以mysql为例：

**事务：**

事务的特性（ACID）：原子性、一致性、隔离性、持久性。

**原子性（Atomicity）**：事务包含的操作，要么全部执行成功，要么失败全部回滚。

**一致性（Consistency）**：事务执行完成后，数据库的数据必须从一个正确的状态变更为另一个正确的状态，本质上是数据的约束条件。如银行转账，AB各有1000元，那么事务执行完成后，一致性的约束条件为AB两者的钱相加为2000元，AB的余额不能为负数，否则回滚事务。

**隔离性（Isolation）**：达到并行事务之间感受不到其它事务在执行的效果。

**持久性（Durability）**：事务在提交后对数据库的改变是永久的，即使数据库出现故障也不会丢失事务的操作。

事务不隔离带来的问题：更新丢失、脏读、不可重复读、虚（幻）读。

事务的隔离级别：读未提交（read uncommitted）、读已提交（read committed）、可重复读（repeatable read）、串行化（serializable）。依次解决上述4个问题。

更新丢失：

1.事务1和事务2同时更新同一条数据，事务1更新完数据A后，事务2又对A进行更新。此时事务1出现错误，回滚操作会覆盖事务2对A的更新。

2.事务1和事务2同时更新同一条数据，先更新的会被后更新的覆盖。

脏读：

指在一个事务处理过程里读取了另一个未提交的事务中的数据。事务1在更新完数据A后，事务2读取数据A。此时事务1出现错误进行回滚，事务2读取到的数据A则变为脏数据。

不可重复读：

事务1在读取数据A后，事务2立即对数据A进行了更新，事务再次对数据A读取时，得到了与第一次读取不一样的数据。

虚（幻）读：

事务1对表A的所有数据执行了将i列由"1"更新为"2"的操作，随后事务2向表A插入了一条i列为"1"的数据。此时事务1再次查询表A的数据时，发现还有一行数据的i列没有变为"2"。

读未提交：

事务写时，禁止其它事务写。

读已提交：

事务写时，禁止其它事务读写。

可重复读：

事务写时，禁止其它事务读写。事务读时，禁止其它事务”写”（其它事务可以写，当前事务读到的值一直为第一次所读到的值）。

串行化：

事务只能一个一个执行，不能并发执行（一个事务未执行完成时，另一事务开始执行会报错并返回）。

Mysql默认事务隔离级别为可重复读。Oracle、sqlserver默认事务隔离级别为读已提交。

SELECT @@tx\_isolation;查看当前事务隔离级别。

**索引：**

索引采用的数据结构：B+树。

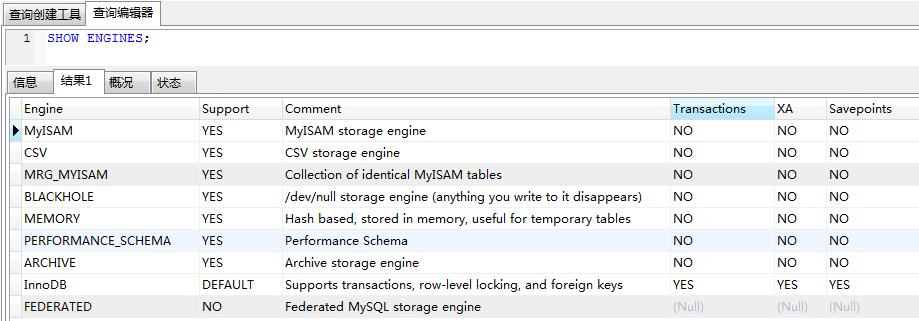
建立索引的原则：

1. 最左匹配原则：mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。
2. =和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式。
3. 尽量选择区分度高的列作为索引。若区分度较低，查询速度无法得到较大提升的同时，数据的插入更新操作效率会被拉低。
4. 索引列不能参与计算，保持列“干净”，比如from\_unixtime(create\_time) = '2014-05-29'就不能使用到索引，因为B+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，成本太大。所以语句应该写成create\_time = unix\_timestamp('2014-05-29')。
5. 尽量扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

不会使用索引的情况：

1. 查询条件中有not in，or，not exist。
2. like查询以%开头。
3. 小表查询会全表扫描，而不会使用索引。

**数据库引擎：**



InnoDB行级锁：

锁定单行数据（若sql查询出多条数据则所有数据都加锁），行级锁是基于索引的，若该sql没有用到索引，将不会使用行级锁，而是直接锁住整表。

InnoDB间隙锁：

间隙锁同样是基于索引的。可以用于在repeatable read下解决幻读问题。

SHOW VARIABLES LIKE 'innodb\_locks\_unsafe\_for\_binlog';查看间隙锁是否开启（默认为关闭）。

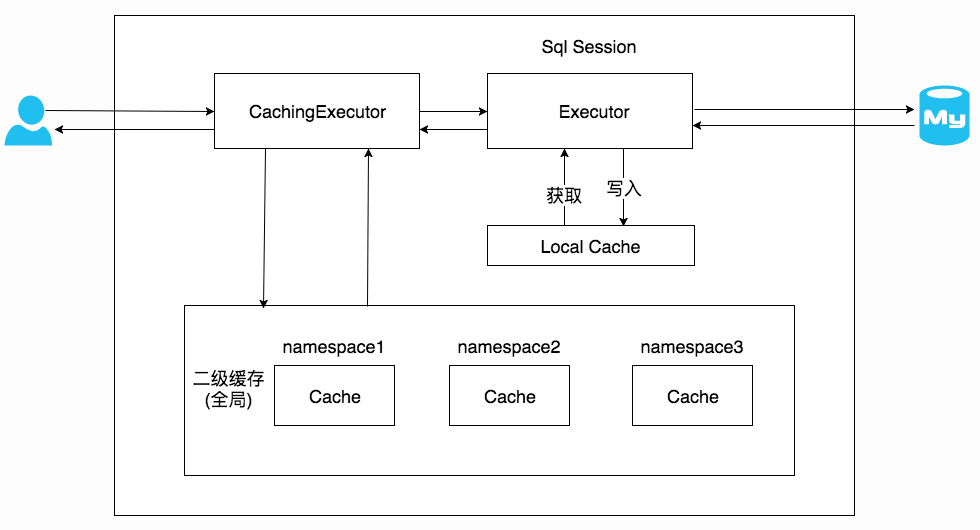
对于指定查询某一条记录的加锁语句，如果该记录不存在，会产生行锁和间隙锁，如果记录存在，则只会产生行锁。

在普通索引列上，不管是何种查询，只要加锁，都会产生间隙锁。



**MySQL缓存：**

mysql查询流程：二级缓存->一级缓存->数据库



**一级缓存：**

<setting name="localCacheScope" value="SESSION"/>

默认值为session，即在一个sqlSession（一次数据库会话(事务)）中的缓存会共享。

另一个值为STATEMENT，即仅在单个statement（sql语句）中缓存。

当值为session时，由于一级缓存仅在单个sqlSession内部中有效，在有多个sqlSession或者分布式的情况下，会出现脏数据的问题。

一级缓存内部设计简单，只是一个没有容量限定的HashMap，在缓存的功能性上有所欠缺。

**二级缓存：**

<setting name="cacheEnabled" value="true"/>

在Mybatis配置文件中开启mysql二级缓存。

<cache blocking="" eviction="" flushInterval="" readOnly="" size="" type=""/>

在mybatis映射文件中使用cache标签声明这个namespace使用二级缓存，并且可以自定义配置。

type：cache使用的类型，默认是PerpetualCache。

SynchronizedCache：同步Cache，实现比较简单，直接使用synchronized修饰方法。

LoggingCache：日志功能，装饰类，用于记录缓存的命中率，如果开启了DEBUG模式，则会输出命中率日志。

SerializedCache：序列化功能，将值序列化后存到缓存中。该功能用于缓存返回一份实例的Copy，用于保存线程安全。

LruCache：采用了LRU算法的Cache实现，移除最近最少使用的Key/Value。

PerpetualCache： 作为为最基础的缓存类，底层实现比较简单，直接使用了HashMap。

eviction： 定义回收的策略，常见的有FIFO，LRU。

flushInterval： 配置一定时间自动刷新缓存，单位是毫秒。

size： 最多缓存对象的个数。

readOnly： 是否只读，若配置可读写，则需要对应的实体类能够序列化。

blocking： 若缓存中找不到对应的key，是否会一直blocking，直到有对应的数据进入缓存。

<cache-ref namespace=""/>

在mybatis映射文件中使用cache-ref标签代表引用其它namespace的cache配置，并且两个namespace使用同一个cache。

二级缓存相对于一级缓存来说，实现了SqlSession之间缓存数据的共享，同时粒度更加的细，能够到namespace级别，通过Cache接口实现类不同的组合，对Cache的可控性也更强。

在分布式环境下，由于默认的MyBatis Cache实现都是基于本地的，分布式环境下必然会出现读取到脏数据，需要使用集中式缓存将MyBatis的Cache接口实现，有一定的开发成本，直接使用Redis、Memcached等分布式缓存可能成本更低，安全性也更高。

遇到的问题：

User:{ id:1, userName: "张三", age: 16}

User user = userMapper.getById(userId);

System.out.print(user.getUserName());

user.setUserName("李四");

User userNew = userMapper.getById(userId);

System.out.print(userNew.getUserName());

输出结果：张三李四

Id为1的user没有更新，正常情况下两次输出应该都为“张三”，但是因为mybatis缓存的原因，导致第二次输出结果为“李四”。

**锁：**

**乐观锁、悲观锁：**

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。适用于多读的应用类型，可以提高吞吐量。

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作。

**锁机制：**

共享锁（读锁）：其他事务可以读，但不能写。(类似于ReentrantReadWriteLock中的ReadLock)

排他锁（写锁）：其他事务不能读取,也不能写。(类似于ReentrantReadWriteLock中的WriteLock)

**粒度锁：**

表锁、页锁、行锁。

表级锁：开销小、加锁快、不会死锁、并发度低。

行级锁：开销大、加锁慢、会出现死锁、并发度高。

**MyISAM的表锁：**

表共享读锁、表独占写锁。

MyISAM会自动加锁，所以用户一般不需要直接用 LOCK TABLE 命令给表显式加锁。在自动加锁的情况下，MyISAM 总是一次获得 SQL 语句所需要的全部锁，所以 MyISAM 表不会出现死锁的情况。

默认情况下，写锁比读锁具有更高的优先级：当一个锁释放时，这个锁会优先给写锁队列中等候的获取锁请求，然后再给读锁队列中等候的获取锁请求。所以MyISAM不适用于有大量更新与查询操作的应用，因为大量的更新操作会造成查询操作很难获得读锁，从而可能永远阻塞。同时，一些需要长时间运行的查询操作，也会使写线程“饿死”。

MyISAM支持并发插入，以减少给定表的读和写操作之间的争用：可以在其他线程进行读操作的时候，同时将行插入到MyISAM表中。

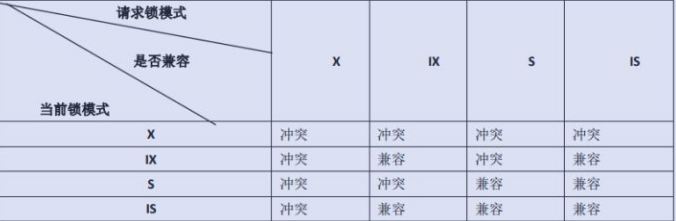
通过concurrent\_insert参数设置，默认为1，即在表中没有间隙时(表的中间没有被删除的行)，允许一个线程读表的同时，另一个线程从表尾插入记录。0：不允许并发插入。2：始终允许并发插入。

**InnoDB的表锁及行锁：**

行锁：共享锁（S）、排他锁（X）。

表锁：意向共享锁（IS）：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的 IS 锁。

意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的 IX 锁。



如果一个事务请求的锁模式与当前的锁兼容， InnoDB 就将请求的锁授予该事务； 反之， 如果两者不兼容，该事务就要等待锁释放。

InnoDB会自动加意向锁。

对于update、insert、delete，会自动加排他锁。

对于select 不会加任何锁。

可以显示添加共享锁或排他锁（注意容易造成死锁）：SELECT …. LOCK IN SHARE MODE加上共享锁，SELECT … FOR UPDATE加上排他锁。

InnoDB的锁只有在事务执行完commit或者rollback后才会释放，并且所有的锁在同一时刻释放。

InnoDB的行锁是基于索引实现的，只有通过索引检索数据才使用行级锁，否则使用表锁。实际上是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫描效率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 将使用表锁，而不是行锁。

**InnoDB的间隙锁：**

当使用范围条件而不是相等条件检索数据（查询参数有索引），并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key锁）。

使用相等条件检索数据时，next-key锁定一个索引记录和它之前的间隙。

在实际应用开发中，尤其是并发插入比较多的应用，要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。

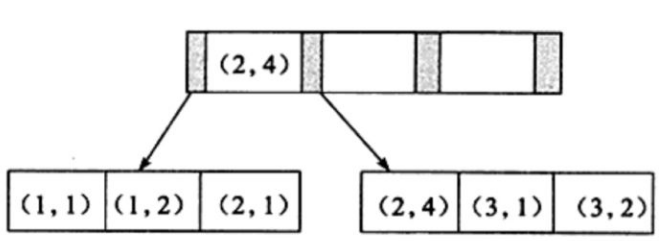
使用间隙锁的原因：

1、防止幻读，使事务隔离级别到达Serializable。

2、满足恢复和复制的需要。

Mysql的恢复和复制都是基于Binlog，Binlog是按照事务的提交先后顺序记录的，所以在一个事务未执行完成时，其他并发事务不能插入满足其锁定条件的任何记录，也就是不允许出现幻读。

**联合索引的存放方式：**



**InnoDB的auto\_increment原理：**

使用auto\_increment必须具有唯一索引，必须设置为not null。

auto\_increment计数器仅存储在内存中，不在磁盘中存储。

所以，当服务重启后，对表进行插入操作时，需要对auto\_increment计数器进行初始化。

初始化InnoDB执行等价如下语句的动作：SELECT MAX(ai\_col) FROM T FOR UPDATE;，会获取表独占锁，并且持续到事务结束。

计数器被初始化之后，如果用户插入一个明确指定该列值的行，而且该值大于当前计数器值，则计数器被设置为指定列值。如果没有明确指定一个值，InnoDB给计数器增加一，并且赋新值给该列。

计数器被初始化之后，当访问计数器时，InnoDB会使用专用的表级AUTO-INC锁，该锁持续到当前SQL语句的结束而不是事务的结束。（所以若事务回滚，则字段值可能出现间隙）

**InnoDB的AUTOCOMMIT和MVCC（多版本并发控制multi-versioned concurrency control）：**

当AUTOCOMMIT设置为1，没有显示开启事务时，每个sql都被视作一个事务。

当AUTOCOMMIT设置为0，没有显示开启事务时，所有sql的执行都被视作在同一个事务中，直到显式调用commit或rollback，此时会结束当前事务并默认开启一个新事务。

MVCC是一种用来解决读-写冲突的无锁并发控制的机制。

MVCC只在READ COMMITTED和REPEATABLE READ两个隔离级别下工作。READ UNCOMMITTED总是读取最新的数据行，而不是符合当前事务版本的数据行。而SERIALIZABLE 则会对所有读取的行都加锁。

InnoDB的MVCC，是通过在每行记录后面保存两个隐藏的列来实现的,这两个列，分别保存了这个行的创建时间（系统版本号），一个保存的是行的删除时间（系统版本号）。（若无主键则多有一个行id列，用于加快查询）

Insert:

InnoDB为新插入的每一行保存当前系统版本号作为创建时间。

Select:

InnoDB只会查找行的创建时间系统版本号小于或等于当前(可见)系统版本号，并且删除时间版本号为未定义或大于当前(可见)系统版本号的数据。这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的。

Delete：

InnoDB为删除的每一行保存当前系统版本号作为删除时间。

Update：

InnoDB执行UPDATE，实际上是新插入了一行记录，并保存其创建时间为当前系统版本号，同时保存当前系统版本号要UPDATE的行的删除时间。



<https://zhuanlan.zhihu.com/p/66791480>;

READ VIEW相关(当前可见系统版本号)