索引

2024年8月7日 20:50

八、索引

8.1 索引的概念

index：目的是为了快速查找数据

8.1.1 常见的索引类型（分类）

1、按照数据结构来的话（B+树索引）

B+树索引的发展过程

查找算法，一开始就是遍历所有数据，后来出现了二分查找（查找的数据量直接减半了）

逐渐出现二叉搜索树（左面的结点小，右面的结点大）（二叉搜索树其实也是利用了二分查找思想），就将需要遍历的数据，变成了按照树的高度去寻找数据，高度是多少，最多就找几次就行，速度提升很大

但是二叉搜索树会出现一个问题，当二叉搜索树不太平衡的时候（全是大于或全是小于的），二叉搜索树就会退化成链表（当然这是一种极端的情况），就算是有几个分叉的结点，那么它的效率也没有很大的提升

因此就发展出了平衡二叉树（AVL树）：任何一个子树的高度差不能大于1，虽然不是绝对的平衡（那也不可能），但也避免了二叉搜索树退化为指针的情况

还有的就是红黑树（一种自平衡的二叉搜索树），相对平衡二叉树没有那么严格，能够提供快速的查找、插入和删除操作。

这样插入删除的代价变大，但是查找更快，数据分布更加稳定，平衡了，这样在查找任意的数据时，时间都是差不多的，不会出现极端情况。

但是，随着数据量的增加，树的高度就越来越高了，由于数据库会访问磁盘，而二叉树，每一个高度，都会有磁盘IO，树的高度变大的话，就会影响效率

这样就发展为了平衡树（可以多叉，每个叉上可以有多个结点），这样就减少了树的高度，也就减少了磁盘IO，同时B树可以将结点大小优化为磁盘块的大小。

后续又发展出了B+树，（它只有叶子结点存储数据，非叶子结点存索引），这样的话，所有的叶子结点就变成了一个有序的结构了，这样的话，我们给所有的叶子结点加个指针，就变成了链表。对于排序和范围查找就非常快速了。

就比如说 找大于3 的数据，B树的话，就得找到3，再去找4，一个一个找后面的数据
而B+树，因为叶子节点是有序的数据，找到3之后，后面所有的数据就都符合范围的数据了

这样的话，就相当于是将链表的优点和树的优点结合到一起了

所以B+树索引是，现在关系型数据库中查找时最常用的也是最有效的索引了，MySQL的引擎 InnoDB使用的也是B+树索引

2、哈希索引

查找非常快（O(1)的时间复杂度），但是由于是散列存储，对于排序和范围查找的效率较低，而一般查找时，大多是要进行排序的。

InnoDB是自适应创建哈希索引的，如果需要的话，它会自动创建哈希索引

3、全文索引

一个表，存的是文章内容，我们在查找时，是需要查文章中的某一段，某一句话，某个词。
倒排索引，把你文章中的关键字进行统计（出现多少次，具体位置），可以进行关键字搜索，这种索引就有很大的限制了，数据量大时，搜索速度很慢，维护成本也高

8.2 使用索引

```
# 创建索引
create index index_sal on emp(sal);
create index index_ename on emp(ename);

# 显示表中的索引
show index from emp;

# 删除索引
drop index index_sal on emp;

# 查看是否使用索引
explain select * from emp where empno = 7788;
desc select * from emp where sal = 3100;
```

explain 个属性的含义

1、id:查询的序列号
2、select_type:查询的类型，主要区别普通查询和联合查询、子查询之类的复杂查询
（1）SIMPLE: 查询中不包含子查询或者UNION
（2）PRIMARY: 查询中包含任何复杂的子部分
（3）SUBQUERY: 作为SELECT 或 WHERE 列表中的子查询

3、table：输出的行所用的表
4、**type**：访问类型
（1）ALL：扫描全表
（2）index：扫描全部索引树
（3）range：扫描部分索引树，索引范围扫描，常见于between、<、>等查询
（4）ref：使用非唯一索引 或 非唯一索引前缀进行的查找
（5）eq_ref：唯一索引扫描，常见于主键或唯一索引扫描
（6）const：单表中最多有一个匹配行，查询起来非常迅速
（7）system：是const类型的特例，表中只有一条数据
（8）NULL：不访问表或者索引，直接就能得到结果

5、**key**：显示MySQL实际决定使用的索引，如果没有索引被选择，是NULL

6、key_len：使用到索引字段的长度，key_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key_len是根据表定义计算而得，不是通过表内检索出的。
7、ref: 显示哪个字段或常数与 key 一起被使用
8、rows:这个数表示 MySQL 要遍历多少数据才能找到，表示 MySQL 根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数，在innodb上可能是不准确的

9、**Extra**：执行情况的说明和描述。包含不适合在其他列中显示但十分重要的额外信息
（1）Using index：表示使用索引，如果只有 Using index，说明他没有查询到数据表。只用索引表就完成了这个查询，这个叫覆盖索引。
（2）Using where：表示条件查询，如果不读取表的所有数据，或不是仅仅通过索引就可以获取所有需要的数据，则会出现 Using where。
（3）Using index condition：索引条件下推(Index Condition Pushdown，ICP)是 MySQL 使用索引的情况的优化。简单来说，在服务器需要扫描表的情况下当没有 ICP 时，存储引擎扫描可以明确地使用索引的条件，将符

合条件的记录返回给服务器。当使用ICP 时，只要条件可以在索引上判断出来，就由存储引擎在索引树上完成判断，再将符合条件的记录返回给服务器。ICP可以减少存储引擎必须访问基本表的次数以及服务器必须访问存储引擎的次数，这是是否使用ICP的最可靠的判断条件

8.3 前缀索引

索引使用开头部分的字符（前缀），可以很大的节约索引的空间，提高索引的效率，（就比如说，abcd和abdc，只需要将前3个字符作为索引值，就可以实现索引）

但是这样就会出现一种情况，（abcd 和 abce，的索引就重复了），这里就是降低了精度，索引的选择性（不重复的索引值➗总行数　得到的比值），唯一索引的选择选择性就是1，1就是最好的索引选择性了，性能也是最高的。

所以在选择前缀索引时，要选择足够长的前缀保证比较高的选择性，同时又不能太长（为了节约空间）

8.4 聚簇索引

就是索引的值 和 数据存到一起

一种存储方式，将数据防止索引的叶子结点，索引和数据在同一个B+树上

但是，你不可能将所有的索引都变成聚簇索引，你创建一个索引就把所有的数据都存一份？这样的话就导致每创建一个索引，数据量就会成倍的增加

这样肯定是不现实的，因此一个表只有一个聚簇索引（主键），其他的索引中，数据存的就是主键，先在其他索引中找到主键，再去聚簇索引中找到完整的数据

在InnoDB 聚簇索引一般是主键，没有主键的话，会找一个唯一非空的索引代替，如果唯一非空也没有，就会隐式定义一个主键作为聚簇索引

优点：

（1）只有一个聚簇索引，其他的索引不用存数据，只需要存主键就行，节省空间
（2）可以把相关的数据保存到一起
（3）访问的速度更快了，索引和数据在同一个结构中

缺点：

（1）插入速度非常依赖插入顺序，如果插入时，影响了聚簇索引树的平衡性，就会可能导致所有的树都需要改变结构
（2）更新聚簇索引的代价也很高（更新之后，也可能会改变树的结构）

（3）插入数据或更新主键时可能面临“页分裂”问题。比如你插入的一个值，这个值的前后页都满了。
这个时候，就需要将某一页分裂，来存储这个值

（4）非聚簇索引需要两次索引查找（但是节省了空间）

如何解决问题上面的问题？

那聚簇索引在插入和更新时的缺点很大，我们就在这两个方面解决就好了嘛

（1）插入时，插入边缘数据，或者是按照顺序插入数据
（2）减少主键的更新，甚至是不让主键更新

我们的使用聚簇索引一般都是 用主键作聚簇索引。

而在定义主键时，我们一般对主键有以下要求
一个表都应该定义主键

主键的值不应该修改
不使用可能会修改值的列作为主键(与业务无关，这通常使用 id 作为主键)

一般还会给他添加自增约束，让他自动增加

主键自动增加，就是在每次插入数据时，在边缘进行插入，不让他随机插入嘛

使用与业务无关的列作为主键，就减少甚至避免了 主键更新的情况

8.5 使用UUID（随机值作为主键）和雪花算法

使用随机值作为主键

尾部热点问题

前面我们说了，一般的聚簇索引是主键，而主键是需要自增的

但是，当数据量很大的时候，一个数据库无法处理全部的数据，就需要将数据表进行分度分表（将表分到多个数据库中，分开处理），（分表就会有多种方法：水平切分（按行），垂直切分（按列），按列分是有上限的，受到字段数量的影响，因此一般都是水平切分，按照行分）

主键自增的话，按照行分表，比如1亿条数据分成100个表，每个表100w条数据，0 - 100w的数据在第一个表中，100w-200w的数据在第二个表。。。，依次类推嘛

这样就会导致一个问题，如果新的数据来了，就会放到最后一个表里，那你写入的压力是没有变小的，所有的压力还是在最后一个表里。而且查找的压力还是大部分在最后一个表中（这个就是尾部热点的问题，一般来说，新创建的数据大概率是更加活跃的）

所以就导致你，最后一个表里，都是新的数据，而且比较活跃，最后一个表的写入、查找压力是很大的

因此就需要使用随机值来作为主键，来平衡压力。使用UUID

雪花算法

我们利用了UUID解决了尾部热点问题，那主键使用随机值又会导致聚簇索引 插入 更新时代价变大。该如何解决这种问题呢?

雪花算法

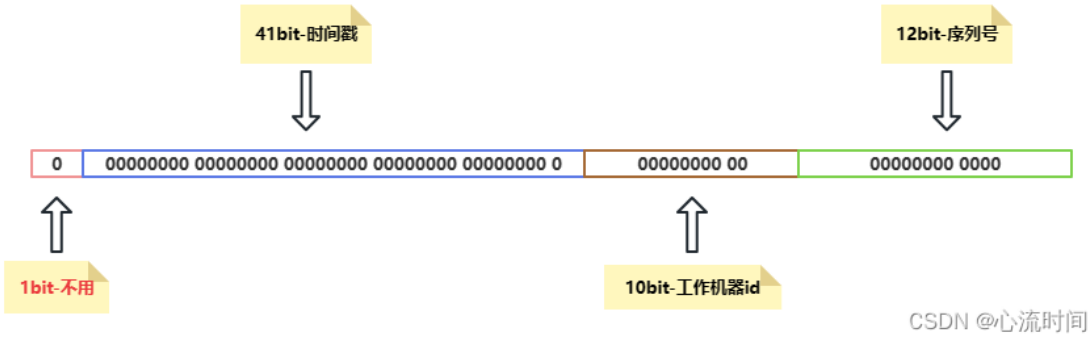
雪花算法的原理就是生成一个的 64 位比特位的 long 类型的唯一 id。

1：最高 1 位是符号位，固定值 0，表示id 是正整数
41：接下来 41 位存储毫秒级时间戳，2^41/(1000606024365)=69，大概可以使用 69 年。

时间戳在不同设备的同一时间上，有可能重复，因此后面还有工作机器和序列号

10：再接下 10 位存储机器码，包括 5 位 datacenterId 和 5 位 workerId。最多可以部署 2^10=1024 台机器。
12：最后 12 位存储序列号。同一毫秒时间戳时，通过这个递增的序列号来区分。即对于同一台机器而言，同一毫秒时间戳下，可以生成 2^12=4096 个不重复 id。

可以将雪花算法作为一个单独的服务进行部署，然后需要全局唯一 id 的系统，请求雪花算法服务获取 id 即可。对于每一个雪花算法服务，需要先指定 10 位的机器码，这个根据自身业务进行设定即可。例如机房号+机器号，机器号+服务号，或者是其他可区别标识的 10 位比特位的整数值都行



8.6 覆盖索引（索引覆盖）

使用聚簇索引之后，其他是索引要查到数据就需要再去查一次聚簇索引（回表）

其他字段在你的非聚簇索引中，已经存在了。不使用聚簇索引（不需要回表），就可以完成查询任务的情况，就是覆盖索引（索引覆盖），只依靠非聚簇索引就能完成查找任务

因为不需要回表，效率提高

8.7 索引失效

我们创建索引，就是为了在查找时更加快速，效率更高

但是，有些数据库语句，就会导致索引失效

我们创建了索引，但是在查找的时候，数据库引擎没有使用索引去查找
因为它判断 使用索引 ，也不能让查找变得更快，这样的情况就是索引失效

因此我们就要避免索引失效，什么情况下会导致索引失效呢？

1、使用or的时候，如果一个是索引，一个不是索引

```
explain select * from emp where sal = 5000 or ename = 'smith';
```

比如在使用where时，判断 工资=5000 或者 名字叫张三 的情况
这个时候，如果工资是有索引的，而名字没有索引

你先按照工资的索引查一遍，如果工资没查到，你还得再去查一遍没有索引的名字。这个时候，就不如直接去查全部的内容了

2、索引存储的时候，是按照前缀存储的，它会先查第一个字母，接着查第二个字母，如果你前缀是模糊的，这样也会导致索引失效

```
# 前缀是模糊的
explain select * from emp where ename like "%建国"; # 前缀模糊 索引失效
explain select * from emp where ename like "张%"; # 后缀模糊 索引有效
```

like查后缀，前缀就模糊了

比如在查找一个人的名字时使用 like 模糊查找 ，查找所有姓张的人，这个时候，前缀是不模糊的，因为索引是从前往后的嘛

那如果你查 所有名字里带 建国 的人，不管它是什么姓，这个时候，你的前缀就是模糊的，就会导致索引失效

3、索引放入表达式 或者函数中，索引也会失效

```
# 索引放入表达式 或者函数中，索引也会失效
explain select * from emp where sal > 3000; # 不会失效
explain select * from emp where sal+1 > 3001; # 索引在表达式，会失效
```

就比如说，查找工资大于3000 和 查找 工资+1 大于 3001，
虽然他们俩表达的都是 工资大于3000的情况，但是 在查找 工资加一 大于3001时就会导致索引失效

确实是没那么智能，加一，他都无法处理，就更别说使用函数了

4、与上面将索引放入表达式或函数差不多，索引进行类型换行，也会导致索引失效

```
# 类型转换也会导致失效
explain select * from emp where ename = 3000; # 字符串转数字（索引类型转换，失效）
explain select * from emp where sal = "3000"; # 字符串转数字（索引没有类型换行，有效）
```

因为在MySQL中，遇到字符串和数字进行比较的时候，是将字符串转换为数字的

比如 在 查一个人的名字时，查找的是 一个数字
查一个人的名字是否等于 3000，这个时候，他就会将名字转换为字符串，也就相当于是使用了强转的函数，会导致索引失效

但是 如果是 查一个人的工资 等于 字符串类型的 3000
他是将 字符串3000 转换为数字 3000 ，没有改变索引的类型，这个时候，索引就不会失效

8.8 最左前缀原则

创建多列的联合索引时，满足最左前缀原则。例如创建(a,b,c)三列的索引，实际上相当于创建了 a、(a,b)、(a,b,c)三个索引。

当不需要考虑排序和分组时，将选择性最高的列放在前面。这时索引的作用只用于优化where 条件的查找，这样设计可以最快过滤需要的行

所以在 创建联合索引时，要注意索引的顺序

MySQL 5.6之后，也支持索引下推 a c也可以

8.9 索引的设计原则

为常作为查询条件的字段建立索引:如果某个字段经常用来做查询条件，那么该字段的查询速度会影响整个表的查询速度。因此，为这样的字段建立索引，可以提高整个表的查询速度。

为经常需要排序、分组和联合操作的字段建立索引:经常需要 ORDER BY、GROUPBY、DISTINCT和 UNION 等操作的字段，排序操作会浪费很多时间。如果为其建立索引，可以有效地避免排序操作。

创建唯一性索引:唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录,

限制索引的数目:每个索引都需要占用用磁盘空间，索引越多，需要的磁盘空间就越大修改表时，对索引的重构和更新很麻烦。

小表不建议索引(如数量级在百万以内):由于数据较小，查询花费的时间可能比遍历索引的时间还要短，索引可能不会产生优化效果。

尽量使用前缀索引:如果索引的值很长，那么查询的速度会受到影响

删除不再使用或者很少使用的索引。

8.10 索引的使用策略

独立的列:索引使用不当会导致索引失效(查询中实际没有使用索引)。如果查询中的列不是独立的，MySQL不会使用索引。
独立的列指查询时索引列不能是表达式的一部分，也不能是函数的参数，这两种情况都会导致索引失效

使用前缀索引:使用前缀索引可以节约索引空间，从而提高索引效率，但是需要平衡索引的选择性

使用联合索引:使用联合索引可以避免回表，实现覆盖索引，可以减少大量 I/O 操作

合适的索引列顺序:创建联合索引时，不同的列顺序会影响索引的性能，通常将选择性高的列放在最前面

合适的主键:最好选择不会修改的列作为主键，不考虑分库分表的情况最好使用自增主键

问题：在大量数据的分页时，效率很低，依靠索引解决

例如在 limit 10000，10（从第1w个开始取，取10个），他不仅仅是取出了10条数据，它内部是取出了1w10条数据，显示了后面的10条。这样就会导致效率很低。

解决方式：

（1）在业务上就避免很大的分页，禁止查询分页很大的数据（在baidu查询的分页中，就禁止显示很多页之后的数据）