐惠去图书馆确认一本叫做《MySQL运维内参》的书是否在藏，这样对她说：“请帮我借一本计算机类的数据库书籍，是属于 MySQL 数据库范畴的，叫做《MySQL运维内参》”。唐惠会根据所属类别，前往存放“计算机”书籍区域的书架，寻找“数据库”类存放位置，再找到一堆讲述“MySQL”的书籍，最后可能发现目标在藏（也可能已外借）。在这个过程中：“计算机”->“数据库”->“MySQL”->“在藏”->《MySQL运维内参》其实就是一个“根据索引查找数据”的典型示例，“计算机”->“数据库”->“MySQL”->“在藏”就是朋友查找书籍的索引。假设没有这个索引，那查找这本书的过程会变成怎样呢？唐惠只能从图书馆入口一个书架一个书架的“遍历”，直到找到《MySQL运维内参》这本书为止。如果幸运，可能在第一个书架就找到。但如果不幸呢，那就惨了，可能要将整个图书馆所有的书架都找一遍才能找到那本书。

注：这个例子中的“索引”是记录在唐惠大脑中的，实际上，每个图书馆都会有一个非常全的实际存在的索引系统（大多位于入口显眼处），由很多个贴上了明显标签的小抽屉构成。这个索引系统中存放这非常齐全详尽的索引数据，标识出我们需要查找的“目标”在某个区域的某个书架上。而且每当有新的书籍入库，旧的书籍销毁以及书记信息修改，都需要对索引系统进行及时的修正。

三、索引优化

- 问题2：索引有哪些“副作用”？
 1. 图书的变更（增，删，改）都需要修订索引，索引存在额外的维护成本；
 2. 查找翻阅索引系统需要消耗时间，索引存在额外的访问成本；
 3. 这个索引系统需要一个地方来存放，索引存在额外的空间成本。

三、索引优化

- 问题3：索引是不是越多越好？

1. 如果我们的这个图书馆只是一个进出中转站，里面的新书进来后很快就会转发去其他图书馆而从这个馆藏中“清除”，那我们的索引就只会不断的修改，而很少会被用来查找图书。

所以，对于类似于这样的存在非常大更新量的数据，索引的维护成本会非常高，如果其检索需求很少，而且对检索效率并没有非常高的要求时，不建议创建索引或者是尽量减少索引。

2. 如果我们的书籍量少到只有几本或者就只有一个书架，索引并不会带来什么作用，甚至可能还会浪费一些查找索引所花费的时间。

所以，对于数据量极小到通过索引检索还不如直接遍历来得快的数据，也并不适合使用索引。

3. 如果我们的图书馆只有一个10平方的面积，现在连放书架都已经非常拥挤，而且馆藏还在不断增加，我们还能考虑创建索引吗？

所以，当我们连存储基础数据的空间都捉襟见肘的时候，我们也应该尽量减少低效或者是去除索引。

三、索引优化

- 问题4：索引该如何设计才高效？

1. 如果我们仅仅只是这样告诉对方的：“帮我确认一本数据库类别的讲述 MySQL 的叫做《MySQL性能调优与架构设计》的书是否在藏”，结果又会如何呢？朋友只能一个大类区域一个大类区域的去寻找“数据库”类别，然后再找到“MySQL”范畴，再看到我们所需是否在藏。由于我们少说了一个“计算机类”，朋友就必须到每一个大类去寻找。

所以，我们应该尽量让查找条件尽可能多的在索引中，尽可能通过索引完成所有过滤，回表只是取出额外的数据字段。

2. 如果我们是这样说的：“帮我确认一本讲述 MySQL 的数据库范畴的计算机丛书，叫做《MySQL性能调优与架构设计》，看是否在藏”。如果这位朋友并不知道计算机是一个大类，也不知道数据库属于计算机大类，那这位朋友就悲剧了。首先他得遍历每个类别确认“MySQL”存在于哪些类别中，然后从包含“MySQL”书籍中再看有哪些是“数据库”范畴的（有可能部分是讲述PHP或者其他开发语言的），然后再排除非计算机类的（虽然可能并没有必要），然后才能确认。

所以，字段的顺序对组合索引效率有至关重要的作用，过滤效果越好的字段需要更靠前。

三、索引优化

3. 如果我们还有这样一个需求（虽然基本不可能）：“帮我将图书馆中所有的计算机图书借来”。朋友如果通过索引来找，每次都到索引柜找到计算机书籍所在的区域，然后从书架上搬下一格（假设只能以一格为单位从书架上取下，类比数据库中以block/page为单位读取），取出第一本，然后再从索引柜找到计算机图书所在区域，再搬下一格，取出一本...如此往复直至取完所有的书。如果他不通过索引来找又会怎样呢？他需要从地一个书架一直往后找，当找到计算机的书，搬下一格，取出所有计算机的书，再往后，直至所有书架全部看一遍。在这个过程中，如果计算机类书籍较多，通过索引来取所花费的时间很可能要大于直接遍历，因为不断往复的索引翻阅所消耗的时间会非常长。

所以，当我们需要读取的数据量占整个数据量的比例较大抑或者说索引的过滤效果并不是太好的时候，使用索引并不一定优于全表扫描。

三、索引优化

4. 如果我们的朋友不知道“数据库”这个类别可以属于“计算机”这个大类，抑或者图书馆的索引系统中这两个类别属性并没有关联关系，又会怎样呢？也就是说，朋友得到的是2个独立的索引，一个是告知“计算机”这个大类所在的区域，一个是“数据库”这个小类所在的区域（很可能是多个区域），那么他只能二者选其一来搜索我的需求。即使朋友可以分别通过2个索引检索然后自己在脑中取交集再找，那这样的效率实际过程中也会比较低。所以，在实际使用过程中，一次数据访问一般只能利用到1个索引，这一点在索引创建过程中一定要注意，不是说一条SQL语句中Where子句里面每个条件都有索引能对应上就可以了。

三、索引优化

5. 最后总结一下法则：不要在建立的索引的数据列上进行下列操作：

- ◆ 避免对索引字段进行计算操作；
- ◆ 避免在索引字段上使用not, !=；
- ◆ 避免在索引列上使用IS NULL和IS NOT NULL；
- ◆ 避免在索引列上出现数据类型转换；
- ◆ 避免在索引字段上使用函数；
- ◆ 避免建立索引的列中使用空值。

其他有关索引的优化举例说明详见二期SQL优化分享中，这里不过多阐述...



SQL执行分析

-
- SQL执行时间分析
 - SQL执行情况分析
 -



一、SQL 执行时间分析——processlist

1、通过 show processlist 来查看系统的执行情况：

```
mysql> show processlist;
```

Id	User	Host	db	Command	Time	State	Info
2	root	localhost	NULL	Query	0	init	show processlist

1 row in set (0.01 sec)

参数：

Id	# ID标识，要kill一个语句的时候很有用
use	# 当前连接用户
host	# 显示这个连接从哪个ip的哪个端口上发出
db	# 数据库名
command	# 连接状态，一般是休眠（sleep）、查询（query）、连接（connect）
time	# 连接持续时间，单位是秒
state	#显示使用当前连接的sql语句的状态，非常关键的列
	mysql手册链接： http://dev.mysql.com/doc/refman/5.0/en/general-thread-states.html
info	# 显示这个sql语句

一、SQL 执行时间分析——profiling

2、通过 profiling 来进行查看SQL 的执行时间，可直观查看SQL执行快慢：

2.1 查看 profiling 是否开启：

```
mysql> select @@profiling;
```

```
+-----+
| @@profiling |
+-----+
| 0           |
+-----+
```

0 代表当前关闭分析功能；1 代表开启分析功能

2.2 打开profiling分析功能：

```
mysql> set profiling=1;
```

Query OK, 0 rows affected, 1 warning (0.01 sec)

```
mysql> select @@profiling;
```

```
+-----+
| @@profiling |
+-----+
| 1           |
+-----+
```

2.3 查看 SQL 的执行时间：

```
mysql> show profiles;
```

```
+-----+-----+-----+
| Query_ID | Duration | Query |
+-----+-----+-----+
| 1        | 0.00173700 | select * from ip |
| 2        | 0.00057500 | select porxy, port from ip |
+-----+-----+-----+
```

一、SQL 执行时间分析——profiling

2.4 查看 SQL 执行耗时详细信息

语法: show profile for query *Query_ID*

```
mysql> show profile for query 1;
+-----+-----+
| Status                | Duration |
+-----+-----+
| starting               | 0.000073 |
| checking permissions   | 0.000031 | ---检查是否在缓存中
| Opening tables         | 0.000207 | ---打开表
| init                   | 0.000067 | ---初始化
| System lock            | 0.000040 | ---锁系统
| optimizing              | 0.000005 | ---优化查询
| statistics             | 0.000021 |
| preparing              | 0.000015 | ---准备
| executing               | 0.000003 | ---执行
| Sending data           | 0.000993 |
| end                    | 0.000006 |
| query end              | 0.000007 |
| closing tables         | 0.000011 |
| freeing items          | 0.000169 |
| cleaning up            | 0.000089 |
+-----+-----+
```

一、SQL 执行时间分析——慢日志

MySQL 的慢查询日志，顾名思义就是把执行时间超过设定值（默认为10s）的 SQL 记录到日志中。这项功能需要手动开启，但是开启后会造成一定的性能损耗。

3.1 查看慢日志是否开启

默认情况下slow_query_log的值为OFF，表示慢查询日志是禁用的，可以通过设置slow_query_log的值来开启。

语法: `set global slow_query_log=1`

```
mysql> show variables like '%slow_query_log%';
```

Variable_name	Value
slow_query_log	OFF
slow_query_log_file	/usr/local/var/mysql/***-slow.log

2 rows in set (0.11 sec)

```
mysql> set global slow_query_log=1;
```

Query OK, 0 rows affected (0.03 sec)

```
mysql> show variables like '%slow_query_log%';
```

Variable_name	Value
slow_query_log	ON
slow_query_log_file	/usr/local/var/mysql/***-slow.log

一、SQL 执行时间分析——慢日志

3.2 设置超时时间

- 设置语法: `set global long_query_time=4`
- 查看语法: `show variables like 'long_query_time'`

注意: 修改后, 需要重新连接或新开一个会话才能看到修改值。

永久生效, 修改 `my.cnf` 配置文件。

- `slow_query_log=1`
- `long_query_time=10`
- `slow_query_log_file=/path/mysql_slow.log`

一、SQL 执行时间分析——慢日志

3.3 其他参数

3.3.1 log_output

参数是指定日志的存储方式。log_output='FILE'表示将日志存入文件，默认值是'FILE'。

log_output='TABLE'表示将日志存入数据库，这样日志信息就会被写入到mysql.slow_log表中。

MySQL数据库支持同时两种日志存储方式，配置的时候以逗号隔开即可，如：

log_output='FILE,TABLE'。日志记录到系统的专用日志表中，要比记录到文件耗费更多的系统资源，因此对于需要启用慢查询日志，又需要能够获得更高的系统性能，那么建议优先记录到文件。

3.3.2 log-queries-not-using-indexes

未使用索引的查询也被记录到慢查询日志中（可选项）。如果调优的话，建议开启这个选项。

另外，开启了这个参数，其实使用full index scan的sql也会被记录到慢查询日志。

3.3.3 log_slow_admin_statements

表示是否将慢管理语句例如ANALYZE TABLE和ALTER TABLE等记入慢查询日志

一、SQL 执行时间分析——慢日志

3.4 分析工具 mysqldumpslow

MySQL 提供了慢日志分析工具 mysqldumpslow。

-s 表示按照何种方式排序：

- c: 访问计数
- l: 锁定时间
- r: 返回记录
- t: 查询时间
- al: 平均锁定时间
- ar: 平均返回记录数
- at: 平均查询时间

-t 是top n的意思，即为返回前面多少条的数据；

-g 后边可以写一个正则匹配模式，大小写不敏感的；

一、SQL 执行时间分析——慢日志

3.4.1 命令示例

得到返回记录集最多的 10 个 SQL:

```
mysql> mysqldumpslow -s r -t 10 /path/mysql/mysql_slow.log
```

得到访问次数最多的 10 个 SQL:

```
mysql> mysqldumpslow -s c -t 10 /path/mysql/mysql_slow.log
```

得到按照时间排序的前10条里面含有左连接的查询语句:

```
mysql> mysqldumpslow -s t -t 10 -g "left join" /path/mysql/mysql_slow.log
```

另外建议在使用这些命令时结合 | 和 more 使用，否则有可能出现刷屏的情况:

```
mysql> mysqldumpslow -s r -t 20 /path/mysql/mysql-slow.log | more
```

二、SQL 执行情况分析

可以通过EXPLAIN命令获取MySQL如何执行SELECT语句的信息，包括在SELECT语句执行过程中表如何连接和连接的顺序。

```
explain select * from ip;
```

```
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | ip    | ALL  | NULL          | NULL | NULL    | NULL | 400  | NULL  |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

select_type	table	type	possible_keys	key	key_len	rows	Extra
表示查询的类型	输出结果集的表	表示表的连接类型	表示查询时，可能使用的索引	表示实际使用的索引	索引字段的长度	扫描出的行数(估算的行数)	执行情况的描述和说明

二、SQL 执行情况分析

· **select_type**:表示SELECT的类型，常见的取值有：

类型	说明
SIMPLE	简单表，不使用表连接或子查询
PRIMARY	主查询，即外层的查询
UNION	UNION中的第二个或者后面的查询语句
SUBQUERY	子查询中的第一个

· **table**:输出结果集的表（表别名）

· **type**:表示MySQL在表中找到所需行的方式，或者叫访问类型。常见访问类型如右表格，从上到下，性能由差到最好：

ALL	全表扫描
index	索引全扫描
range	索引范围扫描
ref	非唯一索引扫描
eq_ref	唯一索引扫描
const,system	单表最多有一个匹配行
NULL	不用扫描表或索引

二、SQL 执行情况分析

- **type=ref**，使用非唯一索引或唯一索引的前缀扫描，返回匹配某个单独值的记录行
- **store_id**字段存在普通索引（非唯一索引）

```
mysql> EXPLAIN SELECT * FROM customer WHERE store_id=10;
```

```
mysql> EXPLAIN SELECT * FROM customer WHERE store id=10;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	ref	idx_fk_store_id	idx_fk_store_id	1	const	1	100.00	NULL

```
1 row in set, 1 warning (0.00 sec)
```

二、SQL 执行情况分析

ref类型还经常会出现join操作中：

customer、**payment**表关联查询，关联字段**customer.customer_id**（主键），**payment.customer_id**（非唯一索引）。表关联查询时必定会有一张表进行全表扫描，此表一定是几张表中记录行数最少的表，然后再通过非唯一索引寻找其他关联表中的匹配行，以此达到表关联时扫描行数最少。

```
mysql> SELECT count(*) from customer;
+-----+
| count(*) |
+-----+
| 599 |
+-----+
1 row in set (0.00 sec)

mysql> SELECT count(*) from payment ;
+-----+
| count(*) |
+-----+
| 16049 |
+-----+
1 row in set (0.01 sec)
```

因为**customer**、**payment**两表中**customer**表的记录行数最少，所以**customer**表进行全表扫描，**payment**表通过非唯一索引寻找匹配行。

```
mysql> EXPLAIN SELECT * FROM customer customer INNER JOIN payment payment ON
customer.customer_id = payment.customer_id;
```

```
mysql> EXPLAIN SELECT * FROM customer customer INNER JOIN payment payment ON customer.customer_id = payment.customer_id;
+-----+
| id | select_type | table | partitions | type | possible keys | key | key_len | ref | rows | filtered | Extra |
+-----+
| 1 | SIMPLE | customer | NULL | ALL | PRIMARY | NULL | NULL | NULL | 599 | 100.00 | NULL |
| 1 | SIMPLE | payment | NULL | ref | idx_fk_customer_id | idx_fk_customer_id | 2 | sakila.customer.customer_id | 26 | 100.00 | NULL |
+-----+
2 rows in set, 1 warning (0.00 sec)
```

二、SQL 执行情况分析

- **type=eq_ref**，类似ref，区别在于使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配eq_ref一般出现在多表连接时使用primary key或者unique index作为关联条件。

film、film_text表关联查询和上一条所说的基本一致，只不过关联条件由非唯一索引变成了主键。

```
mysql> EXPLAIN SELECT * FROM film film INNER JOIN film_text film_text ON film.film_id = film_text.film_id;
```

```
mysql> EXPLAIN SELECT * FROM film film INNER JOIN film_text film_text ON film.film_id = film_text.film_id;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film_text	NULL	ALL	PRIMARY	NULL	NULL	NULL	1000	100.00	NULL
1	SIMPLE	film	NULL	eq_ref	PRIMARY	PRIMARY	2	sakila.film_text.film_id	1	100.00	Using where

2 rows in set, 1 warning (0.00 sec)

- **type=const/system**，单表中最多有一条匹配行，查询起来非常迅速，所以这个匹配行的其他列的值可以被优化器在当前查询中当作常量来处理

const/system出现在根据主键primary key或者 唯一索引 unique index 进行的查询

根据主键primary key进行的查询：

```
mysql> EXPLAIN SELECT * FROM customer WHERE customer_id =10;
```

```
mysql> EXPLAIN SELECT * FROM customer WHERE customer_id =10;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	const	PRIMARY	PRIMARY	2	const	1	100.00	NULL

1 row in set, 1 warning (0.00 sec)

二、SQL 执行情况分析

根据唯一索引unique index进行的查询：

```
mysql> EXPLAIN SELECT * FROM customer WHERE email = 'MARY.SMITH@sakilacustomer.org';
```

```
mysql> ALTER TABLE customer add UNIQUE INDEX uk_email(email);
Query OK, 0 rows affected (0.10 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

```
mysql> DESC customer;
```

Field	Type	Null	Key	Default	Extra
customer_id	smallint(5) unsigned	NO	PRI	NULL	auto_increment
store_id	tinyint(3) unsigned	NO	MUL	NULL	
first_name	varchar(45)	NO		NULL	
last_name	varchar(45)	NO	MUL	NULL	
email	varchar(50)	YES	UNI	NULL	
address_id	smallint(5) unsigned	NO	MUL	NULL	
active	tinyint(1)	NO		1	
create_date	datetime	NO		NULL	
last_update	timestamp	NO		CURRENT_TIMESTAMP	on update CURRENT_TIMESTAMP

9 rows in set (0.00 sec)

```
mysql> EXPLAIN SELECT * FROM customer WHERE email = 'MARY.SMITH@sakilacustomer.org';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customer	NULL	const	uk_email	uk_email	153	const	1	100.00	NULL

1 row in set, 1 warning (0.00 sec)

二、SQL 执行情况分析

- **type=NULL**，MySQL不用访问表或者索引，直接就能够得到结果

```
mysql> mysql> EXPLAIN SELECT 1 FROM dual WHERE 1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	No tables used

1 row in set, 1 warning (0.01 sec)

二、SQL 执行情况分析

- **possible_keys:** 表示查询可能使用的索引
- **key:** 实际使用的索引
- **key_len:** 使用索引字段的长度
- **ref:** 使用哪个列或常数与key一起从表中选择行。
- **rows:** 扫描行的数量
- **filtered:** 存储引擎返回的数据在server层过滤后，剩下多少满足查询的记录数量的比例(百分比)
- **Extra:** 执行情况的说明和描述，包含不适合在其他列中显示，但是对执行计划非常重要的额外信息

二、SQL 执行情况分析

Extra最主要的有以下三种：

Using Index	表示索引覆盖，不会回表查询
Using Where	表示进行了回表查询
Using Index Condition	表示进行了ICP优化
Using Filesort	表示MySQL需额外排序操作, 不能通过索引顺序达到排序效果

什么是ICP?

```
mysql>EXPLAIN SELECT * FROM rental WHERE rental_date='2005-05-25' AND customer_id>=300 AND customer_id<=400;
```

在5.6版本之前:

优化器首先使用复合索引idx_rental_date过滤出符合条件`rental_date='2005-05-25'`的记录，然后根据复合索引idx_rental_date回表获取记录，最终根据条件`customer_id>=300 AND customer_id<=400`过滤出最后的查询结果（在服务层完成）。

在5.6版本之后:

MySQL使用了ICP来进一步优化查询，在检索的时候，把条件`customer_id>=300 AND customer_id<=400`也推到存储引擎层完成过滤，这样能够降低不必要的IO访问。Extra为`Using index condition`就表示使用了ICP优化。

```
mysql> EXPLAIN SELECT * FROM rental WHERE rental_date='2005-05-25' AND customer_id>=300 AND customer_id<=400;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	rental	NULL	ref	idx_fk_customer_id,idx_rental_date	idx_rental_date	5	const	1	16.85	Using index condition

1 row in set, 1 warning (0.00 sec)



SQL语句优化

-
- SQL执行时间分析
 - SQL执行情况分析
 -



一、SQL 执行时间分析

qqq



MySQL服务器及配置优化

-
- SQL执行时间分析
 - SQL执行情况分析
 -



一、SQL 执行时间分析

qqq