1、脏读、不可重复读、幻读

**脏读：**（针对未提交数据）如果一个事务中对数据进行了更新，但事务还没有提交，另一个事务可以“看到”该事务没有提交的更新结果，这样造成的问题就是，如果第一个事务回滚，那么，第二个事务在此之前所“看到”的数据就是一笔脏数据。

**不可重复读：**（针对其他提交前后，读取数据本身的对比）不可重复读取是指同一个事务在整个事务过程中对同一笔数据进行读取，每次读取结果都不同。如果事务1在事务2的更新操作之前读取一次数据，在事务2的更新操作之后再读取同一笔数据一次，两次结果是不同的，所以，Read Uncommitted也无法避免不可重复读取的问题。

**幻读:** （针对其他提交前后，读取数据条数的对比） 幻读是指同样一笔查询在整个事务过程中多次执行后，查询所得的结果集是不一样的。幻读针对的是多笔记录。在Read Uncommitted隔离级别下， 不管事务2的插入操作是否提交，事务1在插入操作之前和之后执行相同的查询，取得的结果集是不同的，所以，Read Uncommitted同样无法避免幻读的问题。

**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）**

**幻读的重点在于新增或者删除！（同样的条件，第一次和第二次读出来的记录数不一样）**

2、事务的四种隔离级别

Read Uncommitted、Read commited、Repeatable read、Serializable，其中再Spring的Isolation属性中一共支持5种事务设置，多了一个default（使用数据库设置的隔离级别）

**Read Ucommitted:** 最低的隔离级别，Read Uncommitted最直接的效果就是一个事务可以读取另一个事务并未提交的更新结果。

**Read commited:** 是大部分数据库采用的默认隔离级别, 它在Read Uncommitted隔离级别基础上所做的限定更进一步， 在该隔离级别下，一个事务的更新操作结果只有在**该事务提交之后，另一个事务才可能读取到同一笔数据更新后的结果**。该隔离级别可以避免脏读、但不可避免不可重复读和幻读。

**Repeatable read:** **mysql默认隔离级别，保证在整个事务的过程中**，**对同一笔数据的读取结果是相同的**，不管其他事务是否同时在对同一笔数据进行更新，也不管其他事务对同一笔数据的更新提交与否。 Repeatable Read隔离级别避免了脏读和不可重复读取的问题，但无法避免幻读。

**Serializable:** 最为严格的隔离级别，**所有的事务操作都必须依次顺序执行，可以避免其他隔离级别遇到的所有问题**，是最为安全的隔离级别， 但同时也是性能最差的隔离级别，因为所有的事务在该隔离级别下都需要依次顺序执行，所以，并发度下降，吞吐量上不去，性能自然就下来了。 因为该隔离级别极大的影响系统性能，所以，很少场景会使用它。通常情况下，我们会使用其他隔离级别加上相应的并发锁的机制来控制对数据的访问，这样既保证了系统性能不会损失太大，也能够一定程度上保证数据的一致性。

**Spring的Isolation**是在TanscationDefinition接口中定义这些属性。在这个接口中还定义了七种事务传播行为。

3、大数据量分页优化

（1）**建立主键或唯一索引，利用索引**

---语句样式: MySQL中,可用如下方法: SELECT \* FROM 表名称 WHERE id\_pk > (pageNum\*10) LIMIT M

---适应场景: 适用于数据量多的情况(元组数上万)

---原因: 索引扫描,速度会很快. 有朋友提出: **因为数据查询出来并不是按照pk\_id排序的，所以会有漏掉数据的情况**，只能方法3

（2**）基于索引再排序**

---语句样式: MySQL中,可用如下方法: SELECT \* FROM 表名称 WHERE id\_pk > (pageNum\*10) ORDER BY id\_pk ASC LIMIT M

---适应场景: 适用于数据量多的情况(元组数上万). 最好ORDER BY后的列对象是主键或唯一索引,使得ORDERBY操作能利用索引被消除但结果集是稳定的(稳定的含义,参见方法1)

---原因: 索引扫描,速度会很快. 但MySQL的排序操作,只有ASC没有DESC(DESC是假的,未来会做真正的DESC,期待...).

（3）**基于索引使用prepare**（第一个问号表示pageNum，第二个？表示每页元组数）

---语句样式: MySQL中,可用如下方法: PREPARE stmt\_name FROM SELECT \* FROM 表名称 WHERE id\_pk > (？\* ？) ORDER BY id\_pk ASC LIMIT M

---适应场景: 大数据量

---原因: **索引扫描,速度会很快. prepare语句又比一般的查询语句快一点**。

(4)利用Mysql支持order操作可以**利用索引快速定位部分元组，避免全表扫描**

比如: 读第1000到1019行元组(pk是主键/唯一键).

SELECT \* FROM your\_table WHERE pk>=1000 ORDER BY pk ASC LIMIT 0,20

（5）**利用"子查询/连接+索引"快速定位元组的位置,然后再读取元组.** 道理同方法5

**利用子查询示例:**

SELECT \* FROM your\_table WHERE id <=

(SELECT id FROM your\_table ORDER BY id desc LIMIT ($page-1)\*$pagesize ORDER BY id desc LIMIT $pagesize

**利用连接示例:**

SELECT \* FROM your\_table AS t1

JOIN (SELECT id FROM your\_table ORDER BY id desc LIMIT ($page-1)\*$pagesize AS t2

WHERE t1.id <= t2.id ORDER BY t1.id desc LIMIT $pagesize;

直接使用limit start,count分页语句，我们能总结出来两点

（1） limit语句的查询时间与起始记录的位置成正比

（2） mysql的limit语句是很方便的，但是对记录很多的表并不适合直接使用

**对limit分页问题的性能优化方法：**

利用表的覆盖索引来加速分页查询，我们都知道，利用索引查询的语句中如果只包含了那个索引列（覆盖索引），那么这种查询会很快，因为利用索引查找有优化算法，且数据就在查询索引上面，不用再去找相关的数据地址了，这样节省了很多时间，另外mysql中也有相关的索引缓存，再并发高的时候利用缓存就效果更好了。但是如果我们要查询所有列，有两种方法，一种是id>=的形式，另外一种就是利用join. 看下实际情况：

SELECT \* FROM product WHERE ID > =(select id from product limit 866613, 1) limit 20

查询时间为0.2秒！

另一种写法

SELECT \* FROM product a JOIN (select id from product limit 866613, 20) b ON a.ID = b.id

查询时间也很短！

**复合索引优化方法：对于有where条件，又想走索引用limit的，必须设计一个索引，将where放在第一位，limit用到的主键放第二位，而且只能select主键。**

4、百万级数据库优化方案

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

2、应该尽量避免在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。

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

4、应该尽量避免在where子句中使用or来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描。

5、in和not in要慎用，否则会导致全表扫描。对于连续的数值，能用between就不要用in了，很多时候exists代替in是一个好选择。

6、如果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，它必须在编译时建立访问计划。

7、应该精良避免在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。

8、应该尽量避免在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。

9、对于多张大数据量的表的join，首先要分页再join,否则逻辑读会很高，性能很差。

10、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以及update的效率，因为insert或update时有可能会重建索引，所以建立索引要慎重。

11、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调整，会耗费相当大的资源。

12、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，并会增加存储的开销，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比较一次就好。

13、尽可能使用varchar/nvarchar代替char/nchar，因为变长字段存储空间小，可节省存储空间，其次对查询来说，在一个相对较小的字段内搜索效率显然高些。

14、任何地方都不要使用select \* from t，不要返回用不到的任何字段。

15、避免频繁创建和删除临时表，以减少系统表资源的消耗。

16、尽量避免使用游标，因为游标的效率较差。

17、尽量避免大事务操作，提高系统的并发能力。

18、尽量避免向客户端返回大数据量，若数据量过大，应重新考虑需求是否合理。

19、合理使用存储过程，采用存储过程可以将业务逻辑装在存储过程中，然后在客户端

直接调用存储过程，减少网络交互的成本。

数据库性能优化基本法则：（漏斗法则）

计算机系统硬件性能从高到低依次为：

**CPU——Cache(L1-L2-L3)——内存——SSD硬盘——网络——硬盘**

1、减少数据访问（减少磁盘访问）

2、返回更少数据（减少网络传输或磁盘访问）

3、减少交互次数（减少网络传输）

4、减少服务器CPU开销(减少CPU及内存开销)

5、利用更多资源（增加资源）

5、聚集索引与非聚集索引

索引是关系型数据库中给数据库表中一列或多列值排序后的存储结构，sql主流的索引结构有B+树以及Hash结构，聚集索引以及非聚集索引用的是B+树索引。

**聚集索引：（clustered索引）：数据行的物理顺序与列值（一般是主键的那列）的逻辑顺序相同，一个表中只能拥有一个聚集索引**

从下图可以看出对于聚集索引，索引的叶子节点就是对应的数据节点，可以直接获取到对应的全部列的数据，而非聚集索引在索引没有覆盖到对应的列的时候需要进行二次查询。

如果不创建索引，系统会自动创建一个隐含列作为表的聚集索引，sql server默认主键为聚集索引，而mysql里主键就是聚集索引。

创建表后添加聚集索引：

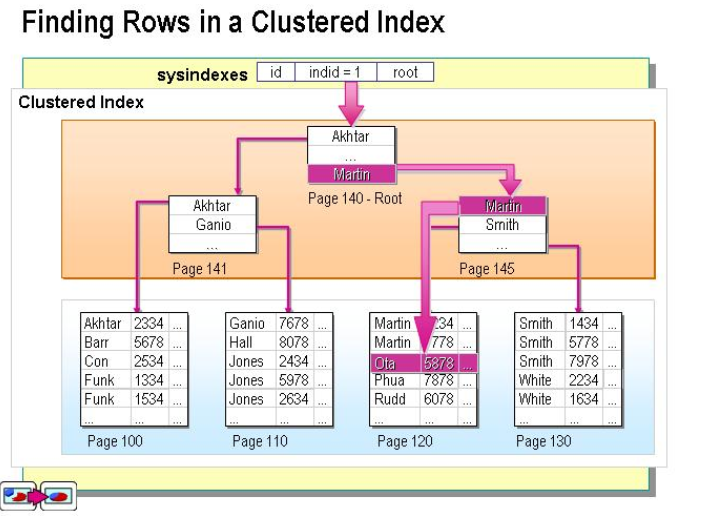
Sql server

create clustered index clustered\_index on table\_name(colum\_name)

Mysql

alter table table\_name add primary key(colum\_name)

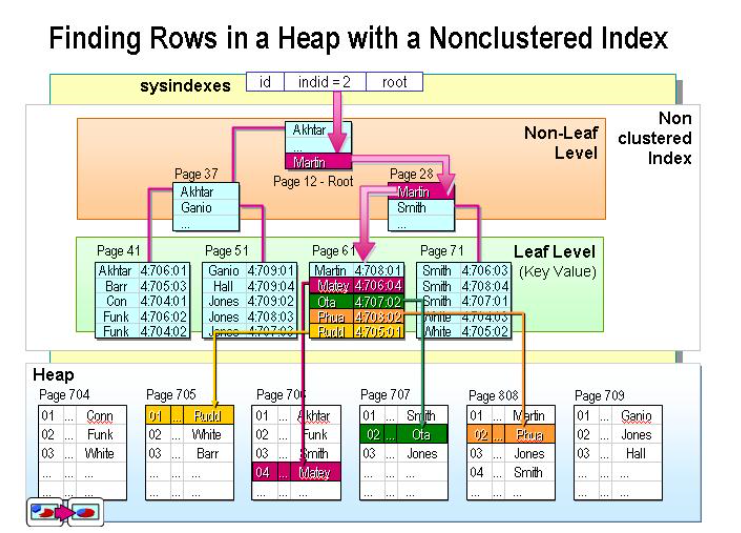
最好在创建表的时候添加聚集索引，由于聚集索引的物理顺序上的特殊性，因此如果再在上面创建索引的时候会根据索引的排列顺序移动全部数据行上的顺序，会非常耗时。



**非聚集索引：该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。**

其实按照定义，除了聚集索引以外的索引都是非聚集索引，只是人们想细分一下非聚集索引，分成普通索引，唯一索引，全文索引。如果非要把非聚集索引比成现实生活中的东西，那么非聚集索引就像新华字典的偏旁字典，他结构顺序与实际存放的顺序不一致。

索引是通过二叉树的数据结构来描述的，我们可以这么理解聚集索引，索引的叶节点就是数据节点，而非聚集索引的叶节点仍是索引节点，只不过有一个指针指向对应数据块。



**索引使用的误区：**

使用聚集索引的最大好处就是能够根据查询要求，迅速缩小查询范围，避免全表扫描，

**1、主键就是聚集索引？**

这种想法我认为是不正确的，虽然sql server默认的是在主键上建立聚集索引的，通常，我们会在每个表中都建立一个id列，以区分每条数据，并且这个id列是自动增长的，步长为1。但我认为这样做的意义不大。显而易见，聚集索引的优势是很明显的，而每个表中只能有一个聚集索引，这使得聚集索引变得更加珍贵，在实际使用中，因为id是自动生成的，物品们并不知道每条记录的id号，所以我们很难在实践中用id号进行查询，这就使得id号这个主键作为聚集索引成为一种资源浪费。

如果您的聚集索引还是盲目地建在ID这个主键上时，您的查询速度是没有这么高的，即使您在“日期”这个字段上建立的索引（非聚合索引）。下面我们就来看一下在1000万条数据量的情况下各种查询的速度表现（3个月内的数据为25万条）：

（1）仅在主键上建立聚集索引，并且不划分时间段：

Select gid,fariqi,neibuyonghu,title from tgongwen

用时：128470毫秒（即：128秒）

（2）在主键上建立聚集索引，在fariq上建立非聚集索引：

select gid,fariqi,neibuyonghu,title from Tgongwen

where fariqi> dateadd(day,-90,getdate())

用时：53763毫秒（54秒）

（3）将聚合索引建立在日期列（fariqi）上：

select gid,fariqi,neibuyonghu,title from Tgongwen

where fariqi> dateadd(day,-90,getdate())

用时：2423毫秒（2秒）

虽然每条语句提取出来的都是25万条数据，各种情况的差异却是巨大的，特别是将聚集索引建立在日期列时的差异。事实上，如果您的数据库真的有1000 万容量的话，把主键建立在ID列上，就像以上的第1、2种情况，在网页上的表现就是超时，根本就无法显示。这也是我摒弃ID列作为聚集索引的一个最重要的因素。得出以上速度的方法是：在各个select语句前加：

declare @d datetime

set @d=getdate()

并在select语句后加：

select [语句执行花费时间(毫秒)]=datediff(ms,@d,getdate())

**2、只要建立索引就能显著提高查询速度？**

事实上，我们可以发现上面的例子中，第2、3条语句完全相同，且建立索引的字段也相同；不同的仅是前者在fariqi字段上建立的是非聚合索引，后者在此字段上建立的是聚合索引，但查询速度却有着天壤之别。所以，并非是在任何字段上简单地建立索引就能提高查询速度。

**3、把所有需要提高查询速度的字段都加进聚集索引，以提高查询速度**

上面已经谈到：在进行数据查询时都离不开字段的是“日期”还有用户本身的“用户名”。既然这两个字段都是如此的重要，我们可以把他们合并起来，建立一个复合索引（compound index）。很多人认为只要把任何字段加进聚集索引，就能提高查询速度，从实验中，我们可以看到如果仅用聚集索引作为起始列作为查询条件和同时用到复合聚集索引的全部列的查询速度几乎是一样的，甚至比用上全部的复合索引列还要快，如果仅用复合聚集索引的非起始列作为查询条件的话，这个索引是不起任何作用的，当然，语句1、2的查询速度一样是因为查询的条目数一样，如果复合索引的所有列都用上，而且查询结果少的话，这样就会形成“索引覆盖”，因而性能可以达到最优。同时请你记住：无论你是否经常使用聚合索引的其他列，但其前导列一定要是使用最频繁的列。

总结：

1、使用聚合索引比非聚合索引的主键速度快，（25w条快1/4）

2、使用聚集索引比用一般主键做order by 时的速度快，特别是在小数据量的情况下。

3、索引有助于提高检索性能，但过多或不当的索引也会导致系统低效。因为用户在表中每加进一个索引，数据库就要做更多的工作。过多的索引甚至会导致索引碎片。我们要建立一个“适当”的索引体系，特别是对聚合索引的创建，更应精益求精，以使您的数据库能得到高性能的发挥。

**4、如何解决非聚集索引的二次查询问题？**

复合索引（覆盖索引）

建立两列以上的索引，即可查询复合索引里的列的数据而不需要进行回表二次查询，如index(col1, col2)，执行下面的语句

select col1, col2 from t1 where col1 = '213';

要注意使用复合索引需要满足最左侧索引的原则，也就是查询的时候如果where条件里面没有最左边的一到多列，索引就不会起作用。

**5、总结**

1、使用聚集索引的查询效率要比非聚集索引的效率要高，但是如果需要频繁去改变聚集索引的值，写入性能并不高，因为需要移动对应数据的物理位置。

2、非聚集索引在查询的时候可以的话就避免二次查询，这样性能会大幅提升。

3、不是所有的表都适合建立索引，只有数据量大表才适合建立索引，且建立在选择性高的列上面性能会更好。

6、五大约束和三大范式

1.主键约束（Primay Key Coustraint） 唯一性，非空性；

2.唯一约束 （Unique Counstraint）唯一性，可以空，但只能有一个；

3.默认约束 (Default Counstraint) 该数据的默认值；

4.外键约束 (Foreign Key Counstraint) 需要建立两表间的关系；

5.非空约束（Not Null Counstraint）:设置非空约束，该字段不能为空。

1NF：数据表中的每一列(字段)，必须是不可拆分的最小单元，也就是确保每一列的原子性。

2NF: 如果一个关系满足1NF，并且除了主键以外的其它列，都依赖与该主键，则满足二范式(2NF)，第二范式要求每个表只描述一件事

3NF: 在1NF基础上，任何非主属性不依赖于其它非主属性（在2NF基础上消除传递依赖）

目前数据库有六种范式，范式越高，冗余越低，再往上表越多，可能导致查询效率降低。

7、InnoDB、MyISAM存储引擎的区别

**1.MyISAM是非事物安全的，InnoDB是事物安全的**

InnoDB具有事务、回滚、和崩溃修复能力的事务安全型表。非事务安全的优点是更快，所需的磁盘空间更小，执行更新时需要的内存更小，但所有发生的改变都是永久的。

**2、MyISAM存储总行数，InnoDB不存储总行数。**

3、**MyISAM锁的粒度是表级的，InnoDB支持行级锁以及表级，默认情况下是采用行级锁**

**粒度小，可以提高并发性能，单开小高，因为如果锁定了多行，则需要持有更多的锁。**

粒度大，会降低并发性能，因为锁定整个表限制了其他事务对表中任意部分的访问，但开销小。

**4、InnoDB支持外键，MyISAM不支持**

MyISAM管理非事务表，提供高速存储和检索以及全文搜索的能力，如果在应用中执行大量的select操作，应选用MyISAM。

InnoDB用于事物处理，具有ACID（原子性，一致性，隔离性，持久性）事物支持等特性，如果在应用中大量使用insert和update操作，应选用InnoDB。

**5、MyISAM是非聚集索引，InnoDB是聚集索引。**

聚集索引是物理地址连续存放的索引，在取区间的时候，查询速度非常快，但同样，插入速度会受到影响。聚集索引的物理位置使用链表进行存储。

8、视图、存储过程、触发器

触发器与存储过程非常相似，触发器也是SQL语句集，两者唯一的区别是触发器不能用EXECUTE语句调用

**使用存储过程的好处：**

1、减少网络通信量，

2、执行速度更快，在存储过程创建的时候，数据库已经对其进行了一次解析和优化，其次存储过程一旦执行，在内存中就会保留一份存储过程。

3、可复用性高

4、增强安全性，通过向用户授予对存储过程的访问权限，它们可以提供特定的数据访问

**缺点：**

**移植性差**

存储过程封装业务逻辑将限制应用程序的可移植性。

9、为什么MySQL数据库选择B+树

Mysql的数据一般时放在磁盘中的，读取数据的时候，肯定有访问磁盘的操作，当有大规模数据存储到磁盘中的时候，定位非常耗时，我们可以需用B树进行优化。

核心思想：B树在提高IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生，B+树只需要去遍历叶子结点就可以实现整棵树的遍历，而在数据库中基于范围的查询是频繁的，而B树的效率太低。

毫无疑问B+树肯定是为了提高查询效率

B树：多路搜索树，关键字集合分布在整棵树中，搜索有可能在非叶子结点结束。

B+树：非叶子节点只保存索引 ，不保存数据，数据都保存在叶子结点中。

对树的查找瓶颈在于树的深度，所以优化的过程中，尽可能把数据结构设计的更为矮胖一点就可以减少访问层数。因为说数据库的索引不可能全部存在内存中，索引往往以索引文件的形式存储在磁盘上，索引的查找过程中要产生磁盘IO消耗，索引评价一个数据结构优劣最重要的指标就是在查找过程中磁盘IO的操作次数，

磁盘顺序读取的效率很高（不需要寻道）因此 对于具有局部性的程序来说，预判可以提高IO效率。

数据库系统巧妙利用了磁盘预读的原理，将一个结点地大小设为一个页，这样每个节点只需要一次IO就可以完全载入

红黑树由于逻辑上很近的节点物理上可能很远，无法利用局部性，所以红黑树的效率比B树差很多。

10、简单的SQL操作

创建表：

create table celebs(id integer, name text, age integer);

插入记录：

insert into celebs(id, name, age) values(1, 'Justin Bieber', 21 );

修改记录：

update celebs set age=22 where id = 1;

修改表结构：

alter table celebs add column twitter\_handle text;