久

杨

珍

精

悟尽归原 几近于道。

create by Yangcl

2018-03-15

1 Mysql基础篇

2 Mysql高级特性

2.1数据库索引与高性能查询

2.1.1 MySQL索引5大原则

**A：最左前缀匹配原则(重要的原则)**

mysql会一直向右匹配直到遇到范围查询（>、<、between、like）就停止匹配。比如a = 1 and b = 2 and c > 3 and d = 4，如果建立（a,b,c,d）顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。 一个查询中，如果第一个查询条件为索引，同时因为使用了>、<、between或like导致他失效了，那么索引整体失效。

**B：=和in可以乱序**

比如a = 1 and b = 2 and c = 3 建立（a,b,c）索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式。

**C：尽量选择区分度高的列作为索引**

区分度的公式是count(distinct column)/count(\*)。表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录。

**D：索引列不能参与计算，保持列“干净”**

比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create\_time = unix\_timestamp(‘2014-05-29’);

**E：尽量的扩展索引，不要新建索引**

比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

2.1.2 索引使用的典型场景(使用索引优化查询)

A：匹配全值

对索引中所有列都指定具体值，即对索引中的所有列都有等值匹配的条件。

#设置组合索引（rental\_date,inventory\_id,customer\_id）为唯一索引。

EXPLAIN

SELECT \* FROM

rental

WHERE rental\_date = '2005-05-25 17:22:10'

AND inventory\_id = 373

AND customer\_id = 343 ;



B：匹配值的范围查询

对索引值进行范围查找，这种情况最常见。

#设置索引idx\_fk\_customer\_id(customer\_id)

EXPLAIN

SELECT

\*

FROM

rental

WHERE customer\_id >= 373

AND customer\_id < 400 ;



**C：匹配最左前缀**

仅仅使用索引中的最左边列进行查询。比如组合索引（col1,col2,col3）能够被col1，col1+col2，col1+col2+col3的等值查询利用到的。

#创建索引idx\_payment\_date(payment\_date,amount,last\_update);

EXPLAIN

SELECT \* FROM

payment

WHERE payment\_date = '2006-02-14 15:16:03'

AND last\_update = '2006-02-15 22:12:32' ;



从结果可以看出利用了索引，但又row为182行，所有只使用了部分索引。换一种方式，如下脚本，将不适用索引，请注意标红色的amount条件

EXPLAIN

SELECT \* FROM

payment

WHERE **amount** = 3.98

AND last\_update = '2006-02-15' ;



从结果看出，这次查询没有利用索引，进行了全表查找。因为我们所创建的索引中没有包含amount参数。

**D：所查询列全为索引**

当查询列都在索引字段中。即select中的列都在索引中。

#创建索引idx\_payment\_date(payment\_date,amount,last\_update);

EXPLAIN

SELECT

last\_update

FROM

payment

WHERE payment\_date = '2005-08-19 21:21:47'

AND amount = 4.99 ;



Extra内容显示为Using index,说明不需要通过索引回表，Using index就是平时说的覆盖索引扫描（即找到索引，就找到了要查询的结果，不用再回表查找了）。

**E：匹配列前缀**

仅仅使用索引的第一列，并且只包含索引第1列的开头部分进行查找。

#创建索引idx\_title\_desc\_part(title(10)，description(20));

EXPLAIN

SELECT

title

FROM

film\_text

WHERE title LIKE 'african%' ;



type=range，范围扫描，基于索引做范围扫描，为诸如BETWEEN，IN，>=，LIKE类操作提供支持。

为提高like的查询效率，请注意创建索引的代码中红色的部分，title(10)大意为匹配前10个字符。大表查询中极少使用LIKE这个关键字。

**F：索引部分等值匹配，部分范围匹配**

EXPLAIN

SELECT

inventory\_id

FROM

rental

WHERE rental\_date = '2006-02-14 15:16:03'

AND customer\_id >= 300

AND customer\_id <= 400 ;



type=ref，说明使用了索引。

**G：列名是索引，column\_name is null，使用索引**

EXPLAIN

SELECT

\*

FROM

payment

WHERE rental\_id IS NULL ;



2.1.3索引存在但不能使用索引的典型场景

**A：以%开头的like查询**

EXPLAIN

SELECT

\*

FROM

actor

WHERE last\_name LIKE '%NI%' ; **#百分号“%”只能在最右的情况索引才不会失效！**



**B：数据类型出现隐式转化，不会使用索引**

EXPLAIN

SELECT

\*

FROM

actor

WHERE last\_name = 1 ;



last\_name为varchar类型存储，但匹配值为int类型的数字，此时索引失效；正确如下：WHERE last\_name = '1' ;

**C：组合索引，不满足最左原则，不使用符合索引**

详见2.1.1 A中的说明。

**D：用索引比全表扫描还慢，则不要使用索引**

如查询以“S”开头的标题的电影，返回记录比例比较大，mysql预估索引扫描还不如全表扫描。

**E：用or分割条件，若or前后只要有一个查询条件的列没有索引，就都不会用索引**

EXPLAIN

SELECT

\*

FROM

payment

WHERE customer\_id = 203

OR amount = 3.96 ;



2.1.4 mysql explain结果中type的含义

|  |  |
| --- | --- |
| type值 | 含义 |
| system | CONST的特例，当表上只有一条元组匹配 |
| const | WHERE条件筛选后表上至多有一条元组匹配时，比如WHERE ID = 2 （ID是主键，值为2的要么有一条要么没有） |
| eq\_ref | 参与连接运算的表是内表（在代码实现的算法中，两表连接时作为循环中的内循环遍历的对象，这样的表称为内表）。  基于索引（连接字段上存在唯一索引或者主键索引，且操作符必须是“=”谓词，索引值不能为NULL）做扫描，使得对外表的一条元组，内表只有唯一一条元组与之对应。 |
| ref | 可以用于单表扫描或者连接。参与连接运算的表，是内表。基于索引（连接字段上的索引是非唯一索引，操作符必须是“=”谓词，连接字段值不可为NULL）做扫描，使得对外表的一条元组，内表可有若干条元组与之对应。 |
| ref\_or\_null | 类似REF，只是搜索条件包括：连接字段的值可以为NULL的情况，比如 where col = 2 or col is null |
| range | 范围扫描，基于索引做范围扫描，为诸如BETWEEN，IN，>=，LIKE类操作提供支持 |
| index\_scan | 索引做扫描，是基于索引在索引的叶子节点上找满足条件的数据（不需要访问数据文件） |
| all | 全表扫描或者范围扫描：**不使用索引**，顺序扫描，直接读取表上的数据（访问数据文件） |
| unique\_subquery | 在子查询中，基于唯一索引进行扫描，类似于EQ\_REF |
| index\_subquery | 在子查询中，基于除唯一索引之外的索引进行扫描 |
| index\_merge | 多重范围扫描。两表连接的每个表的连接字段上均有索引存在且索引有序，结果合并在一起。适用于作集合的并、交操作。 |
| ft | FULL TEXT，全文检索 |

你定义的索引在查询时是否生效取决于这里展示的type的值。

2.2数据库索引深层原理

2.1节从实用性角度总结了在开发中用到的索引知识点，尤其在重构某个功能或者项目模块的时候显得尤为重要。但是并没有从一个深层的角度来认识索引的本质。

这一节中将汇集所有优秀的文章，加以总结：什么是索引的本质。

2.2.1索引的数据结构及算法基础

**A：索引的本质**

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。提取句子主干，就可以得到索引的本质：索引是一种数据结构。数据库查询是数据库的主要功能之一，最基本的查询算法是顺序查找（linear search）时间复杂度为O(n)，显然在数据量很大时效率很低。优化的查找算法如二分查找（binary search）、二叉树查找（binary tree search）等，虽然查找效率提高了。但是各自对检索的数据都有要求：二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织）。所以，在数据之外，数据库系统还维护着满足特定查找算法的数据结构。这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构就是索引。



上图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)的复杂度内获取到相应数据。虽然这是一个货真价实的索引，但是实际的数据库系统几乎没有使用二叉查找树或其进化品种红黑树（red-black tree）实现的，原因会在下文介绍。

**B：B-Tree**

为了描述B-Tree，首先定义一条数据记录为一个二元组[key, data]，key为记录的键值，对于不同数据记录，key是互不相同的；data为数据记录除key外的数据。那么B-Tree是满足下列条件的数据结构：

1. d>=2，即B-Tree的度；
2. h为B-Tree的高；
3. 每个非叶子结点由n-1个key和n个指针组成，其中d<=n<=2d；
4. 每个叶子结点至少包含一个key和两个指针，最多包含2d-1个key和2d个指针，叶结点的指针均为NULL；
5. 所有叶结点都在同一层，深度等于树高h；
6. key和指针相互间隔，结点两端是指针；
7. 一个结点中的key从左至右非递减排列；
8. 如果某个指针在结点node最左边且不为null，则其指向结点的所有key小于v(key1)，其中v(key1)为node的第一个key的值。
9. 如果某个指针在结点node最右边且不为null，则其指向结点的所有key大于v(keym)，其中v(keym)为node的最后一个key的值。
10. 如果某个指针在结点node的左右相邻key分别是keyi和keyi+1且不为null，则其指向结点的所有key小于v(keyi+1)且大于v(keyi)。



由于B-Tree的特性，在B-Tree中按key检索数据的算法非常直观：首先从根节点进行二分查找，如果找到则返回对应节点的data，否则对相应区间的指针指向的节点递归进行查找，直到找到节点或找到null指针，前者查找成功，后者查找失败。B-Tree上查找算法的伪代码如下：

BTree\_Search(node, key) {

if(node == null) return null;

foreach(node.key)

{

if(node.key[i] == key) return node.data[i];

if(node.key[i] > key) return BTree\_Search(point[i]->node);

}

return BTree\_Search(point[i+1]->node);

}

data = BTree\_Search(root, my\_key);

关于B-Tree有一系列有趣的性质，例如一个度为d的B-Tree，设其索引N个key，则其树高h的上限为logd((N+1)/2)，检索一个key，其查找结点个数的渐进复杂度为O(logdN)。从这点可以看出，B-Tree是一个非常有效率的索引数据结构。

https://www.cnblogs.com/tgycoder/p/5410057.html

**C：B+Tree**

http://blog.codinglabs.org/articles/index-condition-pushdown.html

3 MyCat In Action

3.1 数据库集群基本术语

3.1.1 数据切分

Mycat的核心功能，通过某种特定的条件，将放在同一个数据库中的数据分散存放在多个数据库主机中，以达到分散到多台设备负载的效果。

数据切分有两种切分模式，一种是按照不同的表，将数据切分到不同的数据库主机中，这种切分方式称为数据的垂直切分，也叫纵向切分；另一种

则是根据表中的数据逻辑关系，将同一个表中的数据按照某种特定条件(比如：租户id)拆分到多个数据库主机中，这种方式称为数据的水平切分，也

叫做横向切分，当单表数据量达到800万条以上的时候，通常会对该表做横向切分。

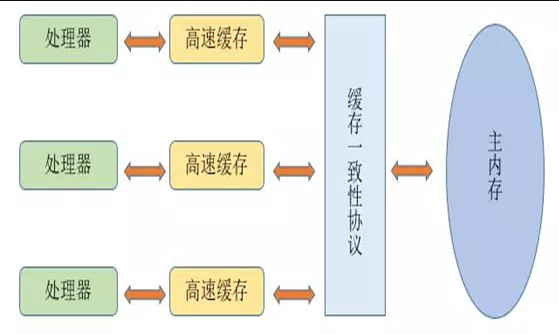
字符串枚举分片

Java高并发底层原理与实际应用

并发原起

绝大多数的运算任务都不可能只靠处理器“计算”就能完成，处理器至少要与内存交互，如读取运算数据、存储运算结果等，这个I/O操作是很难消除的（无法仅靠寄存器来完成所有运算任务）。早期计算机中cpu和内存的速度是差不多的，但在现代计算机中，cpu的指令速度远超内存的存取速度,由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存（Cache）来作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理器就无须等待缓慢的内存读写了。

基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是也为计算机系统带来更高的复杂度，因为它引入了一个新的问题：缓存一致性（Cache Coherence）。在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存（MainMemory）。当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致，举例说明变量在多个CPU之间的共享。如果真的发生这种情况，那同步回到主内存时以谁的缓存数据为准呢？为了解决一致性的问题，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议来进行操作，这类协议有MSI、MESI（Illinois Protocol）、MOSI、Synapse、Firefly及Dragon Protocol等。

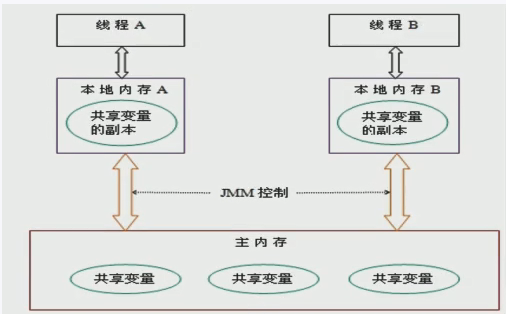
 重新配图

https://www.jianshu.com/p/8a58d8335270

JMM: Java Memory Model

JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM是整个计算机虚拟模型，所以JMM是隶属于JVM的。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。

由于JVM运行程序的实体是线程，而每个线程创建时JVM都会为其创建一个工作内存(也称为：**栈空间**)，工作内存是每个线程的私有数据区域，而Java内存模型中规定所有变量都存储在*主内存*，主内存是共享内存区域，所有线程都可访问，但线程对变量的操作(读取赋值等)必须在栈空间中进行，首先要将变量从主内存拷贝到自己的栈空间，然后对变量进行操作，操作完成再将变量写回主内存，不能直接操作主内存中的变量，各个线程中的工作内存储存着主内存中的变量副本拷贝，因此不同的线程无法访问对方的工作内存,此案成间的通讯(传值) 必须通过主内存来完成,其简要访问过程如下图：



TODO 主内存与堆栈的关系？？？？？？？

**JMM关于同步规定**：

1. 线程解锁前，必须把共享变量的值刷新回主内存。
2. 线程加锁前，必须读取主内存的最新值到自己的工作内存。
3. 加锁解锁是同一把锁。

1 Happen-Before规则【学术讨论】

Java的并发采用的是共享内存模型，使用happen-before规则实现共享变量的同步操作(简称hb规则)。该规则定义了Java多线程操作的有序性和可见性，防止了*编译器重排序* 对程序结果的影响。按照官方的说法：当一个变量被多个线程读取并且至少被一个线程写入时，如果读操作和写操作没有HB关系，则会产生数据竞争问题。要想保证操作B的线程看到操作A的结果(无论A和B是否在一个线程)，那么在A和B之间必须满足HB原则，如果没有，将有可能导致重排序。当缺少HB关系时，就可能出现重排序问题。

**HB有哪些规则**：(http://ifeve.com/java-使用-happen-before-规则实现共享变量的同步操作/)

1. 程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作。
2. 锁定规则：在监视器锁上的解锁操作必须在同一个监视器上的加锁操作之前执行。
3. volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作。
4. 传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C。(*这个规则至关重要，如何熟练的使用传递规则是实现同步的关键*。)
5. 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。
6. 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生。
7. 线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行。
8. 对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始。

volatile

https://www.jianshu.com/p/ccfe24b63d87

synchronized-线程锁

Java Virtual Machine实际应用

Html/Css/Javascript 篇

Servlet/J2EE篇

实用代码篇

* 1. Java代码段

1.1 十六进制和八进制

* 1. Javascript代码段
  2. Mysql代码段

代码样式

|  |  |  |
| --- | --- | --- |
| 方法名 | 返回值 | 功能描述 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |
| --- | --- |
| 方法名 | 功能描述 |
|  |  |
|  |  |