# 第2章 InnoDB关键特性

0. 特点：行锁设计、支持MVCC、支持外键、提供一致型非锁定读

1. InnoDB体系架构：

后台线程

内存

2. InnoDB关键特性

- 插入缓冲

问题：InnoDB中，主键是行唯一的标识符。聚集索引(Primary Key)是物理顺序的，不需要磁盘随机读取。但是非聚簇索引(所有非主键索引)叶子节点的插入不再是顺序的了。(B+树的特性决定了非聚集索引插入的离散性)

解决：对于非聚集索引的插入和更新，不是直接插入到索引页中。而是先判断要插入的非聚集索引**页**是否在缓冲池中。若在，则先把该它放在一个Insert Buffer对象中，伪装已插入。随后再将多个插入merge到一个操作中(因为在一个索引页中)，大大提高了对非聚集索引插入的性能。更新类似。

- 两次写

解决问题：数据页的可靠性

- 自适应哈希索引

B+树查找取决于树高(生产环境3-4层)InnoDB引擎会自动地为某些热点页建立哈希索引(O(1))

- 异步IO

为了提高磁盘操作性能，当前DB系统基本都采用异步IO

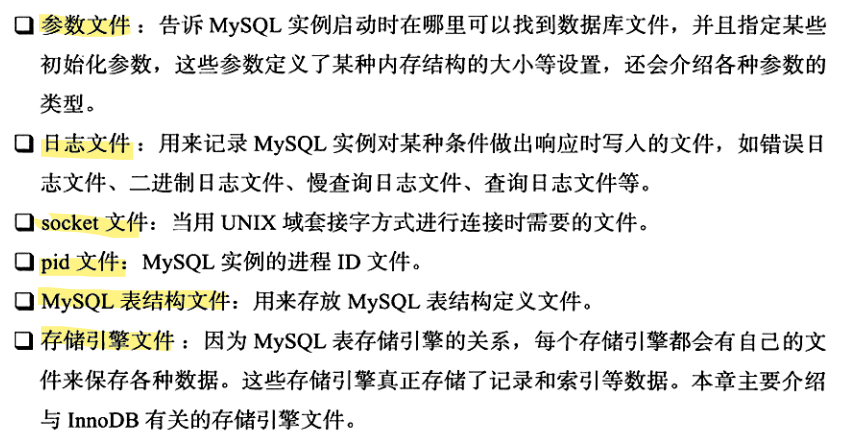
全部IO请求发送完毕后，一起完成IO操作

- 刷新邻接页

当刷新一个脏页时，检查该**页**所在**区**(extent)的所有页，如果也是脏页，一起刷新。

局部性原理。

# 第3章 文件



二进制日志记录了对MySQL数据库执行更改的所有操作，但是不包括select和show这类，因为这类操作对数据本身并没有修改

二进制日志作用：恢复（point-in-time）+复制+审计

# 第4章 表

4.1 索引组织表

表都是按照主键顺序组织存放的。如果没有显式定义主键，InnoDB也会挑一个Unique NOT NULL的作为主键，还没有，自动创建一个6字节的指针。

* 1. InnoDB逻辑存储结构

层次上看：表空间tablespace->段segment->区extent->页page /块block->(行)

1. 表空间：默认有一个共享表空间ibdata1，即所有数据都放这里，初始大小58MB
2. 段：
   1. 常见的段有数据段、索引段、回滚段
   2. 数据段即B+树的叶子节点

\*“InnoDB存储引擎表是索引组织的，因此数据即索引，索引即数据”

* 1. 对段的管理是引擎自身完成的，DBA够不到

1. 区：
   1. 区的大小总是IMB，内部是64个连续的页(16KB大小，2^6\*2^4=2^10)
   2. InnoDB一次从磁盘申请4-5个区，为了保证区中页的连续性
2. 页：
   1. 页是磁盘管理的最小单位。
   2. 默认16KB，可以通过参数innodb\_page\_size设置为4K\8K\16K，只能设置一次。
   3. 常见页类型：数据页、undo页、系统页、事务数据页、插入缓冲位图页、插入缓冲空闲列表页、为也做得二进制大对象页、压缩的二进制大对象页
3. 行
   1. InnoDB存储引擎是面向列的（row-oriented），即数据是按行存放的。
   2. InnoDB行记录格式
4. Compact行记录格式（默认）
   1. MySQL5.0引入，设计目的是高效地存储数据



1. Redundant行记录格式
   1. 为了向前兼容5.0版本之前而保留的

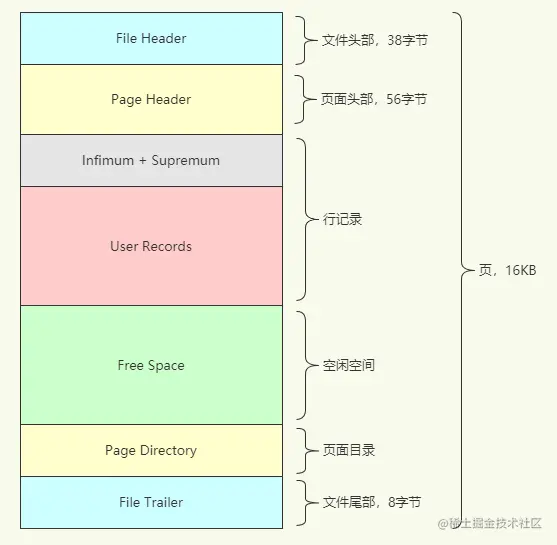


1. 行溢出数据
   1. 存放BLOB、LOB这类的大对象列类型和VARCHAR溢出部分(理论最大65535字节，实际最大65532字节)

4. Compressed和Dynamic行记录格式

* 1. InnoDB数据页结构

1. File Header：
   * + 记录页的头信息
     + checksum值表空间中页的偏移量、上一个页、下一个页、最后修改的日志序列位置LSN、页类型（0x45BF是B+树叶节点，即“数据页”）
2. File Trailer：
   * + 校验页的完整性
     + File Header的checksum值和header中函数校验、LSN后4字节
3. Page Header：
   * + 记录数据页状态信息。
     + 在Page Directory中的槽数、堆的起始、堆中的记录数量、可重用空间的首指针、已删除记录的字节数、最后插入位置、该页中记录数量等！
4. Infimum & Superemum Record
   * + 每个数据也有两个虚拟的行记录，用来限定记录的边界。类似dummy head和dummy tail。 Infimum是比任何主键值都小的值，另一个巨大。
5. User Records & FreeSpace
   * + User Records实际存储行记录的内容
     + FreeSpace空闲空间，是链表结构。一条记录被删除后，就添加到该链表中。
6. Page Directory
   * + 页目录中存放了了记录的相对位置
     + 【注意】B+树索引本身并不能找到具体的记录，只能找到记录所在的页。数据库把页载入内存，然后通过Page Directory进行二分查找。因为内存+二叉查找太快了，所以常常忽略不计。
     + InnoDB中并不是每个记录都有一个槽，InooDB的槽是一个稀疏目录，一个槽中可能有多条目录。



* 1. 约束
  2. 数据库的约束是为了保证数据完整性而实现的一套机制
  3. 完整性三种形式
     1. 实体完整性：保证表中有一个主键
     2. 参照完整性：保证两张表之间的关系。Mysql允许用户定义完整性。
     3. 用户定义的完整性：针对某一具体应用的数据必须满足的语义要求。
  4. 【补充】触发器：
     1. 在执行INSERT、DELETE、UPDATE命令BEFORE或AFTER，自动调用SQL命令或存储过程
     2. 一个表最多6个触发器 2\*3
     3. Mysql只支持按行触发
     4. 目的：业务逻辑的严密性。Eg.用户支付-50元。不建议使用，非常消耗资源，并发和数据量大时容易死锁。在业务代码编写的时候处理就好了。
  5. 视图

是一个虚表，有查询定义

* 1. 分区表

# 第5章 索引与算法

1. B+树简要定义：B+树是为磁盘或其他直接存取辅助设备设计的一种平衡查找树。所有的节点按键值大小顺序放在同一层的叶子节点上，由各叶子结点指针进行连接。

2. 无论聚集索引还是非聚集索引(辅助索引)，一个索引就是一颗B+树。

3. 高扇出(fan out)性->B+树高度通常2-4层，意味着查找一次行记录需要2-4次IO

\*机械键盘每秒可做100次IO

4. 聚集索引：

a) 按照每张表的主键构造一颗B+树

b) 这棵B+树上，value即为行记录数据 ->聚簇索引的叶子节点=数据页

c) 优点：1.能直接在B+树索引的叶子节点上找到数据 2. 对于主键的排序查找（因为双向链表）和范围查找速度特别快（因为叶节点上一层可以得到页的范围）

5. 辅助索引

a) Secondary Index的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，还包含bookmark（就是相应行数据的聚簇索引键）。

b) 可以说，非聚簇索引，是指向索引的索引？——不一定。当查询的列中有该索引未包含的列信息，需要二次索引。

6. Carinality值

什么样的字段使用索引合适？——高选择性的。（取值范围广。姓名；低选择性的有：性别、地区、类型）

7. B+树索引的使用：

1. 不同应用中B+树索引的使用——生产环境

OLTP： 查询操作一般只取得一小部分数据，(例如，订单处理，根据主键值取得1位用户信息)建立B+树索引是有意义的

OLAP： 面向分析的查询，一般是大批数据，例如不用对姓名字段进行索引。

但是对于复杂查询，涉及多张表的连接操作，索引添加是有意义的。

在OLAP中通常会需要对时间字段进行索引，因为常根据时间进行数据筛选。

【注意】：只有查询少量数据，优化器才会选择使用索引

2. 联合索引的优点

1. 对于select \* from .. where a=xx and b=xx，使用(a,b)联合索引，只需一趟查询

2. 联合索引对第二个键值进行了排序处理。例如，对于“查询某个用户购物情况，并按照时间排序”，就可以避免多一次排序。

3. 覆盖索引

a) 是什么？从辅助索引中就能查到的记录，就不需要查询聚集索引。

b) 优点？辅助索引没有包含整行记录的全部信息，大小运远小于聚集索引。减少IO操作。

4. 优化器选择不使用索引的情况？

a) 多发生于范围查找、JOIN链接操作

b) “不使用索引”指：优化器没有选择索引查找数据，而是通过扫描聚集索引，也就是直接进行全表扫描。

5. 索引扫描

MySQL支持INDEX HINT，显式地告诉用户其使用哪个索引

6. MRR优化：Muliti-Range Read优化。减少了磁盘的随机访问，并且将随机访问转化为较为顺序的数据访问

ICP优化：Index Condition Push优化。取出索引的同时根据WHERE过滤， 减少上层SQL层对记录的索取（fetch），从而提高数据库整体性能

8. 哈希算法

a) InnoDB中哈希算法冲突机制：链表方法；哈希函数采用除法散列

b)自适应哈希索引：是存储引擎自己控制的！

官方文档的描述是不支持哈希索引

对字典查找非常快O1，对范围查找无能为力

9. InnoDB支持全文检索

# 第6章 锁

6.1 什么是锁

6.2 lock与latch

6.3 InnoDB存储引擎中的锁

6.4 锁的算法

6.5 锁问题

6.6 阻塞

6.7 死锁

6.8 锁升级

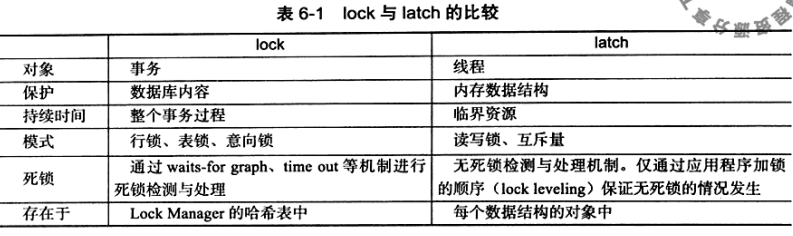
*数据库系统区别于文件系统的关键特性之一：锁——支持对共享资源并发访问；之二：事务——把数据库从一种一致状态转换为另一种一致状态。*

一句话：InnoDB存储引擎提供一致性的非锁定读、行级锁支持。

**lock与latch**

latch：分为mutex（互斥锁）和rwlock（读写锁），其目的是保证并发**线程**操作临界资源的正确性。

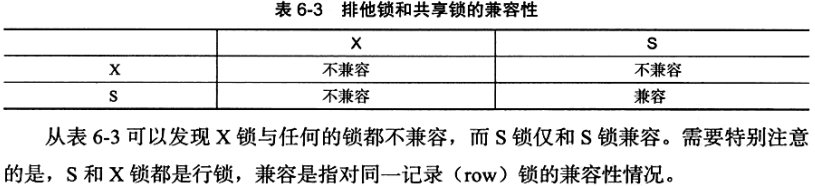
Lock：对象是**事务**，用来锁定的是数据库中的对象（表、页、行）。并且一般lock的对象仅在事务commit或rollback后释放。



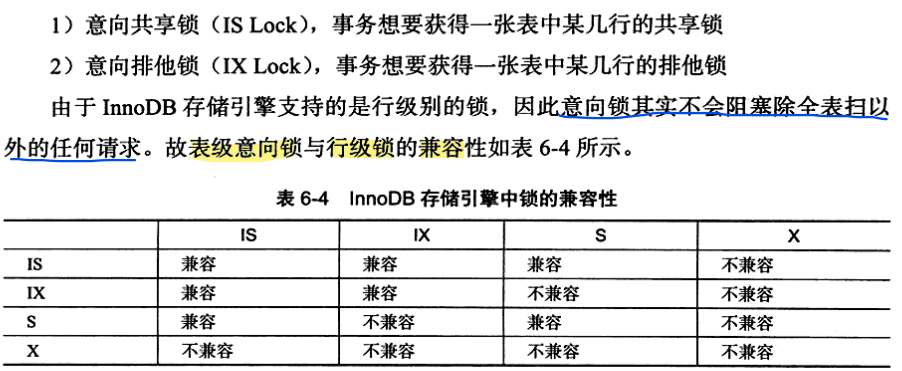
**锁的类型**

1. 两种**行级锁**：

共享锁（S锁）（允许事务读一行）& 排它锁（X锁）（允许事务删除或更新一行）



1. 一种**表级别**的锁：意向锁
   1. 背景：InnoDB支持多粒度锁定——允许事务在行级上的锁和表级上的锁同时存在
   2. 设计目的：为了在一个事务中揭示下一行将被请求的锁类型



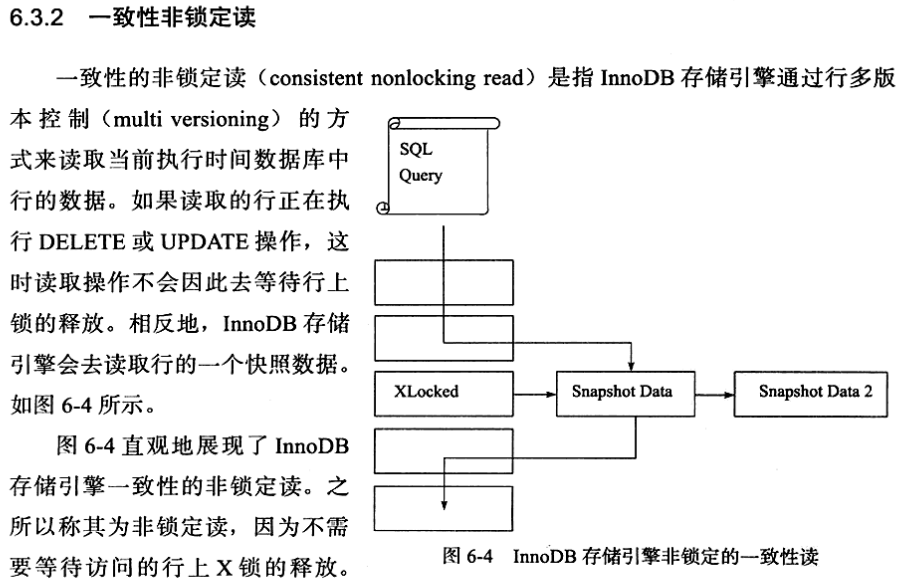
**一致性非锁定读**

1. 一致性非锁定读是InnoDB通过MVCC来读数据——如果要读的行上正在写，不会等待行上的锁释放，而是读行的一个快照数据。

2. MVCC：多版本并发控制。多版本——一个行上不只有一个快照数据/历史版本。由此带来的并发控制。

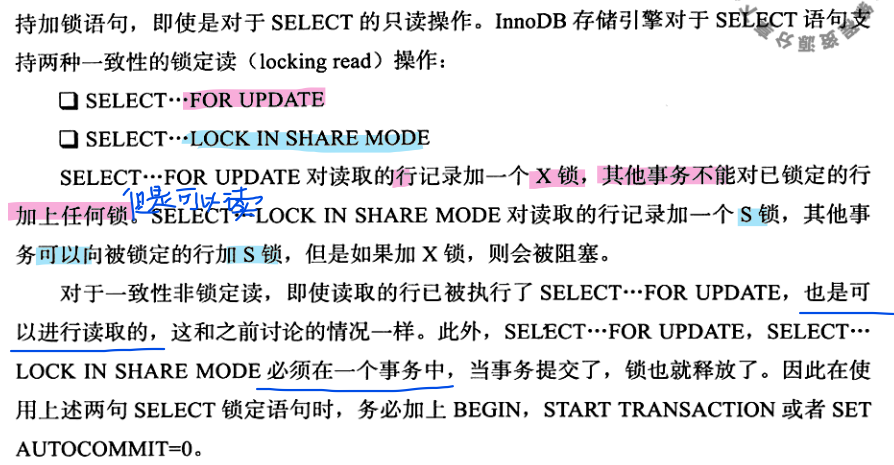
3. 在READ COMMITTED事务隔离级别下，总是读被锁定行的最新的快照。

在REPEATABLE READ事务隔离级别下，总是读事务开始的行数据版本。



**一致性锁定读**

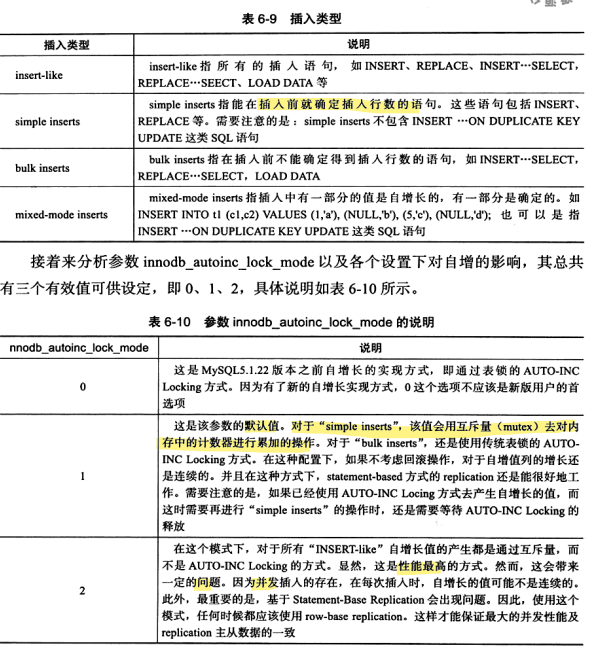
在默认配置下（repeatable read），select用一致性非锁定读。但有时候需要显式地对读操作加锁。



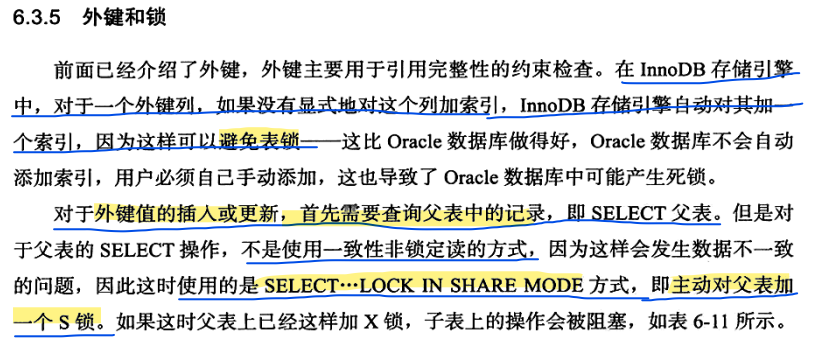
**自增长与锁**

传统的，AUTO-INC Locking——（插入时自增长）为了提高插入的性能，锁不是在一个事务完成后释放，而是在完成对自增长值插入的SQL语句后就释放

新的，InnoDB提供了一种轻量级互斥量的自增长实现机制

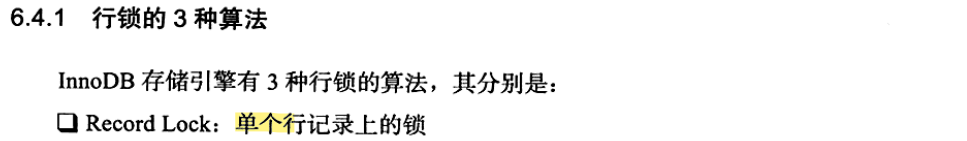


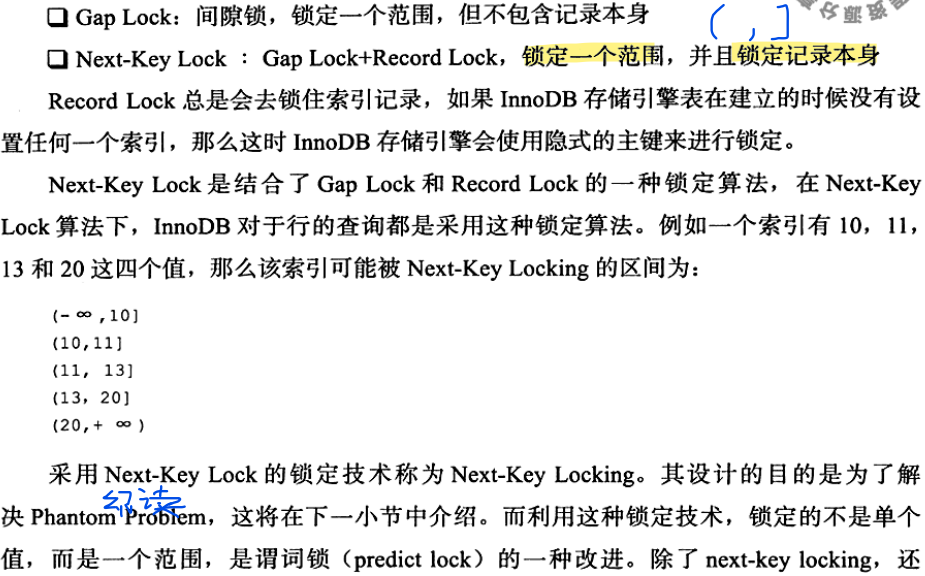
**外键和锁**

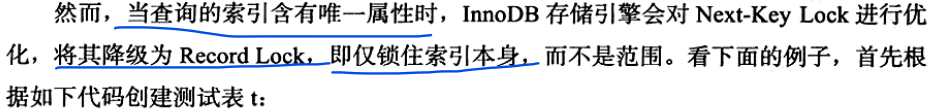


**锁的算法与解决幻读**

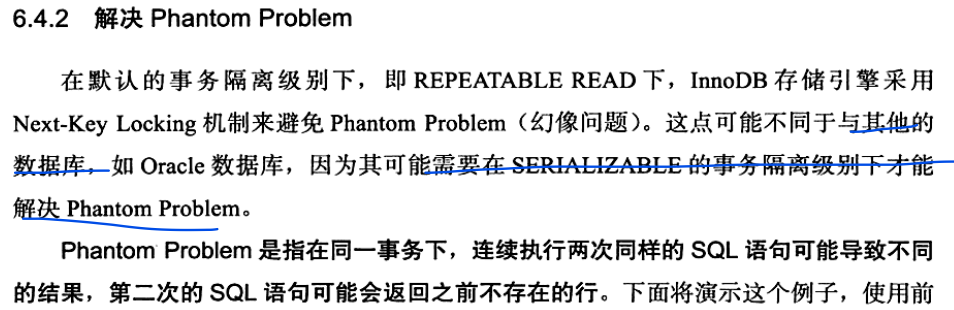
行锁算法 Next-Key Lock，解决幻读问题







索引具有唯一属性：主键、或者标注了unique



解决：通过Next-Key Locking，对查询的到的值与条件值之间加了X锁，不允许插入操作

\*innodb事务隔离级别read committed没有解决幻读，因为其仅采用record lock

**锁问题：脏读+不可重复读+幻读**

**脏读**

1. 区分脏数据&脏页
   1. 脏页：在缓冲池中已被修改的页，还是尚未刷新到磁盘中——对脏页读取是正常的——脏页是因为数据库实例内存和磁盘的异步造成的，并不影响数据的一致性，通过异步提高性能
   2. 脏数据：事务对缓冲池中行记录修改，但是未提交——读到脏数据，影响隔离性
2. 脏读是：在不同的事物下，a事务可以读到b事务未提交的数据
3. 脏读在生产环境中不常发生，因为事务隔离级别read uncommitted不常用。
   1. 什么时候会设置？replication环境中的slave节点，并且在该slave上的查询不要求太精确

**不可重复读**

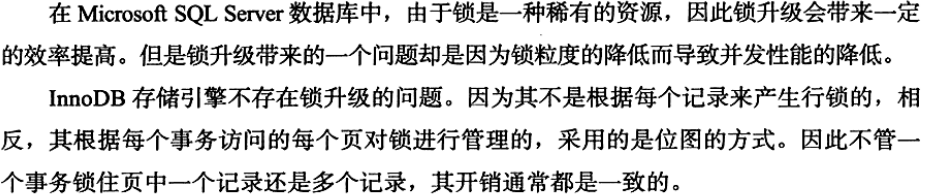
1. 不可重复读是：在同一个事务内，两次读到的数据是不一样的
2. 和脏读的区别：脏读是读到未提交的数据；而不可重复读是读到已提交的数据，但是违反了数据库事务一致性的要求
3. 一般来说，不可重复读是可以接受的。很多数据库厂商默认隔离级别是read committed，允许这种现象
4. 不可重复读与幻读。在一个事务中前后多次读取，前者读到数据内容不一致，后者数据总量不一致。前者update，后者insert和delete。MySQL文档中统一称为phantom problem，即幻象问题，通过next-key lock算法对索引覆盖到的范围加锁来解决。

**死锁**

1. 死锁发生四条件。。
2. 死锁解决——超时——两个事务互相等待时，当一个timeout后，其中一个回滚，另一个就能继续执行
3. 死锁检测——Wait-for graph（等待图）
   1. 原理：保存了锁链表、事务等待链表；存在回路即存在死锁，Innodb将选择回滚undo量最小的事务
   2. 深度优先算法，优化为非递归

锁升级

1. 锁升级是：将当前锁的粒度降低（1000个行锁变一个页锁）



（不懂！）

# 第7章 事务

7.1 认识事务

7.2 事务的分类

7.3 事务控制语句

7.4 隐式提交的SQL语句

7.5 对于事务操作的统计

7.6 事务的隔离级别

7.7 分布式事务

7.8 不好的事务习惯

7.9 长事务

*数据库系统引入事务的主要目的：事务会把数据库从一种一致状态转换为另一种一致状态。在数据库commit时，可以确保要么所有修改都已经保存，要么所有修改都不保存。*

*InnoDB存储引擎中的事务完全符合ACID的特性。（\*指在默认隔离级别read reaptable下）*

*第6章，锁，实现了事务的隔离性。本章主要关注事务的原子性。*

**原子性**——要么都做，要么都不做。

**一致性**——数据库从一种状态转换为下一种一致状态。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**——并发控制、可串行化、锁等

**持久性**——事务一旦提交，结果就是永久性的。即使宕机，数据库也能够恢复。

**分类：**

扁平事务、带有保存点的扁平事务、嵌套事务、分布性事务

**事务的实现**

1. **Redo**
   1. Redo log 重做日志。是对操作的记录。用来保证事务的持久性Duration
      1. 组成：内存中的重做日志缓冲（易失）+重做日志文件（持久）
      2. 机制：Force Log at Commit
   2. 二进制日志binlog和重做日志的区别：
      1. 本质，Redolog是在InnoDB存储引擎层面的；binlog是MySQL数据库上层的，任何存储引擎都对数据库的更改都记录
      2. 内容，二进制日志是一种逻辑日志，记录对应的SQL语句。重做日志是物理格式日志，记录的是对每个页的修改。
      3. 写入磁盘的时间点，二进制日志只在commit后进行一次写入，redolog在事务进行中不断被写入
   3. 重做日志不论在buffer还是文件，都是512字节进行存储的，称为一个block
      1. 由于重做日志块和磁盘扇区大小一样，因此写入可以保证原子性
   4. LSN ：log sequence number，日志序列号，表示事务写入重做日志的字节总数
   5. 恢复：InnoDB存储引擎在启动时不管上次数据库是否正常关闭，都会尝试进行恢复操作。因为redolog记录的是物理日志，因此恢复的速度比逻辑日志，如二进制日志，要快得多。
2. **Undo**
   1. Undo存放在共享表空间的undo端段（segment）中
   2. 作用
      1. Undo用于回滚操作。Undo是逻辑日志，只是将数据库逻辑地恢复到原样。
      2. Undo另一个作用是MVCC。当用户读取某行记录被*其他事务*占用时，*当前事务*通过undo读取之前的行版本数据。
3. **Purge**

Delete和update操作并不直接删除原有数据。

对于delete，在事务中，仅是将符合删除条件的行记录的delete flag设为1，记录并没有被删除，即，记录还是存在B+树中。

对于update，也是将原主键记录标记为已删除，之后插入一条新的纪录。

Purge用于最终完成delete和update操作。

这样设计是因为innodb支持MVCC，所以记录不能在事务提交时立即处理。这时其他事务可能正在引用这行。

1. Group commit

用一次fsync，刷新确保多个事务日志被写入内存。

\*总结redo和undo

1. 都可以视为一种恢复操作

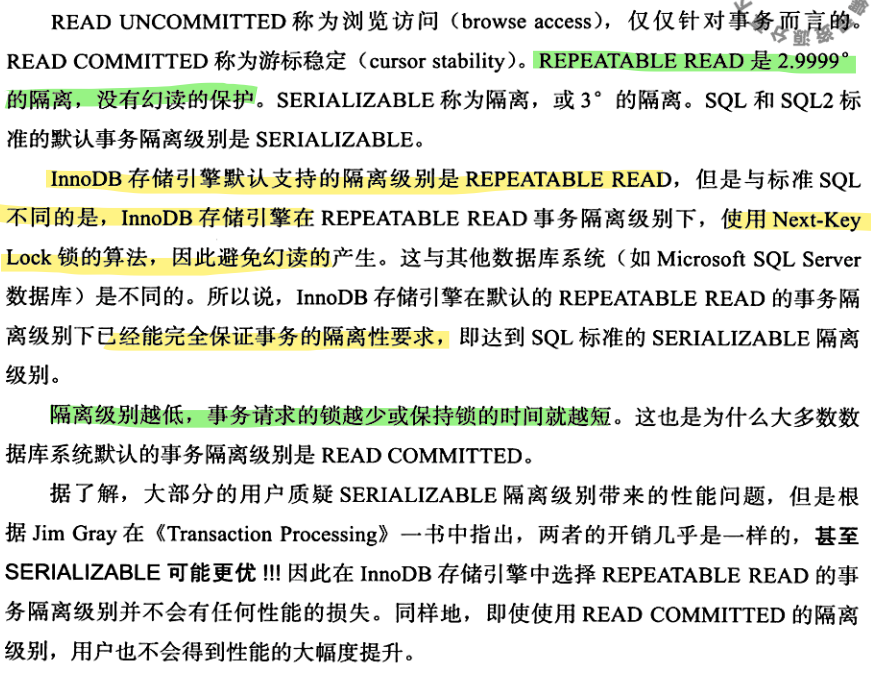
2. redo恢复 提交事务 修改 的页操作；undo回滚行记录到某个特定版本。

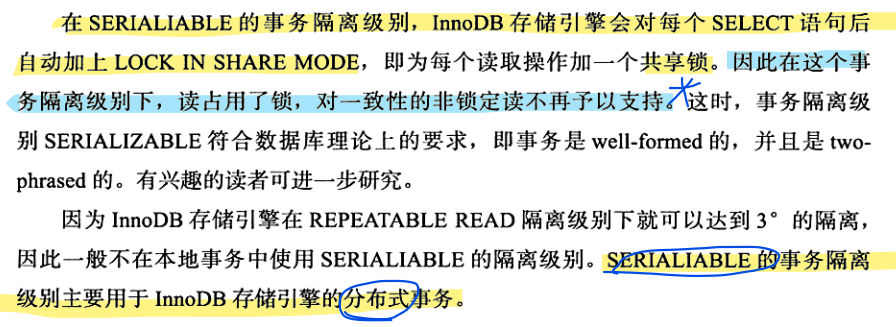
3. redo是物理日志，记录的是页的物理修改日志；undo是逻辑日志，根据行记录进行记录。

**QPS** Question Per Second （应用的）每秒请求数

**TPS** Transaction Per Seond 每秒事务处理能力

**事务的隔离级别**





\* 前文讲的是在repeatable read隔离级别下，显式地对读取操作加锁。因为repeatable read是一致性非锁定读，所以还可以读，~~读的时候不涉及加锁~~。只是不可以再在这行上加X锁。

修订：读的时候加读锁。加了锁就不是一致性非锁定读了。变成一致性锁定读。读的时候不准写。

**分布式事务**

1. 允许多个独立的事务资源参与到全局的事务中。（“事务资源”通常指关系型数据库系统）
2. 举例：上海的工商银行转账到北京的农业银行
3. 方式：prepare-commit/rollback的两段式提交

**不好的事务习惯**

1. 在循环中提交——插入10000条，在第5000条出错
2. 使用自动提交
3. 使用自动回滚

**长事务**：转化为小批量（mini batch）的事务进行处理