MySQL InnoDB存储引擎

前言

总结的顺序按照《MySQL技术内幕InnoDB存储引擎第2版》来编写。蓝色部分是本人觉得的重点，非蓝色部分写出来的是本人觉得可以很好的理解蓝色部分所添加的。有些部分虽然认为是非重点就没有添加，以后在面经上看到会再次添加。

书上很多缩写名词，先不管它具体含义，继续往后看。

1. MySQL体系结构与存储引擎

1.1 定义数据库与实例

数据库：物理操作系统文件或其他形式文件类型的集合。在 MySQL数据库中，数据库文件可以是frm、MYD、MYI、ibd结尾的文件。当使用NDB引擎时，数据库的文件可能不是操作系统上的文件，而是存放于内存之中的文件，但是定义仍然不变。

实例：MySQL数据库由后台线程以及一个共享内存区组成。共享内存可以被运行的后台线程所共享。需要牢记的是，数据库实例才是真正用于操作数据库文件的。

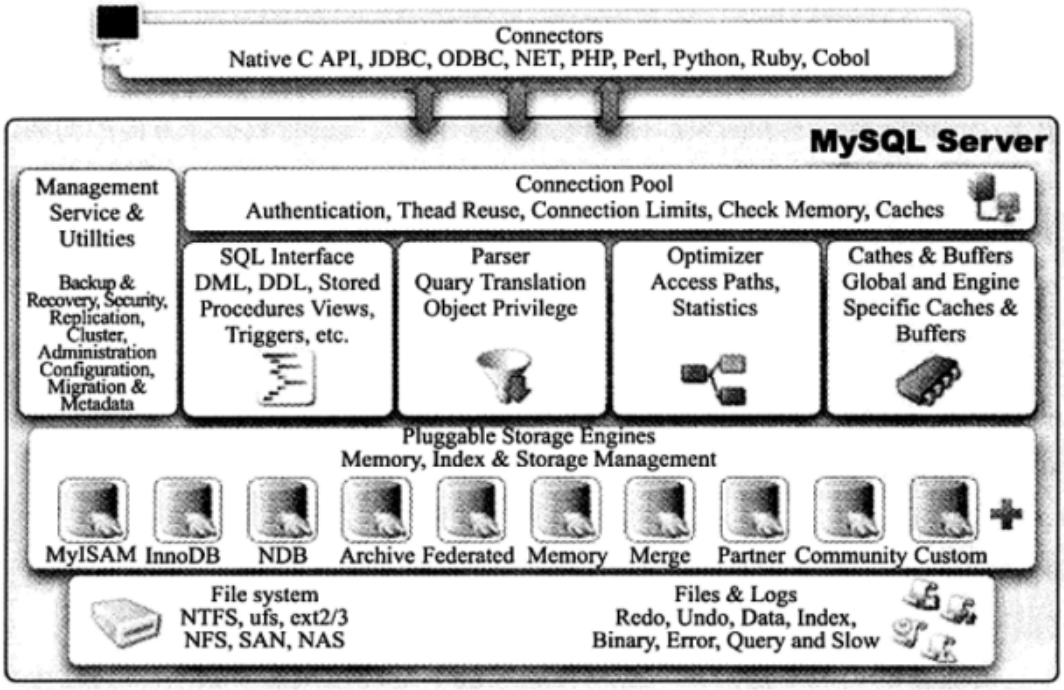
在MySQL数据库中，实例与数据库的关系是一一对应的。通常一个实例对应一个数据库，一个数据库对应一个实例；在集群中，可能存在一个数据库被多个实例使用的情况。

MySQL数据库实例在系统上的表现就是一个进程。它是一个单进程多线程的一个架构。

1.2 MySQL体系结构

从概念上来说，数据库是文件的集合，是依照某种数据模型组织起来并存放二级存储器中的数据集合；数据库实例是程序，是位于用户和操作系统之间的一层数据管理软件。

用户对数据库数据的任何操作（数据库定义，数据查询，数据维护，数据库运行控制）都是在数据库实例下进行的，应用程序只有通过数据库实例才能和数据库打交道。



如图所示MySQL由下面几部分组成（从上到下，从左到右）：

1）连接池组件

2）管理服务和工具组件

3）SQL接口组件

4）查询分析器组件

5）优化器组件

6）缓冲组件

7）插件式存储引擎

8）物理文件

其中，插件式的表存储引擎是MySQL数据库区别于其他数据库的最重要特点。MySQL插件式的存储引擎架构提供了一系列标准的管理和服务支持，这些标准与存储引擎本身无关，可能是每个数据库系统所必须的（比如SQL分析器和优化器）。而存储引擎是底层物理结构的实现。

存储引擎是基于表的，而不是数据库。

1.3 MySQL存储引擎

存储引擎是MySQL区别于其他数据库的一个最重要特性。存储引擎的好处是：每个存储引擎都有各自的特点，能够根据不同的应用场建立不同的引擎表。

1.3.1 InnoDB存储引擎

InnoDB存储引擎支持事务，设计的主要目的是面向在线事务处理（OLTP）的应用。其特点是行锁设计，支持外键，并支持类似于Oracle的非锁定读，即默认读操作不会产生锁。从5.5.8版本开始，InnoDB存储引擎是默认的存储引擎。

InnoDB存储引擎将数据放入一个逻辑的表空间中，这个表空间就像黑盒一样由InnoDB存储引擎自身进行管理。从MySQL4.1版本开始，它可以将每个InnoDB存储引擎的表单独存放到一个独立的ibd文件中。

InnoDB通过多版本并发控制（MVCC）来获得高并发性，并且实现了SQL标准的4种隔离级别，默认为REPEATABLE（可重复读）级别。同时，还使用了一种称为next-key locking的策略来避免幻读现象的产生。

对于表中的数据，InnoDB存储引擎采用聚集的方式，因此每张表的存储都是按主键的顺序进行存放。如果没有显式地在表中定义主键，InnoDB存储引擎会为每一行生成一个6字节的ROWID，并以此为主键。

1.3.2 MyISAM存储引擎

MyISAM存储引擎不支持事务、表锁设计，支持全文索引，主要面向一些OLAP数据库应用。在MySQL5.5.8版本以前，MyISAM存储引擎是默认的存储引擎（windows系统除外）。MyISAM存储引擎的另一个与众不同的地方是它的缓冲池只缓存索引文件，而不缓冲数据文件，这点和大多数数据库大不相同。

MyISAM存储引擎表由MYD和MYI组成。MYD用来存放数据文件，MYI用来存放索引文件。

1.3.3 InnoDB和MyISAM的区别

	MyISAM	InnoDB
事务	不支持	支持
外键	不支持	支持
锁	支持表锁。由于最小粒度是表锁，因此在更新字段的时候会将整个表都锁住，导致其他线程查询该表受限，因此不支持并发	支持行锁，由于最小粒度是行锁，可以保证在更新一行字段的时候其他线程访问该表，这个也是现在MySQL默认为InnoDB的原因。但是在执行一个SQL语句时MySQL不能确定要扫描的范围时，也会将行锁升级成表锁。
索引	非聚集索引。由于使用非聚集索引，因此索引的叶子节点上存放的就是索引和数据文件的指针。	聚集索引。由于使用的是聚集索引，因此索引的叶子节点存放的就是多个一整行的数据，数据的开头是主键的值。

注意：在索引的问题上，书上不看仔细的话会有些误会，这里按照我自己的理解来说。

在书上的后面部分提到了一点，非聚集索引又叫辅助索引，而InnoDB是有辅助索引的，但是为什么统称InnoDB是聚集索引?

索引确实分为聚集索引和非聚集索引（这里只是按照一般的分，如果按照职能还有其他的）。

MyISAM之所以成为非聚集索引，是因为MyISAM本身可以不需要主键，加上叶子节点本身存储的是索引（这个索引是自己创建的）和数据文件的指针（即存储指向每一行的地址），所以无法定义MyISAM使用的聚集索引;

InnoDB本身在没有定义主键的时候，会给每一行生成一个6字节的ROWID，并以此为主键，主键就是索引，加上存储的是一整行数据（叶子节点可以看做一张表，表的第一行就是主键，后面几行就是表中每一列的值），因此使用的是聚集索引。

因此，这里的区别中写的索引区别，在于InnoDB和MyISAM默认的索引结构，InnoDB表创建好就存在聚集索引，MyISAM需要自己去创建索引。两个索引的共同点在于，叶子节点的第一行都是两个结构表的索引；不同点在于，聚集索引叶子节点的第一行后面存放的一整行数据，而非聚集索引的第一行后面存放的到哪（指针）可以找到与索引相对应的行数据。

具体的索引结构将在后面总结。

总结：InnoDB也支持非聚集索引（CREATE INDEX index_name ON table_name (column_list)），这里的非聚集索引就是我们说的普通索引。但是记区别要说InnoDB本身是聚集索引。

2. InnoDB存储引擎

2.1 InnoDB体系架构

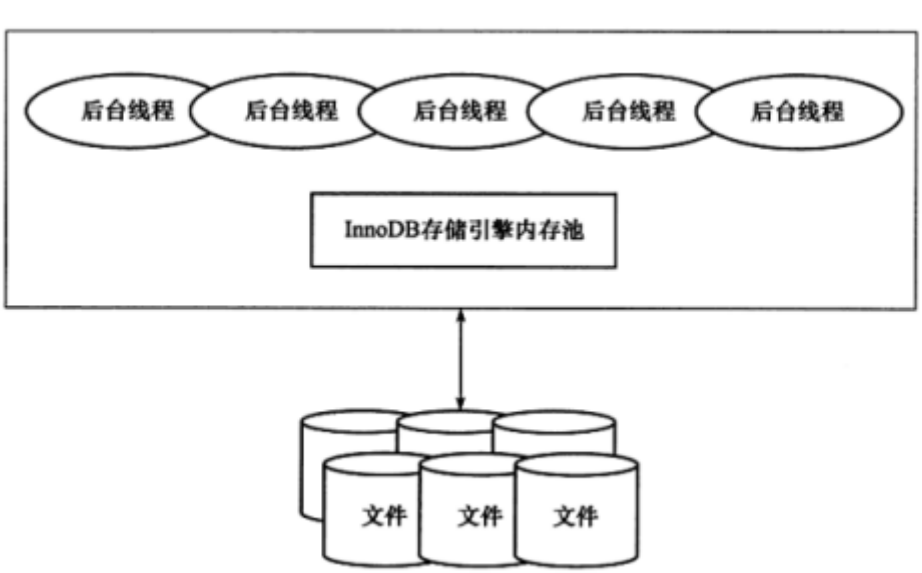
InnoDB引擎存储有多个内存块（可以理解为每个后台线程占用的内存就是一个内存块），可以认为这些内存块组成了一个大的内存池，负责如下工作：

1）维护所有进程/线程需要访问的多个内部数据结构。

2）缓存磁盘上的数据，方便快速的读取，同时在对磁盘文件的数据修改之前在这缓存。

3）重做日志（redo log）缓冲

……



后台线程的主要作用是负责刷新内存池中的数据，保证缓冲池中的内存缓存的是最近的数据。此外将已修改的数据文件刷新到磁盘文件，同时保证在数据库发生异常时InnoDB能恢复到正常运行状态。

2.1.1 后台线程

InnoDB引擎是多线程的模型，因此后台有多个不同的后台线程，负责处理不同的任务。

（1）Master Thread

Master Thread是一个非常核心的后台线程，主要负责将缓冲池中的数据异步刷新到磁盘，保证数据的一致性，包括脏页的刷新、合并插入缓冲、undo页的回收等（基本上有其他后台线程的功能）。

（2）IO Thread

在InnoDB存储引擎中大量使用了AIO来处理写IO请求，这样可以极大提大数据库的性能。而IO Thread的工作主要是负责这些IO请求的回调处理。

（3）Purge Thread

事务被提交后，其所使用的undo log可能不再需要。因此需要Purge Thread来回收已经使用并分配的undo 页。

（4）Purge Cleaner Thread

该线程的作用是将之前版本中脏页的刷新操作都放入单独的线程中来完成。

2.1.2 内存

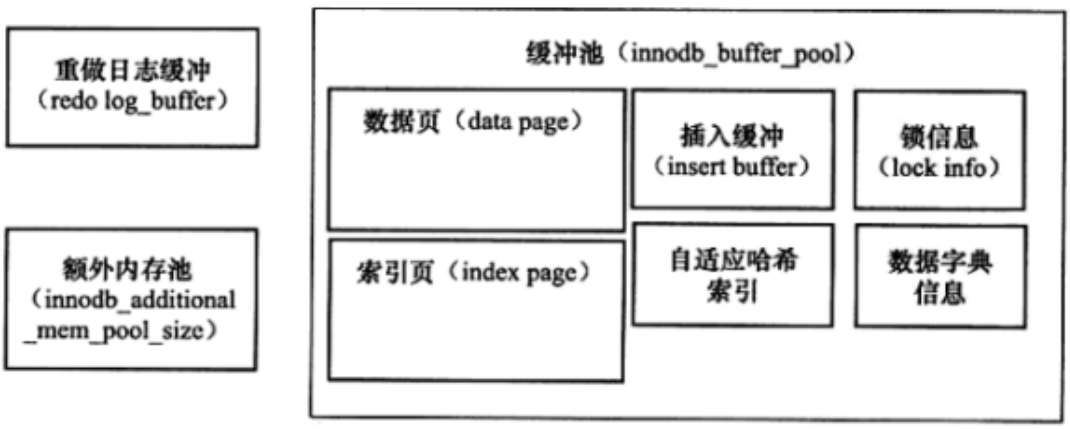
（1）缓冲池

InnoDB是基于磁盘存储的，并将其中的记录按照页的方式进行管理。因此可将其视为基于磁盘的数据库系统。在数据库系统中，由于CPU速度和磁盘速度之间的差距，基于磁盘的数据库系统通常使用缓冲池来提高数据库性能。

缓冲池是一块内存区域，通过内存的速度来弥补磁盘速度较慢对数据库性能的影响。在数据库中进行读取页的操作，首先将从磁盘读到的页存放在缓冲池中，这个过程称为将页"FIX"在缓冲池中。下一次再读相同的页时，首先判断该页是否在缓冲池中。若在缓冲池中，称该页在缓冲池中被命中，直接读取该页。否则，读取磁盘上的页。对于数据库中页的修改操作，则首先修改在缓冲池中的页，然后再以一定的频率刷新到磁盘上。这里需要注意的是，页从缓冲池刷新回磁盘的操作并不是在每次页发生更新时触发，而是通过一种称为 Checkpoint 的机制刷新回磁盘。同样，这也是为了提高数据库的整体性能。

（上面缓冲池工作和TLB很像）

具体来看，缓冲池中缓存的数据页类型有：索引页、数据页、undo页、插入缓冲（insert buffer）、自适应哈希索引（adaptive hash index）、InnoDB存储的锁信息（lock info）、数据字典信息（data dictionary）等。不能简单地认为，缓冲池只是缓存索引页和数据页，它们只是占缓冲池很大的一部分而已。



（2）LRU List、Free List和Flush List

这个不重点总结，大概就是，通过LRU算法来管理缓冲池；如果数据库刚启动，LRU列表是空的，这时页都放到Free列表中；Flush列表里都放着脏页列表（LRU列表里也有脏页，脏页就是内存数据页和磁盘数据页内容不一致），LRU列表用来管理缓冲池中页的可用性，Flush列表用来管理将页刷新回磁盘，二者互不影响。

（3）重做日志缓冲

InnoDB存储引擎首先将重做日志信息先放入到这个缓冲区，然后按一定频率将其刷新到重做日志文件。重做日志缓冲一般不需要设置得很大，因为一般情况下每一秒钟会将重做日志缓冲刷新到日志文件，因此用户只需要保证每秒产生的事务量在这个缓冲大小之内即可。

（4）额外内存池

在 InnoDB存储引擎中，对内存的管理是通过一种称为内存堆（heap）的方式进行的。在对一些数据结构本身的内存进行分配时，需要从额外的内存池中进行申请，当该区域的内存不够时，会从缓冲池中进行申请。

2.2 Checkpoint技术

倘若有一条DML语句，如Update或Delete改变了页中的记录，那么此时页是脏的，即缓冲池中的页的版本要比磁盘的新。数据库需要将新版本的页从缓冲池刷新到磁盘。

倘若每次一个页发生变化，就将新页的版本刷新到磁盘，那么这个开销是非常大的。若热点数据集中在某几个页中，那么数据库的性能将变得非常差。同时，如果在从缓冲池将页的新版本刷新到磁盘时发生了宕机，那么数据就不能恢复了。

为了避免发生数据丢失的问题，当前事务数据库系统普遍都采用了Write Ahead Log策略，即当事务提交时，先写重做日志，再修改页。当由于发生宕机而导致数据丢失时，通过重做日志来完成数据的恢复。这也是事务ACID中D（Durability持久性）的要求。

Checkpoint（检查点）主要解决以下几个问题：

1） 缩短数据库的恢复时间

2） 缓冲池不够用时，将脏页刷新到磁盘

3） 重做日志不够用时，刷新脏页

缩短恢复时间主要应对如下场景：当数据库运行了几个月甚至几年，这时突然发生了怠机，重新应用重做日志的时间会非常久，此时恢复的代价会非常大。

使用Checkpoint检查点后：当数据库发生怠机后，数据库不需要重做所有日志，因为Checkpoint之前的页都刷新回磁盘。故，数据库只需要对Checkpoint后的重做日志进行恢复。

2.3 InnoDB关键特性

InnoDB的关键特性正是InnoDB现在成为默认引擎的原因。很多地方没有加蓝色字体，但是这节看懂了对后面的内容帮助很大。

2.3.1 插入缓冲（insert buffer）

insert buffer是InnoDB独有的功能。这个功能在于提高了非聚集索引的插入性能。

前面讲过，InnoDB默认的是聚集索引，假设有如下的表结构：

```
create table t(
    ID int auto_increment,
    name varchar(20),
    primary key(id)
);
```

对于上述表结构，插入一行或多行数据是很快的，因为主键在默认情况下是自增的，也就是说，数据页中的记录是按ID顺序存放的。

但是假设存在非聚集索引，索引为name：

```
create table t(
    ID int auto_increment,
    name varchar(20),
    primary key(id),
    key(name)
);
```

这个时候发现，数据页的存放仍然是按照ID主键顺序存放（因为插入之后，表还是按照ID的顺序来排序），但是对于非聚集索引叶子节点的插入却不再是顺序的了，这时就要离散的访问非聚集索引页，插入性能降低。（前面提到过，非聚集索引叶子节点的第一行是索引，在上面例子中，第一行就是name，不是顺序的，而构建成非聚集索引B+树叶子节点是顺序的。如果按照字母顺序，就要离散的访问各个叶子节点的索引再插入）

当然，并不是所有情况下的非聚集索引都是无序的，比如ID插入后，多一条时间记录，那么以该时间记录为非聚集索引，就是顺序的。

那么，为了解决这个非聚集索引插入性能问题，InnoDB设计出了插入缓冲技术。对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚集索引页是否在缓冲池中，若在，则直接插入；若不在，则先放入到一个Insert Buffer对象中。然后再以一定的频率和情况进行Insert Buffer和辅助索引页子节点的 merge（合并）操作（这个操作是由后台线程根据一些算法来做的），这时通常能将多个插入合并到一个操作中，这就大大提高了对于非聚集索引插入的性能。

因此，insert buffer的好处显而易见：减少了磁盘的离散读取，将多次插入合并成一次操作。

如果一次次插入，相当于假设本身在一个索引页中的数据（假设该页有5条数据），分5次插入，进行了5次磁盘读取；而如果先放到Insert Buffer对象中，假设先放入了5条数据到insert buffer中，这5条数据万一是一个索引页中的，后面后台线程只需要进行一次离散访问，就算是最差情况也和最初一样为5次。

Insert Buffer 的使用需要同时满足以下两个条件：

1）索引是辅助索引（secondary index）；

2）索引不是唯一（unique）的。

当满足以上两个条件时，InnoDB存储引擎会使用Insert Buffer，这样就能提高插入操作的性能了。不过考虑这样一种情况：应用程序进行大量的插入操作，这些都涉及了不唯一的非聚集索引，也就是使用了Insert Buffer。若此时 MySQL数据库发生了宕机，这时势必有大量的Insert Buffer并没有合并到实际的非聚集索引中去。因此这时恢复可能需要很长的时间，在极端情况下甚至需要几个小时。

辅助索引不能是唯一的，因为在插入缓冲时，数据库并不去查找索引页来判断插入的记录的唯一性。如果去查找肯定又会有离散读取的情况发生，从而导致 Insert Buffer失去了意义。

Insert Buffer的结构是B+Tree。

2.3.2 两次写（doublewrite）

Insert buffer带来的是InnoDB性能上的提升，而两次写则是带来InnoDB数据页的可靠性。

页是InnoDB磁盘管理的最小单位，InnoDB将内存中的数据写到磁盘是以数据页的大小为单位来写的。一般数据页的大小是16kb，而操作系统写文件是以4kb为单位，也就是说，操作系统需要写4次，才能组成一个页的数据到表中。

假设现在有16kb正好一个页的数据要写到磁盘，操作系统只写了一次，也就是4kb，就断电了或者系统崩溃了，也就是说只有部分页是写入的，这种情况称为部分写失效（partial page write），这个页是个不完整的页，页中数据可能会出现不一致的情况。

而这个页失效，没法通过redo log（重做日志）的方式来恢复。这是因为就算是重做日志，也是以页为单位来恢复了，对于页本身无法恢复。

想要解决这个问题，就是在写数据页之前，先把这个数据页写到一块独立的物理文件位置（ibdata），然后再写到数据页。这样一旦出现怠机情况，如果出现数据页损坏，在使用redo log之前，先利用这个副本的页还原页，再redo log重做。上面这个过程就是double write。

2.3.3 自适应hash索引

hash是一种非常快的查找方法（O（1）），即一般一次查询；而B+树的查找取决于树的高度，一般需要3~4次查询。

InnoDB存储引擎会监控对表上各索引页的查询。如果发现建立hash索引可以带来速度上的提升，就会建立hash索引，这个就称为自适应hash索引（adaptive hash index，AHI）。

哈希索引始终基于现有的 InnoDB 辅助索引构建。也就是说，InnoDB会不断监控对辅助索引的查询，当发现某辅助索引被频繁访问，成为了热数据，就会为这个辅助索引生成一个hash索引。而创建这个hash索引很快，因为自适应hash索引就是通过B+树构建来的。

（辅助索引（非聚集索引）本质也是一颗B+树，因此也需要3~4次查询才能定位到主键，最后再通过聚集索引再查询3~4次定位到某一页（叶子节点本身是一段，一段里面有多个页），这样就需要6~8次才能查出数据；而使用hash索引后，只需要4~5次就能查询出数据）

AHI有个要求，即对这个页的连续访问模式必须是一样的。例如对于(a，b)这个联合索引（不明白联合索引可以先看索引那节关于联合索引的概念）页可以这么访问：

```
select * from t where a= xxx and b = xxx;
select * from t where a= xxx ;
```

访问模式一样指的就是查询条件一样，也就是where后面的条件一样。如果上面两条sql语句是交替查询的，那么InnoDB不会构建AHI。

另外AHI和一般的hash索引是一样的，也不能进行范围查询。范围查询也是B+树的一个优点。

2.3.4 异步IO

为了提高磁盘操作性能，当前数据库系统都采用的异步IO（Asynchronous IO，AIO）的方式来处理磁盘操作。InnoDB也是一样。

与AIO相对应的就是同步IO。比如用户发出的是一条索引扫描的查询，假设这个扫描要扫描多个页，那么同步IO的情况下就是先发起一个IO请求，扫描完后再发起新的IO请求，直到扫描完毕。但是AIO不同，AIO就是先发一个IO请求，不等它完成，再发起对新的页进行扫描的请求。在所有的IO请求发送后，只需要等待IO操作的完成。

AIO的另一个优势就是可以进行IO Merge操作，也就是将多个IO合并为1个。例如用户需要访问（space，page_no）为

（8,6），（8,7），（8,8）

每个页的大小是16kb，那么同步IO操作要经过3次。而AIO会判断出这三个页是连续的，因此AIO会发送一个IO请求，从（8,6）开始直接读取48kb的数据。

2.3.5 刷新邻接页

InnoDB存储引擎还提供了Flush Neighbor Page（刷新邻接页）的特性。工作原理为：当刷新一个脏页时，InnoDB会检测脏页所在的区（一个区相当于几个页）的所有页，如果是脏页就一起刷新。

这个操作也没有难度，前面说了AIO可以合并多个IO，可以利用这个特性来解决检测区的问题。

但是仍然需要考虑两个问题：

（1） 是不是可能将不怎么脏的页进行写入，而这个页写入之后，又很快变成了脏页

（2） 固态硬盘有较高的IOPS（IOPS是每秒进行IO的次数，固态的IO次数是机械磁盘的几倍）,是否需要这个特性

为此，InnoDB 存储引擎从 1.2.x 版本开始提供了参数 innodb_flush_neighbors，用来控制是否启用该特性。对于传统机械硬盘建议启用该特性，而对于固态硬盘有着超高 IOPS 性能的磁盘，则建议将该参数设置为 0、即关闭此特性。

3. 文件

本节没有蓝色标注，希望以理解为主。

3.1 日志文件

日志文件记录了影响MySQL数据库的各种类型活动。MySQL数据库中常见的日志文件有：

1）错误日志

2）二进制日志

3）慢查询日志

4）查询日志

3.1.1 错误日志

错误日志文件对MySQL的启动、运行、关闭过程进行了记录。该文件不仅记录了所有的错误信息，也记录一些警告信息或正确信息。

3.1.2 慢查询日志

慢查询日志可以帮忙DBA定位可能存在问题的SQL语句，从而进行SQL语句层面上的优化。例如：在MySQL启动时设一个阈值，将运行时间超过该值的所有SQL语句都记录到该日志文件中。DBA每天检查一次该日志文件，优化存在问题的SQL语句。

3.1.3 查询日志

查询日志记录了所有对MySQL数据库请求的信息，无论这些请求是否得到了正确的执行。

3.1.4 二进制日志

二进制日志记录了对MySQL数据库执行更改的所有操作，但不包括SELECT和SHOW这类操作，因为这类操作对数据本身没有修改，这类操作应该被记录到查询日志中。

总的来说二进制日志主要有以下作用：

1）恢复。某些数据的恢复可能需要二进制日志。例如，在一个数据库全备文件恢复后，用户可以通过二进制日志进行point-in-time的恢复。

2）复制。原理与恢复类似，通过复制和执行二进制日志使一台远程的MySQL数据库（一般称为slave或standby）与一台MySQL数据库（一般称为master或primary）进行实时同步。

3）审计。用户可以通过二进制日志里的信息进行审计，判断是否有对数据库进行注入式攻击的操作。

3.2 InnoDB存储引擎文件

3.2.1 表空间文件

InnoDB采用将存储的数据按表空间进行存放的设计。默认配置下，会有一个初始化大小为10MB、名为ibdata1的文件。该文件就是默认的表空间文件（tablespace file）。

注意，这个默认的表空间文件是共享表空间文件。在设置单独的表空间文件后，InnoDB将会对每张表都创建一个表空间文件。这样，用户就不需要将所有的数据都放入默认表空间中。

单独表空间文件仅储存该表的数据、索引和插入缓冲BITMAP等信息，其他信息比如回滚信息（undo）、插入缓冲索引页、系统事务信息、二次写缓冲等还是存放在默认的表空间中。

3.2.2 重做日志文件

默认情况下，InnoDB存储引擎的数据存放目录下有两个名为ib_logfile0和ib_logfile1文件，这两个文件就是重做日志文件（redo log file），该文件记录了对于InnoDB存储引擎的事务日志。

当实例或介质失败（media failure）时，可以通过重做日志文件进行数据恢复，用来保证InnoDB表存储的数据不会轻易因为宕机而丢失。例如，数据库由于所在主机掉电导致实例失败，InnoDB存储引擎会使用重做日志恢复到掉电前的时刻，以此来保证数据的完整性。（前面提到过的InnoDB的关键特性两次写就是这个）

既然都是记录事务日志，那么和前面提到的二进制日志有什么区别呢?

1）二进制日志记录了所有与MySQL数据库有关的日志记录，包括InnoDB。而InnoDB存储引擎的重做日志文件只存储和InnoDB相关的该引擎本身的事务日志。

2）记录的内容不同。二进制文件记录的是关于一个事务的具体操作内容，即该日志是逻辑日志。而InnoDB存储引擎的重做日志记录的是关于每个页的更改物理情况。

3）写入时间不同。二进制文件仅在事务提交前进行提交，即只写入磁盘一次。而在事务进行的过程中，却不断有重做日志条目被写入到重做日志缓冲（redo log buffer）中，之后再持久化到磁盘。

4. 表

4.1 索引组织表

在InnoDB存储引擎中，表都是根据主键顺序组织存放的，这种存储方式的表称为索引组织表。在InnoDB存储引擎表中，每张表都有个主键，如果没有显式的创建主键，则InnoDB存储引擎会按如下方式选择或者创建主键：

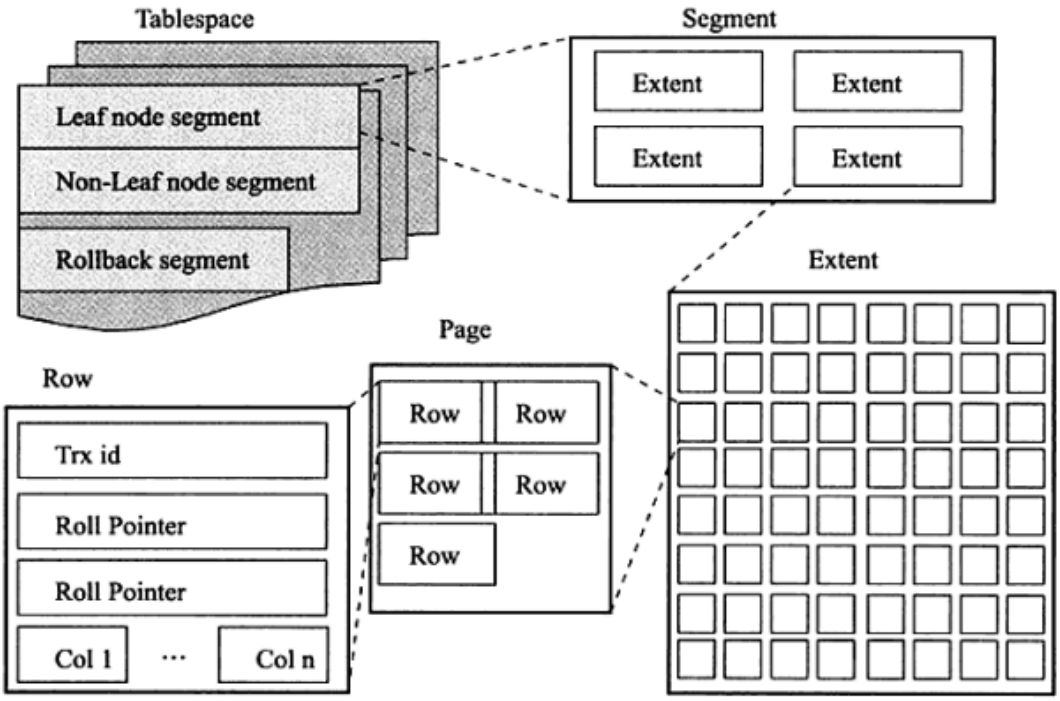
（1） 首先判断表中是否有非空的唯一索引，如果有，则该列为主键。

（2） 如果没有，InnoDB会自动创建一个6字节大的指针作为索引

当表中有多个非空唯一索引时，InnoDB将选择建表时的第一个定义的非空唯一索引作为主键。注意，索引的选择为定义索引的顺序。

4.2 InnoDB逻辑存储结构

从InnoDB存储引擎的逻辑结构来看，所有的数据都被逻辑的放到一个空间中（前面提到过），称为表空间。表空间又由段、区、页组成。如图：



4.2.1 段

表空间是由各个段组成的，常见的段有数据段，索引段，回滚段等。InnoDB存储引擎表是索引组织的，因此数据即索引，索引即数据。数据段就是索引表B+树叶子节点（根据B+树结构，实际上索引数据也在叶子节点），索引段就是非叶子节点。

4.2.2 区

区是由连续的页组成的，在任何情况下每个区的大小为1MB，在默认情况下，一个页是16kb（页可以设置为2,4,8kb），那么一个区就有64个页。

4.2.3 页

页是InnoDB磁盘管理的最小单元。

5. 索引

5.1 InnoDB索引概述

InnoDB存储引擎支持以下几种常见的索引

（1）B+树索引

（2）全文索引

（3）hash索引

前面提到过，InnoDB的自适应hash索引是InnoDB的关键特性，InnoDB引擎会监控辅助索引，如果一旦发现某辅助索引变成热点，就自动（不能人为）生成该辅助索引的自适应hash索引。

B+树（balance tree）索引就是传统意义上的索引。B+树索引类似二叉树，根据键值快速找到数据。

B+树索引并不能找到一个给定键值的具体行。B+树索引只能找到被查找数据行所在的页。然后数据库通过把页读到内存，再在内存中查找具体的行。

5.2 B+树索引

数据库中B+树索引可以分为聚集索引和非聚集索引（辅助索引）。两者不同的是，叶子节点上是否存储着一整行的信息。

5.2.1 聚集索引

InnoDB存储引擎表是索引组织表，即表中数据按照主键顺序存放。而聚集索引就是按照每个表的主键构造成一颗B+树，同时叶子节点存放的即为整张表的行记录，也将聚集索引的叶子节点称为数据页。聚集索引的这个特性也决定了索引组织表中的数据也是索引的一部分。同B+树数据结构一样，每个数据页都通过一个双向链表来进行连接。

在多数情况下，查询优化器倾向于采用聚集索引，因为聚集索引能在B+树的叶子节点上直接找到数据。此外，由于定义了数据的逻辑顺序，聚集索引能很快的访问针对范围的查询。（即一条SQL语句的查询，如果where后面的条件是主键，就查询的特别快）

聚集索引的另一个好处是，它对于主键的排序查找和范围查找速度非常快。叶子节点上的数据就是要查找的数据。比如用户需要查找最后注册的10位用户，由于B+树索引是双向链表，用户可以快速找到最后一个数据页，然后取出10条记录。

补充：

范围查询是B+树的优点。hash的离散型意味着不能进行范围查询，且hash本身是无序的。

1. 相比于B树，B+树由于叶子节点之间用双向链表来连接，因此在做范围查询的时候不需要通过中序遍历（中序遍历会增加查询次数，比查询整个链表要慢），只需要遍历链表。
2. 相比于B树，B+树只有叶子节点存放数据，而B树的非叶子节点也存放了数据，这就意味着：
   a. B+查询效率更加稳定，查询所有的数据都只需要查询到叶子节点，即查询的次数是一样的
   b. B+树的非叶子节点相对B树更小，这样在相同内存大小情况下，能容纳B+树的非叶子节点更多

5.2.2 非聚集索引（辅助索引）

对于辅助索引，叶子节点并不包含行记录的全部数据。叶子节点除了包含键值外，每个叶子节点中的索引行还包括了一个书签。该书签用来告诉InnoDB存储引擎到哪里可以找到与索引相对应的行记录（这个书签就类似于指向聚集索引中主键的指针，之所以是指向聚集索引，是因为只有聚集索引的叶子节点上有整行的数据）。

当通过辅助索引来寻找数据时，InnoDB存储引擎会遍历辅助索引并通过叶级别的指针获得指向主键索引的主键。举个例子来说，如果在一棵高度为3的辅助索引中查找数据，那么需要查询3次查到该数据对应的主键，然后假设聚集索引树高度也是3，那么还需要对聚集索引再查找3次才能查询到数据页，也就是说一共进行了6次IO访问，才找到数据页。

用SQL的方式来说明，类似下面的例子。假设name设置为辅助索引：

```
select * from t where name = 'xxx';
```

在进行上面语句查找数据时，实际上是先查找主键，再通过主键查找数据：

```
select id from t where name = 'xxx';
select * from t where id = 'xxx';
```

5.3 Cardinality值

并不是在所有的查询条件中出现的列都需要添加索引。一般的经验是，在访问表中很少一部分时（比如查询出的结果不是多数），使用B+树索引才有意义。比如查找性别，地区，类型等这些字段，取值范围就很小，称为低选择性：

```
select * from t where sex = 'M';
```

对于这种查询，男女比例为50%，也就是说，可能会查出整张表的50%的数据，这样B+树索引就显得没有那么必要（因为可能还是要遍历全部数据）。如果某个字段的取值范围很广，几乎没有重复，即属于高选择性，此时使用B+索引最适合。

Cardinality值表示索引中不重复记录数量的预估值。在实际应用中，Cardinality值应尽可能接近1，如果非常小，就没有必要建立索引。

5.4 B+树索引的应用

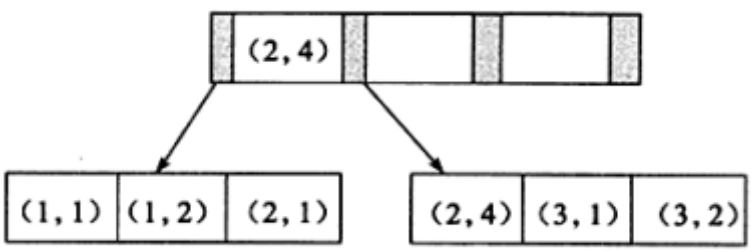
5.4.1 联合索引

联合索引是指对表上的多个列进行索引。前面讨论的情况都是对一个列进行索引。联合索引的创建方法和单个索引创建的方法一样，不同之处在于有多个索引列。例如：

```
CREATE TABLE t(
    a INT,
    b INT,
    PRIMARY KEY(a),
    KEY idx_a_b (a,b)
)ENGINE=INNODB
```

如上，创建一个表，且创建一个索引idx_a_b，这个索引就是联合索引，联合的列是a和b。

从本质上来说，联合索引也是一颗B+树，不同的是联合索引的键值数量不是1，而是大于或等于2。



如图所示，叶子节点的数据为（1,1），（1,2），（2,1），（2,4），（3,1），（3,2），数据按照（a,b）进行存放。由于a是有序的，因此像：

```
select * from t where a = xxx and b = xxx;
select * from t where a = xxx;
```

这两条查询语句是可以用（a,b）这个索引的。而：

```
select * from t where b = xxx;
```

则不能使用该索引，因为b：1,2,1,4,1,2是无序的。

但是b对于相同的a来说，又是有序的。这就是联合索引的好处。假如现在有张表

```
CREATE TABLE t(
    user_id INT,
    buy_date date,
    .....
    PRIMARY KEY(user_id),
    KEY idx_a_b (user_id,buy_date)
)ENGINE=INNODB
```

现在有个需求，让你查询最近三天时间user_id = 1的用户买了什么东西。如果不使用联合索引，则在查询出相同的user_id之后还要额外进行一次排序。但是如果用了排序索引，因为buy_date对于user_id来说又是有序的，那么就省去了排序的操作：

```
select * from t where user_id = 1 order by buy_date DESC limit 3
```

对于联合索引（a,b,c）来说，下列语句同样可以直接通过联合索引得到结果：

```
select * from t where a = xxx order by b
select * from t where a = xxx and b = xxx order by c
```

但是像：

```
select * from t where a = xxx order by c
```

就不能使用（a,b,c）来直接得到结果。这是因为b对于相同的a是排序的，c对于相同的b是排序的，而c对于相同的a并没有排序。

5.4.2 覆盖索引

InnoDB存储引擎支持覆盖索引（或称索引覆盖），即从辅助索引中就可以得到查询的记录，而不需要再去查询聚集索引。使用覆盖索引的一个好处就是辅助索引不包含整行记录的所有信息，故其大小远小于聚集索引，因此可以减少大量的IO操作。（这个减少IO操作我的理解是，首先IO操作的时间会更短，因为进入内存的数据会更少；其次由于辅助索引中已经查询到记录，并不需要再去查询聚集索引）

简单点说，覆盖索引就是在查询时，查询的列恰好是辅助索引的一部分，比如主键，和索引本身。看下面一个例子：

```
CREATE TABLE t(
    a INT,
    b INT,
    c VARCHAR(20),
    PRIMARY KEY(a),
    KEY b (b)
)ENGINE=INNODB
```

假设有表如上所示，现在有如下三条查询语句：

```
select a from t where b = xxx;
select a,b from t where b = xxx;
select a,b,c from t where b = xxx;
```

上面三条语句，前两条由于a和b都是覆盖索引叶子节点的一部分，a是主键，b是索引本身，因此只需要在一棵树上查询就能得到结果，而并不需要再去查询聚集索引。但是最后一条不同，c并不在辅助索引树上，因此要通过主键a再去查询聚集索引树得到c。这种再去查询聚集索引的操作称为回表。

如果要避免回表操作，可以使用联合索引来解决。例如上面例子，如果设定索引为（b,c），那么辅助索引树就会有两个键值对，即存了b和c的数据，这时再去用第三条语句查询，就可以直接查询到，而不需要回表。

（虽然联合索引能解决这种问题，但是对于低选择性条件还是不要用好些）

此外，针对统计问题时，覆盖索引也有好处：

```
select count(*) from t;
```

如上例子，InnoDB在做统计的时候，并不会去查询聚集索引树，而是选择辅助索引树去统计，因为辅助索引树大小远小于聚集索引，因此减少了大量的IO操作。

5.4.3 优化器选择不使用索引的情况

在某些情况下，使用EXPLAIN命令进行SQL语句分析的时候，会发现优化器并没有选择索引去查找数据，而是通过扫描聚集索引，也就是全表扫描。这种情况多发生与范围查找、JOIN连接操作等情况下。

（1）范围查找

比如：

```
CREATE TABLE t(
    id INT,
    orderid INT,
    ......
    PRIMARY KEY(id),
    KEY b (orderid)
)ENGINE=INNODB
```

现在要查找订单号大于10000小于102000的订单：

```
select * from t where orderid < 102000 and orderid >10000;
```

按照一般思路，orderid是单独的索引，通过索引条件，InnoDB应该会从orderid的索引树上寻找。但是进行SQL语句分析时发现并没有按照orderid索引树来寻找，而是对整表进行了扫描，即对聚集索引进行了寻找。

这是因为用户要得到的信息是整行的数据，而辅助索引树上并没有提供整行的数据，因此在对辅助索引寻找后还要对聚集索引进行一次寻找。虽然orderid索引树上orderid是顺序放的，但是查找出的10000<orderid<102000的主键在聚集索引上并不是按顺序排放的，即InnoDB对orderid范围查询是离散随机的，这增加了IO操作。

因此，InnoDB与其选择这种离散查找方式，还不如直接全表扫描，毕竟在聚集索引中，还是按照顺序排放的，可以查找到某一个叶子后，再通过链表查找找到其他数据。

当需要查找的数据占整个表中数据蛮大一部分（一般是20%左右）时，优化器就会使用全表查找。

（2）联合索引非最左前缀

详见联合索引。

（3）联合索引中，最左前缀，但是中间有范围查询，那么范围查询后面的列都用不到索引

比如联合索引为（a,b,c）

```
select a from t where a = 1 and b>1 and c=2;
```

像这样的条件就无法使用联合索引。这是因为，当b对于相同的a来说是有序的，因此通过a = 1过滤掉一部分的值，剩下a=1的行数据，这时b仍然是有序的。但是当在b>1进行过滤时，剩下的b>1的行数据InnoDB就无法判断是否有序了。那么对于c来说，如果b>1的行数据无序，则c无序，那么还是要经过聚集索引来查询。

（4）条件中使用or，如果or的前面条件用到了索引，而后面条件没有用到，就不会使用索引。

（5）join连接操作。

当使用join连接操作时，如果主键和外键的索引数据类型不一致，也不能使用索引。

（上面讲了5种常见的索引不能使用情况，还有很多种没有提到）

6. 锁

6.1 什么是锁

锁是数据库系统区别于文件系统的一个关键特性。锁机制用于管理对共享资源的并发访问。InnoDB支持行级锁，但并不是只有在表上才会加锁，另外在比如说操作缓冲池的LRU列表，添加，删除，移动LRU列表中的元素，为了保证一致性，必须要有锁的介入。

6.2 InnoDB存储引擎中的锁

6.2.1 锁的类型

InnoDB存储引擎实现了如下两种标准的行级锁：

（1）共享锁（S），允许事务读一行数据

（2）排他锁（X），允许事务删除或者更新一行数据

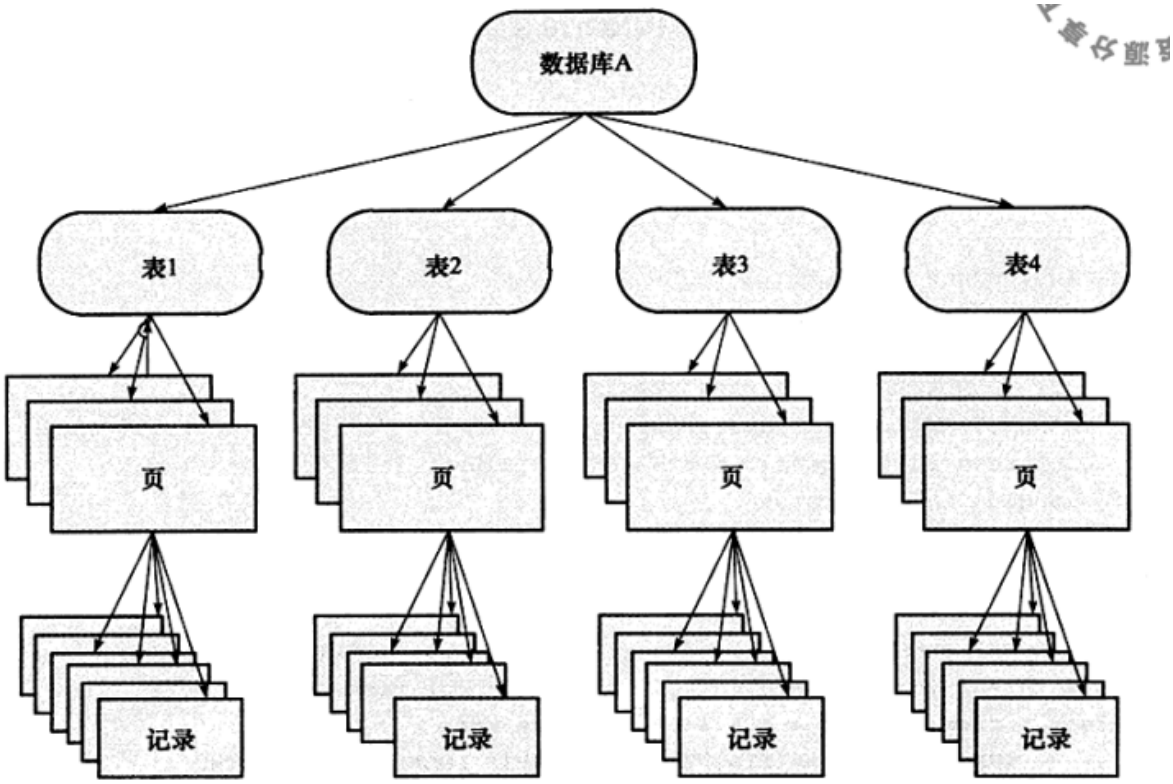
假设事务T1已经获得了行r的共享锁，那么另外的事务T2可以立即获得行r的共享锁，因为读取并没有改变行r的数据，称这种情况为锁兼容。

但若有其他的事务T3想获得行r的排它锁，就必须等待T1和T2释放行r上的共享锁，这种情况称为锁不兼容。

	X	S
X	不兼容	不兼容
S	不兼容	兼容

可以发现，X锁与所有锁都不兼容，而S锁只和S锁兼容。注意：兼容与否情况是发生在同一行数据上。

此外，InnoDB存储引擎支持多粒度锁定。这种锁定允许事务在行级上的锁和表级上的锁同时存在。为了支持这种多粒度的锁定，InnoDB存储引擎支持了另外一种额外的锁方式，称为意向锁。意向锁是将锁定的对象分为多个层次，意向锁意味着事务希望在更细粒度上上锁。



如图所示，如果想对记录r上X锁，那么就需要对该记录所在页，表，数据库A上意向锁IX，最后再对r上X锁。若在上意向锁的过程中出现了等待，则需要等待造成该过程的部分结束等待。例如：如果需要对表1加IX锁，但是表一现在已经被加上了S锁，由于S和IX不兼容，则就必须等待S锁的释放。

意向锁是表级别的锁。InnoDB支持两种意向锁：

（1）意向共享锁（IS）。事务想要获得一张表中某几行的共享锁

（2）意向排他锁（IX）。事务想要获得一张表中某几行的排他锁

	IS	IX	S	X
IS	兼容	兼容	兼容	不兼容
IX	兼容	兼容	不兼容	不兼容
S	兼容	不兼容	兼容	不兼容
X	不兼容	不兼容	不兼容	不兼容

6.2.2 一致性非锁定读

一致性非锁定读是指InnoDB存储引擎通过多版本控制（MVC）的方式来读取当前执行时间数据库中行的数据。如果读取的行正在执行DELETE或UPDATE操作，这时读取操作不会因此去等待行上锁的释放。相反，InnoDB存储引擎会去读取行上的一个快照数据。

之所以称为非锁定读，是因为读取时并不需要等待需要访问的行上X锁的释放。

快照数据是指该行的之前版本的数据，该实现是通过undo段来完成。而undo用来在事务中回滚数据，因此快照数据本身是没有额外的开销。此外读取快照上的数据是不需要上锁的，因为没有事务需要对历史的数据进行修改操作。

可以看到，非锁定读机制极大的提高了数据库的并发性。在InnoDB存储引擎的默认设置下，这是默认的读取方式，即读取不会占用和等待表上的锁。但是在不同事务隔离级别下，读取的方式不同，并不是在每个事务隔离级别下都是采用非锁定的一致性读。

快照数据其实就是当前数据之前的历史版本，每行记录可能有多个版本。即一个行记录可能有多个快照数据，一般称这种技术为行多版本技术，由此带来的并发控制，称之为多版本并发控制。

在事务隔离级别为READ COMMITTED和REPEATABLE READ（InnoDB默认事务隔离级别），InnoDB存储引擎使用非锁定一致性读。在事务隔离级别为READ COMMITTED的情况下，非一致性读总是读取被锁定行的最新一份快照数据；而在REPEATABLE READ下，非一致性读总是读取事务开始时的行数据版本。

6.2.3 一致性锁定读

在默认配置下，事务隔离界别为REPEATABLE READ的情况下，InnoDB存储引擎的SELECT操作使用非锁定一致性读。但是某种情况下，用户需要显式的对数据库读取操作进行加锁以保证数据逻辑的一致性。而这要求数据库支持加锁语句。

InnoDB存储引擎对于SELECT语句支持两种一致性的锁定读操作。

（1） SELECT ... FOR UPDATE

（2） SELECT ... LOCK IN SHARE MODE

第一种对读取的行记录加一个X锁，其他事务不能对已锁定的行加任何锁。

第二种对读取的行记录加一个S锁，其他事务可以向被锁定的行加S锁，但是不能加X锁。

6.3 锁问题

6.3.1 脏读

（1）脏页。脏页是指在缓冲池中已修改的页，但是还没有刷新到磁盘中。即数据库实例内存中的页和磁盘中的页的数据是不一致的。

（2）脏数据。脏数据是指事务对缓冲池中行记录的修改，并且还未提交。

（3）脏读。在不同的事务下，当前事务可以读到另外事务未提交的数据（所有事务提交前都要经过缓冲池）。即读到了脏数据。

脏读现象在生产环境中并不常发生，脏读发生的条件需要事务在READ UNCOMMITTED，而现在InnoDB默认的事务隔离等级为READ REPEATABLE。

6.3.2 不可重复读

不可重复读是指在一个事务内多次读取同一数据集合。在这个事务还没结束时，另外一个事务也访问该同一数据集合，并做了一些DML操作。

在第一个事务的两次读数据之间，由于第二个事务的修改，导致第一个事务读到的两次数据是不一致的。这种情况称为不可重复读。

不可重复读和脏读的区别：

脏读是读到未提交的数据；不可重复读读到的是已提交的数据，但是在两次读的情况下出现了数据不一致的情况。

6.3.3 幻读

在InnoDB默认事务隔离级别下，还是会出现幻读问题。

幻读是指，在同一事务下，连续执行两次相同的SQL语句却出现了不同的结果，第二次的SQL语句可能会返回之前不存在的行。注意：幻读特指查询出多余的行，即有事务在这个操作过程中添加了行数据。例如下面例子：

时间	Session1	Session2
	begin	
T1	select * from t where a > 2; 结果：（1,3）（2,4）	
T2		insert into t values(3,5)
T3	select * from t where a > 2; 结果：（1,3）（2,4）	
T4	select * from t where a > 2; 结果：（1,3）（2,4）（3,5）	

来分析一下：

（1）T1时刻，Session1读取a>2的数据，此时返回两条

（2）T2时刻，Session2插入了一条数据，并且满足a>2

（3）T3时刻，Session1再次查询a>2的数据，此时返回还是两条

（4）T4时刻，Session1再次查询a>2的数据，发现变成三条了

那么，为什么会出现事务更新后，查询还是两条的结果呢（幻读问题）？

这是因为之前提过的MVCC（多版本并发控制）。在这个机制下，当Session2修改数据后，Session1由于Session2加了X锁，此时就会去读取当前行记录的一个快照版本，而这个版本就是结果为两条的版本。最后T4时刻，读的就不是快照版本了，而是当前行记录版本。因此，引起幻读的原因就在于当前读和快照读混淆使用，导致结果不一致。

解决方法自然就要用到事务隔离级别的最后一种机制，串行化。在串行化的条件下，无论多少事务，都按照顺序往下走，每一个事务对表进行操作时，都会加上不同粒度的锁。虽然这样降低了并发性，但是也避免了幻读的问题。

6.3.4 丢失更新

丢失更新是另一个锁导致的问题，简单点说其就是一个事务的更新操作会被另一个事务的更新操作所覆盖，从而导致数据不一致情况。（类似Java的ABA问题）

但是，在当前数据库的任何隔离级别下，都不会导致数据库理论意义上的丢失更新问题。这是因为，即使是最低级别的事务隔离级别READ UNCOMMITTED，对于行的DML操作，也会去加上行锁或者粗粒度的表锁等。因此，当第一个事务去更新行记录的时候，第二个事务并不能去更新行记录，而是堵塞住，直到第一个事务事务提交为止。

虽然数据库本身能靠加锁解决这个问题，但是这个问题在实际应用下还是会产生。比如：

（1） 事务T1查询一行数据，放入本地内存，并显示给一个终端用户User1

（2） 事务T2也查询了该行数据，并将取得的数据显示给终端用户User2

（3） User1修改这行记录，更新数据库并提交

（4） User2修改这行记录，更新数据库并提交

显然，User1修改的这行记录，丢失了。

6.3.5 总结

	脏读	**不可重复读**	**幻读**
读未提交（READ UNCOMMITTED）	是	是	是
读已提交（READ COMMITTED）	否	是	是
可重复读（READ REPEATABLE）	否	否	是
串行化（SERIALIZABLE）	否	否	否

脏读（不同事务针对未提交的数据）

不可重复读（同一事务读取数据本身的对比）

幻读（同一事务读取结果条数的对比）

7. 事务

7.1 概述

事务可由一条SQL语句组成，也可以由一组复杂的SQL语句组成。事务是访问并更新数据库中各种数据项的一个程序单元。在事务中的操作，要么都做修改，要么都不做。

理论上说，事务要满足四个特性（ACID）

（1）A（Atomicity），原子性：原子性指整个数据库事务是不可分割的工作单位。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。

（2）C（consistency），一致性：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。

（3）I（isolation），隔离性：隔离性是指多个用户并发访问数据库时，每个事务对其他操作事务的操作对象能相互分离，隔离性通常使用锁来实现。

（4）D（durability），持久性：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。

7.2 事务的实现

事务的四大特性中，隔离性是使用锁来实现，而原子性，一致性，持久性则是通过数据库的redo log和undo log来实现的。

redo log称为重做日志，用来保证事务的原子性和持久性。

undo log称为回滚日志，用来保证事务的一致性。

7.2.1 redo

重做日志（redo log）用来实现事务的持久性，其由两部分组成：

（1）内存中的重做日志缓冲（redo log buffer），其是易失的

（2）重做日志文件（redo log file），其实持久的

当事务提交时，必须先将该事务的所有日志写入到重做日志文件进行持久化，等到事务的commit操作完成才算完成。redo log用来保证事务的持久性，undo log用来帮助事务回滚及MVCC的功能。

为了确保每次日志都写入重做日志文件，在每次将重做日志缓冲写入重做日志文件后，InnoDB存储引擎都需要调用一次fsync操作。由于重做日志文件打开并没有使用O_DIRECT选项，因此重做日志缓冲先写入文件系统缓存，为了确保重做日志写入磁盘，必须进行一次fsync操作。该操作的效率取决于磁盘的性能。因此，磁盘的性能决定了事务提交的性能，也就是数据库的性能。

MySQL中还有一种二进制日志，表面上看和redo log很像，都是记录了对于数据库操作的日志，但是从本质上来看，还是有很大不同：

（1）redo log是在InnoDB存储引擎层产生；二进制日志则是在MySQL数据库上层产生。即无论什么存储引擎都会产生二进制日志，而redo log只有InnoDB才会产生。

（2）两种日志记录形式不同。二进制日志记录的是一种逻辑日志，记录的是对应的SQL语句；InnoDB存储引擎层重做的是一种物理日志，记录的是对每个页的修改。

（3）写入时间不同。二进制日志只在事务提交完成后进行一次写入；redo log则是在事务的进行中不断写入。

InnoDB存储引擎在启动时不管上次数据库运行时是否正常关闭，都会尝试进行恢复操作。

7.2.2 undo

重做日志记录了事务的行为，可以很好的通过其对页进行重做操作。但是事务还应该有回滚操作，这时就需要undo。因此在对数据库进行修改时，InnoDB不仅会产生redo log，还会产生一定量的undo。这样当事务失败，或者用户使用一条ROLLBACK语句请求回滚，就可以利用这些undo信息将数据回滚到修改之前的样子。

redo存放在重做日志文件中，而undo存放在数据库内部的一个特殊段中，称为回滚段（undo 段）。undo段位于共享表空间内。

undo并不是将事务恢复成执行语句或者事务之前的样子，而是将数据库逻辑的恢复成原来的样子。这一系列看上去做的是和之前相反的操作。比如对于每个Insert，回滚操作意味着它会完成一个delete；对于每个delete，回滚操作意味着它会完成一个insert；对于每个update，它会执行一个相反的update，将修改前的行放回去。

数据库逻辑恢复意味着对于上面例子的insert来说，表空间增大了，但是回滚操作并不会将表空间恢复到原样。

此外，如上面例子的delete操作，在没回滚时，做delete操作，并不是直接删除记录，而是将记录标记为删除。最终删除是在purge操作中完成的。而更新操作即先删除该记录（也是记录标记为删除），然后产生一条undo log，再插入一条新纪录，插入操作又产生了一条undo log。

undo log的另一个作用就是MVCC。当用户读取一行信息时，若该记录已经被其他事务占用，当前事务可以通过undo读取之前的行版本信息，以此实现非锁定读。

undo log会产生redo log，这是因为undo log也需要持久性的保护。

7.2.3 purge

上面提到过，delete或者update操作可能并不直接删除原有的数据，最终删除操作是在purge中完成的。

purge用于最终完成delete和update操作，这样设计是因为InnoDB存储引擎支持MVCC（因为需要回滚），因此记录不能在事务提交时立即完成。这时，可能有其他事务正在引用此行，所以InnoDB存储引擎需要保存记录之前的版本。若该行记录没有被任何事务引用，就可以真正删除此记录。

7.2.4 group commit

前面提到过，若事务非只读事务，则事务每次提交的时候，都需要进行一次fsync操作，以此保证重做日志都已经写入磁盘。当数据库怠机时，可以通过重做日志（redo log）进行恢复（当然，如果操作系统怠机了，则丢失的数据就丢失了无法恢复）。该操作的效率取决于磁盘的性能，因此，为了提高fsync的效率，当前数据库都提供了group commit功能，即一次fsync可以刷新确保多个事务日志被写入文件。