# 存储引擎

MyIsam：不支持事务，读多写少（查询效率极高），比如电商平台的商品表

InnoDB：支持事务，写多读少，比如电商平台的订单表

## 概述

InnoDB是事务安全的MySQL存储引擎，是OLTP应用中核心表的首选存储引擎。

## 对比

1、InnoDB 是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高，但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，否则其他索引也会很大。而 MyISAM 是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针，主键索引和辅助索引是独立的。

MyISAM和InnoDB都使用B+树来实现索引：

1.1 MyISAM的索引与数据分开存储

1.2 MyISAM的索引叶子存储指针，主键索引与普通索引无太大区别

1.3 InnoDB的聚集索引和数据行统一存储

1.4 InnoDB的聚集索引存储数据行本身，普通索引存储主键

1.5 InnoDB一定有且只有一个聚集索引

1.6 InnoDB建议使用趋势递增整数作为PK，而不宜使用较长的列作为PK

2、InnoDB支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败。

3、InnoDB在MySQL 5.6之前不支持全文索引，而MyISAM一直都支持，如果你用的是老版本，查询效率上MyISAM要高。

4、InnoDB锁粒度是行锁，而MyISAM是表锁。

5、InnoDB支持事务，MyISAM不支持，对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务。

6、InnoDB不保存表的具体行数，执行select count(\*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数（类似MySQL中的伪列rownum），执行上述语句时只需要读出该变量即可，速度很快，但如果上述语句还包含了where子句，那么两者执行效率是一样的。

## 选择

两种存储引擎的选择，要结合你的业务场景来做选型，可以参考以下基本原则：

1、是否要支持事务，如果要请选择Innodb，如果不需要可以考虑MyISAM。

2、如果表中绝大多数都是读查询（有人总结出读:写比率大于100:1），可以考虑 MyISAM，如果既有读又有写，而且也挺频繁，请使用InnoDB。

3、系统崩溃后，MyISAM恢复起来更困难，能否接受。

4、MySQL5.5开始InnoDB 已经成为MySQL的默认引擎(之前是MyISAM )，说明其优势是有目共睹的，如果你不知道用什么，那就用InnoDB，至少不会差。

# 体系架构

InnoDB存储引擎有多个内存块，可以认为这些内存块组成了一个大的内存池，作用包括：

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，方便快速地读取，同时在对磁盘文件的数据修改之前在这里缓存；
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中，同时保证在数据库发生异常的情况下InnoDB能恢复到正常运行状态。

## 后台线程

### Master Thread

Master Thread是一个非常核心的后台线程，主要负责将缓存池中的数据异步刷新到磁盘，保持数据的一致性，包括**脏页的刷新、合并插入缓冲（INSERT BUFFER）、UNDO页的回收**等。

### IO Thread

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

### Purge Thread

事务被提交后，所使用的undolog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。

用户可以在MySQL数据库的配置文件中添加innodb\_purge\_threads=1命令来启用独立的Purge Thread。在InnoDB1.1版本中，即使将innodb\_purge\_threads设为大于1，InnoDB存储引擎启动时也会将其设为1。从InnoDB1.2版本开始，InnoDB支持多个Purge Thread，这样做的目的是为了进一步加快undo页的回收。同时由于Purge Thread需要离散地读取undo页，这样也能更进一步利用磁盘的随机读取性能。

### Page Cleaner Thread

Page Cleaner Thread作用是将之前版本中脏页的刷新操作都放入到单独的线程中来完成，其目的为了减轻Master Thread的工作及对于用户查询线程的阻塞，进一步提高InnoDB存储引擎的性能。

## 内存

### 缓冲池

InnoDB存储引擎是基于磁盘存储的，并将其中的记录按照页的方式进行管理。因此，可以将其视为基于磁盘的数据库系统（Disk-base Database）。在数据库系统中，由于CPU速度与磁盘速度之间的差异，基于磁盘的数据库系统通常使用缓冲池技术来提高数据库的整体性能。

对于InnoDB存储引擎而言，其缓冲池的配置通过参数innodb\_buffer\_pool\_size来设置。

存储池中缓存的数据页类型有：**索引页、数据页、插入缓存（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 directory）**等。

### LRU List/Free List/Flush List

通常来说，数据库中的缓冲池是通过LRU（Latest Recent Used，最近最少使用）算法来进行管理的。即最频繁使用的页在LRU列表的前端，而最少使用的页在LRU列表的尾端。当缓冲池不能存放新读取到的页时，将首先释放LRU列表中尾端的页。

### 重做日志缓冲

### 额外的内存池

# CheckPoint技术

为了避免在缓冲池到磁盘之间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。

# 关键特性

## 插入缓冲

## 两次写

## 自适应哈希索引

## 异步IO

## 刷新邻接页

# 启动/关闭/恢复

# 行格式

## 概述

## 使用

我们可以在创建或修改表的语句中指定行格式：

CREATE TABLE 表名 (列的信息) ROW\_FORMAT=行格式名称

ALTER TABLE 表名 ROW\_FORMAT=行格式名称

## 分类

一行记录可以以不同的格式存在InnoDB中，行格式分别是Compact、Redundant、Dynamic、Compressed行格式。

### COMPACT

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 变长字段长度列表 | NULL标志位 | 记录头信息 | 列1数据 | 列2数据 | … |

#### 记录的额外信息

这部分信息时服务器为了描述这条记录而不得不额外添加的一些信息，这些额外信息分为3类：

变长字段长度列表

NULL值列表

记录头信息

##### 变长字段长度列表

MySQL支持一些变长的数据类型，比如VARCHAR(M)、VARBINARY(M)、TEXT类型、BLOB类型，这些数据类型修饰列称为变长字段，变长字段中存储多少字节的数据不是固定的，所以我们在存储真实数据的时候需要顺便把这些数据占用的字节数也存起来。在Compact行格式中，把所有变长字段的真实数据占用的字节长度都存放在记录的开头部位，从而形成一个变长字段长度列表。

注：CHAR是一种固定长度的类型，VARCHAR是一种可变长度的类型。VARCHAR(M)，M代表最大能存储多少个字符（MySQL5.0.3以前是字节，以后就是字符）。

##### NULL值列表

Compact行格式会把可以为NULL的列统一管理起来，存一个标记位在NULL值列表中，如果表中没有允许存储NULL的列，则NULL值列表页不存在了。

二进制位的值为1时，代表该列的值为NULL。

二进制位的值为0时，代表该列的值不为NULL。

##### 记录头信息

除了变长字段长度列表、NULL值列表外，还有一个用于描述记录的记录头信息。它是由固定的5个字节组成，5个字节也就是40个二进制位，不同的位代表不同的意思，如图：

|  |  |  |
| --- | --- | --- |
| 名称 | 大小(单位bit) | 描述 |
| 预留位1 | 1 | 未使用 |
| 预留位2 | 1 | 未使用 |
| delete mask | 1 | 标记该记录是否被删除 |
| min\_rec\_mask | 1 | B+树的每层非叶子节点中的最小记录都会添加该标记 |
| n\_owned | 4 | 表示当前记录拥有的记录数 |
| heap\_no | 13 | 表示当前记录在记录堆的位置信息 |
| record\_type | 3 | 表示当前记录的类型：  0表示普通记录，  1表示B+树非叶子节点记录，  2表示最小记录，  3表示最大记录 |
| next\_record | 16 | 表示下一条记录的相对位置 |

#### 记录的真实数据

##### 隐藏列

记录的真实数据除了我们自己定义的列的数据以外，这会有三个隐藏列：

|  |  |  |  |
| --- | --- | --- | --- |
| 列名 | 是否需要 | 占用空间 | 描述 |
| row\_id | 否 | 6字节 | 行ID，唯一标识一条记录 |
| transaction\_id | 是 | 6字节 | 事务ID |
| roll\_pointer | 是 | 7字节 | 回滚指针 |

实际上这几列的真正名称其实是DB\_ROW\_ID、DB\_TRX\_ID、DB\_ROLL\_PTR。一个表没有手动定义主键，则会选取一个Unique键作为主键，如果连Unique键都没有的话，则会为表默认添加一个名为row\_id的隐藏列作为主键。所以row\_id是在没有自定义主键以及Unique键的情况下才会存在的。

##### 行溢出数据

VARCHAR(M)类型的列最多可以占用65535个字节，其中的M代表该类型最多存储的字符数量，如果我们使用ASCII字符集的话，一个字符就代表一个字节。

在CREATE TABLE的时候如创建的字段长度指定65535则提示Row size too large，是因为前面的额外信息占据3字节，如果创建65533则成功。

一个页大小一般是16KB即16384字节，而一个VARCHAR(M)类型的列最多可以存储65533个字节，这样就可能出现一个页存放不下一条记录。即这一行65533字节会占用多个页，这就是行溢出。

在跨多页的查询时，有两种方法：

1. 第一个页存储部分数据+下一页的地址，这样就可以实现多页的读取（其实就是B树底层实现）；
2. 第一页存储地址，快速定位到真正的数据地址，最后存储数据（B+树的底层实现）。

在Compact和Redundant行格式中，对于占用存储空间非常大的列，在记录的真实数据处只会存储该列的一部分数据，把剩下的数据分散存储在几个其他的页中，然后记录的真实数据处用20个字节存储指向这些页的地址（当然这20个字节中还包括这些分散在其他页面中的数据占用的字节数），从而可以找到剩余数据所在的页。

### Redundant

### Dynamic

Dynamic和Compressed行格式类似于Compact行格式，只不过在处理行溢出时有点分歧，它们不会在记录的真实数据处存储一部分数据，而是把所有的数据都存储到其他页面中（非聚集），只在记录的真实数据处存储其他页面的地址。另外，Compressed行格式会采用压缩算法对页面进行压缩。

### Compressed

# 数据页

## 概述

要了解页，需要了解局部性原理：从磁盘读取数据的时候，不是按照你需要的实际大小取数据，而是按照页为单位取数据（避免频繁的磁盘访问）。

**页是InnoDB管理存储空间的基本单位，一个页的大小默认是16KB（操作系统默认是4KB**，不要混淆）：

SHOW GLOBAL STATUS like ‘Innodb\_page\_size’;

## 结构



|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 中文名 | 占用空间 | 描述 |
| File Header | 文件头部 | 38字节 | 页的通用信息 |
| Page Header | 页面头部 | 56字节 | 数据页专有的信息 |
| Infimum+Supermum | 最小和最大记录 | 26字节 | 两个虚拟的行记录 |
| User Records | 用户记录 | 不确定 | 实际存储的行记录内容 |
| Free Space | 空闲空间 | 不确定 | 页中尚未使用的空间 |
| Page Directory | 页面目录 | 不确定 | 页中某些记录相对位置 |
| File Trailer | 文件尾部 | 8字节 | 校验页是否完整 |

### File Header

### Page Header

### Infimum+Supremum Records

### User Records

### Free Space

### Page Directory

### File Trailer

## 数据插入/查找



基本过程：

1. 首先第一条数据插入到用户数据区域；
2. 第二条数据插入前，首先进行排序（按照主键），然后确定插入位置（不一定第二条数据就在第一条数据后面）；
3. 以此类推；

注：这样的插入操作并不高，因为需要额外的排序，这样做的目的是提高查询的速度（比如插入的是1,2,5,8，在查找数据是否=4这种等值查找，只需要查找到5就可以确定不再这里了，不需要继续遍历）。

1. 查询的时候，先把数据所在的那一页数据全部从磁盘读取到内存，然后从记录中依次查找（但是极端情况下可能会遍历所有行数据的链表）；
2. 此时需要页目录进行快速定位（目录会用专门的算法对用户行数据进行分组），目录项中存储的是该组中主键（或索引）的最小值；
3. 这样查找数据的时候，先根据值的大小从目录项查找，然后具体定位到组，进而查找用户行数据；如果在对应组中没有找到，则不需要继续查找；
4. 如果数据非常大，则需要多个页：



1. 如果存储数据非常大，则页也非常多，页的链表很长，极端情况下数据查找可能会遍历整个页的链表，此时需要一个管理页的结构（每个结构存储所管理的页的最小值）：



上面叫目录页（大小也是16KB），主要存储的是目录的信息，下面的数据页，主要存储的是具体的数据。

数据页的next指针用于大于>操作，prev指针用于小于<操作（上述结构可以认为是B+树，但是与真正的B+树有所不同，严格的B+树只有next指针）。

注：经过上述分析可以总结一下B+树的特点：

1. 一个节点可以存储多个数据；
2. 所有非叶子节点的数据都冗余一份在叶子节点；

注：非叶子节点的数据是为了快速定位叶子节点/页的位置，而叶子节点上的数据是为了快速定位该页中数据的具体位置。

1. 叶子节点通过指针连接。

# 索引

索引其实就是可以帮助我们实现快速查找的数据结构。

# 自适应哈希

# 锁信息

# 数据字典