# MySQL

## 函数

Concat()，将几个字符拼接成一个长的字符

Trim() 去除两边空格，Rtrim去除右边空格，LTrim

Date() 年月日

Year() 年

Month()月

可以用where Date(order\_date) Between ‘2009-1-1’ and ‘2009-1-30’表示订单下单的日期为1月，也可以用 Year(order\_date) = 2005 and Month(order\_date) = 9 来表示九月份下的订单

## 存储过程

是一组为了完成特定功能的SQL语句集，经过编译后存储在数据库中，用户通过编译时指定的存储过程的名字并给定参数（如果有参数的话）来调用它

优点：

* 增强SQL语言的功能和灵活性：可以将一些复杂的SQL语句封装到存储过程中
* 标准组件式编程：存储过程被创建后，可以在程序中多次被调用，不必重新编写该存储过程的SQL语句。而程序员可以随时对存储过程进行修改，对应用程序源码毫无影响
* 较快的执行速度：存储过程是预编译一次，然后可以多次使用，而批处理是每次都要预编译所以要慢一些
* 减少网络流量
* 作为一种安全机制来充分使用：存储过程能够避免没有权限的用户的访问，保证数据的安全

### 语法

**CREATE PROCEDURE**  **过程名([[IN|OUT|INOUT] 参数名 数据类型[,[IN|OUT|INOUT] 参数名 数据类型…]]) [特性 ...] 过程体**

**DELIMITER //  
 CREATE PROCEDURE myproc(OUT s int)  
 BEGIN  
 SELECT COUNT(\*) INTO s FROM students;  
 END  
 //  
DELIMITER ;**

### 分隔符

MySQL默认以";"为分隔符，如果没有声明分割符，则编译器会把存储过程当成SQL语句进行处理，因此编译过程会报错，所以要事先用“DELIMITER //”声明当前段分隔符，让编译器把两个"//"之间的内容当做存储过程的代码，不会执行这些代码；“DELIMITER ;”的意为把分隔符还原。

## 游标

## 4.索引

### 索引是什么?

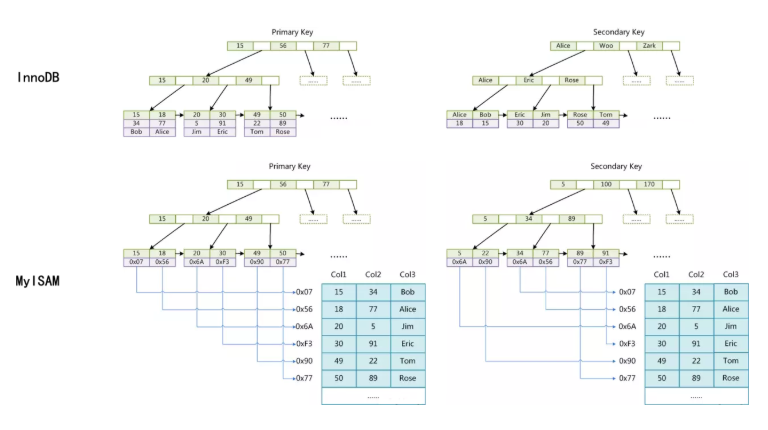
索引是帮助MySQL高效获取数据的数据结构。

### 索引类型

索引可以分为聚簇索引和非聚簇索引，细分可以分为普通索引，唯一索引，主键索引，联合索引，外键索引，全文索引几种。

**聚簇索引不是一种索引，而是一种数据存储组织方式 ！！！**

InnoDB可以看做是聚簇索引，因为它的B+树的节点包含了完整的数据记录。InnoDB的数据文件本身就是索引文件，数据表文件本身就是按B+Tree组织的一个索引结构，这颗树的叶节点data域保存了完整的数据记录。这个索引的Key是数据表的主键，因此InnoDB表数据文件本身就是索引文件。InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为Data域



以上是InnoDB跟MyISAM存储引擎的数据存储区别，可以看出InnoDB在B+Tree的叶子节点中保存的索引跟数据，而MyISAM保存的是索引跟数据的地址，故称为非聚集索引。所以InnoDB表数据文件本身就是一个索引表

在 InnoDB 中，又有聚簇索引和普通索引之分，聚簇索引根据主键来构建，叶子节点存放的是该主键对应的这一行记录，根据主键查询可以直接利用聚簇索引定位到所在记录。而普通索引根据申明这个索引时候的列来构建，叶子节点存放的是这一行记录对应的主键的值，根据普通索引查询需要先在普通索引上找到对应的主键的值，然后根据主键值去聚簇索引上查找记录，俗称回表。如果我们查询一整行记录的话，一定要去聚簇索引上查找，而如果我们只需要根据普通索引查询主键的值，由于这些值在普通索引上已经存在，所以并不需要回表，这个称为索引覆盖，在一定程度上可以提高查询效率。

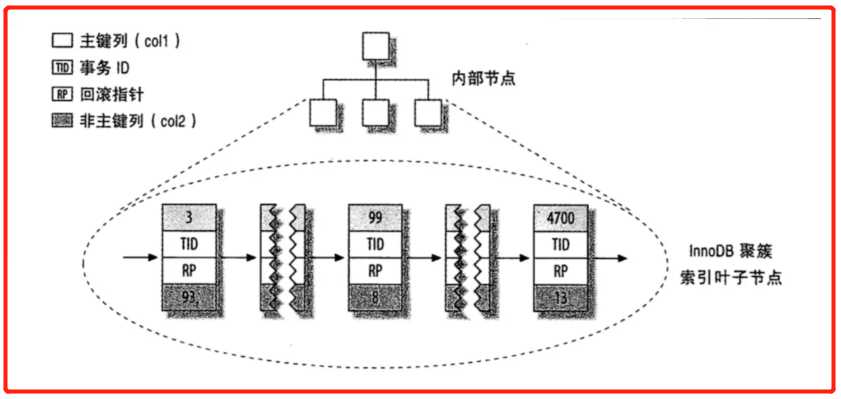
### 聚簇索引与非聚簇索引的区别

主键 col1 和 普通索引 col2。那么这俩索引跟聚簇非聚簇有啥关系呢？

会生成一个聚簇索引跟一个非聚簇索引（二级索引），也就是说会组织两颗索引树。主键索引会生成聚簇索引的树，而非主键索引（普通索引）则会生成非聚簇索引的树

**InnoDb 将通过主键来实现聚簇索引** ，如果没有主键则会选选一个唯一非空索引来实现。如果没有唯一非空索引则会隐式生成一个主键。

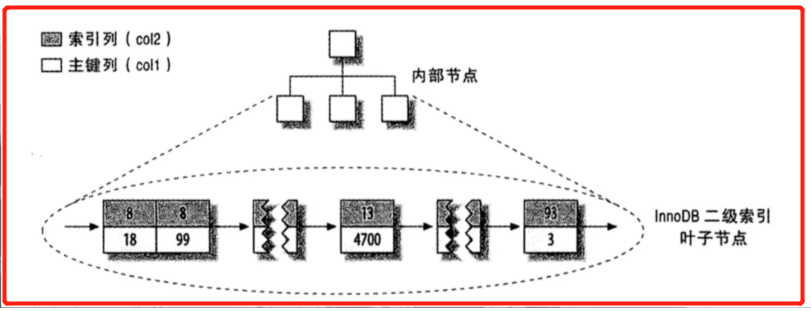
**索引列是主键 col1**



可以看出叶子结点除了存储索引值 列col1 (3~99~4700)值 之外还存储了其他列的值，如列col2(92~8~13),如果还有别的列的话也会存储，或者换句话说聚簇索引在叶子节点上存储了某个索引值对应的一行数据

**下图是非聚簇索引(二级索引)的数据组织方式**

**索引列是 col2**

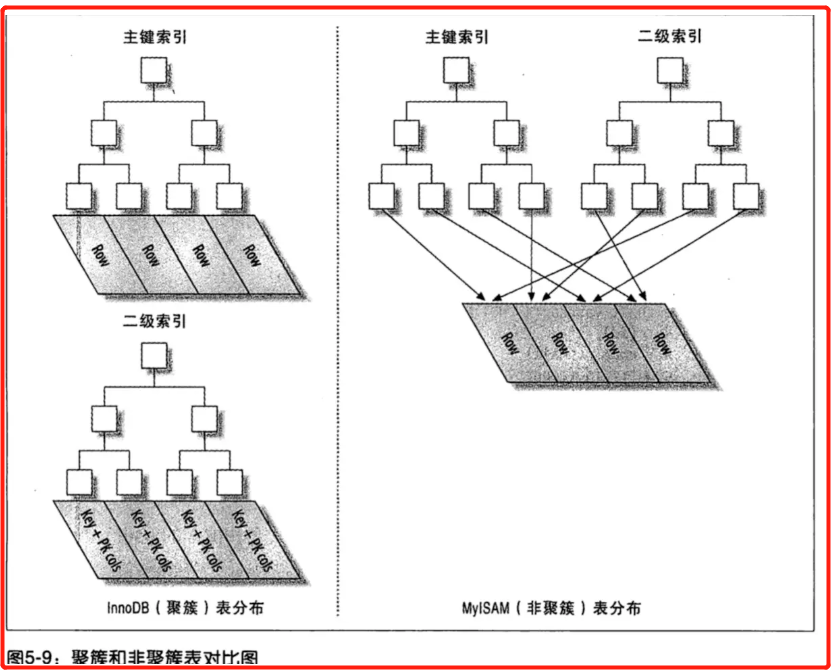


与聚簇索引不同的是非聚簇索引在索引树上除了索引值之外只存了主键值。而聚簇索引则存了一行数据

假如有一条sql 语句 select \* from test where col2=93;  
上面这条语句会经历两次从索引树查找过程

1. 第一步从非聚簇索引的索引树上找到包含col2=93的叶子结点，并定位到行的主键3
2. 从主键3在聚簇索引上找到对应这一行的数据

以上说的都是基于InnoDb存储引擎的,**MyISAM**是不支持聚簇索引的，因为他的数据文件和索引文件都是相互独立存放的**MyISAM**存储引擎的索引树的叶子结点不会存储主键值，而存一个指向对应行的地址或者指针，然后再从数据表文件里去找，如下图所示：



结论：

* 聚簇索引:  
  通常由主键或者非空唯一索引实现的，叶子节点存储了一整行数据
* 非聚簇索引：  
  又称二级索引，就是我们常用的普通索引，叶子节点存了索引值和主键值，在根据主键从聚簇索引查

### 索引选择性

对索引列和字符串前缀长度，都参考选择性（Selectivity）这个指标来确定：选择性定义为不重复的索引值和数据总记录条数的比值，其选择性越高，那么索引的查询效率也越高，譬如对于性别这种参数，建立索引根本没有意义。

Index Selectivity = Cardinality

显然选择性的取值范围为 (0, 1]，选择性越高的索引价值越大，这是由 B+Tree 的性质决定的。在实际的数据库中，我们可以通过以下语句来计算某列的选择性：

SELECT count(DISTINCT(title))/count(\*) AS Selectivity FROM titles;

### 主键

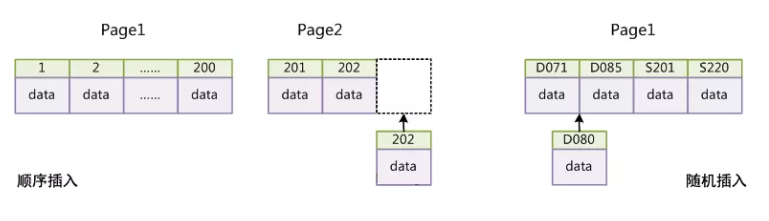
在InnoDB内部，表数据是优化主键快速查询而排列分布的，其查询速度最快，该索引中键值的逻辑顺序决定了表中相应的物理顺序。即使表中没有合适做主键的列，也推荐采用一个自动增长的整数主键（代理键），那么这个表在增加数据的时候是顺序存放的，而后续在别的表参考改外键查询的时候也会得到优化。

如果在创建表时没有显示地定义主键（Primary Key）,则InnoDB存储引擎会按照如下方式选择或创建主键：

* 首先表中是否有非空的唯一索引（Unique NOT NULL），如果有，则该列即为主键。
* 不符合上述条件，InnoDB 存储引擎自动创建一个 6 个字节大小的指针，用户不能查看或访问。

### 主键的选择

在[分布式 ID https://url.wx-coder.cn/tQ5eH](https://link.juejin.im/?target=https%3A%2F%2Furl.wx-coder.cn%2FtQ5eH) 一文中我们讨论过分布式场景下的分布式 ID 的选择策略，而在数据库中，我们同样会有这样的考量。首先，MySQL官方有明确的建议主键要尽量越短越好，36个字符长度不符合要求，如果主键是一个很长的字符串并且建立了很多普通索引，将造成普通索引占用很大的物理空间。并且主键最好是顺序递增的，否则在InnoDB引擎下，UUID的无序性可能会引起数据位置的频繁变动，严重影响性能。



自增ID在插入的时候可以保证相邻的两条记录可能在同一个数据块中，而订单这样的业务相关的连续性设计上可能没有自增ID好，导致连续插入可能在多个数据块中，增加了磁盘读写次数。

* 唯一性：自增 ID 很容易会被暴力破解，数据迁移的时候，特别是发生表格合并这种操作的时候，会不可避免地存在冲突。UUID 则能够保证唯一性，彻底避免冲突。
* 键长度：自增字段的长度较 UUID 小很多，这会对检索的性能有较大影响。Innodb 引擎进行数据检索时，也是先根据索引找到主键，然后根据主键找到记录；这样在主键长度短的情况下，会有较好的读性能。
* 并发性：自增 ID 并且高并发的情况下，竞争自增锁会降低数据库的吞吐能力。UUID 则能够在应用层生成 UUID，提高数据库的吞吐能力。

### 主键与唯一索引

主键就是唯一索引，但唯一索引不一定是主键，唯一索引可以为空，但是空值只能有一个，主键不能为空。对于单列索引，要求改列所有数据都不相同，但允许有Null值，对于多列的联合索引，要求这些列的组合是唯一的。唯一索引其本身既可以作为索引，实际中也可以用以产生数据约束，防止增加或者修改后产生相同数据，从而保证数据的完整性。

对于字符串类型，可以指定索引前缀长度(且对于 BLOB/TEXT 前缀长度参数是必须的)，在 InnoDB 表中其前缀长度最长是 767 bytes，且参数 M 是用 bytes 计量的。所以太长的字符串，建立 B+Tree 索引浪费比较大，这时候用手动模拟 HASH 索引是个方法，不过这种方式对字符串无法灵活的使用前缀方式查询(例如 LIKE 这类的操作)。

### 联合索引

创建一张数据表

CREATE TABLE `student` (  `studentId` int(11) NOT NULL,  `studentName` varchar(255) DEFAULT NULL,  `gradeId` int(11) DEFAULT NULL,  `schoolId` int(11) DEFAULT NULL,  PRIMARY KEY (`studentId`),  KEY `s\_g\_s` (`schoolId`,`gradeId`,`studentId`)

)

假设每个班有50名学生，一个年级有10个班，一所学校有4个年级，共有3所学校，那么总共会有6000名学生

若要查询出第2所学校3年级学生的姓名，SQL如下  
SELECT studentName FROM student WHERE schoolId=2 AND gradeId=3  
如果使用名为s\_g\_s的联合索引，那通过索引，MySQL可筛选掉大部分不满足查询条件的学生信息，在这个例子中，可筛掉5500条记录，这样MySQL只需回表查询剩余的500条记录即可得到结果。  
如果只使用schoolId上的单列索引，只能筛掉4000条记录，需回表扫描过滤剩余的2000条记录才能得到结果，从数量上看差了一个数量级。性能自然不佳。

若要查询出第2所学校3年级学生的姓名并按照studentId倒排，SQL如下  
SELECT studentName FROM student WHERE schoolId=2 AND gradeId=3 ORDER BY studentId DESC  
这条SQL若没有多列索引，在较大数据量下性能会很差。但有了s\_g\_s索引，排序可以在索引上直接完成，不用MySQL取回记录后，再在内存或者磁盘上进行一次排序。性能提升很大。

若要查询出第2所学校3年级学生的studentId，SQL如下  
SELECT studentId FROM student WHERE schoolId=2 AND gradeId=3  
对于这条查询，s\_g\_s索引包含所有需要查询的字段的值，MySQL根本不需要再去读取表中的记录，直接全部在索引上完成，这是性能最高的一种索引，通常称为“覆盖索引”。

InnoDB 的联合索引只支持从左到右：联合索引又叫复合索引。对于复合索引:Mysql从左到右的使用索引中的字段，一个查询可以只使用索引中的一部份，但只能是最左侧部分。例如索引是key index (a,b,c). 可以支持**a** |**a,b**| **a,b,c** 3种组合进行查找，但不支持 b,c进行查找 .当最左侧字段是常量引用时，索引就十分有效。

MyISAM：可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增.(百度没有搜索证明)

### 前缀索引

MySQL 的前缀索引可以分为三类：联合索引前缀，like 前缀和字符串前缀。

### 联合索引前缀与最左匹配（Leftmost Prefix）

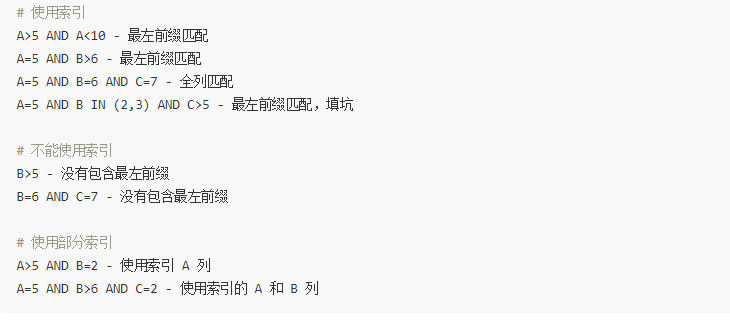
联合索引前缀指的是在建立多列索引的时候，必须按照从左到右的顺序使用全部或部分的索引列，才能充分的使用联合索引。比如：比如：(col1, col2, col3) 使用 (col1)、(col1, col2)、(col1, col2, col3) 有效。在查询语句中会一直向右匹配直到遇到范围查询(>,<,between,like)就停止匹配，其后的索引列将不会使用索引来优化查找。

以 (name, city, interest) 三个字段联合的索引为例，如果查询条件为 where name='Bush'; 那么就只需要根据 B+树定位到 name 字段第一个 Bush 所在的值，然后顺序扫描后续数据，直到找到第一个不为 Bush 的数据即可，扫描过程中将该索引片的数据 id 记录下来，最后根据 id 查询聚簇索引获取结果集。同理对于查询条件为 where name='Bush' and city='Chicago'; 的查询，MySQL 可以根据联合索引直接定位到中间灰色部分的索引片，然后获取该索引片的数据 id，最后根据 id 查询聚簇索引获取结果集。

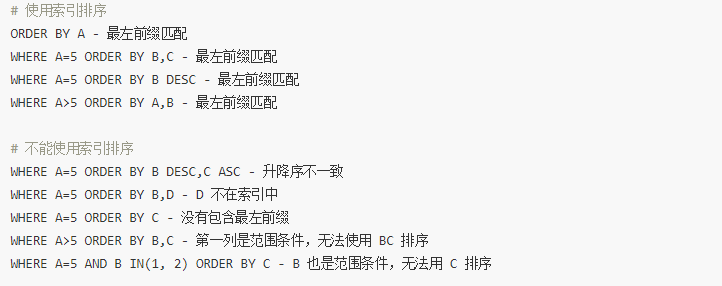
由此我们可以得出联合索引前缀的注意点：

* 无法跨越字段使用联合索引，如 where name='Bush' and interest='baseball';，对于该查询，name 字段是可以使用联合索引的第一个字段过滤大部分数据的，但是对于 interest 字段，其无法通过 B+ 树的特性直接定位第三个字段的索引片数据，比如这里的 baseball 可能分散在了第二条和第七条数据之中。最终，interest 字段其实进行的是覆盖索引扫描。
* 对于非等值条件，如 >、<、!= 等，联合索引前缀对于索引片的过滤只能到第一个使用非等值条件的字段为止，后续字段虽然在联合索引上也无法参与索引片的过滤。这里比如 where name='Bush' and city>'Chicago' and interest='baseball';，对于该查询条件，首先可以根据 name 字段过滤索引片中第一个字段的非 Bush 的数据，然后根据联合索引的第二个字段定位到索引片的 Chicago 位置，由于其是非等值条件，这里 MySQL 就会从定位的 Chicago 往下顺序扫描，由于 interest 字段是可能分散在索引第三个字段的任何位置的，因而第三个字段无法参与索引片的过滤。

因此 B-Tree 的列顺序非常重要，上述使用规则都和列顺序有关。对于实际的应用，一般要根据具体的需求，创建不同列和不同列顺序的索引。假设有索引 Index(A,B,C):



使用索引对结果进行排序，需要索引的顺序和 ORDER BY 子句中的顺序一致，并且所有列的升降序一致(ASC/DESC)。如果查询连接了多个表，只有在 ORDER BY 的列引用的是第一个表才可以(需要按序 JOIN)。



### like 前缀

对于 like 前缀，其是指在使用 like 查询时，如果使用的表达式为 first\_name like 'rMq%';那么其是可以用到 first\_name 字段的索引的。但是对于 first\_name like '%Chu%';，其就无法使用 first\_name 的索引。

对于like前缀，MySQL底层实际上是使用了补全策略来使用索引的，比如：first\_name like 'rMq%'; ，MySQL 会将其补全为两条数据：rMqAAAAA 和 rMqzzzzz，后面补全的部分长度为当前字段的最大长度。在使用索引查询时，Mysql就使用这两条数据进行索引定位，最后需要的结果就是这两个定位点的中间部分的数据。如下图所示：

### 覆盖索引

覆盖索引指的是对于查询中使用的除去参与索引过滤扫描的所有字段将其加入到该查询所使用的索引尾部的索引。覆盖索引扫描的在于由于查询中所使用的所有字段都在同一索引的字段，因而在进行查询时只需要在索引中获取相关数据即可，而不需要回磁盘扫描相应的数据，从而避免了查询中最耗时的磁盘 I/O 读取。对于如下查询：

**select** a, b, c **from** t **where** a='a' **and** b='b';

该查询中如果建立联合索引(a, b, c)，那么这就是使用了覆盖扫描的索引，因为对于该查询，可以使用索引的前两个字段 a 和 b 根据 where 条件进行索引片的过滤，对过滤后的索引片直接在索引中读取 a, b, c 三个字段的值即可，而无需回表扫描。

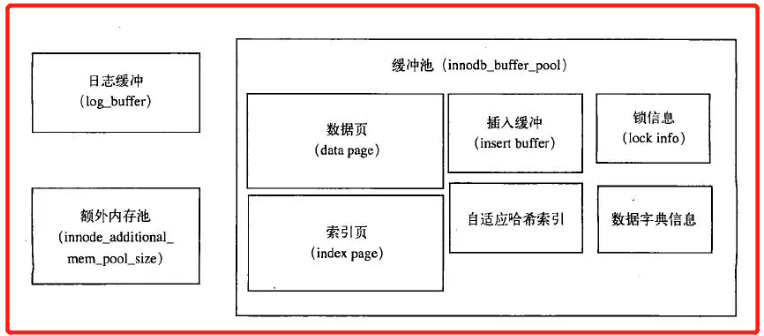
**覆盖索引好处**  
1.避免了对主键索引(聚簇)的二次查询  
2.由于不需要回表查询(从表数据文件)所以大大提升了Mysql缓存的负载

### 索引存储结构

MySQL 查询的时候会先通过索引定位到对应的数据页，然后检测数据页是否在缓冲池内，如果在就直接返回，如果不在就去聚簇索引中通过磁盘 IO 读取对应的数据页并放入缓冲池。一个数据页会包含多个数据行。缓存池通过 LRU 算法对数据页进行管理，也就是最频繁使用的数据页排在列表前面，不经常使用的排在队尾，当缓冲池满了的时候会淘汰掉队尾的数据页。从磁盘新读取到的数据页并不会放在队列头部而是放在中间位置，这个中间位置可以通过参数进行修。缓冲池也可以设置多个实例，数据页根据哈希算法决定放在哪个缓冲池。

### Memory Architecture | 内存架构

InnoDB 的内存主要有以下几个部分组成：缓冲池 (buffer pool)、重做日志缓冲池(redo log buffer）以及额外的内存池（additional memory pool），如下图所示：

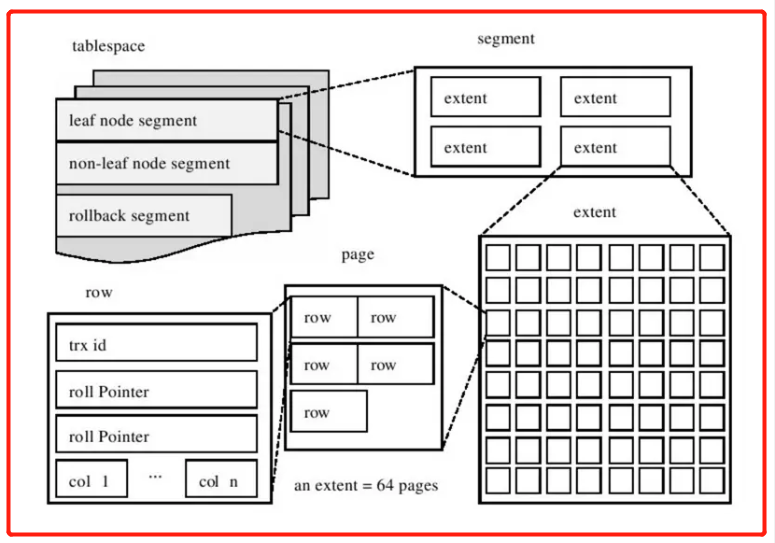


其中缓冲池占最大块内存，用来缓存各自数据，数据文件按页（每页 16K）读取到缓冲池，按最近最少使用算法（LRU）保留缓存数据。缓冲池缓冲的数据类型有：数据页、索引页、插入缓冲、自适应哈希索引、锁信息、数据字典信息等，其中数据页和索引页占了绝大部分内存。日志缓冲将重做日志信息先放入这个缓冲区，然后按一定频率（默认为 1s）将其刷新至重做日志文件。

InnoDB 通过一些列后台线程将相关操作进行异步处理，同时借助缓冲池来减小 CPU 和磁盘速度上的差异。当查询的时候会先通过索引定位到对应的数据页，然后检测数据页是否在缓冲池内，如果在就直接返回，如果不在就去聚簇索引中通过磁盘 IO 读取对应的数据页并放入缓冲池。一个数据页会包含多个数据行。缓存池通过 LRU 算法对数据页进行管理，也就是最频繁使用的数据页排在列表前面，不经常使用的排在队尾，当缓冲池满了的时候会淘汰掉队尾的数据页。从磁盘新读取到的数据页并不会放在队列头部而是放在中间位置，这个中间位置可以通过参数进行修。缓冲池也可以设置多个实例，数据页根据哈希算法决定放在哪个缓冲池。

### Storage Architecture | 存储结构

InnoDB 存储引擎的逻辑存储结构和 Oracle 大致相同，所有数据都被逻辑地存放在一个空间中，我们称之为表空间（tablespace）。表空间又由段（segment）、区（extent）、页（page）组成。页在一些文档中有时也称为块（block），1 extent = 64 pages，InnoDB 存储引擎的逻辑存储结构大致如图所示：



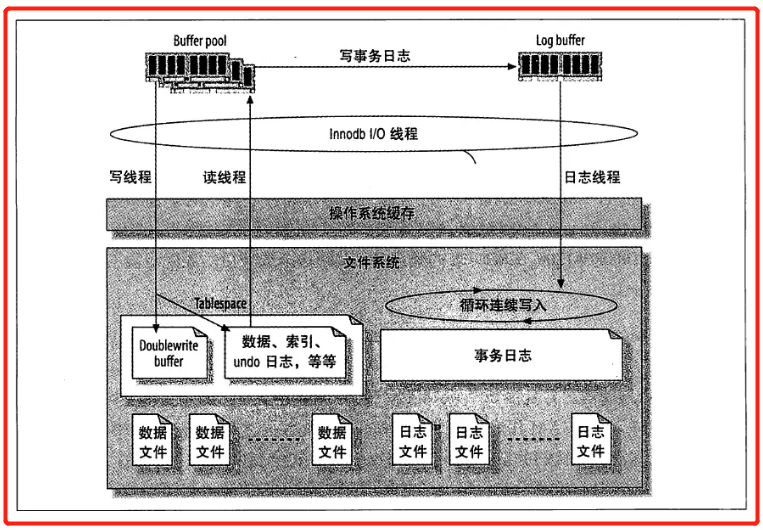
表空间作为存储结构的最高层，所有数据都存放在表空间中，默认情况下用一个共享表空间 ibdata1 ，如果开启了 innodb\_file\_per\_table 则每张表的数据将存储在单独的表空间中，也就是每张表都会有一个文件，

表空间由各个段构成，InnoDB 存储引擎由索引组织的，而索引中的叶子节点用来记录数据，存储在数据段，而非叶子节点用来构建索引，存储在索引段。区是由连续的页组成，任何情况下一个区都是 1MB，一个区中可以有多个页，每个页默认为 16KB ，所以默认情况下一个区中可以包含 64 个连续的页，页的大小是可以通过 innodb\_page\_size 设置，页中存储的是具体的行记录。一行记录最终以二进制的方式存储在文件里。

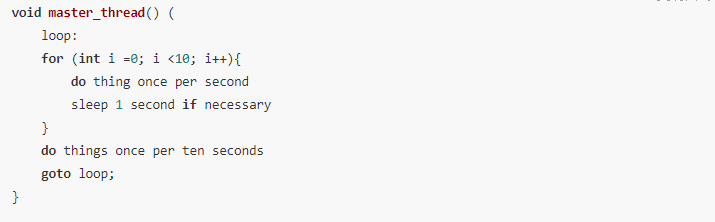
从物理意义上来看，InnoDB 表由共享表空间、日志文件组（更准确地说，应该是 Redo 文件组）、表结构定义文件组成。若将 innodb\_file\_per\_table 设置为 on，则每个表将独立地产生一个表空间文件，以 ibd 结尾，数据、索引、表的内部数据字典信息都将保存在这个单独的表空间文件中。表结构定义文件以 frm 结尾，这个是与存储引擎无关的，任何存储引擎的表结构定义文件都一样，为 .frm 文件。

### Process Architecture | 进程架构

默认情况下，InnoDB 的后台线程有 7 个，其中 4 个 IO thread, 1 个 Master thread, 1 个 Lock monitor thread, 一个 Error monitor thread。InnoDB 的主要工作都是在一个单独的 Master 线程里完成的。Master 线程的优先级最高，它主要分为以下几个循环：主循环（loop）、后台循环（background loop）、刷新循环（flush loop）、暂停循环（suspend loop）。



其中主循环的伪代码如下：



 其中每秒一次的操作包括：刷新日志缓冲区（总是），合并插入缓冲（可能），至多刷新 100 个脏数据页（可能），如果没有当前用户活动，切换至 background loop （可能）。

 其中每 10 秒一次的操作包括：合并至多 5 个插入缓冲（总是），刷新日志缓冲（总是），刷新 100 个或 10 个脏页到磁盘（总是），产生一个检查点（总是），删除无用 Undo 页 （总是）。

 后台循环，若当前没有用户活动或数据库关闭时，会切换至该循环执行以下操作：删除无用的 undo 页（总是），合并 20 个插入缓冲（总是），跳回到主循环（总是），不断刷新 100 个页，直到符合条件跳转到 flush loop（可能）。

 如果 flush loop 中也没有什么事情可做，边切换到 suspend loop，将 master 线程挂起。

### 最佳索引使用策略

**独立的列**

独立的列不是指单列索引，而是指索引列不能是表达式的一部分或者是函数的一部分。

select \* FROM test where col1 + 1 =100; // 不能是表达式一部分

select \* FROM test where ABS(col1) =100; // 不能是函数一部分

**最左匹配原则**

假如有个联合索引 key (col1,col2)。那么以下查询是索引无效的

select \* from test where col2 = 3;

select \* from test where col1 like '%3';

对于最左匹配原则，大家想一下B+树叶子结点的关联就差不多知道为啥需要最左匹配原则了，因为B+的叶子结点，从左到右以链表的形式关联的，所以我们查询的时候要么范围查询，要么有明确的左边一个开始的索引值，不能跳过或者不明确如like '%XYZ'这种查询。

**索引值不能是null值**

单列索引有null值会导致索引无效  
多列索引只要有个列有null值会导致索引无效

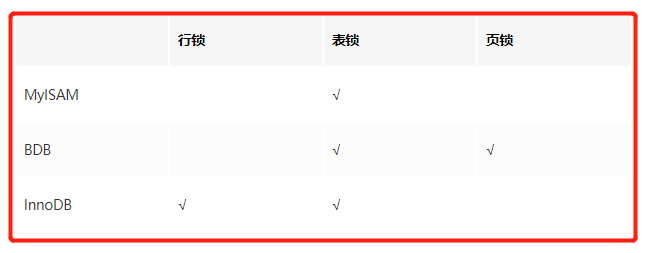
PS:Null有效，但是仅对Is null这个函数有效

**使用聚簇索引和覆盖索引大大提升读取性能**

因为聚簇索引和覆盖索引的索引树上就有了需要的字段，所以不需要回表文件查询，所以提升了查询速度

**使用短索引**  
如果很长的字符串进行查询，只需匹配一个前缀长度，这样能够节省大量索引空间

## 5.索引与锁

MySQL 为我们提供了行锁、表锁、页锁三种级别的锁，其中表锁开销小，加锁快，不会出现死锁，锁定粒度大，发生锁冲突概率高，并发度最低。行锁开销大，加锁慢，会出现死锁，锁定粒度小，发生锁冲突概率低，并发度高；页锁开销和加锁速度介于表锁和行锁之间；会出现死锁；锁定粒度介于表锁和行锁之间，并发度一般。每个存储引擎都可以有自己的锁策略，每个存储引擎都可以有自己的锁策略，例如 MyISAM 引擎仅支持表级锁，而 InnoDB 引擎除了支持表级锁外，也支持行级锁（默认）。  


InnoDB行锁是通过给索引上的索引项加锁，这一点 MySQL 与 Oracle 不同，后者是通过在数据块中对相应数据行加锁实现的，InnoDB 这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB 才使用行级锁，否则，InnoDB 将使用表锁，同样地，当 for update 的记录不存在会导致锁住全表。当表有多个索引时，不同的事务可以使用不同的索引锁定不同的行，另外，无论是使用主键索引

InnoDB 的加锁过程比较复杂，将所有扫描到的记录都加锁，范围查询会加间隙锁，然后加锁过程按照两阶段锁 2PL 来实现，也就是先加锁，然后所有的锁在事物提交的时候释放。加锁的策略会和数据库的隔离级别有关，在默认的可重复读的隔离级别的情况下，加锁的流程还会和查询条件中是否包含索引，是主键索引还是普通索引，是否是唯一索引等有关。

譬如对于 select \* from o\_order where order\_sn = '201912102322' for update; 这条 SQL 语句，在不同的索引情况下其加锁策略也不一致：

 order\_sn 是主键索引，这种情况将在主键索引上的 order\_sn = 201912102322 这条记录上加排他锁。

 order\_sn 是普通索引，并且是唯一索引，将会对普通索引上对应的一条记录加排他锁，对主键索引上对应的记录加排他锁。

 order\_sn 是普通索引，并且不是唯一索引，将会对普通索引上 order\_sn = 201912102322 一条或者多条记录加锁，并且对这些记录对应的主键索引上的记录加锁。这里除了加上行锁外，还会加上间隙锁，防止其他事务插入 order\_sn = 201912102322 的记录，然而如果是唯一索引就不需要间隙锁，行锁就可以。

 order\_sn 上没有索引，innoDB 将会在主键索引上全表扫描，这里并没有加表锁，而是将所有的记录都会加上行级排他锁，而实际上 innoDB 内部做了优化，当扫描到一行记录后发现不匹配就会把锁给释放，当然这个违背了 2PL 原则在事务提交的时候释放。这里除了对记录进行加锁，还会对每两个记录之间的间隙加锁，所以最终将会保存所有的间隙锁和 order\_sn = 201912102322 的行锁。

 order\_sn = 201912102322 这条记录不存在的情况下，如果 order\_sn 是主键索引，则会加一个间隙锁，而这个间隙是主键索引中 order\_sn 小于 201912102322 的第一条记录到大于 201912102322 的第一条记录。试想一下如果不加间隙锁，如果其他事物插入了一条 order\_sn = 201912102322 的记录，由于 select for update 是当前读，即使上面那个事物没有提交，如果在该事物中重新查询一次就会发生幻读。

 如果没有索引，则对扫描到的所有记录和间隙都加锁，如果不匹配行锁将会释放只剩下间隙锁。回忆一下上面讲的数据页的结果中又一个最大记录和最小记录，Infimum 和 Supremum Record，这两个记录在加间隙锁的时候就会用到。