**MYSQL笔记**

1. **索引**

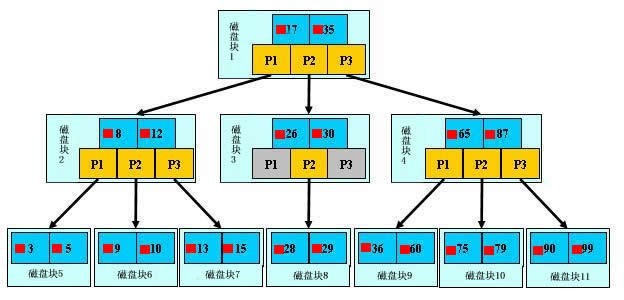
**索引**（Index）是帮助MySQL高效获取数据的数据结构。提取句子主干，就可以得到索引的本质：索引是数据结构。

**1.1 索引的存储分类**

**B-Tree索引**

最常见的索引类型，大部分引擎都支持B树索引。

###详解b+树



如上图，是一颗b+树，关于b+树的定义可以参见B+树，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。

###b+树的查找过程

如图所示，如果要查找数据项29，那么首先会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时间因为非常短（相比磁盘的IO）可以忽略不计，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

###b+树性质

1.通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。

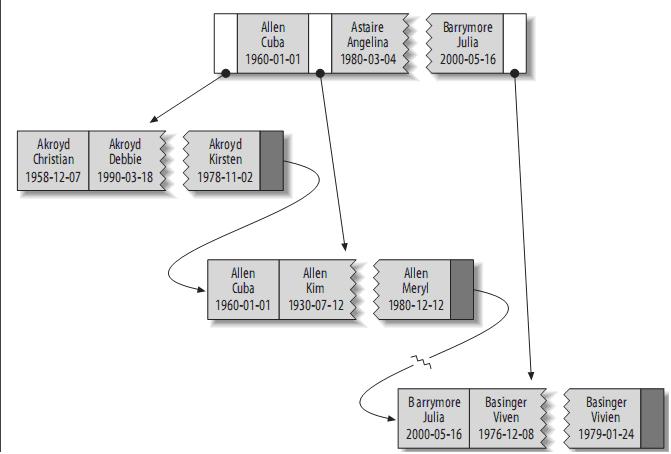
2.当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最后得到检索的数据；但当(20,F)这样的没有name的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是F的数据了， 这个是非常重要的性质，即索引的最左匹配特性。

**例子：**

假设有如下一个表：

|  |
| --- |
| CREATE TABLE People (     last\_name varchar(50)    not null,     first\_name varchar(50)    not null,     dob        date           not null,     gender     enum('m', 'f') not null,     key(last\_name, first\_name, dob)  ); |

 其索引包含表中每一行的last\_name、first\_name和dob列。其结构大致如下：



  索引存储的值按索引列中的顺序排列。可以利用B-Tree索引进行全关键字、关键字范围和关键字前缀查询，当然，如果想使用索引，你必须保证按索引的最左边前缀(leftmost prefix of the index)来进行查询。

(1)匹配全值(Match the full value)：对索引中的所有列都指定具体的值。例如，上图中索引可以帮助你查找出生于1960-01-01的Cuba Allen。

(2)匹配最左前缀(Match a leftmost prefix)：你可以利用索引查找last name为Allen的人，仅仅使用索引中的第1列。

(3)匹配列前缀(Match a column prefix)：例如，你可以利用索引查找last name以J开始的人，这仅仅使用索引中的第1列。

(4)匹配值的范围查询(Match a range of values)：可以利用索引查找last name在Allen和Barrymore之间的人，仅仅使用索引中第1列。

(5)匹配部分精确而其它部分进行范围匹配(Match one part exactly and match a range on another part)：可以利用索引查找last name为Allen，而first name以字母K开始的人。

(6)仅对索引进行查询(Index-only queries)：如果查询的列都位于索引中，则不需要读取元组的值。

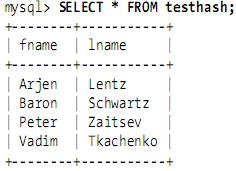
由于B-树中的节点都是顺序存储的，所以可以利用索引进行查找(找某些值)，也可以对查询结果进行ORDER BY。当然，使用B-tree索引有以下一些限制：

(1) 查询必须从索引的最左边的列开始。关于这点已经提了很多遍了。例如你不能利用索引查找在某一天出生的人。

(2) 不能跳过某一索引列。例如，你不能利用索引查找last name为Smith且出生于某一天的人。

(3) 存储引擎不能使用索引中范围条件右边的列。例如，如果你的查询语句为WHERE last\_name="Smith" AND first\_name LIKE 'J%' AND dob='1976-12-23'，则该查询只会使用索引中的前两列，因为LIKE是范围查询。

**Hash索引**

MySQL中，只有Memory存储引擎显示支持hash索引，是Memory表的默认索引类型，尽管Memory表也可以使用B-Tree索引。Memory存储引擎支持非唯一hash索引，这在数据库领域是罕见的，如果多个值有相同的hash code，索引把它们的行指针用链表保存到同一个hash表项中。  
假设创建如下一个表：  
CREATE TABLE testhash (  
   fname VARCHAR(50) NOT NULL,  
   lname VARCHAR(50) NOT NULL,  
   KEY USING HASH(fname)  
) ENGINE=MEMORY;  
包含的数据如下：  


假设索引使用hash函数f( )，如下：

|  |
| --- |
| f('Arjen') = 2323  f('Baron') = 7437  f('Peter') = 8784  f('Vadim') = 2458 |

此时，索引的结构大概如下：



 Slots是有序的，但是记录不是有序的。当你执行mysql> SELECT lname FROM testhash WHERE fname='Peter'; MySQL会计算’Peter’的hash值，然后通过它来查询索引的行指针。因为f('Peter') = 8784，MySQL会在索引中查找8784，得到指向记录3的指针。因为索引自己仅仅存储很短的值，所以，索引非常紧凑。Hash值不取决于列的数据类型，一个TINYINT列的索引与一个长字符串列的索引一样大。

Hash索引有以下一些限制：

(1)由于索引仅包含hash code和记录指针，所以，MySQL不能通过使用索引避免读取记录。但是访问内存中的记录是非常迅速的，不会对性造成太大的影响。

(2)不能使用hash索引排序。

(3)Hash索引不支持键的部分匹配，因为是通过整个索引值来计算hash值的。 (4)Hash索引只支持等值比较，例如使用=，IN( )和<=>。对于WHERE price>100并不能加速查询。

**空间(R-Tree)索引**

MyISAM支持空间索引，主要用于地理空间数据类型，例如GEOMETRY。

**全文(Full-text)索引**

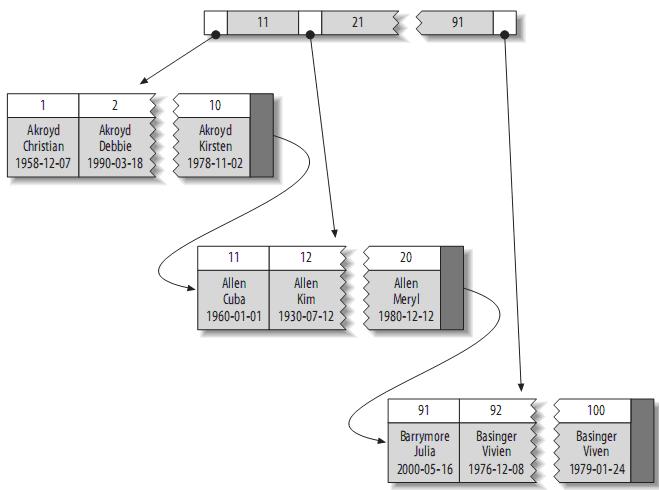
全文索引是MyISAM的一个特殊索引类型，主要用于全文检索。



**1.2 高性能的索引策略**

**聚簇索引(Clustered Indexes)**

聚簇索引保证关键字的值相近的元组存储的物理位置也相同（所以字符串类型不宜建立聚簇索引，特别是随机字符串，会使得系统进行大量的移动操作），且一个表只能有一个聚簇索引。因为由存储引擎实现索引，所以，并不是所有的引擎都支持聚簇索引。目前，只有solidDB和InnoDB支持。

聚簇索引的结构大致如下： 

  注：叶子页面包含完整的元组，而内节点页面仅包含索引的列(索引的列为整型)。一些DBMS允许用户指定聚簇索引，但是MySQL的存储引擎到目前为止都不支持。InnoDB对主键建立聚簇索引。如果你不指定主键，InnoDB会用一个具有唯一且非空值的索引来代替。如果不存在这样的索引，InnoDB会定义一个隐藏的主键，然后对其建立聚簇索引。一般来说，DBMS都会以聚簇索引的形式来存储实际的数据，它是其它二级索引的基础。

聚簇索引表最大限度地提高了I/O密集型应用的性能，但它也有以下几个限制：

1）插入速度严重依赖于插入顺序，按照主键的顺序插入是最快的方式，否则将会出现页分裂，严重影响性能。因此，对于InnoDB表，我们一般都会定义一个自增的ID列为主键。

2）更新主键的代价很高，因为将会导致被更新的行移动。因此，对于InnoDB表，我们一般定义主键为不可更新。

3）二级索引访问需要两次索引查找，第一次找到主键值，第二次根据主键值找到行数据。

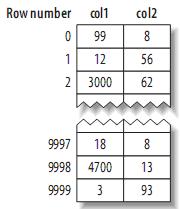
二级索引的叶节点存储的是主键值，而不是行指针，这是为了减少当出现行移动或数据页分裂时二级索引的维护工作，但会让二级索引占用更多的空间。

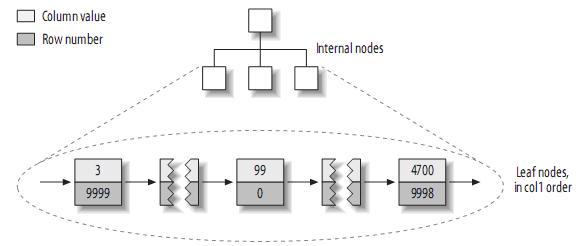
**InnoDB和MyISAM的数据布局的比较**

为了更加理解聚簇索引和非聚簇索引，或者primary索引和second索引(MyISAM不支持聚簇索引)，来比较一下InnoDB和MyISAM的数据布局，对于如下表：

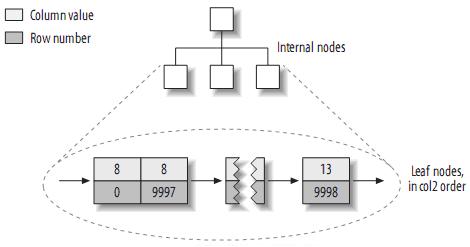
|  |
| --- |
| CREATE TABLE layout\_test (     col1 int NOT NULL,     col2 int NOT NULL,     PRIMARY KEY(col1),     KEY(col2)  ); |

 假设主键的值位于1---10,000之间，且按随机顺序插入，然后用OPTIMIZE TABLE进行优化。col2随机赋予1---100之间的值，所以会存在许多重复的值。  
(1) MyISAM的数据布局

其布局十分简单，MyISAM按照插入的顺序在磁盘上存储数据，如下：  


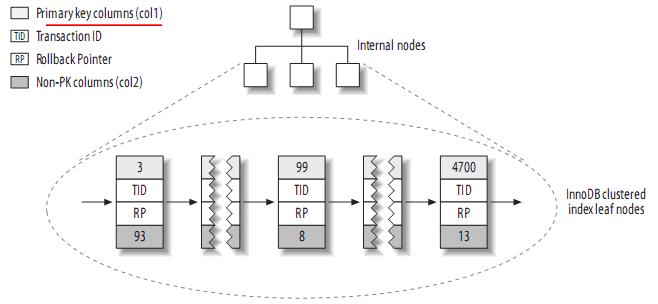
 注：左边为行号(row number)，从0开始。因为元组的大小固定，所以MyISAM可以很容易的从表的开始位置找到某一字节的位置。  
据些建立的primary key的索引结构大致如下：  


 注：MyISAM不支持聚簇索引，索引中每一个叶子节点仅仅包含行号(row number)，且叶子节点按照col1的顺序存储。

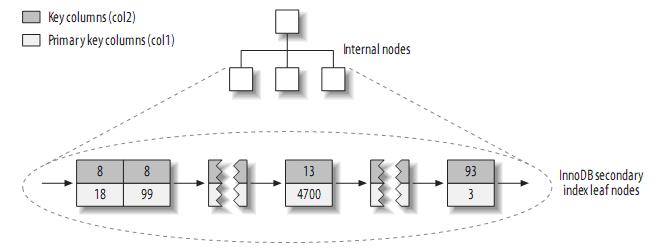
来看看col2的索引结构： 

 实际上，在MyISAM中，primary key和其它索引没有什么区别。Primary key仅仅只是一个叫做PRIMARY的唯一，非空的索引而已。

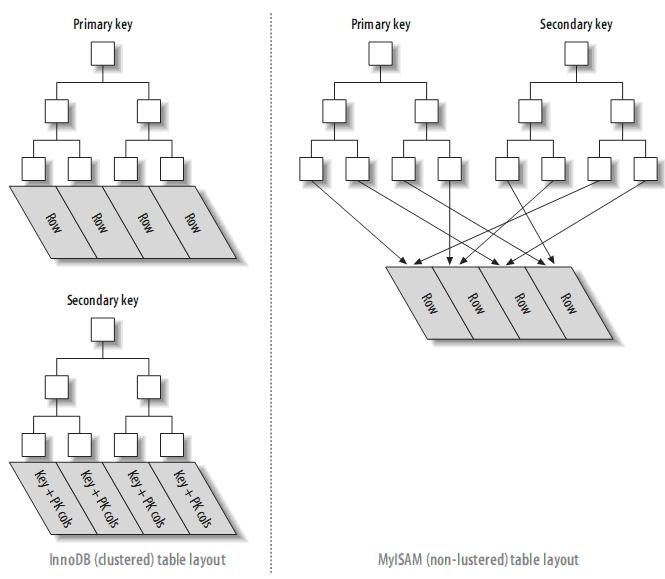
(2) InnoDB的数据布局

InnoDB按聚簇索引的形式存储数据，所以它的数据布局有着很大的不同。它存储表的结构大致如下： 

 注：聚簇索引中的每个叶子节点包含primary key的值，事务ID和回滚指针(rollback pointer)——用于事务和MVCC，和余下的列(如col2)。

相对于MyISAM，二级索引与聚簇索引有很大的不同。InnoDB的二级索引的叶子包含primary key的值，而不是行指针(row pointers)，这减小了移动数据或者数据页面分裂时维护二级索引的开销，因为InnoDB不需要更新索引的行指针。其结构大致如下：  
  


 聚簇索引和非聚簇索引表的对比：



对于普通的堆组织表来说（右图），表数据和索引是分成存储的，主键索引和二级索引存储上没有任何区别。

而对于聚簇索引表来说（左图），表数据是和主键一起存储的，主键索引的叶结点存储行数据，二级索引的叶结点存储行的主键值。

**聚集索引和非聚集索引的区别：**

其实，我们的汉语字典的正文本身就是一个聚集索引。比如，我们要查“安”字，就会很自然地翻开字典的前几页，因为“安”的拼音是“an”，而按照拼音排序汉字的字典是以英文字母“a”开头并以“z”结尾的，那么“安”字就自然地排在字典的前部。如果您翻完了所有以“a”开头的部分仍然找不到这个字，那么就说明您的字典中没有这个字；同样的，如果查“张”字，那您也会将您的字典翻到最后部分，因为“张”的拼音是“zhang”。也就是说，字典的正文部分本身就是一个目录，您不需要再去查其他目录来找到您需要找的内容。

我们把这种正文内容本身就是一种按照一定规则排列的目录称为“聚集索引”。

如果您认识某个字，您可以快速地从自动中查到这个字。但您也可能会遇到您不认识的字，不知道它的发音，这时候，您就不能按照刚才的方法找到您要查的字，而需要去根据“偏旁部首”查到您要找的字，然后根据这个字后的页码直接翻到某页来找到您要找的字。但您结合“部首目录”和“检字表”而查到的字的排序并不是真正的正文的排序方法，比如您查“张”字，我们可以看到在查部首之后的检字表中“张”的页码是672页，检字表中“张”的上面是“驰”字，但页码却是63页，“张”的下面是“弩”字，页面是390页。很显然，这些字并不是真正的分别位于“张”字的上下方，现在您看到的连续的“驰、张、弩”三字实际上就是他们在非聚集索引中的排序，是字典正文中的字在非聚集索引中的映射。我们可以通过这种方式来找到您所需要的字，但它需要两个过程，先找到目录中的结果，然后再翻到您所需要的页码。

我们把这种目录纯粹是目录，正文纯粹是正文的排序方式称为“非聚集索引”。

**覆盖索引(Covering Indexes)**

如果索引包含满足查询的所有数据，就称为覆盖索引。覆盖索引是一种非常强大的工具，能大大提高查询性能。只需要读取索引而不用读取数据有以下一些优点：  
(1)索引项通常比记录要小，所以MySQL访问更少的数据；  
(2)索引都按值的大小顺序存储，相对于随机访问记录，需要更少的I/O；  
(3)大多数据引擎能更好的缓存索引。比如MyISAM只缓存索引。  
(4)覆盖索引对于InnoDB表尤其有用，因为InnoDB使用聚集索引组织数据，如果二级索引中包含查询所需的数据，就不再需要在聚集索引中查找了。

覆盖索引不能是任何索引，只有B-TREE索引存储相应的值。而且不同的存储引擎实现覆盖索引的方式都不同，并不是所有存储引擎都支持覆盖索引(Memory和Falcon就不支持)。

对于索引覆盖查询(index-covered query)，使用EXPLAIN时，可以在Extra一列中看到“Using index”。例如，在sakila的inventory表中，有一个组合索引(store\_id,film\_id)，对于只需要访问这两列的查询，MySQL就可以使用索引，如下：

|  |
| --- |
| mysql> EXPLAIN SELECT store\_id, film\_id FROM sakila.inventory\G  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*             id: 1   select\_type: SIMPLE          table: inventory           type: index  possible\_keys: NULL            key: idx\_store\_id\_film\_id        key\_len: 3            ref: NULL           rows: 5007          Extra: Using index  1 row in set (0.17 sec) |

在大多数引擎中，只有当查询语句所访问的列是索引的一部分时，索引才会覆盖。但是，InnoDB不限于此，InnoDB的二级索引在叶子节点中存储了primary key的值。因此，sakila.actor表使用InnoDB，而且对于是last\_name上有索引，所以，索引能覆盖那些访问actor\_id的查询，如：

|  |
| --- |
| mysql> EXPLAIN SELECT actor\_id, last\_name      -> FROM sakila.actor WHERE last\_name = 'HOPPER'\G  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*             id: 1   select\_type: SIMPLE          table: actor           type: ref  possible\_keys: idx\_actor\_last\_name            key: idx\_actor\_last\_name        key\_len: 137            ref: const           rows: 2          Extra: Using where; Using index |

**利用索引进行排序**

MySQL中，有两种方式生成有序结果集：一是使用filesort，二是按索引顺序扫描。利用索引进行排序操作是非常快的，而且可以利用同一索引同时进行查找和排序操作。当索引的顺序与ORDER BY中的列顺序相同且所有的列是同一方向(全部升序或者全部降序)时，可以使用索引来排序。如果查询是连接多个表，仅当ORDER BY中的所有列都是第一个表的列时才会使用索引。其它情况都会使用filesort。

|  |
| --- |
| create table actor(  actor\_id int unsigned NOT NULL AUTO\_INCREMENT,  name      varchar(16) NOT NULL DEFAULT '',  password        varchar(16) NOT NULL DEFAULT '',  PRIMARY KEY(actor\_id),   KEY     (name)  ) ENGINE=InnoDB  insert into actor(name,password) values('cat01','1234567');  insert into actor(name,password) values('cat02','1234567');  insert into actor(name,password) values('ddddd','1234567');  insert into actor(name,password) values('aaaaa','1234567'); |

|  |
| --- |
| mysql> explain select actor\_id from actor order by actor\_id \G  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*             id: 1   select\_type: SIMPLE          table: actor           type: index  possible\_keys: NULL            key: PRIMARY        key\_len: 4            ref: NULL           rows: 4          Extra: Using index  1 row in set (0.00 sec)    mysql> explain select actor\_id from actor order by password \G  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*             id: 1   select\_type: SIMPLE          table: actor           type: ALL  possible\_keys: NULL            key: NULL        key\_len: NULL            ref: NULL           rows: 4          Extra: Using filesort  1 row in set (0.00 sec)    mysql> explain select actor\_id from actor order by name \G  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*             id: 1   select\_type: SIMPLE          table: actor           type: index  possible\_keys: NULL            key: name        key\_len: 18            ref: NULL           rows: 4          Extra: Using index  1 row in set (0.00 sec) |

当MySQL不能使用索引进行排序时，就会利用自己的排序算法(快速排序算法)在内存(sort buffer)中对数据进行排序，如果内存装载不下，它会将磁盘上的数据进行分块，再对各个数据块进行排序，然后将各个块合并成有序的结果集（实际上就是外排序）。对于filesort，MySQL有两种排序算法。  
(1)两遍扫描算法(Two passes)

实现方式是先将须要排序的字段和可以直接定位到相关行数据的指针信息取出，然后在设定的内存（通过参数sort\_buffer\_size设定）中进行排序，完成排序之后再次通过行指针信息取出所需的Columns。

注：该算法是4.1之前采用的算法，它需要两次访问数据，尤其是第二次读取操作会导致大量的随机I/O操作。另一方面，内存开销较小。  
(2)一次扫描算法(single pass)

该算法一次性将所需的Columns全部取出，在内存中排序后直接将结果输出。注：从 MySQL 4.1 版本开始使用该算法。它减少了I/O的次数，效率较高，但是内存开销也较大。如果我们将并不需要的Columns也取出来，就会极大地浪费排序过程所需要的内存。在 MySQL 4.1 之后的版本中，可以通过设置 max\_length\_for\_sort\_data 参数来控制 MySQL 选择第一种排序算法还是第二种。当取出的所有大字段总大小大于 max\_length\_for\_sort\_data 的设置时，MySQL 就会选择使用第一种排序算法，反之，则会选择第二种。为了尽可能地提高排序性能，我们自然更希望使用第二种排序算法，所以在 Query 中仅仅取出需要的 Columns 是非常有必要的。

当对连接操作进行排序时，如果ORDER BY仅仅引用第一个表的列，MySQL对该表进行filesort操作，然后进行连接处理，此时，EXPLAIN输出“Using filesort”；否则，MySQL必须将查询的结果集生成一个临时表，在连接完成之后进行filesort操作，此时，EXPLAIN输出“Using temporary;Using filesort”。

**索引与加锁**

索引对于InnoDB非常重要，因为它可以让查询锁更少的元组。这点十分重要，因为MySQL 5.0中，InnoDB直到事务提交时才会解锁。有两个方面的原因：首先，即使InnoDB行级锁的开销非常高效，内存开销也较小，但不管怎么样，还是存在开销。其次，对不需要的元组的加锁，会增加锁的开销，降低并发性。

InnoDB仅对需要访问的元组加锁，而索引能够减少InnoDB访问的元组数。但是，只有在存储引擎层过滤掉那些不需要的数据才能达到这种目的。一旦索引不允许InnoDB那样做（即达不到过滤的目的），MySQL服务器只能对InnoDB返回的数据进行WHERE操作，此时，已经无法避免对那些元组加锁了：InnoDB已经锁住那些元组，服务器无法解锁了。

来看个例子：

|  |
| --- |
| create table actor(  actor\_id int unsigned NOT NULL AUTO\_INCREMENT,  name      varchar(16) NOT NULL DEFAULT '',  password        varchar(16) NOT NULL DEFAULT '',  PRIMARY KEY(actor\_id),   KEY     (name)  ) ENGINE=InnoDB  insert into actor(name,password) values('cat01','1234567');  insert into actor(name,password) values('cat02','1234567');  insert into actor(name,password) values('ddddd','1234567');  insert into actor(name,password) values('aaaaa','1234567'); |

|  |
| --- |
| SET AUTOCOMMIT=0;  BEGIN;  SELECT actor\_id FROM actor WHERE actor\_id < 4  AND actor\_id <> 1 FOR UPDATE; |

 该查询仅仅返回2---3的数据，实际已经对1---3的数据加上排它锁了。InnoDB锁住元组1是因为MySQL的查询计划仅使用索引进行范围查询（而没有进行过滤操作，WHERE中第二个条件已经无法使用索引了）：

|  |
| --- |
| mysql> EXPLAIN SELECT actor\_id FROM test.actor      -> WHERE actor\_id < 4 AND actor\_id <> 1 FOR UPDATE \G  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*             id: 1   select\_type: SIMPLE          table: actor           type: index  possible\_keys: PRIMARY            key: PRIMARY        key\_len: 4            ref: NULL           rows: 4          Extra: Using where; Using index  1 row in set (0.00 sec)    mysql> |

 表明存储引擎从索引的起始处开始，获取所有的行，直到actor\_id<4为假，服务器无法告诉InnoDB去掉元组1。  
为了证明row 1已经被锁住，我们另外建一个连接，执行如下操作：

|  |
| --- |
| SET AUTOCOMMIT=0;  BEGIN;  SELECT actor\_id FROM actor WHERE actor\_id = 1 FOR UPDATE; |

 该查询会被挂起，直到第一个连接的事务提交释放锁时，才会执行（这种行为对于基于语句的复制(statement-based replication)是必要的）。

如上所示，当使用索引时，InnoDB会锁住它不需要的元组。更糟糕的是，如果查询不能使用索引，MySQL会进行全表扫描，并锁住每一个元组，不管是否真正需要。

**什么样的sql不走索引**

**要尽量避免这些不走索引的sql**

SELECT `sname` FROM `stu` WHERE `age`+10=30;-- 不会使用索引,因为所有索引列参与了计算

SELECT `sname` FROM `stu` WHERE LEFT(`date`,4) <1990; -- 不会使用索引,因为使用了函数运算,原理与上面相同

SELECT \* FROM `houdunwang` WHERE `uname` LIKE'后盾%' -- 走索引

SELECT \* FROM `houdunwang` WHERE `uname` LIKE "%后盾%" -- 不走索引

-- 正则表达式不使用索引,这应该很好理解,所以为什么在SQL中很难看到regexp关键字的原因

-- 字符串与数字比较不使用索引;

CREATE TABLE `a` (`a` char(10));

EXPLAIN SELECT \* FROM `a` WHERE `a`="1" -- 走索引

EXPLAIN SELECT \* FROM `a` WHERE `a`=1 -- 不走索引

对于多列索引，不是使用的第一部分，则不会使用索引

谓词上的索引列条件上使用了<>，NOT IN操作符 不走索引

MySQL查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

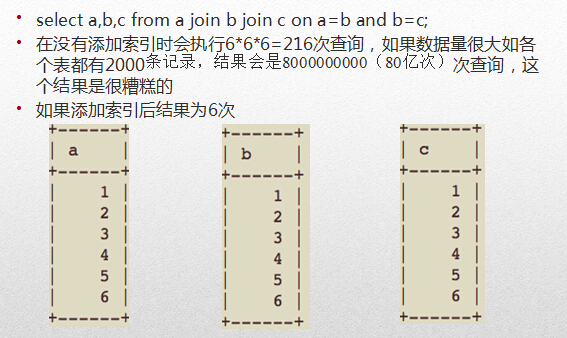
select \* from dept where dname='xxx' or loc='xx' or deptno=45 --如果条件中有or,即使其中有条件带索引也不会使用。换言之,就是要求使用的所有字段,都必须建立索引, 我们建议大家尽量避免使用or 关键字

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

-- 如果mysql估计使用全表扫描要比使用索引快,则不使用索引

最后总结一下，MySQL只对一下操作符才使用索引：<,<=,=,>,>=,between,in,以及某些时候的like(不以通配符%或\_开头的情形)。多表关联时的索引效率

* SELECT `sname` FROM `stu` WHERE LEFT(`date`,4) <1990; — 不会使用索引,因为使用了函数运算,原理与上面相同
* SELECT \* FROM `houdunwang` WHERE `uname` LIKE’后盾%’ — 走索引
* SELECT \* FROM `houdunwang` WHERE `uname` LIKE “%后盾%” — 不走索引



从上图可以看出,所有表的type为all,表示全表索引;也就是6 6 6,共遍历查询了216次;

除第一张表示全表索引(必须的,要以此关联其他表),其余的为range(索引区间获得),也就是6+1+1+1,共遍历查询9次即可;

所以我们建议在多表join的时候尽量少join几张表,因为一不小心就是一个笛卡尔乘积的恐怖扫描,另外,我们还建议尽量使用left join,以少关联多.因为使用join 的话,第一张表是必须的全扫描的,以少关联多就可以减少这个扫描次数.

**索引的弊端**

不要盲目的创建索引,只为查询操作频繁的列创建索引,创建索引会使查询操作变得更加快速,但是会降低增加、删除、更新操作的速度,因为执行这些操作的同时会对索引文件进行重新排序或更新;

但是,在互联网应用中,查询的语句远远大于DML的语句,甚至可以占到80%~90%,所以也不要太在意,只是在大数据导入时,可以先删除索引,再批量插入数据,最后再添加索引。

**二、事务**

**事务**是访问并更新数据库中各中数据项的一个程序执行单元。在事务中的操作，要么都修改，要么都不执行，这就是事务的目的，也是事务模型区别于文件系统的重要特征之一。

**2.1 事务的特性**

**原子性 (atomicity)**

原子性是指一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。只有使事务中所有的数据库操作执行都成功，才算整个事务成功。只要有一个sql语句执行失败，那么在这个事务中已经执行的sql语句都必须撤销，数据库状态应该退回到执行事务前的状态。

**一致性(consistency)**

事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。在事务开始之前和结束之后，数据库的完整性约束没有被破坏。

**隔离性 (isolation)**

一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰，这些通过锁来实现。

**持久性(durability)**

持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障（比如说宕机等）不应该对其有任何影响。

**2.2 事务的实现**

       隔离性 (isolation)通过锁来实现，其他3个原子性atomicity，一致性consistency，持久性durability通过数据库的redo和undo来完成。

**2.2.1  redo**

  在Innodb存储引擎中，事务日志通过redo日至文件和Innodb存储引擎的日志缓冲(InnoDBLog Buffer)来实现。当开始一个事务时候，会纪录该事务的一个LSN（Log Sequence Number日志序列号）；当事务执行时候，会往InnoDB存储引擎的日志缓冲里插入事务日志；当事务提交时，必须将InnoDB存储引擎的日志缓冲写入disk（默认的实现，即innodb\_flush\_log\_at\_trx\_commit=1）。也就是在写数据之前，需要先写日志，这种方式称为预写日志方式WAL（Write-Ahead Logging）。

  InnoDB存储引擎通过预写日志的方式来保证事务的完整性。这意味着disk上存储的数据页和内存缓冲池中的页是不同步的，对于内存缓冲池中页的修改，先是写入redo日志文件，然后再写入磁盘，因此是一种异步的方式，可以通过show engine innodb status来观察当前disk和log的差距。

**2.2.2 undo**

undo的记录正好与redo的相反，insert变成delete，update变成相反的update，redo放在redo file里面。而undo放在一个内部的一个特殊segment上面，存储与共享表空间内（ibdata1或者ibdata2中）。参考：[http://blog.csdn.net/mchdba/article/details/8664943](http://blog.csdn.net/mchdba/article/details/8664943" \t "_blank)

**undo不是物理恢复，是逻辑恢复**，因为它是通过执行相反的dml语句来实现的。而且不会回收因为insert和upate而新增加的page页的，undo页的回收是通过masterthread线程来实现的。

## 多事务运行时的并发问题

**1．丢失更新**：两个事务同时更新一行数据，最后一个事务的更新会覆盖掉第一个事务的更新，从而导致第一个事务更新的数据丢失，这是由于没有加锁造成的；

**2. 脏读**：脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这3时，另外一个事务也访问这个数据，然后使用了这个数据。

1. **不可重复读**：是指在一个事务内，多次读同一数据。在这个事务还没有结束

时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。

1. **幻读**: 是指当事务不是独立执行时发生的一种现象，例如第一个事务对一

个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样。

**不可重复读的重点是修改**:  
 同样的条件, 你读取过的数据,再次读取出来发现值不一样了  
**幻读的重点在于新增或者删除** 同样的条件, 第 1 次和第 2 次读出来的记录数不一样

## 数据库锁

## MySQL中锁的种类很多，有常见的表锁和行锁，也有新加入的Metadata Lock等等,表锁是对一整张表加锁，虽然可分为读锁和写锁，但毕竟是锁住整张表，会导致并发能力下降，一般是做ddl处理时使用。

## 行锁则是锁住数据行，这种加锁方法比较复杂，但是由于只锁住有限的数据，对于其它数据不加限制，所以并发能力强，MySQL一般都是用行锁来处理并发事务。这里主要讨论的也就是行锁。

|  |  |  |
| --- | --- | --- |
| **资源上已放置的锁** | **第二个事务读操作** | **第二个事务更新操作** |
| 无 | 立即获取共享锁 | 立即获取独占锁 |
| 共享锁 | 立即获取共享锁 | 等待第一个事务解除共享锁 |
| 独占锁 | 等待第一个事务解除独占锁 | 等待第一个事务解除独占锁 |

### 共享锁（read 读锁）

用于读数据操作，他是非独占的，允许其他事务同时读取其锁定的资源，但不允许其他事务更新他。

1. 加锁条件：当一个事务执行select语句时。
2. 解锁条件：默认情况下，数据读取后，数据库系统立即释放共享锁。
3. 兼容性：放置共享锁后还可放置共享锁和更新锁。
4. 并发性：具有良好的并发性能。当多个事务读取相同的数据时，每个事务都会获得一把共享锁，因此可以同时读锁定的数据。

### 独占锁（write 写锁 或排他锁）

也叫排他锁，使用与修改数据的场合。他锁定的资源，其他事务不能进行读写。

1. 加锁条件：当一个事务执行insert,update,delete时，数据库系统会自动对被操作的数据使用独占锁。如果该数据已有其他锁存在，则不能放置独占锁。
2. 解锁条件：事务结束。
3. 兼容性：不和其他锁兼容。
4. 并发性：并发性差，只允许有一个事务访问锁定数据。其他事务需等待，直到当前事务结束。

### 更新锁

更新操作的初始阶段用来锁定可以能要被修改资源的锁。更新锁可避免使用共享锁造成的死锁现象。

1. 加锁条件：当执行update时，数据库系统会先为事务分配一个更新锁。
2. 解锁条件：当读取数据完毕，执行更新操作时，更新锁升级为独占锁。
3. 兼容性：与共享锁兼容。一个资源可同时放置更新锁和共享锁，但只能放置一把更新锁。
4. 并发性：允许多个事务同时读锁定的资源，但不允许其他事务修改。

**不同锁的优缺点及选择**

**行级锁的优点及选择 ：**

1 ）在很多线程请求不同记录时减少冲突锁。

2 ）事务回滚时减少改变数据。

3 ）使长时间对单独的一行记录加锁成为可能。

**行级锁的缺点 ：**

1 ）比页级锁和表级锁消耗更多的内存。

2 ）当在大量表中使用时，比页级锁和表级锁更慢，因为他需要请求更多的所资源。

3 ）当需要频繁对大部分数据做 GROUP BY 操作或者需要频繁扫描整个表时，就明显的比其它锁更糟糕。

4 ）使用更高层的锁的话，就能更方便的支持各种不同的类型应用程序，因为这种锁的开销比行级锁小多了。

5 ）可以用应用程序级锁来代替行级锁，例如 MySQL 中的 GET\_LOCK() 和 RELEASE\_LOCK() 。但它们是劝告锁（原文： These are advisory locks ），因此只能用于安全可信的应用程序中。

6 ）对于 InnoDB 和 BDB 表， MySQL 只有在指定用 LOCK TABLES 锁表时才使用表级锁。在这两种表中，建议最好不要使用 LOCK TABLES ，因为 InnoDB 自动采用行级锁， BDB 用页级锁来保证事务的隔离。

**表锁的优点及选择：**

1 ）很多操作都是读表。

2 ）在严格条件的索引上读取和更新，当更新或者删除可以用单独的索引来读取得到时： UPDATE tbl\_name SET column=value WHERE unique\_key\_col=key\_value;DELETE FROM tbl\_name WHERE unique\_key\_col=key\_value;

3 ） SELECT 和 INSERT 语句并发的执行，但是只有很少的 UPDATE 和 DELETE 语句。

4 ）很多的扫描表和对全表的 GROUP BY 操作，但是没有任何写表。

**表锁的缺点：**

1 ）一个客户端提交了一个需要长时间运行的 SELECT 操作。

2 ）其他客户端对同一个表提交了 UPDATE 操作，这个客户端就要等到 SELECT 完成了才能开始执行。

3 ）其他客户端也对同一个表提交了 SELECT 请求。由于 UPDATE 的优先级高于 SELECT ，所以 SELECT 就会先等到 UPDATE 完成了之后才开始执行，它也在等待第一个 SELECT 操作。

## 如何防止死锁

1. 合理安排表访问顺序；
2. 使用短事务（包含尽可能少的能在短时间内完成操作的事务）；
   * 将事务分解为多个小事务，让后分别执行。这样可保证每个小事务的快速完成，减少对数据资源锁定的时间；
   * 减少执行事务过程中等待需要更新的数据。
3. 如果对数据要求不高，可以允许脏读。脏读不需要对数据资源加锁，可以避免锁冲突；
4. 如果可能，错开多个事务访问相同数据资源的时间，防止锁冲突；
5. 使用尽可能低的事务隔离级别。

一般来说，要出现死锁问题需要满足以下条件：

1. 互斥条件：一个资源每次只能被一个线程使用。

2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

3. 不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。

4. 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

**死锁解决方案**

死锁是由四个必要条件导致的，所以一般来说，只要破坏这四个必要条件中的一个条件，死锁情况就应该不会发生。

1. 如果想要打破互斥条件，我们需要允许进程同时访问某些资源，这种方法受制于实际场景，不太容易实现条件；
2. 打破不可抢占条件，这样需要允许进程强行从占有者那里夺取某些资源，或者简单一点理解，占有资源的进程不能再申请占有其他资源，必须释放手上的资源之后才能发起申请，这个其实也很难找到适用场景；
3. 进程在运行前申请得到所有的资源，否则该进程不能进入准备执行状态。这个方法看似有点用处，但是它的缺点是可能导致资源利用率和进程并发性降低；
4. 避免出现资源申请环路，即对资源事先分类编号，按号分配。这种方式可以有效提高资源的利用率和系统吞吐量，但是增加了系统开销，增大了进程对资源的占用时间。

如果我们在死锁检查时发现了死锁情况，那么就要努力消除死锁，使系统从死锁状态中恢复过来。消除死锁的几种方式：

1. 最简单、最常用的方法就是进行系统的重新启动，不过这种方法代价很大，它意味着在这之前所有的进程已经完成的计算工作都将付之东流，包括参与死锁的那些进程，以及未参与死锁的进程；

2. 撤消进程，剥夺资源。终止参与死锁的进程，收回它们占有的资源，从而解除死锁。这时又分两种情况：一次性撤消参与死锁的全部进程，剥夺全部资源；或者逐步撤消参与死锁的进程，逐步收回死锁进程占有的资源。一般来说，选择逐步撤消的进程时要按照一定的原则进行，目的是撤消那些代价最小的进程，比如按进程的优先级确定进程的代价；考虑进程运行时的代价和与此进程相关的外部作业的代价等因素；

3. 进程回退策略，即让参与死锁的进程回退到没有发生死锁前某一点处，并由此点处继续执行，以求再次执行时不再发生死锁。虽然这是个较理想的办法，但是操作起来系统开销极大，要有堆栈这样的机构记录进程的每一步变化，以便今后的回退，有时这是无法做到的。

## 数据库事务隔离级别

1. Serializalble: 串行化。一个事务在执行过程完全看不到其他事务对数据库所做的更新。当两个事务同时访问相同数据时，第一个事务必须等第二个事务完成后才能访问。
2. Repeatable Read: 可重复读。MySQL的默认事务隔离级别。事务在执行过程中可以看到其他事务已提交的新插入记录，但不能看到其他事务已提交的对已有记录的更新。
3. Read Committed: 读已提交数据。事务在执行过程中可以看到其他事务已提交的新插入记录，也可看到其他事务已提交的对已有记录的更新。
4. Read Uncommited: 读未提交数据。事务在执行过程既可以看到其他事务没有提交的新插入数据，也可看到其他事务已经提交的对已有记录的更新。

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读（Dirty Read） | 不可重复读（NonRepeatable Read  ） | 幻读（Phantom Read） |
| 未提交读（Read uncommitted） | 可能 | 可能 | 可能 |
| 已提交读（Read committed） | 不可能 | 可能 | 可能 |
| 可重复读（Repeatable read） | 不可能 | 不可能 | 可能 |
| 可串行化（Serializable ） | 不可能 | 不可能 | 不可能 |

其中**Serializalble**的隔离级别最高，但并发性能最差。而**Read Uncommited**的隔离级别最低，但并发性能最好。隔离级别越高，越能保证数据的完整性和一致性，但并发性能则会降低。对于多数程序，可以优先考虑把数据库系统的隔离级别设置为Read Committed，它可以避免脏读，而且具有较好的并发性能。

**数据库隔离级别及其实现原理**

### READ\_UNCOMMITED 的原理:

* 事务对当前被读取的数据不加锁；
* 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加 **行级共享锁**，直到事务结束才释放。

**表现：**

* 事务1读取某行记录时，事务2也能对这行记录进行读取、更新；当事务2对该记录进行更新时，事务1再次读取该记录，能读到事务2对该记录的修改版本，即使该修改尚未被提交。
* 事务1更新某行记录时，事务2不能对这行记录做更新，直到事务1结束。

### READ\_COMMITED 的原理:

* 事务对当前被读取的数据加 **行级共享锁（当读到时才加锁）**，一旦读完该行，立即释放该行级共享锁；
* 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加 **行级排他锁**，直到事务结束才释放。

**表现：**

* 事务1读取某行记录时，事务2也能对这行记录进行读取、更新；当事务2对该记录进行更新时，事务1再次读取该记录，读到的只能是事务2对其更新前的版本，要不就是事务2提交后的版本。
* 事务1更新某行记录时，事务2不能对这行记录做更新，直到事务1结束。

### REPEATABLE READ 的原理:

* 事务在读取某数据的瞬间（就是开始读取的瞬间），必须先对其加 **行级共享锁**，直到事务结束才释放；
* 事务在更新某数据的瞬间（就是发生更新的瞬间），必须先对其加 **行级排他锁**，直到事务结束才释放。

**表现：**

* 事务1读取某行记录时，事务2也能对这行记录进行读取、更新；当事务2对该记录进行更新时，事务1再次读取该记录，读到的仍然是第一次读取的那个版本。
* 事务1更新某行记录时，事务2不能对这行记录做更新，直到事务1结束。

### SERIALIZABLE 的原理:

* 事务在读取数据时，必须先对其加 **表级共享锁** ，直到事务结束才释放；
* 事务在更新数据时，必须先对其加 **表级排他锁** ，直到事务结束才释放。

**表现：**

* 事务1正在读取A表中的记录时，则事务2也能读取A表，但不能对A表做更新、新增、删除，直到事务1结束。
* 事务1正在更新A表中的记录时，则事务2不能读取A表的任意记录，更不可能对A表做更新、新增、删除，直到事务1结束。