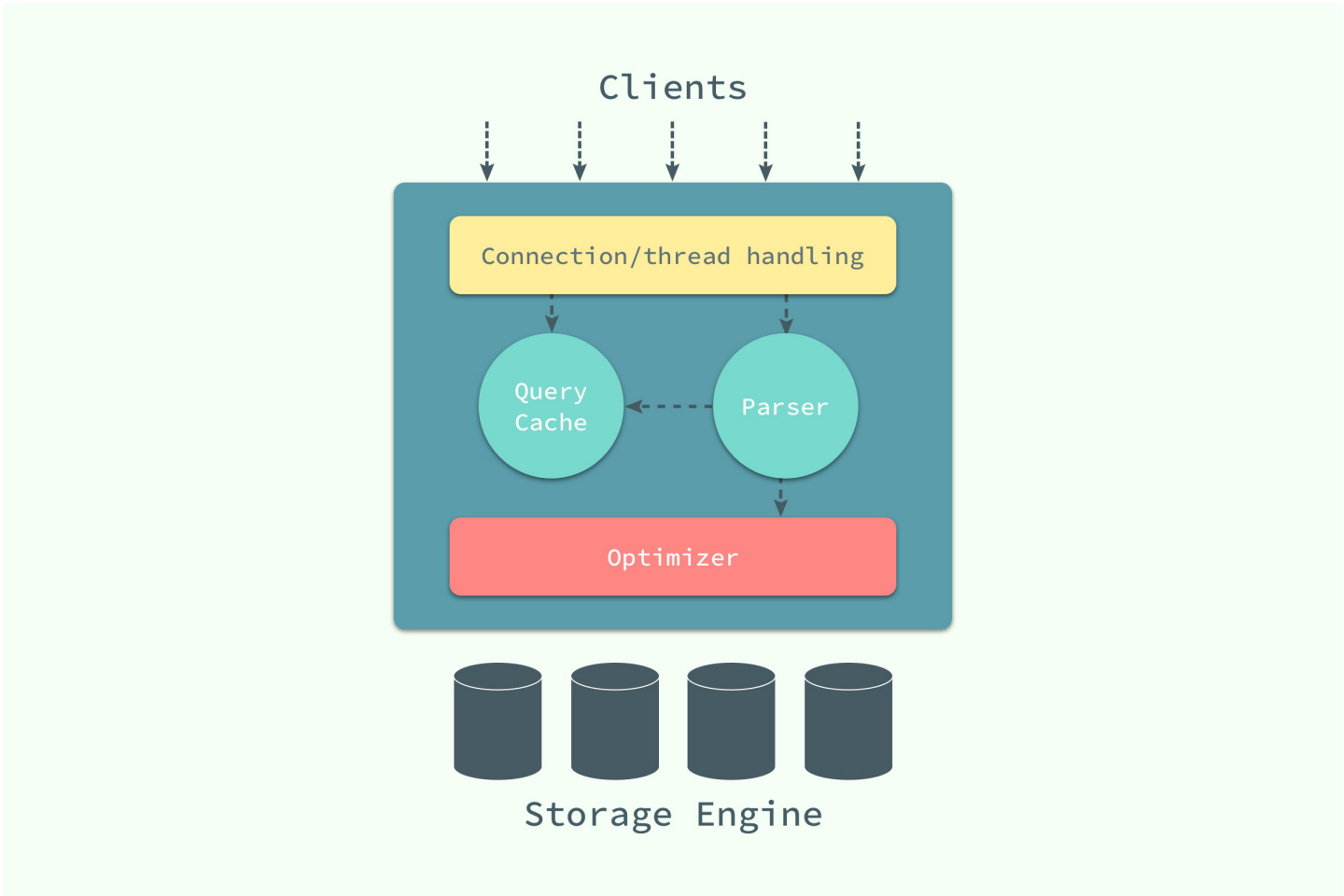
MySQL 的架构

MySQL 从第一个版本发布到现在已经有了 20 多年的历史，在这么多年的发展和演变中，整个应用的体系结构变得越来越复杂：



最上层用于连接处理、授权认证、安全的部分并不是MySQL所独有的，大多数基于网络的客户端/服务器的工具或者服务都有类似的架构。每个客户端连接都会在服务器进程中拥有一个线程，这个连接的查询只会在这个单独的线程中执行，该线程只能轮流在某个CPU核心或者CPU中运行；

第二层中包含了大多数 MySQL 的核心服务，包括了对 SQL 的解析、分析、优化和缓存等功能，存储过程、触发器和视图都是在这里实现的。MySQL会解析查询，并创建内部数据结构（解析树），然后对其进行各种优化，包括重写查询、决定表的读取顺序，以及选择合适的索引等。对于SELECT语句，在解析查询之前，服务器会先检查查询缓存（Query Cache），如果能够在其中找到对应的查询，服务器就不必再执行查询解析、优化和执行的整个过程，而是直接返回查询缓存中的结果集；

而第三层就是 MySQL 中真正负责数据的存储和提取的存储引擎，例如：InnoDB、MyISAM等。

MyISAM存储引擎

MyISAM是MySQL的默认数据库引擎（5.5版之前），由早期的 ISAM所改良。虽然性能极佳，而且提供了大量的特性，包括全文索引、压缩、空间函数等，但MyISAM不支持事务和行级锁，而且最大的缺陷就是崩溃后无法安全恢复。

MyISAM特点：

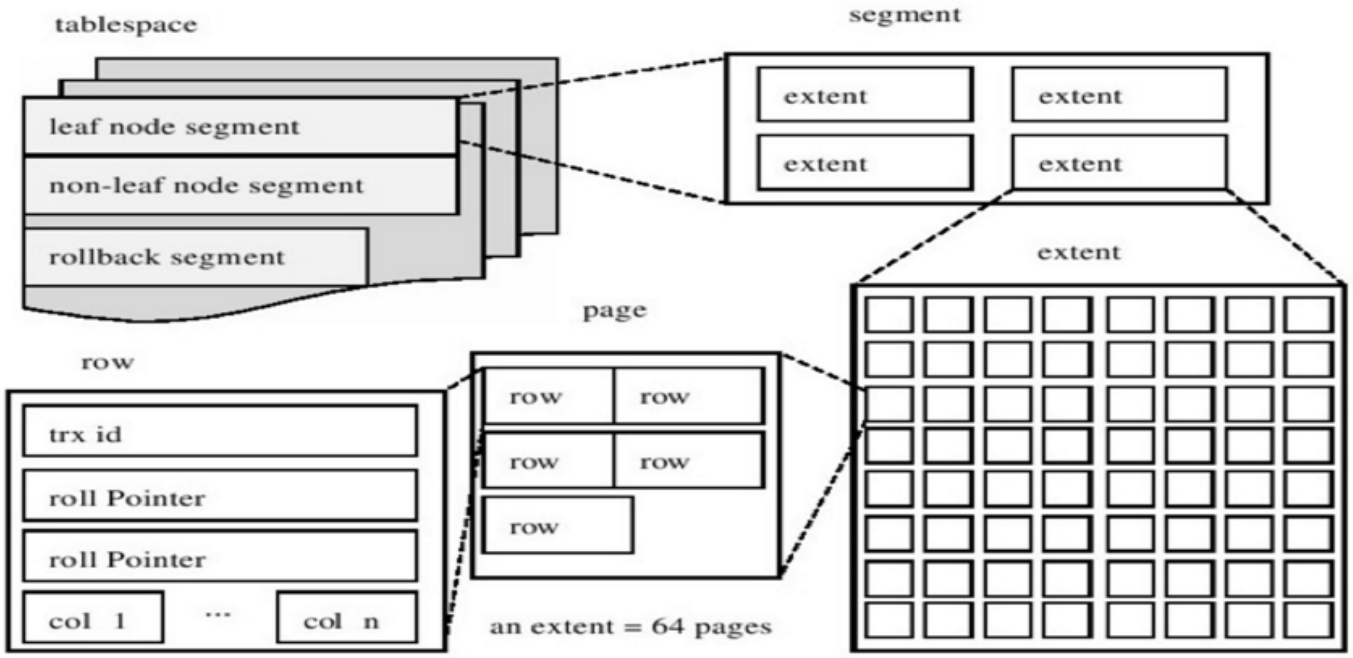
- 不支持行锁(MyISAM只有表锁)，读取时对需要读到的所有表加锁，写入时则对表加排他锁,如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据；
- 不支持事务；
- 不支持崩溃后的安全恢复；
- 不支持外键；
- 数据文件和索引文件分开存储，数据文件(.MYD),索引文件(.MYI)和结构文件(.frm)；
- 支持全文索引；
- 主键索引和二级索引完全一样都是B+树的数据结构，只有是否唯一的区别（主键和唯一索引有唯一属性，其他普通索引没有唯一属性。B+树叶子节点存储的是数据记录的地址；
- 支持延迟更新索引，极大提升写入性能;
- 对于不会进行修改的表，支持压缩表，极大减少磁盘空间占用；

InnoDB存储引擎

InnoDB存储引擎最早由Innobase Oy公司开发，被包括在MySQL数据库所有的二进制发行版本中，从MySQL 5.5版本开始是默认的表存储引擎（之前的版本InnoDB存储引擎仅在Windows下为默认的存储引擎）。是第一个完整支持ACID事务的MySQL存储引擎。也是最重要、使用最广泛的存储引擎。它被设计用来处理大量的短期事务，短期事务大部分情况是正常提交的，很少会被回滚。InnoDB的性能和自动崩溃恢复特性，使得它在非事务型存储的需求中也很流行。除非有非常特别的原因需要使用其他的存储引擎，否则应该优先考虑InnoDB引擎。
InnoDB相较其它存储引擎的主要特点有：支持事务、支持高并发、自动崩溃恢复、基于聚簇索引组织表数据等。

InnoDB逻辑存储结构

在 InnoDB 存储引擎中，所有的数据都被逻辑地存放在表空间中，表空间（tablespace）是存储引擎中最高的存储逻辑单位，在表空间的下面又包括段（segment）、区（extent）、页（page）：



- 表空间（tablespace）
  表空间是Innodb存储引擎逻辑的最高层，所有的数据都存放在表空间中，Innodb存储引擎有一个共享表空间ibdata1,即所有数据都存放在这个表空间中内。如果启用了innodb_file_per_table参数（ mysql5.6.6及其后续版本默认开启），则每张表内的数据可以单独放到一个表空间内，但请注意，只有数据、索引、和插入缓冲Bitmap放入单独表内，其他数据，比如回滚(undo)信息、插入缓存索引页、系统事物信息，二次写缓冲等还是放在原来的共享表内的。
- 段(segment)
  表空间是由不同的段组成的，常见的段有：数据段，索引段，回滚段等等，因为InnoDB存储引擎表是索引组织的，因此数据即索引，索引即数据。数据段即为B+树的叶子结点，索引段即为B+树的非索引结点。在InnoDB存储引擎中对段的管理都是由引擎自身所完成，DBA也没必要对其进行控制。
- 区(extent)
  区是由连续页组成的空间，每个区的固定大小为1MB,为保证区中页的连续性，InnoDB会一次从磁盘中申请4~5个区，在默认不压缩的情况下，一个区可以容纳64个连续的页。但是在开始新建表的时候，空表的默认大小为96KB,是由于为了高效的利用磁盘空间，在开始插入数据时表会先利用32个页大小的碎片页来存储数据，当这些碎片使用完后，表大小才会按照MB倍数来增加。
- 页(page)
  页是InnoDB存储引擎的最小管理单位，每页大小默认是16KB，从InnoDB 1.2.x版本开始，可以利用innodb_page_size来改变页size，但是改变只能在初始化InnoDB实例前进行修改，之后便无法进行修改，除非mysqldump导出创建新库，常见的页类型有：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。
- 行(row)
  行对应的是表中的行记录，每页存储最多的行记录也是有硬性规定的最多16KB/2-200,即7992行。

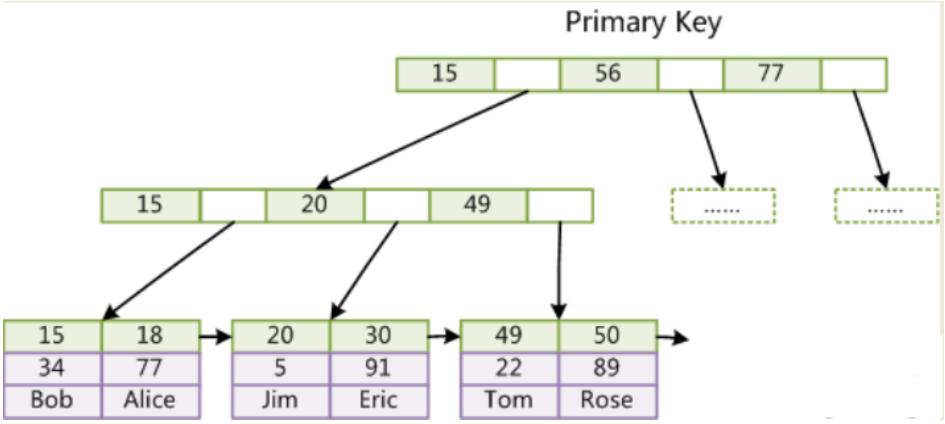
InnoDB索引

InnoDB存储引擎支持以下几种常见索引：B+树索引、全文索引、哈希索引。我们平时在开发中使用最多的，就是B+树索引。

聚集索引(Clustered Index)

InnoDB数据文件本身是按照B+Tree组织的索引结构，并且叶子节点存放的是整张表的行记录数据。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键，如果不存在，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。因为B+树的特点，数据节点通过一个双向的链表连接起

来，使得查找更加快速，由于实际的数据页只能按照一颗B+树来存放，所以，一张表只能拥有一个聚集索引，在多数情况下，查询优化器倾向于采用聚集索引。B+树索引并不能找到一个给定键值的具体行。B+树索引能够找到的只是被查找的数据行所在的页。然后通过数据库把页读入内存，在内存中进行查找，最后得到所需的数据。

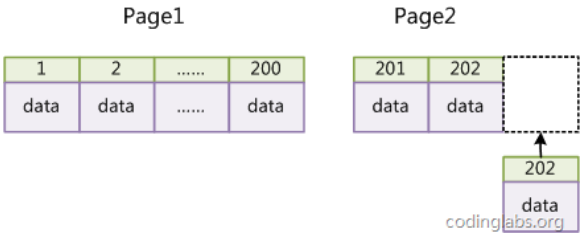


无特殊需求下Innodb建议使用与业务无关的自增ID作为主键

InnoDB引擎使用聚集索引，数据记录本身被存于主索引（一颗B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放。

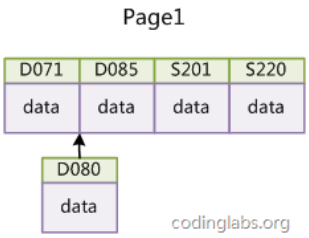
因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）。

如果使用自增主键：每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页。如下图所示：



这样就会形成一个紧凑的索引结构，近似顺序填满。由于每次插入时也不需要移动已有数据，因此效率很高，也不会增加很多开销在维护索引上。

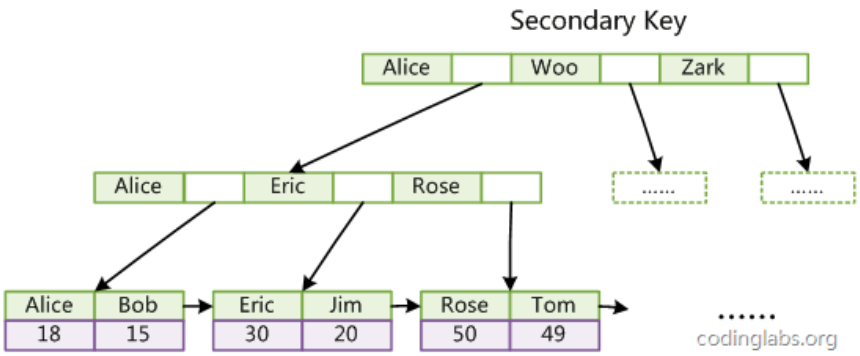
如果使用非自增主键：由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置：



此时MySQL不得不为了将新记录插到合适位置而移动数据。甚至目标页面可能已经被回写到磁盘上而从缓存中清掉了，此时又要从磁盘上读回来。这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

辅助索引(Secondary Index)

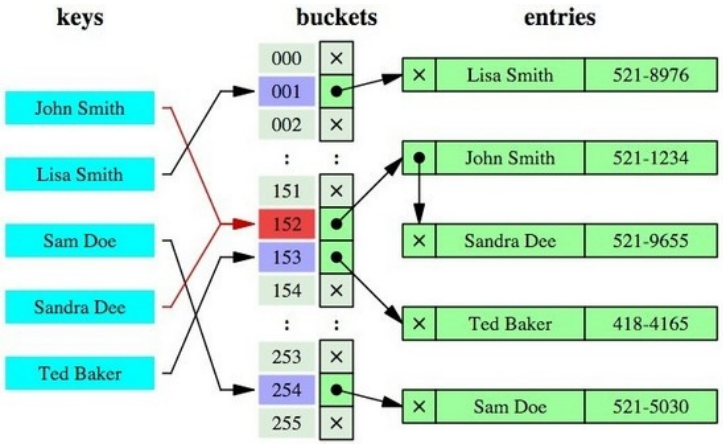
InnoDB中所有除聚集索引以外的所有索引都被称为辅助索引。对于辅助索引，叶子节点并不包含行记录的全部数据。叶子节点除了包含键值以外，每个叶子节点的索引行中还包含一个主键值。通过主键值InnoDB存储引擎可以再次在聚集索引中找到对应索引项的行数据。辅助索引的存在不会影响数据在聚集索引中的组织，因此每张表上可以存在多个辅助索引。当通过辅助索引来查找数据时，InnoDB存储引擎会遍历辅助索引并获得页中的指向主键索引的主键，然后再通过主键索引依次找到一个完整的行记录。因此，这就意味着通过辅助索引查找行，存储引擎先找到辅助索引的叶子节点，得到对应的主键值，然后根据该主键值去聚集索引上查找对应的行数据。也即一次辅助索引查找需要两次对B+树进行查找。



Hash 索引（Hash Index）

哈希索引基于哈希表实现，只有精确匹配索引所有列的查询才有效。对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码（hash code），哈希码是一个较小的值，并且不同键值的行计算出来的哈希码也不一样。哈希索引将所有的哈希码存储在索引中，同时在哈希表中保存指向每个数据行的指针。在MySQL中，只有Memory引擎显式支持哈希索引(NDB也支持，但这个不常用）。

InnoDB引擎有一个特殊的功能叫做“自适应哈希索引（adaptive hash index）”。当InnoDB注意到某些索引值被使用得非常频繁时，它会在内存中基于B-Tree索引之上再创建一个哈希索引，这样就让B-Tree索引也具有哈希索引的一些优点，比如快速的哈希查找。这是一个完全自动的、内部的行为，用户无法控制或者配置，不过如果有必要，完全可以关闭该功能。



哈希索引的限制

- 哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。不过，访问内存中的行的速度很快，所以大部分情况下这一点对性能的影响并不明显。
- 哈希索引并不是按照索引值顺序存储的，所以也就无法用于排序。
- 哈希索引也不支持部分索引列匹配查找，因为哈希索引始终是使用索引列的全部内容来计算哈希值的。例如，在数据列（A，B）上建立哈希索引，如果查询只有数据列A，则无法使用该索引。
- 哈希索引只支持等值比较查询，包括=、in、<=>（注意<>和<=>是不同的操作）。也不支持任何范围查询，例如where Price>100。
- 访问哈希索引的数据非常快，除非有很多哈希冲突（不同的索引列值却有相同的哈希值）。当出现哈希冲突的时候，存储引擎必须遍历链表中所有的行指针，逐行进行比较，直到找到所有符合条件的行。
- 如果哈希冲突很多的话，一些索引维护操作的代价也会很高。例如，如果在某个选择性很低（哈希冲突很多）的列上建立哈希索引，那么当从表中删除一行时，存储引擎需要遍历对应哈希值的链表中的每一行，找到并删除对应行的引用，冲突越多，代价越大。

全文索引(FullText)

MySQL从版本5.6开始支持InnoDB引擎的全文索引，不过“从5.7.6版本开始才提供一种内建的全文索引ngram parser，支持CJK字符集（中文、日文、韩文，CJK有个共同点就是单词不像英语习惯那样根据空格进行分解的，因此传统的内建分词方式无法准确的对类似中文进行分词）”，我们使用的MySQL版本为5.6.28，并且需要建全文索引的数据部分是中文，所以这是个问题。

覆盖索引

覆盖索引（covering index）指一个查询语句的执行只用从索引中就能够取得，不必从数据表中读取。也可以称之为实现了索引覆盖。辅助索引不包含一整行的记录，因此可以大大减少IO操作。对于索引覆盖查询(index-covered query)，使用EXPLAIN时，可以在Extra一列中看到“Using index”。

建立索引的常用技巧

- 最左前缀匹配原则，非常重要的原则，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。=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。
- 尽量选择区分度高的列作为索引,区分度的公式是count(distinct col)/count(*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录。
- 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，原因很简单，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，显然成本太大。所以语句应该写成create_time = unix_timestamp(’2014-05-29’);。
- 尽量的扩展索引，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，当然要考虑原有数据和线上使用情况。

InnoDB Buffer Pool

InnoDB维护一个称为缓冲池（Buffer Pool）的内存存储区域，主要用来存储访问过的数据页面，它就是向操作系统申请的一块连续的内存空间，通过一定的算法可以使这块内存得到有效的管理。它是数据库系统中拥有最大块内存的系统模块。InnoDB存储引擎中数据的访问是按照页（也可以叫块，默认为16KB）的方式从数据库文件读取到Buffer Pool中的，然后在内存中用同样大小的内存空间来做一个映射。为了提高数据访问效率，数据库系统预先就分配了很多这样的空间，用来与文件中的数据进行交换。

innodb_buffer_pool_size是MySQL配置参数，它指定MySQL分配给InnoDB缓冲池的内存量。这是MySQL配置中最重要的设置之一，应根据可用的系统RAM进行配置。理想情况下，您可以将缓冲池的大小设置为尽可能大的值，从而为服务器上的其他进程留出足够的内存，而无需过多的分页。缓冲池越大，InnoDB 内存数据库的行为就越多，从磁盘读取数据一次，然后在后续读取期间从内存中访问数据。最常用的做法是将此值设置为系统RAM的70% - 80%。

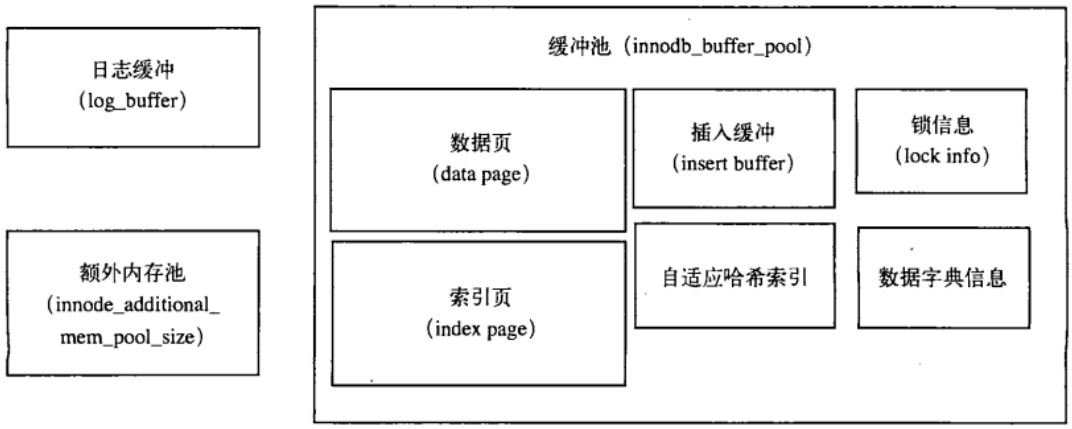


图2-3 InnoDB存储引擎内存结构

InnoDB缓冲池LRU算法(最近最少使用)

InnoDB管理buffer poll是将buffer pool作为一个list管理，基于LRU算法的管理。当有新的页信息要读入到buffer pool里面的时候，buffer pool就将最近最少使用的页信息从buffer pool当中驱逐出去，并且将新页加入到list的中间位置，这就是所谓的"中点插入策略"。一般情况下list头部存放的是热数据，就是所谓的young page（最近经常访问的数据），list尾部存放的就是old page（最近不被访问的数据）。

Buffer Pool Instances

在64位操作系统的情况下，可以拆分缓冲池成多个部分，这样可以在高并发的情况下最大可能的减少争用。配置多个Buffer Pool Instances能在很大程度上能够提高MySQL在高并发的情况下处理事物的性能，优化不同连接读取缓冲页的争用。我们可以通过设置 innodb_buffer_pool_instances来设置Buffer Pool Instances。当InnoDB buffer pool 足够大的时候，你能够从内存中读取时候能有一个较好的性能，但是也有可能碰到多个线程同时请求缓冲池的瓶颈。这个时候设置多个Buffer Pool Instances能够尽量减少连接的争用。这能够保证每次从内存读取的页都对应一个Buffer Pool Instances，而且这种对应关系是一个随机的关系。并不是热数据存放在一个Buffer Pool Instances下，内部也是通过hash算法来实现这个随机数的。每一个Buffer Pool Instances都有自己的free lists，LRU和其他的一些buffer pool的数据结构，各个Buffer Pool Instances是相对独立的。

innodb_buffer_pool_instances 的设置必须大于一才算得上是多配置，但是这个功能起作用的前提是innodb_buffer_pool_size的大小必须大于1G，理想情况下innodb_buffer_pool_instances的每一个instance都保证在1G以上。

innodb buffer pool预读

我们可以控制MySQL何时以何种方式预读数据进入buffer pool。 预读就是IO异步读取多个页数据读入buffer pool的一个过程，并且这些页被认为是很快就会被读取到的，当需要读取这些数据的时候就会将需要的页放在一个区当中，IInnoDB使用两种预读算法来提高I/O性能。

线性预读：能够预测将有那些数据很快能被读到的一种技术，因为buffer pool中的页数据是顺序访问的。我们可以通过设置innodb_read_ahead_threshold参数控制MySQL何时进行预读，也可以控制MySQL预读数据时候对于数据的敏感度，如果一个extent中的被顺序读取的page超过或者等于该参数变量时，Innodb将会异步的将下一个extent读取到buffer pool中，innodb_read_ahead_threshold可以设置为0-64的任何值，默认值为56，值越高，访问模式检查越严格。

随机预读：随机预读方式则是表示当同一个extent中的一些page在buffer pool中发现时，Innodb会将该extent中的剩余page一并读到buffer pool中，由于随机预读方式给Innodb code带来了一些不必要的复杂性，同时在性能也存在不稳定性，在5.5中已经将这种预读方式废弃。要启用此功能，请将配置变量设置innodb_random_read_ahead为ON。

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（INSERT BUFFER）、UNDO页的回收等。

插入缓冲（insert buffer）：

InnoDB是基于主键的聚簇索引。通常应用程序中行纪录的插入顺序是按照主键递增的顺序进行插入的。因此，插入聚集索引一般是顺序的，不需要磁盘的随机读写。对于非聚集索引的插入和更新，非实时更新索引页中，而是把若干对同一页面的更新缓存起来，合并为一次性更新操作。然后再以一定的频率执行插入缓冲和非聚集索引页子节点的合并操作，这时通常能将多个插入合并到一个操作中，这就大大提高了对非聚集索引执行插入和修改操作的性能。

插入缓冲使用的条件：

1、索引是辅助索引；
2、索引不是唯一的；

为什么必须不能是唯一索引呢？ 因为插入的时候，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的读取的情况发生，岂不是得不偿失，那Insert Buffer也失去了意义。

Change Buffer

change buffering是MySQL5.5加入的新特性，change buffering是insert buffer的加强，insert buffer只针对insert有效，change buffering对insert、delete、update(delete+insert)、purge都有效。当修改一个索引块(secondary index)时的数据时，索引块在buffter pool中不存在，修改信息就会被cache在change buffer中，当通过索引扫描把需要的索引块读取到buffer pool时，会和change buffer中修改信息合并，再择机写回disk。

您可以InnoDB 使用innodb_change_buffering 配置参数控制执行更改缓冲 的范围。您可以为插入，删除操作（当索引记录最初标记为删除时）和清除操作（物理删除索引记录时）启用或禁用缓冲。更新操作是插入和删除的组合。在MySQL 5.5及更高版本中，默认 innodb_change_buffering值从更改inserts为all。

允许的innodb_change_buffering 值包括：

- all

  默认值：缓冲区插入，删除标记操作和清除。
- none

  不要缓冲任何操作。
- inserts

  缓冲插入操作。
- deletes

  缓冲区删除标记操作。

- changes

  缓冲插入和删除标记操作。
- purges

  缓冲后台发生的物理删除操作。

两次写（double write）

InnoDB的page size一般是16KB，其数据校验也是针对这16KB来计算的，将数据写入到磁盘是以page为单位进行操作的。操作系统写文件是以4KB作为单位的，那么每写一个InnoDB的page到磁盘上，操作系统需要写4个块。而计算机硬件和操作系统，在极端情况下（比如断电）往往并不能保证这一操作的原子性，16K的数据，写入4K时，发生了系统断电或系统崩溃，只有一部分写是成功的，这种情况下就是partial page write（部分页写入）问题。这时page数据出现不一样的情形，从而形成一个"断裂"的page，使数据产生混乱。这个时候InnoDB对这种块错误是无 能为力的.

有人会认为系统恢复后，MySQL可以根据redo log进行恢复，而MySQL在恢复的过程中是检查page的checksum，checksum就是pgae的最后事务号，发生partial page write问题时，page已经损坏，找不到该page中的事务号，就无法恢复。

如果说插入缓冲带给InnoDB存储引擎的是性能，那么两次写带给InnoDB存储引擎的是数据的可靠性。主要用来解决部分写失败(partial page write)。doublewrite有两部分组成，一部分是内存中的doublewrite buffer，大小为2M，另外一部分就是物理磁盘上的共享表空间中连续的128个页，即两个区，大小同样为2M。在将页面写入数据文件之前，InnoDB首先将它们写入连续的表空间区域（称为doublewrite缓冲区）。 只有在双写缓冲写入和刷新完成之后，InnoDB才会将页面写入数据文件中的正确位置。 如果操作系统在页面写入过程中崩溃，InnoDB稍后可以在恢复期间从doublewrite缓冲区中找到正确的页面副本。

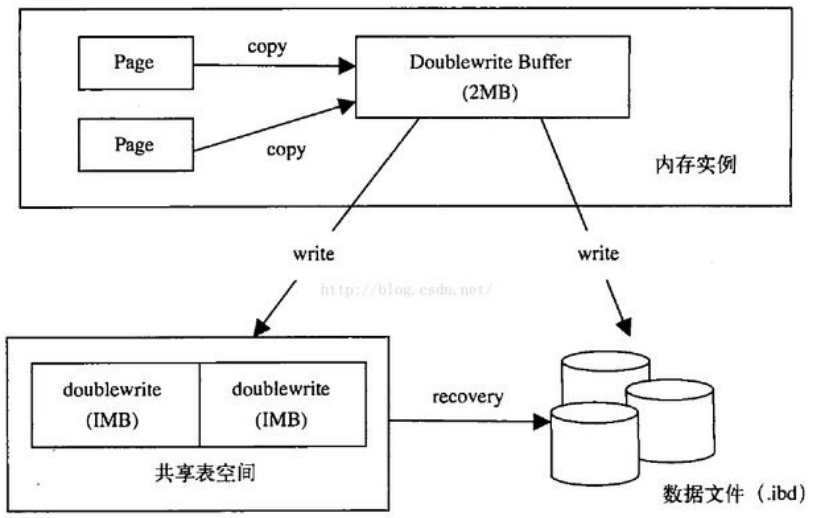


图2-4 InnoDB存储引擎doublewrite架构

注：以下命令可以查看doublewrite的使用 情况。mysql> show global status like ‘innodb_dblwr%’;slave上可以通过设置skip_innodb_doublewrite参数关闭两次写功能来提高性能，但是master上一定要开启此功能，保证数据安全。

事务

数据库事务（Transaction）是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。一方面，当多个应用程序并发访问数据库时，事务可以在应用程序间提供一个隔离方法，防止互相干扰。另一方面，事务为数据库操作序列提供了一个从失败恢复正常的方法。

事务具有四个特性：原子性（Atomicity）、一致性（Consistency）、隔离型（Isolation）、持久性（Durability），简称ACID。

隔离级别：

READ UNCOMMITTED（未提交读）

在READ UNCOMMITTED级别，事务中的修改，即使没有提交，对其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读（Dirty Read）。这个级别会导致很多问题，从性能上来说，READ UNCOMMITTED不会比其他的级别好太多，但却缺乏其他级别的很多好处，除非真的有非常必要的理由，在实际应用中一般很少使用。

READ COMMITTED（提交读）

大多数数据库系统的默认隔离级别都是READ COMMTTED（但MySQL不是）。READ COMMITTED满足前面提到的隔离性的简单定义：一个事务开始时，只能"看见"已经提交的事务所做的修改。换句话说，一个事务从开始直到提交之前，所做的任何修改对其他事务都是不可见的。这个级别有时候叫做不可重复读（nonrepeatble read），因为两次执行同样的查询，可能会得到不一样的结果

REPEATABLE READ(可重复读)

REPEATABLE READ解决了脏读的问题。该隔离级别保证了在同一个事务中多次读取同样记录结果是一致的。但是理论上，可重复读隔离级别还是无法解决另外一个幻读（Phantom Read）的问题。所谓幻读，指的是当某个事务在读取某个范围内的记录时，另一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行（Phantom Row）。InnoDB和XtraDB存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）解决了幻读的问题。

SERIALIZABLE（可串行化）

SERIALIZABLE是最高的隔离级别。它通过强制事务串行执行，避免了前面说的幻读的问题。简单来说，SERIALIZABLE会在读取每一行数据都加锁，所以可能导致大量的超时和锁争用问题。实际应用中也很少用到这个隔离级别，只有在非常需要确保数据的一致性而且可以接受没有并发的情况下，才考虑采用该级别。

多版本并发控制（MVCC)

大多数事务型存储引擎实现的都不是简单的行级锁。基于提升并发性能的考虑，它们一般都同时实现了多版本并发控制,MVCC是Multi Version Concurrency Control的简称。

可以认为MVCC是行级锁的一个变种，但是它在很多情况下避免了加锁操作，因此开销更低。虽然实现机制有所不同，但大都实现了非阻塞的读操作，写操作也只锁定必要的行。MVCC的实现，是通过保存数据在某个时间点的快照来实现的。也就是说，不管需要执行多长时间，每个事务看到的数据都是一致的。根据事务开始的时间不同，每个事务对同一张表，同一时刻看到的数据可能是不一样的。如果之前没有这方面的概念，这句话听起来就有点迷惑。熟悉了以后会发现，这句话其实还是很容易理解的。

前面说到不同存储引擎的MVCC实现是不同的，典型的有乐观（optimistic）并发控制和悲观（pessimistic）并发控制。下面我们通过InnoDB的简化版行为来说明MVCC是如何工作的。

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，一个保存了行的创建时间，一个保存行的过期时间（或删除时间）。当然存储的并不是实际的时间值，而是系统版本号（system version number）。每开始一个新的事务，系统版本号都会自动递增。事务开始时刻的系统版本号会作为事务的版本号，用来和查询到的每行记录的版本号进行比较。下面看一下在REPEATABLE READ隔离级别下，MVCC具体是如何操作的。

SELECT

InnoDB会根据以下两个条件检查每行记录：

- InnoDB只查找版本早于当前事务版本的数据行（也就是，行的系统版本号小于或等于事务的系统版本号），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。
- 行的删除版本要么未定义，要么大于当前事务版本号。这可以确保事务读取到的行，在事务开始之前未被删除。

只有符合上述两个条件的记录，才能返回作为查询结果。

INSERT
InnoDB为新插入的每一行保存当前系统版本号作为行版本号。

DELETE

InnoDB为删除的每一行保存当前系统版本号作为行删除标识。

UPDATE

InnoDB为插入一行新记录，保存当前系统版本号作为行版本号，同时保存当前系统版本号到原来的行作为行删除标识。

保存这两个额外系统版本号，使大多数读操作都可以不用加锁。这样设计使得读数据操作很简单，性能很好，并且也能保证只会读取到符合标准的行。不足之处是每行记录都需要额外的存储空间，需要做更多的行检查工作，以及一些额外的维护工作。

MVCC只在REPEATABLE READ和READ COMMITTED两个隔离级别下工作。其他两个隔离级别都和MVCC不兼容，因为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。

Redo log

InnoDB 有两块非常重要的日志，一个是undo log，另外一个是redo log，前者用来保证事务的原子性以及InnoDB的MVCC，后者用来保证事务的持久性。

任何对Innodb表的变动, redo log都要记录对数据的修改，redo日志就是记录要修改后的数据。redo 日志是保证事务一致性非常重要的手段，同时也可以使在bufferpool修改的数据不需要在事务提交时立刻写到磁盘上减少数据的IO从而提高整个系统的性能。这样的技术推迟了bufferpool页面的刷新，从而提升了数据库的吞吐，有效的降低了访问时延。带来的问题是额外的写redo log操作的开销。而为了保证数据的一致性，都要求WAL（Write Ahead Logging），即在持久化数据文件前，保证之前的redo日志已经写到磁盘。而redo 日志也不是直接写入文件，而是先写入redo log buffer，当需要将日志刷新到磁盘时（如事务提交）,将许多日志一起写入磁盘。

默认情况下，对应的物理文件位于数据库的data目录下的ib_logfile1&ib_logfile2

然后会通过以下三种方式将innodb日志缓冲区的日志刷新到磁盘
1，Master Thread 每秒一次执行刷新Innodb_log_buffer到重做日志文件。
2，每个事务提交时会将重做日志刷新到重做日志文件。
3，当重做日志缓存可用空间 少于一半时，重做日志缓存被刷新到重做日志文件

重做日志都是以512字节进行存储的。重做日志文件都是以块的方式进行保存，称为重做日志块。由于重做日志块的大小和磁盘扇区的大小一样，都是512字节，因此重做日志的写入可以保证原子性，不需要doublewrite技术。

redo log buffer

InnoDB存储引擎的内存区域除了有缓冲池外，还有重做日志缓冲（redo log buffer）。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。该值可由配置参数innodb_log_buffer_size控制，默认为8MB。

Checkpoint技术

数据库为了提高性能，数据页在内存修改后并不是每次都会刷到磁盘上。checkpoint之前的数据页保证一定落盘了，这样之前的日志就没有用了(由于InnoDB redolog日志循环使用，这时这部分日志就可以被覆盖)，checkpoint之后的数据页有可能落盘，也有可能没有落盘，所以checkpoint之后的日志在崩溃恢复的时候还是需要被使用的。InnoDB会依据脏页的刷新情况，定期推进checkpoint，从而减少数据库崩溃恢复的时间。检查点的信息在第一个日志文件的头部。

InnoDB 通过 Log Sequence Number, LSN 来标记版本，这是 8 字节的数字，每个页有 LSN，重做日志中也有 LSN，Checkpoint 也有 LSN，这个是联系三者的关键变量。

Checkpoint（检查点）也通过LSN的形式来保存，其表示页已经刷新到磁盘的LSN位置，当数据库重启时，仅需从检查点开始进行恢复操作。若检查点的LSN与重做日志相同，表示所有页都已经刷新到磁盘，不需要进行恢复操作了。用Redo Log LSN减去Checkpoint LSN就是需要恢复的数据。

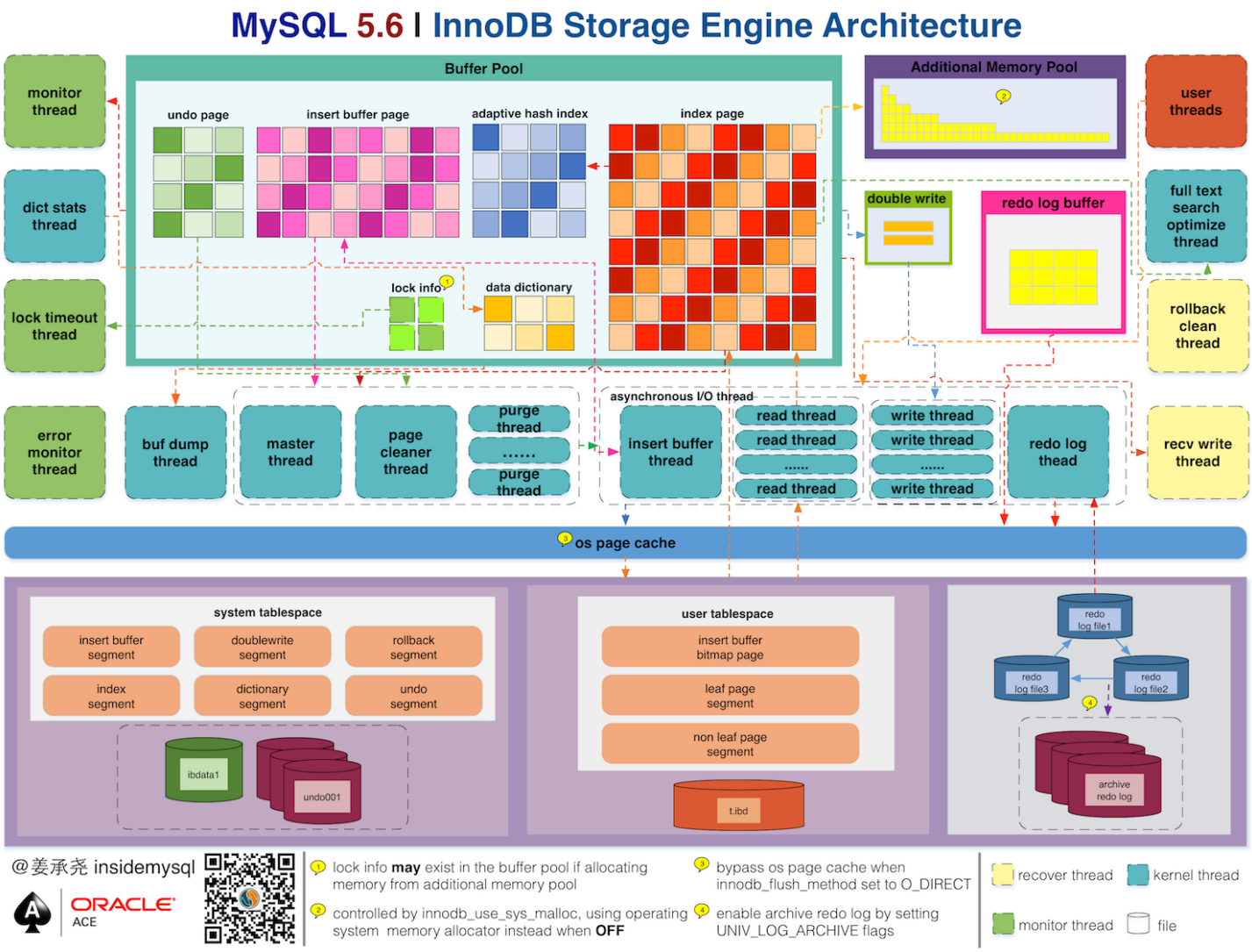
Undo log

Undo log是InnoDB MVCC事务特性的重要组成部分。当我们对记录做了变更操作时就会产生undo记录，Undo记录默认被记录到系统表空间(ibdata)中，但从5.6开始，也可以使用独立的Undo 表空间。undo只是逻辑地将数据库恢复，比如用insert恢复delete，而不是把页回滚为事务开始的样子，因为可能别的事务对同一个页进行修改。

Undo记录中存储的是老版本数据，当一个旧的事务需要读取数据时，为了能读取到老版本的数据，需要顺着undo链找到满足其可见性的记录。当版本链很长时，通常可以认为这是个比较耗时的操作（例如bug#69812）。

大多数对数据的变更操作包括INSERT/DELETE/UPDATE，其中INSERT操作在事务提交前只对当前事务可见，因此产生的Undo日志可以在事务提交后直接删除（谁会对刚插入的数据有可见性需求呢！！），而对于UPDATE/DELETE则需要维护多版本信息，在InnoDB里，UPDATE和DELETE操作产生的Undo日志被归成一类，即update_undo。

InnoDB 架构图


MySQL 5.6 I InnoDB Storage Engine Architecture
monitor thread
dict stats thread
lock timeout thread
error monitor thread
buf dump thread
master thread
page cleaner thread
purge thread
.....
purge thread
Buffer Pool
undo page
insert buffer page
adaptive hash index
index page
lock info
data dictionary
Additional Memory Pool
double write
redo log buffer
user threads
full text search optimize thread
rollback clean thread
recv write thread
redo log thead
asynchronous I/O thread
insert buffer thread
read thread
read thread
.....
read thread
write thread
write thread
.....
write thread
os page cache
system tablespace
insert buffer segment
doublewrite segment
rollback segment
index segment
dictionary segment
undo segment
ibdata1
undo001
user tablespace
insert buffer bitmap page
leaf page segment
non leaf page segment
t.ibd
redo log file1
redo log file3
redo log file2
archive redo log
@姜承尧 insidemysql
ORACLE ACE
lock info may exist in the buffer pool if allocating memory from additional memory pool
controlled by innodb_use_sys_malloc, using operating system memory allocator instead when OFF
bypass os page cache when innodb_flush_method set to O_DIRECT
enable archive redo log by setting UNIV_LOG_ARCHIVE flags
recover thread
kernel thread
monitor thread
file