数据库

2017年8月21日 9:31

1. **有哪些锁，各自的区别及应用场景**

数据库中的锁是网络数据库中的一个非常重要的概念，它主要用于多用户环境下保证数据库完整性和一致性。各种大型数据库所采用的锁的基本理论是一致的，但在具体实现上各有差别。目前，大多数数据库管理系统都或多或少具有自我调节、自我管理的功能，因此很多用户实际上不 清楚锁的理论和所用数据库中锁的具体实现。在数据库中加锁时，除了可以对不同的资源加锁，还可以使用不同程度的加锁方式，即锁有多种模式，SQL Server中锁模式包括：

1）共享锁

SQL Server中，共享锁用于所有的只读数据操作。共享锁是非独占的，允许多个并发事务读取其锁定的资源。默认情况下，数据被读取后，SQL Server立即释放共享锁。例如，执行查询“SELECT * FROM my_table”时，首先锁定第一页，读取之后，释放对第一页的锁定，然后锁定第二页。这样，就允许在读操作过程中，修改未被锁定的第一页。但是，事务 隔离级别连接选项设置和SELECT语句中的锁定设置都可以改变SQL Server的这种默认设置。例如，“ SELECT * FROM my_table HOLDLOCK”就要求在整个查询过程中，保持对表的锁定，直到查询完成才释放锁定。

2）修改锁

修 改锁在修改操作的初始化阶段用来锁定可能要被修改的资源，这样可以避免使用共享锁造成的死锁现象。因为使用共享锁时，修改数据的操作分为两步，首先获得一 个共享锁，读取数据，然后将共享锁升级为独占锁，然后再执行修改操作。这样如果同时有两个或多个事务同时对一个事务申请了共享锁，在修改数据的时候，这些 事务都要将共享锁升级为独占锁。这时，这些事务都不会释放共享锁而是一直等待对方释放，这样就造成了死锁。如果一个数据在修改前直接申请修改锁，在数据修 改的时候再升级为独占锁，就可以避免死锁。修改锁与共享锁是兼容的，也就是说一个资源用共享锁锁定后，允许再用修改锁锁定。

3）独占锁

独占锁是为修改数据而保留的。它所锁定的资源，其他事务不能读取也不能修改。独占锁不能和其他锁兼容。

4）结构锁

结构锁分为结构修改锁（Sch-M）和结构稳定锁（Sch-S）。执行表定义语言操作时，SQL Server采用Sch-M锁，编译查询时，SQL Server采用Sch-S锁。

5）意向锁

意 向锁说明SQL Server有在资源的低层获得共享锁或独占锁的意向。例如，表级的共享意向锁说明事务意图将独占锁释放到表中的页或者行。意向锁又可以分为共享意向锁、独占意向锁和共享式独占意向锁。共享意向锁说明事务意图在共享意向锁所锁定的低层资源上放置共享锁来读取数据。独占意向锁说明事务意图在共享意向锁所锁定 的低层资源上放置独占锁来修改数据。共享式独占锁说明事务允许其他事务使用共享锁来读取顶层资源，并意图在该资源低层上放置独占锁。

2. **数据库的四大特性**

(1) 原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

(2) 一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

(3) 隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务T1和T2，在事务T1看来，T2要么在T1开始之前就已经结束，要么在T1结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

(4) 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用JDBC操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

3. **数据库的隔离级别**

SQL标准定义了4类隔离级别，包括了一些具体规则，用来限定事务内外的哪些改变是可见的，哪些是不可见的。低级别的隔离级一般支持更高的并发处理，并拥有更低的系统开销。

Read Uncommitted（读取未提交内容）

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单

定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读（ Nonrepeatable Read ），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。

Repeatable Read（可重读）

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（ Phantom Read ）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（ MVCC，Multiversion Concurrency Control ）机制解决了该问题。

Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

这四种隔离级别采取不同的锁类型来实现，若读取的是同一个数据的话，就容易发生问题。例如：

脏读(Drity Read)：某个事务已更新一份数据，另一个事务在此时读取了同一份数据，由于某些原因，前一个RollBack了操作，则后一个事务所读取的数据就会是不正确的。

不可重复读(Non-repeatable read):在一个事务的两次查询之中数据不一致，这可能是两次查询过程中间插入了一个事务更新的原有的数据。

幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致，例如有一个事务查询了几列(Row)数据，而另一个事务却在此时插入了新的几列数据，先前的事务在接下来的查询中，就会发现有几列数据是它先前所没有的。

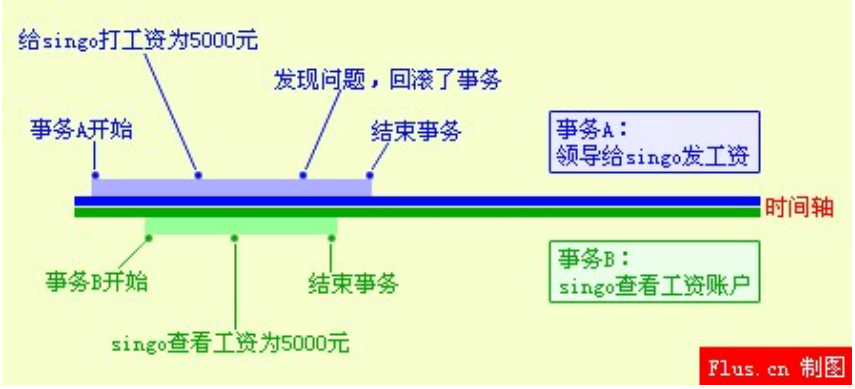
在MySQL中，实现了这四种隔离级别，分别有可能产生问题如下所示：

隔离级别	脏读	不可重复读	幻读
读未提交（Read uncommitted）	V	V	V
读已提交（Read committed）	X	V	V
可重复读（Repeatable read）	X	X	V
可串行化（Serializable）	X	X	X

来自 <http://xm-king.iteye.com/blog/770721>

Read uncommitted 读未提交

公司发工资了，领导把5000元打到singo的账号上，但是该事务并未提交，而singo正好去查看账户，发现工资已经到账，是5000元整，非常高兴。可是不幸的是，领导发现发给singo的工资金额不对，是2000元，于是迅速回滚了事务，修改金额后，将事务提交，最后singo实际的工资只有2000元，singo空欢喜一场。



出现上述情况，即我们所说的脏读，两个并发的事务，“事务A：领导给singo发工资”、“事务B：singo查询工资账户”，事务B读取了事务A尚未提交的数据。

当隔离级别设置为Read uncommitted时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

Read committed 读提交

singo拿着工资卡去消费，系统读取到卡里确实有2000元，而此时她的老婆也正好在网上转账，把singo工资卡的2000元转到另一账户，并在singo之前提交了事务，当singo扣款时，系统检查到singo的工资卡已经没有钱，扣款失败，singo十分纳闷，明明卡里有钱，为何.....

出现上述情况，即我们所说的不可重复读，两个并发的事务，“事务A：singo消费”、“事务B：singo的老婆网上转账”，事务A事先读取了数据，事务B紧接了更新了数据，并提交了事务，而事务A再次读取该数据时，数据已经发生了改变。

当隔离级别设置为Read committed时，避免了脏读，但是可能会造成不可重复读。

大多数数据库的默认级别就是Read committed，比如Sql Server，**Oracle**。如何解决不可重复读这一问题，请看下一个隔离级别。

Repeatable read 重复读

当隔离级别设置为Repeatable read时，可以避免不可重复读。当singo拿着工资卡去消费时，一旦系统开始读取工资卡信息（即事务开始），singo的老婆就不可能对该记录进行修改，也就是singo的老婆不能在此时转账。

虽然Repeatable read避免了不可重复读，但还有可能出现幻读。

singo的老婆工作在银行部门，她时常通过银行内部系统查看singo的信用卡消费记录。有一天，她正在查询到singo当月信用卡的总消费金额（select sum(amount) from transaction where month = 本月）为80元，而singo此时正好在外面胡吃海塞后在收银台买单，消费1000元，即新增了一条1000元的消费记录（insert transaction ... ），并提交了事务，随后singo的老婆将singo当月信用卡消费的明细打印到A4纸上，却发现消费总额为1080元，singo的老婆很诧异，以为出现了幻觉，幻读就这样产生了。

注：**MySQL**的默认隔离级别就是Repeatable read。

Serializable 序列化

Serializable是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

来自 <http://blog.csdn.net/fg2006/article/details/6937413>

4. **MySQL数据库中默认的隔离级别为Repeatable read (可重复读)。**
5. **Mysql两种存储引擎区别，在什么情况下使用**

MyISAM：

1. 不支持事务，但是每次查询都是原子的；
2. 支持表级锁，即每次操作是对整个表加锁；
3. 存储表的总行数；
4. 一个MYISAM表有三个文件：索引文件、表结构文件、数据文件；
5. 采用菲聚集索引，索引文件的数据域存储指向数据文件的指针。辅索引与主索引基本一致，但是辅索引不用保证唯一性。

InnoDb：

1. 支持ACID的事务，支持事务的四种隔离级别；
2. 支持行级锁及外键约束：因此可以支持写并发；
3. 不存储总行数；
4. 一个InnoDb引擎存储在一个文件空间（共享表空间，表大小不受操作系统控制，一个表可能分布在多个文件里），也有可能为多个（设置为独立表空，表大小受操作系统文件大小限制，一般为2G），受操作系统文件大小的限制；
5. 主键索引采用聚集索引（索引的数据域存储数据文件本身），辅索引的数据域存储主键的值；因此从辅索引查找数据，需要先通过辅索引找到主键值，再访问辅索引；最好使用自增主键，防止插入数据时，为维持B+树结构，文件的大调整。

来自 <http://www.cnblogs.com/wangdake-qq/p/7358322.html>

主要区别：

1、MyIASM是非事务安全的，而InnoDB是事务安全的

2、MyIASM锁的粒度是表级的，而InnoDB支持行级锁

3、MyIASM支持全文类型索引，而InnoDB不支持全文索引

4、MyIASM相对简单，效率上要优于InnoDB，小型应用可以考虑使用MyIASM

5、MyIASM表保存成文件形式，跨平台使用更加方便

应用场景：

1、MyIASM管理非事务表，提供高速存储和检索以及全文搜索能力，如果再应用中执行大量select操作，应该选择MyIASM 2、InnoDB用于事务处理，具有ACID事务支持等特性，如果在应用中执行大量insert和update操作，应该选择InnoDB

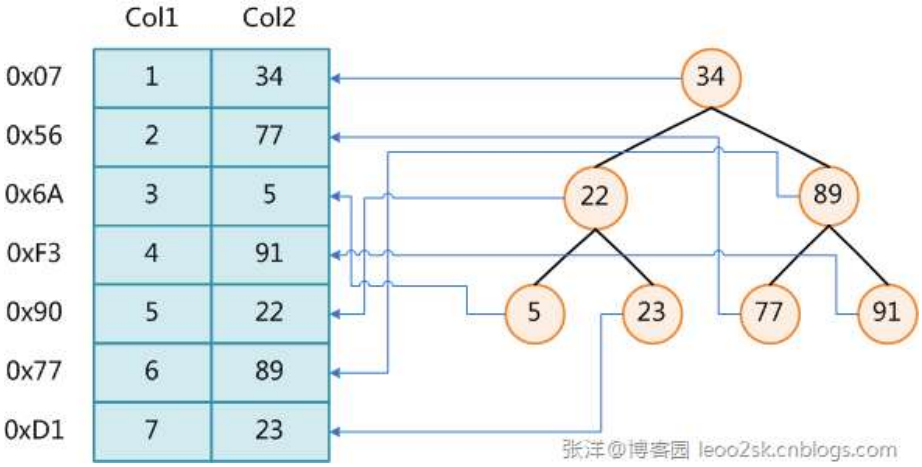
来自 <http://www.2cto.com/database/201703/616547.html>

? 6. 索引的实现，hash和B树的不同，优缺点，应用常见

数据库索引，是数据库管理系统中一个排序的**数据结构**，以协助快速查询、更新数据库表中数据。**索引的实现通常使用B树及其变种B+树**。

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。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



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$O(\log_2 n)$的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。**一般来说，应该在这些列上创建索引：**在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需

要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。**一般来说，不应该创建索引的的这些列具有下列特点：**
第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。
第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。
第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在数据库设计器中创建三种索引：**唯一索引、主键索引和聚集索引**。
唯一索引
唯一索引是不允许其中任何两行具有相同索引值的索引。
当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在employee表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。
主键索引
数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。
在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。
聚集索引
在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。
如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的**局部性原理**：**当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。**

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

B-/+Tree索引的性能分析

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O($\log_d N$)。

一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

综上所述，用B-Tree作为索引结构效率是非常高的。

应该花时间学习B-树和B+树数据结构

==========

1）B树

B树中每个节点包含了键值和键值对于的数据对象存放地址指针，所以成功搜索一个对象可以不用到达树的叶节点。

成功搜索包括节点内搜索和沿某一路径的搜索，成功搜索时间取决于关键码所在的层次以及节点内关键码的数量。

在B树中查找给定关键字的方法是：首先把根结点取来，在根结点所包含的关键字K1,...,kj查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查的关键字在某个Ki或Ki+1之间，于是取Pi所指的下一层索引节点块继续查找，直到找到，或指针Pi为空时查找失败。

2）B+树

B+树非叶节点中存放的关键码并不指示数据对象的地址指针，非也节点只是索引部分。所有的叶节点在同一层上，包含了全部关键码和相应数据对象的存放地址指针，且叶节点按关键码从小到大顺序链接。如果实际数据对象按加入的顺序存储而不是按关键码次数存储的话，叶节点的索引必须是稠密索引，若实际数据存储按关键码次序存放的话，叶节点索引时稀疏索引。

B+树有2个头指针，一个是树的根节点，一个是最小关键码的叶节点。

所以 B+树有两种搜索方法：

一种是按叶节点自己拉起的链表顺序搜索。

一种是从根节点开始搜索，和B树类似，不过如果非叶节点的关键码等于给定值，搜索并不停止，而是继续沿右指针，一直查到叶节点上的关键码。所以无论搜索是否成功，都将走完树的所有层。

B+ 树中，数据对象的插入和删除仅在叶节点上进行。

这两种处理索引的数据结构的不同之处：

a，B树中同一键值不会出现多次，并且它有可能出现在叶结点，也有可能出现在非叶结点中。而B+树的键一定会出现在叶结点中，并且有可能在非叶结点中也有可能重复出现，以维持B+树的平衡。

b，因为B树键位置不定，且在整个树结构中只出现一次，虽然可以节省存储空间，但使得在插入、删除操作复杂度明显增加。B+树相比来说是一种较好的折中。

c，B树的查询效率与键在树中的位置有关，最大时间复杂度与B+树相同(在叶结点的时候)，最小时间复杂度为1(在根结点的时候)。而B+树的时候复杂度对某建成的树是固定的。

来自 <http://blog.csdn.net/kennyrose/article/details/7532032>

在MySQL里常用的索引数据结构有B+树索引和哈希索引两种，我们来看下这两种索引数据结构的区别及其不同的应用建议。

二者区别

备注：先说下，**在MySQL文档里，实际上是把B+树索引写成了BTREE**，例如像下面这样的写法：

```
CREATE TABLE t(
aid int unsigned not null auto_increment,
userid int unsigned not null default 0,
username varchar(20) not null default '',
detail varchar(255) not null default '',
primary key(aid),
unique key(uid) USING BTREE,
key (username(12)) USING BTREE — 此处 uname 列只创建了最左12个字符长度的部分索引
)engine=InnoDB;
```

一个经典的**B+树索引数据结构**见下图：

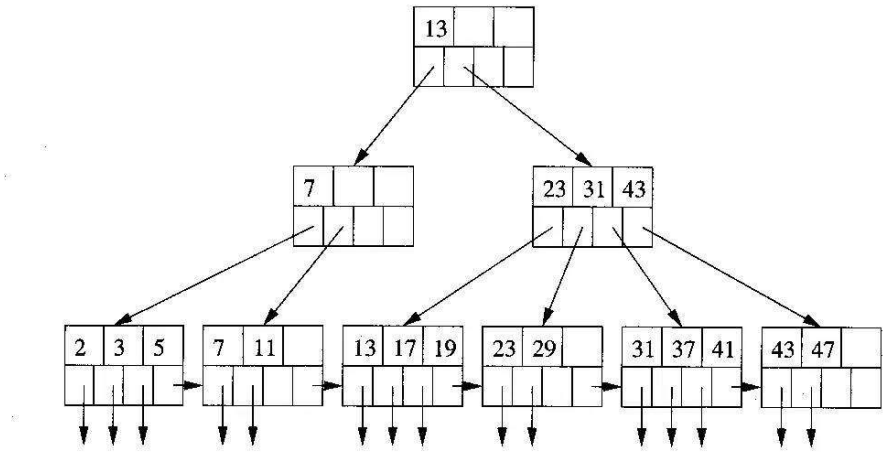


Figure 13.23: A B-tree

(图片源自网络)

B+树是一个平衡的多叉树，从根节点到每个叶子节点的高度差值不超过1，而且同层级的节点间有指针相互链接。

在B+树上的常规检索，从根节点到叶子节点的搜索效率基本相当，不会出现大幅波动，而且基于索引的顺序扫描时，也可以利用双向指针快速左右移动，效率非常高。

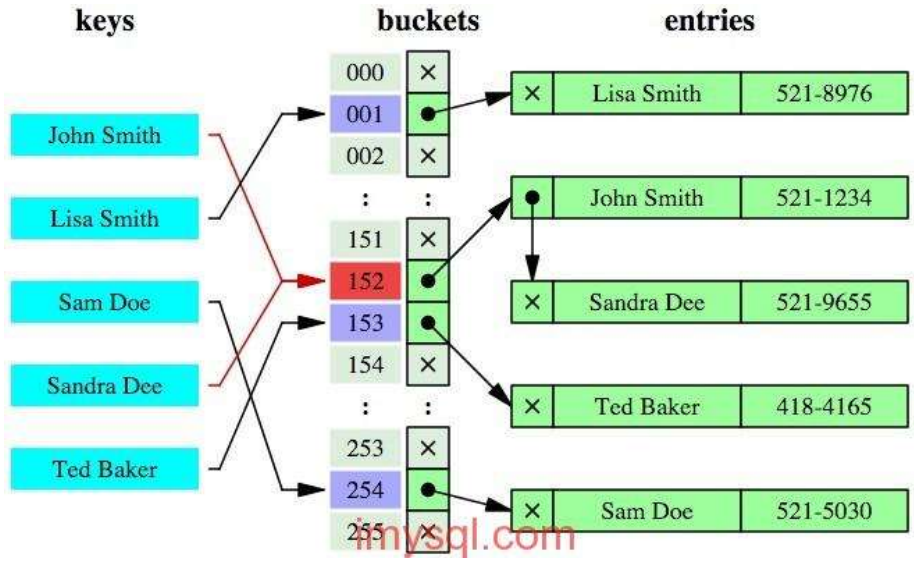
因此，B+树索引被广泛应用于数据库、文件系统等场景。顺便说一下，xfs文件系统比ext3/ext4效率高很多的原因之一就是，它的文件及目录索引结构全部采用B+树索引，而ext3/ext4的文件目录结构则采用Linked list, hashed B-tree、Extents/Bitmap等索引数据结构，因此在高I/O压力下，其IOPS能力不如xfs。

详细可参见：

https://en.wikipedia.org/wiki/Ext4

https://en.wikipedia.org/wiki/XFS

而**哈希索引的示意图**则是这样的：



(图片源自网络)

简单地说，**哈希索引就是采用一定的哈希算法**，把键值换算成新的哈希值，检索时不需要类

似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，速度非常快。

从上面的图来看，B+树索引和哈希索引的明显区别是：

- **如果是等值查询，那么哈希索引明显有绝对优势**，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据；
- 从示意图中也能看到，**如果是范围查询检索，这时候哈希索引就毫无用武之地了**，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；
- 同理，**哈希索引也没办法利用索引完成排序**，以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；
- **哈希索引也不支持多列联合索引的最左匹配规则**；
- B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，**在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题。**

后记

在MySQL中，只有HEAP/MEMORY引擎表才能显式支持哈希索引（NDB也支持，但这个不常用），InnoDB引擎的自适应哈希索引（adaptive hash index）不在此列，因为这不是创建索引时可指定的。

还需要注意到：HEAP/MEMORY引擎表在mysql实例重启后，数据会丢失。

通常，B+树索引结构适用于绝大多数场景，像下面这种场景用哈希索引才更有优势：

在HEAP表中，如果存储的数据重复度很低（也就是说基数很大），对该列数据以等值查询为主，没有范围查询、没有排序的时候，特别适合采用哈希索引

例如这种SQL：

SELECT … FROM t WHERE C1 = ?; — 仅等值查询

在大多数场景下，都会有范围查询、排序、分组等查询特征，用B+树索引就可以了。

来自 <http://www.cnblogs.com/zengkefu/p/5647279.html>

? 7. 优化sql语句

1.对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

2.应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

3.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：
select id from t where num is null
可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：
select id from t where num=0

4.应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：
select id from t where num=10 or num=20
可以这样查询：
select id from t where num=10
union all
select id from t where num=20

5.下面的查询也将导致全表扫描：
select id from t where name like '%abc%'
若要提高效率，可以考虑全文检索。

6.in 和 not in 也要慎用，否则会导致全表扫描，如：
select id from t where num in(1,2,3)
对于连续的数值，能用 between 就不要用 in 了：
select id from t where num between 1 and 3

7.如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：
select id from t where num=@num
可以改为强制查询使用索引：
select id from t with(index(索引名)) where num=@num

8.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：
select id from t where num/2=100
应改为:
select id from t where num=100*2

9.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：
select id from t where substring(name,1,3)='abc'--name以abc开头的id
select id from t where datediff(day,createdate,'2005-11-30')=0--'2005-11-30'生成的id

应改为:
select id from t where name like 'abc%'
select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'

10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

12.不要写一些没有意义的查询，如需要生成一个空表结构：
select col1,col2 into #t from t where 1=0
这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：
create table #t(...)

13.很多时候用 exists 代替 in 是一个好的选择：
select num from a where num in(select num from b)
用下面的语句替换：
select num from a where exists(select 1 from b where num=a.num)

14.并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。

15.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

16.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

17.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

18.尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

19.任何地方都不要使用 select * from t ，用具体的字段列表代替“*”，不要返回用不到的任何字段。

20.尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

21.避免频繁创建和删除临时表，以减少系统表资源的消耗。

22.临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

23.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

24.如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

25.尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

26.使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28.在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE_IN_PROC 消息。

29.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

30.尽量避免大事务操作，提高系统并发能力。

来自 <http://www.jb51.net/article/39221.htm>

参考：http://blog.csdn.net/e3002/article/details/1817941

? **8. 数据库序列化级别有什么优缺点**

9. 关系型数据库和非关系型数据库

关系与非关系型数据库

NoSQL,指的是非关系数据库。由上面的叙述可以看到关系型数据库中的表都是存储一下格式化的数据结构，每个元组字段的组成都是一样的，即使不是每个元组都需要所有的字段，但数据库会为每个元组都分配所有的字段，这样的结构可以便于表与表之间进行连接等操作，但从另一个角度来说它也是关系数据库性能瓶颈的一个因素。而非关系数据库以键值对存储，它的结构不固定，每一个元组可以有不一样的字段，每个元组可以根据需要增加或减少一些自己的键值对，这样就不会局限于固定的结构，可以减少一些时间和空间的开销。

关系型数据库以行和列的形式存储数据，以便于用户理解。这一系列的行和列被称为表，一组表组成了数据库。用户用查询(Query)来检索数据库中的数据。一个Query是一个用于指定数据库中行和列的SELECT语句。关系型数据库通常包含下列组件：

客户端应用程序(Client)

数据库服务器(Server)

数据库(Database)

Structured Query Language(SQL)Client端和Server端的桥梁，Client用SQL来象Server端发送请求，Server返回Client端要求的结果。现在流行的大型关系型数据库有IBM DB2、IBM UDB、Oracle、SQL Server、SyBase、Informix等。

关系型数据库管理系统中储存与管理数据的基本形式是二维表。

关系型数据库是一组已经被组织为表结构的信息的集合。这些信息以表的形式被存储于磁盘、磁带等物理介质中。每个表可以有多行，而每行又被拆分成多列。

关系型数据库一整套数学理论基础，例如关系代数和关系运算是关系型数据库的只要理论基础。

日常生活中我们对表结构非常熟悉，例如学生的成绩表，课程表等，这些表格都是以行和列的二维方式来将信息组织在一起。这些信息可以以各种形式存在，例如打印在纸上，显示在电脑的屏幕上，记录在人们的脑海里，存在服务器的磁盘里等等。

现在需要一种方便的手段来管理这些信息，最好是随时能查询，新增，删除和更新的，这就是数据

关系：

·关系是满足一定条件的二维表，表中的一行称为关系的一个元组，用来存储事物的一个实例；表中

的一列称为关系的一个属性，用来描述实体的某一特征。表是由一组相关实体组成的集合。所以表和

实体集这两个词常常可以交替使用。

·关系是一个行与列交叉的二维表，每一列（属性）的所有数据都是同一种数据类型，每一列都有唯

一的列名，列在表中的顺序无关紧要；表中的任意两行（元组）不能相同，行在表中的顺序也无关紧

要

关系的特征：

·关系的每一行定义实体集的一个实体，每一列定义实体的一个属性

·每一行必须有一个主码，主码是一个属性组（可以是一个属性），它能唯一标识一个实体

·每一列表示一个属性，且列名不能重复

·列的每个值必须与对应属性的类型相同

·列有取值范围，称为域

·列是不可分割的最小数据项

·行、列的顺序对用户无关紧要

关系型数据库与NOSQL

关系型数据库把所有的数据都通过行和列的二元表现形式表示出来。

关系型数据库的优势：

1. 保持数据的一致性（事务处理）

2.由于以标准化为前提，数据更新的开销很小（相同的字段基本上都只有一处）

3. 可以进行Join等复杂查询

其中能够保持数据的一致性是关系型数据库的最大优势。

关系型数据库的不足：

不擅长的处理

1. 大量数据的写入处理

2. 为有数据更新的表做索引或表结构（schema）变更

3. 字段不固定时应用

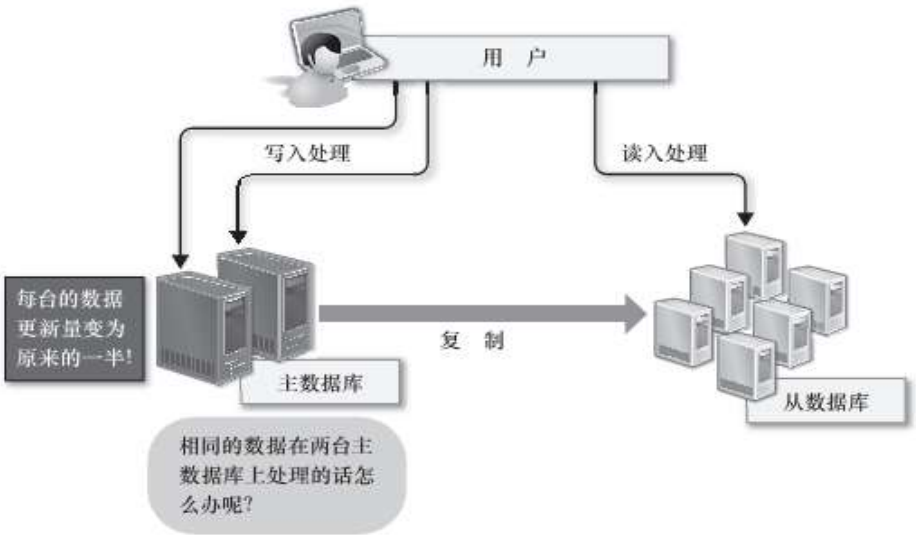
4. 对简单查询需要快速返回结果的处理

--大量数据的写入处理

读写集中在一个数据库上让数据库不堪重负，大部分网站已使用主从复制技术实现读写分离，以提高读写性能和读库的可扩展性。

所以在进行大量数据操作时，会使用数据库主从模式。数据的写入由主数据库负责，数据的读入由从数据库负责，可以比较简单地通过增加从数据库来实现规模化，但是数据的写入却完全没有简单的方法来解决规模化问题。

第一，要想将数据的写入规模化，可以考虑把主数据库从一台增加到两台，作为互相关联复制的二元主数据库使用，确实这样可以把每台主数据库的负荷减少一半，但是更新处理会发生冲突，可能会造成数据的不一致，为了避免这样的问题，需要把对每个表的请求分别分配给合适的主数据库来处理。



第二，可以考虑把数据库分割开来，分别放在不同的数据库服务器上，比如将不同的表放在不同的数据库服务器上，数据库分割可以减少每台数据库服务器上的数据量，以便减少硬盘IO的输入、输出处理，实现内存上的高速处理。但是由于分别存储字不同服务器上的表之间无法进行Join处理，数据库分割的时候就需要预先考虑这些问题，数据库分割之后，如果一定要进行Join处理，就必须要在程序中进行关联，这是非常困难的。

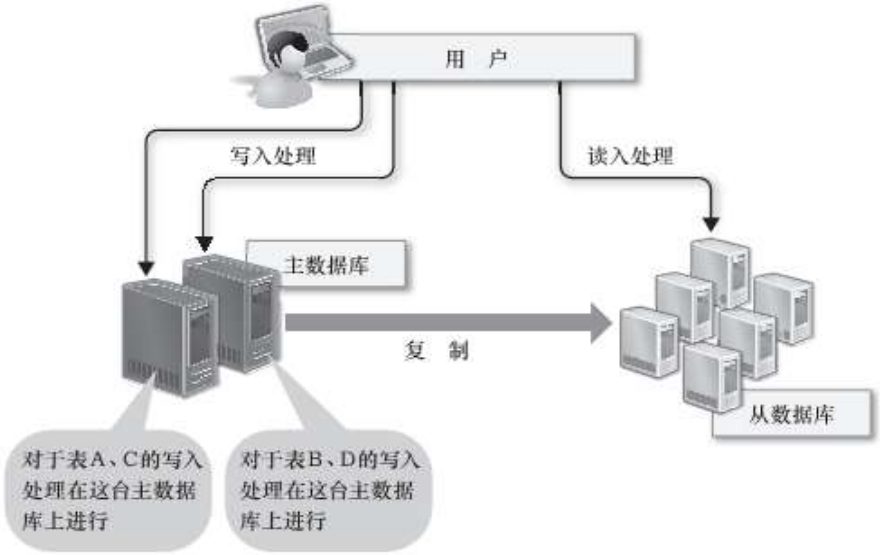


图 二元主数据库问题的解决办法

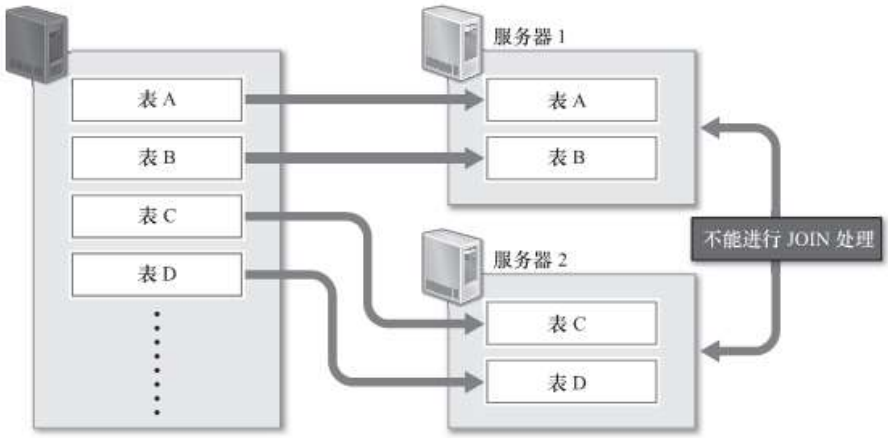


图　　数据库分割

--为有数据更新的表做索引或表结构变更

在使用关系型数据库时，为了加快查询速度需要创建索引，为了增加必要的字段就一定要改变表结构，为了进行这些处理，需要对表进行共享锁定，这期间数据变更、更新、插入、删除等都是无法进行的。如果需要进行一些耗时操作，例如为数据量比较大的表创建索引或是变更其表结构，就需要特别注意，长时间内数据可能无法进行更新。

名称	锁的影响范围	别名
共享锁	其他连接可以对数据进行读取但是不能修改数据	读锁
排他锁	其他连接无法对数据进行读取和修改操作	写锁

--字段不固定时的应用

如果字段不固定，利用关系型数据库也是比较困难的，有人会说，需要的时候加个字段就可以了，这样的方法也不是不可以，但在实际运用中每次都进行反复的表结构变更是非常痛苦的。你也可以预先设定大量的预备字段，但这样的话，时间一长很容易弄不清除字段和数据的对应状态，即哪个字段保存有哪些数据。

--对简单查询需要快速返回结果的处理 （这里的“简单”指的是没有复杂的查询条件）

这一点称不上是缺点，但不管怎样，关系型数据库并不擅长对简单的查询快速返回结果，因为关系型数据库是使用专门的sql语言进行数据读取的，它需要对sql与越南进行解析，同时还有对表的锁定和解锁等这样的额外开销，这里并不是说关系型数据库的速度太慢，而只是想告诉大家若希望对简单查询进行高速处理，则没有必要非使用关系型数据库不可。

NoSQL数据库

关系型数据库应用广泛，能进行事务处理和表连接等复杂查询。相对地，NoSQL数据库只应用在特定领域，基本上不进行复杂的处理，但它恰恰弥补了之前所列举的关系型数据库的不足之处。

优点：

易于数据的分散

各个数据之间存在关联是关系型数据库得名的主要原因，为了进行join处理，关系型数据库不得不把数据存储在同一个服务器内，这不利于数据的分散，这也是关系型数据库并不擅长大数据量的写入处理的原因。相反NoSQL数据库原本就不支持Join处理，各个数据都是独立设计的，很容易把数据分散在多个服务器上，故减少了每个服务器上的数据量，即使要处理大量数据的写入，也变得更加容易，数据的读入操作当然也同样容易。

典型的NoSQL数据库

临时性键值存储（memcached、Redis）、永久性键值存储（ROMA、Redis）、面向文档的数据库（MongoDB、CouchDB）、面向列的数据库（Cassandra、HBase）

一、 键值存储

它的数据是以键值的形式存储的，虽然它的速度非常快，但基本上只能通过键的完全一致查询获取数据，根据数据的保存方式可以分为临时性、永久性和两者兼具 三种。

（1）临时性

所谓临时性就是数据有可能丢失，memcached把所有数据都保存在内存中，这样保存和读取的速度非常快，但是当memcached停止时，数据就不存在了。由于数据保存在内存中，所以无法操作超出内存容量的数据，旧数据会丢失。总结来说：

。在内存中保存数据

。可以进行非常快速的保存和读取处理

。数据有可能丢失

（2）永久性

所谓永久性就是数据不会丢失，这里的键值存储是把数据保存在硬盘上，与临时性比起来，由于必然要发生对硬盘的IO操作，所以性能上还是有差距的，但数据不会丢失是它最大的优势。总结来说：

。在硬盘上保存数据

。可以进行非常快速的保存和读取处理（但无法与memcached相比）

。数据不会丢失

（3）两者兼备

Redis属于这种类型。Redis有些特殊，临时性和永久性兼具。Redis首先把数据保存在内存中，在满足特定条件（默认是 15分钟一次以上，5分钟内10个以上，1分钟内10000个以上的键发生变更）的时候将数据写入到硬盘中，这样既确保了内存中数据的处理速度，又可以通过写入硬盘来保证数据的永久性，这种类型的数据库特别适合处理数组类型的数据。总结来说：

。同时在内存和硬盘上保存数据

。可以进行非常快速的保存和读取处理

。保存在硬盘上的数据不会消失（可以恢复）

。适合于处理数组类型的数据

二、面向文档的数据库

MongoDB、CouchDB属于这种类型，它们属于NoSQL数据库，但与键值存储相异。

（1）不定义表结构

即使不定义表结构，也可以像定义了表结构一样使用，还省去了变更表结构的麻烦。

（2）可以使用复杂的查询条件

跟键值存储不同的是，面向文档的数据库可以通过复杂的查询条件来获取数据，虽然不具备事务处理和Join这些关系型数据库所具有的处理能力，但初次以外的其他处理基本上都能实现。

三、 面向列的数据库

Cassandra、HBae、HyperTable属于这种类型，由于近年来数据量出现爆发性增长，这种类型的NoSQL数据库尤其引入注目。

普通的关系型数据库都是以行为单位来存储数据的，擅长以行为单位的读入处理，比如特定条件数据的获取。因此，关系型数据库也被成为面向行的数据库。相反，面向列的数据库是以列为单位来存储数据的，擅长以列为单位读入数据。

表 1-3　面向行的数据库和面向列的数据库比较

数据类型	数据存储方式	优势
面向行的数据库	以行为单位	对少量行进行读取和更新
面向列的数据库	以列为单位	对大量行少数列进行读取，对所有行的特定列进行同时更新

面向列的数据库具有搞扩展性，即使数据增加也不会降低相应的处理速度（特别是写入速度），所以它主要应用于需要处理大量数据的情况。另外，把它作为批处理程序的存储器来对大量数据进行更新也是非常有用的。但由于面向列的数据库跟现行数据库存储的思维方式有很大不同，故应用起来十分困难。

总结：关系型数据库与NoSQL数据库并非对立而是互补的关系，即通常情况下使用关系型数据库，在适合使用NoSQL的时候使用NoSQL数据库，让NoSQL数据库对关系型数据库的不足进行弥补。

来自 <http://www.cnblogs.com/monion/p/5520945.html>

10. 乐观锁和悲观锁，怎么实现的额

1、悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠**数据库**提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系

统不会修改数据）。

2、乐观锁（ Optimistic Locking ）

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

而乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

本质上，数据库的乐观锁做法和悲观锁做法主要就是解决下面假设的场景，避免丢失更新问题：

一个比较清楚的场景

下面这个假设的实际场景可以比较清楚的帮助我们理解这个问题：

1. 假设当当网上用户下单买了本书，这时数据库中有条订单号为001的订单，其中有个status字段是’有效’,表示该订单是有效的；
2. 后台管理人员查询到这条001的订单，并且看到状态是有效的
3. 用户发现下单的时候下错了，于是撤销订单，假设运行这样一条SQL: update order_table set status = ‘取消’ where order_id = 001;
4. 后台管理人员由于在b这步看到状态有效的，这时，虽然用户在c这步已经撤销了订单，可是管理人员并未刷新界面，看到的订单状态还是有效的，于是点击”发货”按钮，将该订单发到物流部门，同时运行类似如下SQL，将订单状态改成已发货:update order_table set status = ‘已发货’ where order_id = 001

其实之前已经分别对乐观锁的做法和悲观锁的做法做了详细的分析，

这里引用wiki的定义做更权威的引用说明

http://en.wikipedia.org/wiki/Lock_%28database%29

There are mechanisms employed to manage the actions of multiple concurrent users on a database – **the purpose is to prevent lost updates and dirty reads**. The two types of locking are Pessimistic and Optimistic Locking.

- **Pessimistic locking**: **A user who reads a recor**d, with the intention of updating it, **places an exclusive lock on the record to prevent other users from manipulating i**t. This means no one else can manipulate that record until the user releases the lock. The downside is that users can be locked out for a very long time, thereby slowing the overall system response and causing frustration.
    - **Where to use pessimistic locking**: This is mainly used in environments where data-contention (the degree of users request to the database system at any one time) is heavy; where the cost of protecting data through locks is less than the cost of rolling back transactions if concurrency conflicts occur. Pessimistic concurrency is best implemented when lock times will be short, as in programmatic processing of records. **Pessimistic concurrency requires a persistent connection to the database** and is not a scalable option when users are interacting with data, because records might be locked for relatively large periods of time. **It is not appropriate for use in web application development**.

本质上，这里wiki的意思就是，悲观锁和乐观锁都是为了解决丢失更新问题或者是脏读。悲观锁和乐观锁的重点就是是否在读取记录的时候直接上锁。悲观锁的缺点很明显，需要一个持续的数据库连接，这在web应用中已经不适合了。

观点1：只有冲突非常严重的系统才需要悲观锁；

分析：这是更准确的说法；我在原文中说到：

“所有悲观锁的做法都适合于状态被修改的概率比较高的情况，具体是否合适则需要根据实际情况判断。”，表达的也是这个意思，不过说法不够准确；的确，之所以用悲观锁就是因为两个用户更新同一条数据的概率高，也就是冲突比较严重的情况下，所以才用悲观锁。

观点2：最后提交前作一次select for update检查，然后再提交update也是一种乐观锁的做法

分析：这是更准确的说法；

的确，这符合传统乐观锁的做法，就是到最后再去检查。但是wiki在解释悲观锁的做法的时候，’**It is not appropriate for use in web application development**.’，现在已经很少有悲观锁的做法了，所以我自己将这种二次检查的做法也归为悲观锁的变种，因为这在所有乐观锁里面，做法和悲观锁是最接近的，都是先select for update，然后update

来自 <http://blog.csdn.net/sdyy321/article/details/6183412/>

参考：http://www.digpage.com/lock.html

11. 主键和外键作用

关系型**数据库**中的一条记录中有若干个属性，若其中某一个属性组(注意是组)能唯一标识一条记录，该属性组就可以成为一个主键

比如

学生表(学号，姓名，性别，班级)

其中每个学生的学号是唯一的，学号就是一个主键

课程表(课程编号,课程名,学分)

其中课程编号是唯一的,课程编号就是一个主键

成绩表(学号,课程号,成绩)

成绩表中单一一个属性无法唯一标识一条记录，学号和课程号的组合才可以唯一标识一条记录，所以 学号和课程号的属性组是一个主键

成绩表中的学号不是成绩表的主键，但它和学生表中的学号相对应，并且学生表中的学号是学生表的主键，则称成绩表中的学号是学生表的外键

同理 成绩表中的课程号是课程表的外键

定义主键和外键主要是为了维护关系**数据库**的完整性，总结一下：

1.主键是能确定一条记录的唯一标识，比如，一条记录包括身份正号，姓名，年龄。

身份证号是唯一能确定你这个人的，其他都可能有重复，所以，身份证号是主键。

2.外键用于与另一张表的关联。是能确定另一张表记录的字段，用于保持数据的一致性。

比如，A表中的一个字段，是B表的主键，那他就可以是A表的外键。

二、 主键、外键和索引的区别

主键、外键和索引的区别？

	主键	外键	索引
定义：	唯一标识一条记录，不能	表的外键是另一表的主键, 外	该字段没有重复值，

	有重复的，不允许为空	键可以有重复的, 可以是空值	但可以有一个空值
作用：	用来保证数据完整性	用来和其他表建立联系用的	是提高查询排序的速
个数：	主键只能有一个	一个表可以有多个外键	一个表可以有多个惟一索引

聚集索引和非聚集索引的区别？

聚集索引一定是唯一索引。但唯一索引不一定是聚集索引。

聚集索引，在索引页里直接存放数据，而非聚集索引在索引页里存放的是索引，这些索引指向专门的数据页的数据。

三、数据库中主键和外键的设计原则

主键和外键是把多个表组织为一个有效的关系数据库的粘合剂。主键和外键的设计对物理数据库的性能和可用性都有着决定性的影响。

必须将数据库模式从理论上的逻辑设计转换为实际的物理设计。而主键和外键的结构是这个设计过程的症结所在。一旦将所设计的数据库用于了生产环境，就很难对这些键进行修改，所以在开发阶段就设计好主键和外键就是非常必要和值得的。

主键：

关系数据库依赖于主键---它是数据库物理模式的基石。

主键在物理层面上只有两个用途：

1. 惟一地标识一行。

2. 作为一个可以被外键有效引用的对象。

基于以上这两个用途，下面给出了我在设计物理层面的主键时所遵循的一些原则：

1. 主键应当是对用户没有意义的。如果用户看到了一个表示多对多关系的连接表中的数据，并抱怨它没有什么用处，那就证明它的主键设计地很好。

2. 主键应该是单列的，以便提高连接和筛选操作的效率。

注：使用复合键的人通常有两个理由为自己开脱，而这两个理由都是错误的。其一是主键应当具有实际意义，然而，让主键具有意义只不过是给人为地破坏数据库提供了方便。其二是利用这种方法可以在描述多对多关系的连接表中使用两个外部键来作为主键，我也反对这种做法，理由是：复合主键常常导致不良的外键，即当连接表成为另一个从表的主表，而依据上面的第二种方法成为这个表主键的一部分，然，这个表又有可能再成为其它从表的主表，其主键又有可能成了其它从表主键的一部分，如此传递下去，越靠后的从表，其主键将会包含越多的列了。

3. 永远也不要更新主键。实际上，因为主键除了惟一地标识一行之外，再没有其他的用途了，所以也就没有理由去对它更新。如果主键需要更新，则说明主键应对用户无意义的原则被违反了。

注：这项原则对于那些经常需要在数据转换或多数据库合并时进行数据整理的数据并不适用。

4. 主键不应包含动态变化的数据，如时间戳、创建时间列、修改时间列等。

5. 主键应当有计算机自动生成。如果由人来对主键的创建进行干预，就会使它带有除

了惟一标识一行以外的意义。一旦越过这个界限，就可能产生认为修改主键的动机，这样，这种系统用来链接记录行、管理记录行的关键手段就会落入不了解数据库设计的人的手中。

四、数据库主键选取策略

我们在建立数据库的时候，需要为每张表指定一个主键，所谓主键就是能够唯一标识表中某一行的属性或属性组，一个表只能有一个主键，但可以有多个候选索引。因为主键可以唯一标识某一行记录，所以可以确保执行数据更新、删除的时候不会出现张冠李戴的错误。当然，其它字段可以辅助我们在执行这些操作时消除共享冲突，不过就不在这里讨论了。主键除了上述作用外，常常与外键构成参照完整性约束，防止出现数据不一致。所以数据库在设计时，主键起到了很重要的作用。

来自 <http://blog.csdn.net/harbor1981/article/details/53449435>

12. mysql并发怎么解决？

事务，隔离级别，锁

13. mysql主键索引的区别

普通索引：最基本的索引，没有任何限制

唯一索引：与"普通索引"类似，不同的就是：索引列的值必须唯一，但允许有空值。

主键索引：它 是一种特殊的唯一索引，不允许有空值。

全文索引：仅可用于 MyISAM 表，针对较大的数据，生成全文索引很耗时好空间。

组合索引：为了更多的提高mysql效率可建立组合索引，遵循” 最左前缀“原则。

来自 <http://www.cnblogs.com/lonelyxmas/p/4594624.html>

14. 聚簇索引和非聚簇索引的区别

15. varchar和vhar的最大长度

char是一种固定长度的类型，varchar则是一种可变长度的类型，它们的区别是： char(M)类型的数据列里，每个值都占用M个字节，如果某个长度小于M，MySQL就会在它的右边用空格字符补足．（在检索操作中那些填补出来的空格字符将被去掉）在varchar(M)类型的数据列里，每个值只占用刚好够用的字节再加上一个用来记录其长度的字节（即总长度为L+1字节）．

在MySQL中用来判断是否需要进行对据列类型转换的规则

1、在一个数据表里，如果每一个数据列的长度都是固定的，那么每一个数据行的长度也将是固定的．

2、只要数据表里有一个数据列的长度的可变的，那么各数据行的长度都是可变的．

3、如果某个数据表里的数据行的长度是可变的，那么，为了节约存储空间，MySQL会把这个数据表里的固定长度类型的数据列转换为相应的可变长度类型．例外：长度小于4个字符的char数据列不会被转换varchar类型

在mysql中,char和varchar都表示字符串类型.但他们存储和检索数据的方式并不相同.

在表结构定义中声明char和varchar类型时,必须指定字符串的长度.也就是该列所能存储多

少个字符（**不是字节,是字符**）.例如：**char(10)和varchar(10)都能存储10个字符.**

声明为char的列长度是固定的,char的长度可选范围在0-255之间.也就是char最大能存储255个字符.如果该列是utf8编码,则该列所占用的字节数=字符数*3.如果是gbk编码则该列所占用的字节数=字符数*2.

声明为varchar的列长度是可变的,在mysql5.0.3之前varchar的长度范围为0-255,mysql5.0.3之后varchar的长度范围为0-65535个字节.采用varchar类型存储数据需要1-2个字节(长度超过255时需要2个字节)来存储字符串的实际长度.如果该列的编码为gbk,每个字符最多占用2个字节,最大长度不能超过32766个字符.如果该列的编码为utf8,每个字符最多占3个字节,最大字符长度为21845.

总结:

① char和varchar最大的不同就是一个是固定长度,一个是可变长度.由于是可变长度,因此存储的是实际字符串再加上一个记录字符串长度的字节。如果分配给char或varchar列的值超过 列的最大长度,则对值进行裁剪.

②varchar(M)和char(M),M都表示字符数.varchar的最大长度为65535个字节,不同的编码所对应的最大可存储的字符数不同.char最多可以存放255个字符,不同的编码最大可用字节数不同.

来自 <http://www.cnblogs.com/jewave/p/6214540.html>

16. 内连接和外连接

1、内联接（典型的联接运算，使用像 = 或 <> 之类的比较运算符）。包括相等联接和自然联接。

内联接使用比较运算符根据每个表共有的列的值匹配两个表中的行。例如，检索 students和courses表中学生标识号相同的所有行。

2、外联接。外联接可以是左向外联接、右向外联接或完整外部联接。

在 FROM子句中指定外联接时，可以由下列几组关键字中的一组指定：

1）LEFT JOIN或LEFT OUTER JOIN

左向外联接的结果集包括 LEFT OUTER子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值。

2）RIGHT JOIN 或 RIGHT OUTER JOIN

右向外联接是左向外联接的反向联接。将返回右表的所有行。如果右表的某行在左表中没有匹配行，则将为左表返回空值。

3）FULL JOIN 或 FULL OUTER JOIN

完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。如果表之间有匹配行，则整个结果集行包含基表的数据值。

3、交叉联接

交叉联接返回左表中的所有行，左表中的每一行与右表中的所有行组合。交叉联接也称作笛卡尔积。

FROM 子句中的表或视图可通过内联接或完整外部联接按任意顺序指定；但是，用左或右向

外联接指定表或视图时，表或视图的顺序很重要。有关使用左或右向外联接排列表的更多信息，请参见使用外联接。

例子：

---

a表 id name b表 id job parent_id

1 张3 1 23 1

2 李四 2 34 2

3 王武 3 34 4

a.id同parent_id 存在关系

1）内连接

select a.*,b.* from a inner join b on a.id=b.parent_id

结果是

1 张3 1 23 1

2 李四 2 34 2

2）左连接

select a.*,b.* from a left join b on a.id=b.parent_id

结果是

1 张3 1 23 1

2 李四 2 34 2

3 王武 null

3）右连接

select a.*,b.* from a right join b on a.id=b.parent_id

结果是

1 张3 1 23 1

2 李四 2 34 2

null 3 34 4

4）完全连接

select a.*,b.* from a full join b on a.id=b.parent_id

结果是

1 张3 1 23 1

2 李四 2 34 2

null 3 34 4

3 王武 null

一、交叉连接（CROSS JOIN）

交叉连接（CROSS JOIN）：有两种，显式的和隐式的，不带ON子句，返回的是两表的乘积，也叫笛卡尔积。

例如：下面的语句1和语句2的结果是相同的。

语句1：隐式的交叉连接，没有CROSS JOIN。

SELECT O.ID, O.ORDER_NUMBER, C.ID, C.NAME

```
FROM ORDERS O , CUSTOMERS C
WHERE O.ID=1;
```

语句2：显式的交叉连接，使用CROSS JOIN。

```
SELECT O.ID,O.ORDER_NUMBER,C.ID,
C.NAME
FROM ORDERS O CROSS JOIN CUSTOMERS C
WHERE O.ID=1;
```

语句1和语句2的结果是相同的，查询结果如下：

二、内连接（INNER JOIN）

内连接（INNER JOIN）：有两种，显式的和隐式的，返回连接表中符合连接条件和查询条件的数据行。（所谓的链接表就是数据库在做查询形成的中间表）。

例如：下面的语句3和语句4的结果是相同的。

语句3：隐式的内连接，没有INNER JOIN，形成的中间表为两个表的笛卡尔积。

```
SELECT O.ID,O.ORDER_NUMBER,C.ID,C.NAME
FROM CUSTOMERS C,ORDERS O
WHERE C.ID=O.CUSTOMER_ID;
```

语句4：显示的内连接，一般称为内连接，有INNER JOIN，形成的中间表为两个表经过ON条件过滤后的笛卡尔积。

```
SELECT O.ID,O.ORDER_NUMBER,C.ID,C.NAME
FROM CUSTOMERS C INNER JOIN ORDERS O ON C.ID=O.CUSTOMER_ID;
```

语句3和语句4的查询结果：

三、外连接（OUTER JOIN）：外连不但返回符合连接和查询条件的数据行，还返回不符合条件的一些行。外连接分三类：左外连接（LEFT OUTER JOIN）、右外连接（RIGHT OUTER JOIN）和全外连接（FULL OUTER JOIN）。

三者的共同点是都返回符合连接条件和查询条件（即：内连接）的数据行。不同点如下：

左外连接还返回左表中不符合连接条件单符合查询条件的数据行。

右外连接还返回右表中不符合连接条件单符合查询条件的数据行。

全外连接还返回左表中不符合连接条件单符合查询条件的数据行，并且还返回右表中不符合连接条件单符合查询条件的数据行。全外连接实际是上左外连接和右外连接的数学合集（去掉重复），即“全外=左外 UNION 右外”。

说明：左表就是在“（LEFT OUTER JOIN）”关键字左边的表。右表当然就是右边的了。在三种类型的外连接中，OUTER 关键字是可省略的。

下面举例说明：

语句5：左外连接（LEFT OUTER JOIN）

```
SELECT O.ID,O.ORDER_NUMBER,O.CUSTOMER_ID,C.ID,C.NAME
FROM ORDERS O LEFT OUTER JOIN CUSTOMERS C ON C.ID=O.CUSTOMER_ID;
```

语句6：右外连接（RIGHT OUTER JOIN）

```
SELECT O.ID,O.ORDER_NUMBER,O.CUSTOMER_ID,C.ID,C.NAME
FROM ORDERS O RIGHT OUTER JOIN CUSTOMERS C ON C.ID=O.CUSTOMER_ID;
```

注意：WHERE条件放在ON后面查询的结果是不一样的。例如：

语句7：WHERE条件独立。

```
SELECT O.ID,O.ORDER_NUMBER,O.CUSTOMER_ID,C.ID,C.NAME
FROM ORDERS O LEFT OUTER JOIN CUSTOMERS C ON C.ID=O.CUSTOMER_ID
WHERE O.ORDER_NUMBER<>'MIKE_ORDER001';
```

语句8：将语句7中的WHERE条件放到ON后面。

```
SELECT O.ID,O.ORDER_NUMBER,O.CUSTOMER_ID,C.ID,C.NAME
FROM ORDERS O LEFT OUTER JOIN CUSTOMERS C ON C.ID=O.CUSTOMER_ID AND
O.ORDER_NUMBER<>'MIKE_ORDER001';
```

从语句7和语句8查询的结果来看，显然是不相同的，语句8显示的结果是难以理解的。因此，推荐在写连接查询的时候，ON后面只跟连接条件，而对中间表限制的条件都写到WHERE子句中。

语句9：全外连接（FULL OUTER JOIN）。

```
SELECT O.ID,O.ORDER_NUMBER,O.CUSTOMER_ID,C.ID,C.NAME
FROM ORDERS O FULL OUTER JOIN CUSTOMERS C ON C.ID=O.CUSTOMER_ID;
```

注意：MySQL是不支持全外的连接的，这里给出的写法适合Oracle和DB2。但是可以通过左外和右外求合集来获取全外连接的查询结果。下图是上面SQL在Oracle下执行的结果：

语句10：左外和右外的合集，实际上查询结果和语句9是相同的。

```
SELECT O.ID,O.ORDER_NUMBER,O.CUSTOMER_ID,C.ID,C.NAME
FROM ORDERS O LEFT OUTER JOIN CUSTOMERS C ON C.ID=O.CUSTOMER_ID
UNION
SELECT O.ID,O.ORDER_NUMBER,O.CUSTOMER_ID,C.ID,C.NAME
FROM ORDERS O RIGHT OUTER JOIN CUSTOMERS C ON C.ID=O.CUSTOMER_ID;
```

来自 <http://www.jb51.net/article/39432.htm>

17.