教育经历模块

这篇论文是哪个期刊？

这篇论文是发表咋IEEE Transaction On Big Data上，这篇期刊在2023年的中科院sci分区表中是二区

纵向联邦学习和横向联邦学习的区别？

这篇论文的创新点?

基本技能模块

练习sql语句

<https://www.nowcoder.com/exam/oj?page=1&tab=SQL%E7%AF%87&topicId=199&fromPut=pc_kol_wenqlgd>

介绍一下mysql索引？

mysql的索引是一种数据结构，用于加快数据的查询效率，类似于目录，通过索引可以快速定位到数据。（索引定义）

索引是数据引擎层实现的，不同的数据引擎层由不同的索引。对于InnoDB数据引擎，索引由主键索引、单列索引、符合索引、唯一索引。对于MyISAM存储引擎，索引有全文索引。（索引分类）

我们创建索引可以在建表时或建表后创建，通过show index指令可以查看一张表的索引。（创建索引）

mysql索引底层是由B+树数据结构实现的。B+树是多路平衡查找树，由B树发展而来，一个m阶的B树最多有m个孩子阶段即m-1个关键字，在存储时，数据根据关键字的大小选中不同分叉的节点存放。B+树与B树最大的区别在于只有叶子节点存放数据，而非叶子节点只存放关键字和指针。叶子节点之间是有序的，并用指针连接，叶子节点内部也是有序的。（索引的底层原理）

对于InnnoDB存储引擎， 主键索引，关键字和数据存放在一起，非叶子节点存放关键字，叶子节点存放数据。而辅助键索引的非叶子节点存放关键字，但是叶子节点不存放数据，存放对于的主键id，在利用辅助键索引查询数据时，需要查询两次，第一次时通过辅助键索引的B+树找到对应的主键，然后拿着主键去主键索引对应的B+树走一遍。这也被称作回表。

为什么索引要采用 B+ 树，用其他数据结构不可以吗，比如哈希表，B树，红黑树？

用其它数据也可以，在早期，索引的实现也是经过了一次次探索，并不是一来就使用B+树。从索引的定义来看，哈希表是比较符合人们直觉的一种数据结构，哈希表在做单值查询的时候，效率非常快。但是哈希表有两个问题，一是哈希冲突，在空间有限的前提下，哈希函数无论设计得再好，都是无法避免哈希冲突的，索引当数据量变非常大，哈希冲突增多，如果采用链地址法解决哈希冲突，即哈希表的每一个槽位都是一个链条，对于哈希值相同的元素，会放在相同索引位置的链条上。链条变长时，查询退化成O(n)的线性查询效率。第二，哈希表不能做范围查询，对于范围查询，只能遍历所有位置所有链条，然后一个一个判断。后来，人们使用二叉搜索树，BST，BST是一个排序树，使用折半查找的原理，将数据的查询速度降低到O(logn)，但是二叉搜索树有一个缺点，当数据以有序状态一个个插入时，二叉树会变成一条链，极度不平衡，查询效率降为logn，所以人们使用平衡二叉树AVL树，但是AVL树要求的平衡很严格， 虽然查询效率上稳定了，但是构建树的、删除节点后恢复平衡的效率下降，所以人们发明了红黑树，红黑树不遵守严格平衡条件，但是构建和复衡的效率提高了。由于索引和数据是存放在磁盘上的，读取一个树的结点，就会进行一次IO,在程序中，磁盘IO的效率是很低的，索引IO次数越少越好，由于二叉树分叉的限制，即每个节点最多两个分叉，当数据量增大时，树的高度会快速增加，而树的高度增加会导致从根节点到某个子节点的路径变长，路径上的节点变多，IO次数变多，导致查询效率降低，为了解决这个二叉树本质的问题，引入了多路平衡查找树，即B树。但是，由于B的非叶子节点存储了指针、关键字和数据，导致每个节点的分叉也不能达到很大，所以当数据量继续增大时树的高度还是会上升，而B+树解决了这个问题，由于B+树的数据都存放在叶子节点，非叶子节点只存放关键字和指针，所以一个非叶子节点存储的关键字就大大增加，分叉数量变多，只需要较低的树高，就可以存储大量数据，而树地会让磁盘的IO次数减少。所以最后B+树成为了索引的底层结构。

用一个数据说明B+树比B树的存储数量多

InnoDB存储引擎是按页存储的，

假设一页的大小是16KB

主键是int类型，关键字大小是，8Byte

假设指针大小是2Byte

数据大小是1KB

如果按照B树存储，非叶子节点可以存储16 KB / (1KB + 8B + 2B) = 16个元素，三层高的B+树可以存储，16 \* 16 \* 16 = 4096条数据

如果按照B+树存储，非叶子节点可以存储 16 KB / (8B + 2B) = 1600个元素

那么，三层树高，1600的三次方，大概是四十几亿数据。

介绍一下B+树是怎么分裂的？

一开始，只有一个根节点时，数据会按照关键字顺序排列在一个页中，当加入新的数据页变满时，节点会分裂，从按照关键字顺序排列的数据中选择一个关键字，将其提升为一个新的根节点，放到一个新的页中，然后原理的页根据选择的关键字分裂成两个新的页，左边是小于当前根节点关键字的数据，右边是大于等于当前更节点关键字的数据。

对于一个已经有结构的B+树，如果向叶节点插入数据导致分裂条件满足，那么分类会从叶节点开始，接着是其父节点，一层一层地往上分裂，直到每个节点的孩子数不超过B+数的阶数即可。每个节点在分裂时，都会选择节点中间大小元素分裂，将这个元素向上移至父节点，然后检查父节点的孩子节点个数是否大于B+树的阶数，如果大于，那么重复之前的过程，如果不大于则结束。

刚才说到了 B 树，B 树和 B+ 树的主要区别是什么可以讲一讲 B 树的应用场景吗？

主要区别如下:

1. B+非叶子节点只保存关键字和指针，不保存数据
2. B+树数据只保存在叶子节点中
3. B+树叶子节点之间用指针连接

字段加索引，你是否在自己的项目中用过呢？你觉得什么样的字段适合加索引？

1. 使用很多的字段（查询频率），项目中举例说明
2. 区分度较高的字段，因为mysql在选择索引的时候是通过估计扫描行数来，通过基数来估计，也就是索引的区分度，所以区分度越高的字段，mysql选错索引的概率就越小
3. 字段的长度，主要是针对字符类型的索引，因为字符串索引具有前缀索引的功能，太长占空间，太短占区分度不够

mysql怎么创建索引？

|  |  |  |
| --- | --- | --- |
| 索引类型 | 建表时 | 建表后 |
| 主键索引 | create table t(id int primary key);  create table t(id int, primary key(id)); | alter table t add primary key(id); |
| 单值索引 | create table t(a int, index(a));  create table t(a int, key(a));  create table t(a int, index index\_name (a));  create table t(a int, key key\_name (a)); | alter table t add index (a);  alter table t add key (a);  alter table t add index index\_name (a);  alter table t add key key\_name (a);  create index index\_name on t(a); |
| 唯一索引 | create table t(a int unique);  create table t(a int, unique(a));  create table t(a int, unique unique\_name (a));  create table t(a int, unique index index\_name (a));  create table t(a int, unique key key\_name (a)); | alter table t add unique (a);  alter table t add unique unique\_name (a);  alter table t add unique index (a);  alter table t add unique key (a);  alter table t add unique index index\_name (a);  alter table t add unique key key\_name (a);  create unique index index\_name on t(a); |
| 复合索引 | create table t(a int, b int, index(a,b));  create table t(a int, b int, key(a,b));  create table t(a int, b int, index index \_name (a,b));  create table t(a int, b int, key key\_name (a,b)); | alter table t add index (a,b);  alter table t add key (a,b);  alter table t add index index\_name (a,b);  alter table t add key key\_name (a,b);  create index index\_name on t(a,b); |

那你觉得，字段加了索引，查找的时候一定会走索引吗？

不一定，以下是不会走索引的情况：

1. 使用like语句查询字段为字符串类型的索引，如果通配符%放在最前面，无法使用索引。
2. 使用复合索引查询时，如果where条件里面的条件不满足最左前缀原则且者经sql优化器优化后仍不满足最左前缀原则， 也无法使用索引。
3. where条件中使用or，如果两边的字段有一个没有索引，也无法使用索引。必须or两边的条件的列都是索引才会用到索引。

刚才你的索引失效的例子，都是因为人为没有写好 sql 导致的，那如果排除人为的情况，sql 正确书写，那就一定会走索引吗？

不一定，因为走不走索引，走哪个索引是由优化器权衡抉择的，选择索引是优化器的工作。

优化器选择索引的目的，是为了找到一个最优的执行方案，并用最小的代价去执行语句。影响执行代价的因素有很多，比如扫描行数、回表次数、是否使用临时表、是否排序等等。

对于扫描行数越少，意味着访问磁盘的次数越少，消耗的CPU资源也就越少。mysql在开始真正执行语句前，并不能精确地知道满足条件的记录有多少条，只能更加统计信息来估算记录数。

扫描行数估计不准确会导致mysql优化器选错索引。而导致扫描行数不正确有两种因素：1 统计信息不准确；2 其他优化器误判

对于第一种，我们可以使用analyze table解决

对于第二种，我们可以使用force index强制让mysql选择某个索引执行。

这个统计信息就是索引的区分度。

一个索引上，不同值越多，区分度就越好。而一个索引上不同值的个数被称为基数。也就是说，基数（索引不同值的个数）越大，区分度就越好。可以通过show index from T来查看基数大小，cardinality就是基数含义。基数的计算是通过采用统计方法，计算某些页的不同值的平均值再乘以总页数



如果我想要强制走某个索引，能实现吗？可以怎么做？

1. 采用force index强行选择一个索引，如果我们判断应该走某个索引，但是优化器没有走，可以使用force index让优化器强制选择该索引。force index写好之后如果索引名称改变，也需要force index里面的索引名，所以不够灵活。
2. 修改sql语句，引导mysql使用我们期望的索引
3. 新建一个更合适的索引给优化器选择，或者删除选错的索引

如何一条 sql 执行的很慢，我们可以怎么来排查原因？

这需要分为两种情况讨论，偶尔执行很慢还是一直都执行很慢。如果是偶尔执行很慢，我认为sql语句没有问题，可能有其它原因，比如redo log 写满了，此时系统会停止所有更新操作，把redo log的checkpoint往前推，在把check point往前推的过程需要将内存中的脏页数据flush到磁盘，在这个过程中，会发现一条正常执行的sql语句会执行的特别慢。还有就是等待其它并发事务占用的锁，这个锁可能是MDL锁，有可能是行锁。

对于一直都执行很慢的sql语句，考虑是不是没有使用索引，如果使用的条件所在列没有索引，可以向表添加索引。如果所在列有索引，但mysql没有选择，这也分几种情况，首先是sql语句写的不够准确，比如对于索引字段为字符串类型的，使用like，如果通配符%在最前面，是无法使用索引的。对于复合索引，如果不满足最左前缀原则，也不会走索引。对于or，最有两边必须同时包含索引。

当然如果上面的情况都排处，即sql语句写的完全正确，逻辑上优化器应该选择某个索引，但是mysql优化器还是选错了，可能是由于统计信息不正确导致mysql扫描行数预估错误，此时可以使用analyze table来。

对于其它原因导致优化器选错索引，可以使用force index方法。

刚才说到了模糊匹配失效，为什么使用模糊匹配会失效，你能给我解释一下底层原理吗？

这种情况一般发生在使用like关键字时，将通配符%放在了开头，导致索引用不上，只能走全表扫描，效率很低。

原理是InnoDB存储引擎底层是一棵B+树，对于字符串类型来说，每个叶子节点的数据都按照字符串首字母排序，如果首字母相同，再比较第二个字母，依次类推。我们再进行模糊匹配时，如果把%放在开头，会导致最左N个字母是不确定的。无法根据索引的有序性定位到某个索引，只能进行全表扫描。

对于like语句，通配符%在最前面，会导致模糊匹配失效，无法使用索引

因为mysql底层的数据结构是B+树，在执行搜索算法时，会从根节点出发，依次比较关键字的大小，然后选择对应的分叉，往下一层搜索，直到找到符合条件的节点。如果通配符在最前面，就无法比较两个字符串关键字的大小。从而导致索引失效，只能走全表扫描。因为字符串比较大小是通过最左边的字母比较，然后依次向右比较，而通配符会匹配所有的字符，通配符放在最前，索引的有序性就利用不上了。

（1）说明什么时候会失效。（2）索引底层结构（3）字符串比较大小的方式（4）%导致最左N个字母失效（5）最终只能走全表扫描

你是怎么理解事务的，说一下你的理解？

事务的四大特性了解吗，介绍一下解释一下？

ACID

事务有好几种隔离级别，介绍一下常见的有哪些，主要解决了哪些问题呢，又各自存在哪些问题呢？

|  |  |  |  |
| --- | --- | --- | --- |
|  | 脏读 | 不可重复读 | 幻读 |
| 读未提交 | 1 | 1 | 1 |
| 读提交 | 0 | 1 | 1 |
| 可重复读 | 0 | 0 | 1 |
| 串行化 | 0 | 0 | 0 |

快照了解吗，介绍一下？

1. mysql数据库的快照：数据库在某个时间点的状态记录，可以保留数据库在特定时间的一致性状态，通过快照可以对数据进行恢复、备份等功能。
2. 快照读中的快照：并不是对整个数据库的快照。它的本质是利用undo log+MVCC实现的一致性视图。

在可重复读隔离级别下，当一个事务开始时，会生成唯一的事务id，当事务需要更新数据，mysql并不会直接修改原始数据。

mysql快照是数据库在某一个时刻的状态，与备份不同，备份时将数据库的数据全部复制一份到另外的地方。快照就像照相以向，记录此刻数据库的状态。

mysql 快照读中的快照，并不是对整个数据库的快照，它本质使用undo log +MVCC实现的一致性视图。undo log记录了某行记录的不同版本，每个事务都有一个唯一的事务id，当事务更新数据时，并不是在原来的行数据上之间更新，而是记录一次新的undo log，将其回滚指针指向上一个数据的版本，每个行数据都有trx\_id、roll pointer字段分别表示产生该版本记录的事务和回滚指针，直到事务提交，才会把更新记录的正在的数据行上。undo log可以用于事务回滚，也可以结合MVCC一致性视图实现读提交或者可重复读隔离级别。

Mysql 是怎么保证原子性的？

原子性含义：一个事务中的所有操作，要么同时成功，要么同时失败。

对于一个事务，更新数据并不会直接在原来的行数据上进行修改，由于undo log的存在，事务每一次修改数据都会生成一个新的数据版本，这个行数据由trx\_id和roll pointer，分别时事务id，表示当前修改数据的事务，回滚指针，指向上一个数据的版本。只有当事务提交时，才会将修改写入内存页中的行数据。如果事务发生异常回滚，会根据undo log版本链回滚到事务开始前的状态，保证了事务的一致性。

Mysql 怎么保证持久性的？

持久性：每一个提交的事务都应该持久化，保存到磁盘，它对数据的改变应该是永久性的，即使mysql异常重启，也可以恢复。

InnoDB主要通过redo log保证事务的持久性，其中还有WAL技术

WAL指的是日志先写技术，它的关键是mysql的操作并不是立即写入磁盘，而是先写入日志。然后在合适的时间写入磁盘。具体来说，当有一条 数据需要更新时，InnoDB存储引擎会先把数据写入redo log里，并更新内存里面的数据，这个时候整个记录的更新就算完成了。后续，InnoDB引擎会在适当的时候，由后台线程，将缓存在Buffer Pool里面的数据刷到磁盘上去，这个时候往往时磁盘空闲的时候。

有了redo log，当系统发生崩溃时，即使脏页数据还没来得及持久化。但redo log已经持久化了。mysql可以根据redo log记录的内容进行数据恢复，恢复到崩溃前的状态，这就是crash-safe

快照读在提交读和可重复读级别下有什么区别？

在读提交的隔离级别下，事务中每次读都会创建一个视图

在可重复度的隔离级别下，事务一开始就会创建一个视图，在后面的select中会复用这个视图。

Mysql 怎么保证隔离性的？

|  |  |  |
| --- | --- | --- |
|  | 锁 | MVCC |
| 读未提交 | 无 | 无 |
| 读提交 |  |  |
|  |  |  |

项目经历模块

科研项目的步骤？