

MySql笔记-innodb无特殊说明，基于5.7版本

[零 参考资料 4](#_Toc65506370)

[一 概述 4](#_Toc65506371)

[1.1 概述 4](#_Toc65506372)

[1.2 创建一张表 4](#_Toc65506373)

[二 表的逻辑存储 5](#_Toc65506374)

[1 概述 5](#_Toc65506375)

[2 I/O和File Space Management 6](#_Toc65506376)

[2.1 Read-Ahead 6](#_Toc65506377)

[2.2 File Space Management 6](#_Toc65506378)

[2.3 Checkpoint 7](#_Toc65506379)

[3 文件格式 7](#_Toc65506380)

[3.1 概述 7](#_Toc65506381)

[4 System tablespace 7](#_Toc65506382)

[5 file-per-table Tablespaces 8](#_Toc65506383)

[5.1 概述 8](#_Toc65506384)

[5.2 优势 8](#_Toc65506385)

[5.3 劣势 8](#_Toc65506386)

[6 General Tablespaces 8](#_Toc65506387)

[6.1 概述 8](#_Toc65506388)

[6.2 限制 9](#_Toc65506389)

[7 Undo Tablespaces 9](#_Toc65506390)

[8 Temporary Tablespace 9](#_Toc65506391)

[9 Segment 10](#_Toc65506392)

[10 Extent 10](#_Toc65506393)

[11 Page 10](#_Toc65506394)

[11.1 概述 10](#_Toc65506395)

[11.2 File Header 12](#_Toc65506396)

[11.3 Page Header 12](#_Toc65506397)

[11.4 Infimun+Supremum Records 13](#_Toc65506398)

[11.5 User Records 13](#_Toc65506399)

[11.6 Free Space 13](#_Toc65506400)

[11.7 Page Directory 13](#_Toc65506401)

[11.8 File Trailer 14](#_Toc65506402)

[12 Row Format 14](#_Toc65506403)

[12.1 概述 14](#_Toc65506404)

[12.2 REDUNDANT 15](#_Toc65506405)

[12.3 COMPACT 16](#_Toc65506406)

[12.4 DYNAMIC 17](#_Toc65506407)

[12.5 COMPRESSED 17](#_Toc65506408)

[三 索引 18](#_Toc65506409)

[1 概述 18](#_Toc65506410)

[13 Sorted Index Builds 18](#_Toc65506411)

[14 Clustered index 19](#_Toc65506412)

[14.1 概述 19](#_Toc65506413)

[15 Secondary index 19](#_Toc65506414)

[15.1 概述 19](#_Toc65506415)

[16 Adaptive Hash Index 19](#_Toc65506416)

[17 Full-Text Index 20](#_Toc65506417)

[17.1 概述 20](#_Toc65506418)

[17.2 倒排索引(Inverted Index) 20](#_Toc65506419)

[17.3 Auxiliary index tables(FTS\_${table\_id}\_${index\_id}\_INDEX\_${1-6}) 21](#_Toc65506420)

[二 InnoDB Buffer 21](#_Toc65506421)

[1 概述 21](#_Toc65506422)

[1.1 InnoDB In-Memory Structures 21](#_Toc65506423)

[2 Buffer Pool 21](#_Toc65506424)

[3 Change Buffer 22](#_Toc65506425)

[4 Log Buffer 22](#_Toc65506426)

[5 Doublewrite Buffer 22](#_Toc65506427)

[三 Lock 23](#_Toc65506428)

[1 概述 23](#_Toc65506429)

[2 Shared and Exclusive Locks 23](#_Toc65506430)

[3 Intention Locks 23](#_Toc65506431)

[4 Record Locks 24](#_Toc65506432)

[5 Gap Locks 24](#_Toc65506433)

[6 Next-Key Locks 24](#_Toc65506434)

[7 Insert Intention Locks 24](#_Toc65506435)

[8 AUTO-INC Locks 25](#_Toc65506436)

[8.1 概述 25](#_Toc65506437)

[8.2 AUTO\_INCREMENT 25](#_Toc65506438)

[9 Predicate Locks 25](#_Toc65506439)

[四 日志(Log) 25](#_Toc65506440)

[1 redo log 25](#_Toc65506441)

[1.1 概述 25](#_Toc65506442)

[2 undo log 25](#_Toc65506443)

[2.1 概述 25](#_Toc65506444)

[3 Binary Log 26](#_Toc65506445)

[4 Slow Query Log 27](#_Toc65506446)

[5 Error Log 28](#_Toc65506447)

[6 DDL Log 28](#_Toc65506448)

[7 General Query Log 28](#_Toc65506449)

[五 事务(Transaction) 29](#_Toc65506450)

[1 概述 29](#_Toc65506451)

[2 ACID 29](#_Toc65506452)

[2.1 A(atomicity) 29](#_Toc65506453)

[2.2 C(cosistency) 29](#_Toc65506454)

[2.3 I(isolation) 30](#_Toc65506455)

[2.4 D(durability) 30](#_Toc65506456)

[3 Transaction Isolation 30](#_Toc65506457)

[3.1 概述 30](#_Toc65506458)

[3.2 READ UNCOMMITED 30](#_Toc65506459)

[3.3 READ COMMITTED 30](#_Toc65506460)

[3.4 REPEATABLE READ 31](#_Toc65506461)

[3.5 SERIALIZABLE 31](#_Toc65506462)

# 零 参考资料

《Mysql技术内幕 InnoDB存储引擎》

<https://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html>

<https://dev.mysql.com/doc/internals/en/innodb.html>

PS：未特别说明，基于5.7

# 概述

### 概述

* + - 1. 整体架构图



### 创建一张表

https://dev.mysql.com/doc/refman/5.7/en/using-innodb-tables.html

https://dev.mysql.com/doc/refman/5.7/en/innodb-data-dictionary.html

* + - 1. CREATE TABLE t1 (a INT, b CHAR (20), PRIMARY KEY (a)) ENGINE=InnoDB;
      2. 如果数据库实例默认引擎非Innodb的话，需要在每次建表时添加ENGINE=Innodb指定当前表使用innodb
      3. 每创建一张表:
         1. 会在实例的data目录下的database目录中创建一个xxxx.frm表文件，存储了table data dictionary等信息
         2. 关于tablespace

默认情况下创建在System tablespace中：

如果创建在file-per-table tablespace中，则会在database目录中创建一个xxx.ibd的tablespace文件

如果创建在 general tablespace中，则会关联到已存在的xxx.ibd的tablespace文件

* + - * 1. 关于row\_format

如果建表时未指定ROW\_FORMAT且实例也未配置innodb\_default\_row\_format，则默认为DYNAMIC

如果建表时未指定ROW\_FORMAT，但实例配置了innodb\_default\_row\_format，则值为innodb\_default\_row\_format的值

如果指定了ROW\_FORMAT，则以ROW\_FORMAT值为准

* + - * 1. 每张表都必须有个主键，可以是一个或多个字段，需要满足下列特性：

左边不能有blank

不能重复

很少变动

* + - * 1. 同时在内部，会在innodb data dictionary增加名为${databaseName}/${tableName}的entry记录表的信息： 比如在test数据库创建一张t1表，这个entry就是’test/t1’
      1. 可以通过SHOW\_TABLE\_STATUS statement查看表属性
      2. 也可以通过查询SELECT \* FROM INFORMATION\_SCHEMA.INNODB\_SYS\_TABLES WHERE NAME='${databaseName}/${tableName}'

# 表的逻辑存储

## 概述

<https://dev.mysql.com/doc/refman/5.7/en/innodb-file-space.html>

* + - 1. 所有数据都被逻辑地放在表空间中(tablespace),表空间又由段(segment),区(Extent),页(page,或者称为块(block))
      2. 每个tablespace都有N个pages组成，即innodb最小的管理单位
         1. 默认情况每个page大小为16kb
         2. 也可以通过innodb\_page\_size参数配置page大小,但一旦配置生效，则不能在修改，除非将实例重新初始化
         3. 有些page可能包含其他page的bitmap
      3. N个页被分组成多个extents，接着N个extents分组成多个Segment(这里的segment，不是rollback segment)，最后由多个segment组成tablespace
      4. 当tablespace新增一个segment时：
         1. innodb每次首先分配32个page给segment
         2. 然后分配对应的extents，但每次最多分配4个extents到segment以保证数据良好的连续性
         3. 但因为有些page包含其他page的bitmap，因此，有些extent可能不能整个分配给segment，只能个别的页分配
      5. 针对index，将分配两种segment：
         1. 组成B TREE结构的non-leaf node segment
         2. 另一个就是包含表实际数据的leaf node segment



## I/O和File Space Management

https://dev.mysql.com/doc/refman/5.7/en/innodb-disk-management.html

### Read-Ahead

* + - 1. 当 innodb判断数据大概率被马上使用到，就会执行Read-Ahead操作将数据预先加载到 buffer pool：
         1. sequential read-ahead:当innodb判断segment访问是顺序的，会将对应的页提前加载
         2. random read-ahead: 当innodb判断当前读取数据所在的空间中，其余很可能后续会立马访问，则会把剩余的数据都提前加载

### File Space Management

https://dev.mysql.com/doc/refman/5.7/en/innodb-file-space.html

* + - 1. 可以通过 innodb\_data\_file\_path参数配置system tablespace文件目录，当中的各个文件通过逻辑紧密连接从而形成system tablespace
      2. 新建的system tablespace，从第一个数据文件开始分配空间，用户无法控制表在system tablespace何处分配
      3. 为避免所有表和索引都存储在system tablespace，可以通过：
         1. 配置innodb\_file\_per\_table参数，将新建表自动放到各自独立的tablespace
         2. 用CREATE TABLESPACE语句建立新tablespace，并在建表时指定TABLESPACE，将新表建在指定的tablespace中
      4. 可以通过SHOW TABLE STATUS查看extents可用剩余空间，但这个值不包含innodb为清理或内部操作所预留的extents
      5. 当删除表数据时：
         1. 根据情况，innodb会收缩对应的B-tree索引
         2. 是否释放空间或对其他用户可见，根据删除模式是否释放tablespace中的page或extent确定
      6. 删除一张表或清理表中所有数据，可以释放可用空间给其他mysql用户，但物理空间仅能通过在不需要事务回滚或consistent read时自动执行的purge操作释放
      7. 最大row长度小于等于page大小一半
         1. 当行实际数据小于最大长度，则都存在一个page中
         2. 当行实际数据大于最大长度，variable-length columns将选择性存储在external off-page，直到行数据在最大长度之内
      8. variable-length columns具体使用规则根据Row format确定
         1. https://dev.mysql.com/doc/refman/5.7/en/innodb-row-format.html

### Checkpoint

https://dev.mysql.com/doc/refman/5.7/en/innodb-checkpoints.html

https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool-flushing.html#innodb-adaptive-flushing

* + - 1. innodb的checkpoints实现由两种：
         1. fuzzy checkpoint，具体类型：

Master thread checkpoint

FLUSH\_LRU\_LIST checkpoint

Async/Sync Flush Checkpoint

Dirty page too much Checkpoint

* + - * 1. Sharp checkpoint：

当innodb判断需要重用部分日志时，会产生一个sharp checkpoint

当日志文件满了时，也会触发一个sharp checkpoint

当innodb\_fast\_shutdown=1时，数据库关闭时将所有的dirtypage刷新回磁盘，会触发一个sharp checkpoint

* + - 1. 目的，系统崩溃恢复期间：
         1. 不需要重做所有日志
         2. innodb通过查找日志中的checkpoints label获知在这个label前所有数据库的修改
         3. 然后从这个checkpoint开始向前扫描，将所有修改都恢复
      2. 当日志文件大小配置很大时可以减少checkpointing期间的I/O，因此它通常被设置为大于等于buffer pool大小

## 文件格式

https://dev.mysql.com/doc/refman/5.7/en/innodb-file-format.html

### 概述

* + - 1. Innodb目前支持两种文件格式：Antelope、Barracuda
      2. Antelope
         1. 是最初的文件格式，之前甚至没有名字。。。
         2. 仅支持COMPACT和REDUNDANT行格式
      3. Barracuda，支持所有Row format，是当前mysql默认的file format
      4. 可以通过启动时修改innodb\_file\_format参数或运行时SET GLOBAL innodb\_file\_format，指定File format

## System tablespace

https://dev.mysql.com/doc/refman/5.7/en/innodb-system-tablespace.html

* + - 1. 可以有多个system tablespace数据文件，
         1. 默认情况下，为一个名为ibdata1的在数据目录下的初始大小为12M的单文件，并以8MB自动扩展
         2. 可以通过innodb\_data\_file\_path参数配置
      2. 默认情况下，存储了Innodb的InnoDB data dictionary，doublewrite buffer，change buffer，undo logs以及一些创建在此空间的table和index的
      3. 当建表时，innodb\_file\_per\_table参数禁用且未指定表空间时，默认在此空间创建表
      4. 当操作引起表空间数据减少时，不会释放物理存储空间，只是将释放出来的空间标记为可用

## file-per-table Tablespaces

https://dev.mysql.com/doc/refman/5.7/en/innodb-file-per-table-tablespaces.html

### 概述

* + - 1. 默认是启用
      2. 当innodb\_file\_per\_table启用且未指定表空间时，会为每个创建表动作在数据库目录下独立创建一个对应的文件名为${表名}.idb的表空间文件

### 

### 优势

* + - 1. 对表执行truncating 或 dropping操作后，可以直接释放物理存储空间
      2. 表ALTER TABLE操作，会释放操作过程中额外增加的空间需求
      3. TRUNCATE TABLE操作性能相对更好
      4. 此表空间文件可以创建在不同的存储空间中
      5. 可以从其他Mysql直接迁移指定的表
      6. 可以使用Barracuda文件格式
      7. 当数据异常时可以更快速修复以及提高恢复概率
      8. 在不打断其他表使用的情况下，可以迅速恢复/备份表
      9. 可以通过监视表空间文件，监视文件系统上的表
      10. 在Linux系统中，当设置innodb\_flush\_method值为O\_DIRECT时，可以提高性能
      11. 每个表都是64TB大小极限(Tables in a shared tablespace are limited in size by the 64TB tablespace size limit. By comparison, each file-per-table tablespace has a 64TB size limit, which provides plenty of room for individual tables to grow in size.)

### 劣势

* + - 1. 此表空间只能被指定表使用，导致资源浪费
      2. 会导致有很多fsync操作
      3. mysqld必须保存很多文件句柄，导致影响性能
      4. 每个表需要更多文件描述符
      5. 如果没管理好，会产生很多碎片，影响dropTable和表扫描操作。反之会提高性能
         1. 当drop table操作导致扫描缓冲池时，会用到一个内部锁，所以在此模式下，可能会延迟其他操作执行
         2. 配置innodb\_autoextend\_increment只应用于共享表空间(system tablespace/general tablespace)文件，这个空间固定每次扩展4MB

## General Tablespaces

https://dev.mysql.com/doc/refman/5.7/en/general-tablespaces.html

### 概述

* + - 1. 此空间是个通过CREATE TABLESPACE创建，可共享的表空间：
         1. 文件后缀为.isl
         2. 不属于任何数据库，即使空间中的表都是某特定数据库的
      2. 会一直保存tablespace metadata，因此存储表空间元数据消耗内存比file-per-table tablespaces少
      3. 表空间文件可以存在相对目录或独立的数据目录
      4. 支持Antelop和Barracuda文件格式，因此支持所有行格式和相关功能
      5. 参数innodb\_file\_format 和 innodb\_file\_per\_table不影响此表空间
      6. 在执行Create table时，可以指定tablespace
      7. 在执行Alter table时，可以通过Tablespace，对表做表空间迁移
      8. 由page格式不同，压缩和非压缩的表不能在同一个表空间共存
      9. 表空间如果包含压缩表，FILE\_BLOCK\_SIZE必须指定具体值，同时innodb\_page\_size不能为32k或64k,而且KEY\_BLOCK\_SIZE必须等于FILE\_BLOCK\_SIZE/1024
      10. 此表空间不属于特定database，drop database操作能删掉此空间的表，但删不掉此空间
      11. 和系统空间一样，修改/删除表数据不会改变空间物理大小，但会在表空间内部释放资源
      12. 表空间创建语句

1. CREATE TABLESPACE tablespace\_name
2. ADD DATAFILE 'file\_name'
3. [FILE\_BLOCK\_SIZE = value]
4. [ENGINE [=] engine\_name]

### 限制

* + - 1. 不能将其他已存在的表空间改成General tablespace
      2. 不支持创建临时General tablespace
      3. Generaltable不支持临时表
      4. 当tablespace中还有表时，不能删除tablespace，且仅能在空间中没有表后，手动删除
      5. 此空间中的表不支持DISCARD TABLESPACE 和IMPORT TABLESPACE操作
      6. MySQL 5.7.24以后可能会不支持此表空间中的表分区
      7. 不支持同主机主从复制

## Undo Tablespaces

https://dev.mysql.com/doc/refman/5.7/en/innodb-undo-tablespaces.html

* + - 1. 可以代替system tablespace 存放undo log
      2. 仅可以在实例启动前通过innodb\_undo\_tablespaces参数修改此tablespace数量，不能再运行时改变
      3. 可以通过innodb\_rollback\_segments参数配置定义每个undo tablespace分配多少个rollback segment
      4. 每个实例至少要有两个undo tablespace

## Temporary Tablespace

https://dev.mysql.com/doc/refman/5.7/en/innodb-temporary-tablespace.html

* + - 1. 对于未压缩临时表，用户创建的临时表和内部临时表都存在共享的临时表空间中
      2. 对于压缩表，会在指定目临时表目录下的file-pre-table tablespace 创建
      3. 如果没有配置innodb\_temp\_data\_file\_path，默认情况下，会在innodb\_data\_home\_dir 指定的目录下创建初始大小12MB的自动扩展的ibtmp1文件
      4. 会在初始化失败或者正常关闭系统情况下被自动移除，并可以在系统运行的任何时间重建
      5. 如果因为系统崩溃，导致temporary tablespace没被移除，则可以手动删除或通过重启服务自动移除和重建temporary tablespace

## Segment

<https://dev.mysql.com/doc/refman/5.7/en/innodb-file-space.html>

<https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_segment>

* + - 1. 在Innodb中，段的管理由引擎自动完成
      2. 这里的段跟回滚段不同
      3. 如果表空间是目录，段就是目录中文件
      4. 段每次扩容都一次1个区，每次释放也是一次一个区
      5. file-per-table 表空间，数据都在一个段中，对应的索引在各自的段中
      6. system tablespace表空间则包含各种不同的段
      7. 在Innodb中每个索引分配两个段，一个用于Btree的非叶子节点，一个用于叶子节点
      8. 个别Extent会分配在多个Segment中

## Extent

<https://dev.mysql.com/doc/refman/5.7/en/innodb-file-space.html>

https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos\_extent

* + - 1. 由连续页组成的空间，
         1. 当页大小为4KB、8KB、16KB时，区大小固定为1MB；
         2. 当页大小为32KB，对应的区大小为2MB，
         3. 页大小为64KB,对应的区大小为4MB
      2. 默认情况下，一个区包含64个大小为16KB的page

## Page

<https://dev.mysql.com/doc/refman/5.7/en/innodb-file-space.html>

<https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_page>

### 概述

* + - 1. 常见page类型：B-tree Page、undo Log page,System page, Transaction system page,Insert Buffer bitmap, Insert buffer free list, uncompressed blob page, compressed blob page
      2. 结构(5.6版本)：File Header，Page Header，Infimun和Supremum Records，User Records，Free Space，Page Directory，File Trailer



### File Header

* + - 1. 组成部分

|  |  |  |
| --- | --- | --- |
| 名称 | size(byte) | 说明 |
| FIL\_PAGE\_SPACE\_OR\_CHKSUM | 4 | checksum值 |
| FIL\_PAGE\_OFFSET | 4 | page在tablespace中的偏移值 |
| FIL\_PAGE\_PREV | 4 | 上一页 |
| FIL\_PAGE\_NEXT | 4 | 下一页 |
| FIL\_PAGE\_LSN | 8 | 最后修改LSN |
| FIL\_PAGE\_TYPE | 2 | 页类型；0x45BF，表示数据页 |
| FIL\_PAGE\_FILE\_FLUSH\_LSN | 8 | 在非独立表空间，代表文件至少被更新到了该LSN  对于独立表空间，都为0 |
| FIL\_PAGE\_ARCH\_LOG\_NO\_OR\_SPACE\_ID | 4 | 表示属于哪个tablespace |

* + - 1. FIL\_PAGE\_TYPE

|  |  |  |
| --- | --- | --- |
| 名称 | 16进制 | 说明 |
| FIL\_PAGE\_INDEX | 0x45BF | B+tree 叶节点 |
| FIL\_PAGE\_UNDO\_LOG | 0x0002 | undo log |
| FIL\_PAGE\_INODE | 0x0003 | 索引节点 |
| FIL\_PAGE\_IBUF\_FREE\_LIST | 0x0004 | Insert Buffer空闲列表 |
| FIL\_PAGE\_TYPE\_ALLOCATED | 0x0000 | 最新分配 |
| FIL\_PAGE\_IBUF\_BITMAP | 0x0005 | Insert buffer 位图 |
| FIL\_PAGE\_TYPE\_SYS | 0x0006 | 系统页 |
| FIL\_PAGE\_TYPE\_TRX\_SYS | 0x0007 | 事务系统数据 |
| FIL\_PAGE\_TYPE\_FSP\_HDR | 0x0008 | File space header |
| FIL\_PAGE\_TYPE\_XDES | 0x0009 | 扩张描述页 |
| FIL\_PAGE\_TYPE\_BLOB | 0x000A | BLOB |

### Page Header

|  |  |  |
| --- | --- | --- |
| 名称 | size(byte) | 说明 |
| PAGE\_N\_DIR\_SLOTS | 2 | slot数量 |
| PAGE\_HEAP\_TOP | 2 | 堆中第一个指针 |
| PAGE\_N\_HEAP | 2 | 堆中记录数，第15bit表示记录格式 |
| PAGE\_FREE | 2 | 指向可重用空间的首指针 |
| PAGE\_GARBAGE | 2 | 已删除的记录数 |
| PAGE\_LAST\_INSERT | 2 | 最后插入记录位置 |
| PAGE\_DIRECTION | 2 | 最后插入的方向：  PAGE\_LEFT(0x01)  PAGE\_RIGHT(0x02)  PAGE\_SAME\_REC(0x03)  PAGE\_SAME\_PAGE(0x04)  PAGE\_NO\_DIRECTION(0x05) |
| PAGE\_N\_DIRECTION | 2 | 一个方向连续插入的数量 |
| PAGE\_N\_RECS | 2 | 记录数 |
| PAGE\_MAX\_TRX\_ID | 8 | 最大事务ID，仅在secondary Index中定义 |
| PAGE\_LEVEL | 2 | 该页在索引数中的位置，0x00表示叶节点 |
| PAGE\_INDEX\_ID | 8 | 所属索引ID |
| PAGE\_BTR\_SEG\_LEAF | 10 | B+树数据页非叶子节点所在段的segment header，仅在Root页中定义 |
| PAGE\_BTR\_SEG\_TOP | 10 | B+树数据页所在段的segment header，仅在Root页中定义 |

### Infimun+Supremum Records

https://dev.mysql.com/doc/internals/en/innodb-infimum-and-supremum-records.html

* + - 1. 每个数据页都有两个虚拟记录行，用来限定边界：
         1. Infimum：比该页中任何主键值都要小的值，Greatest Lower Bound
         2. supremum：比该页任何可能大的值都大的值，Least Upper Bound
      2. 这两个值在页创建时建立，且不可能被删除



### User Records

https://dev.mysql.com/doc/internals/en/innodb-user-records.html

* + - 1. 用户所有数据都在这块
      2. 有两种方式浏览这块内容：
         1. 无序：基于堆实现，innodb不会依赖B-tree key排序，会将数据插入最后一行的末尾或随机插入因行被删除而释放出来的空间中
         2. 有序：基于B-tree，所有记录必须按照key值排序，因此每个记录都有额外的指针指向下个记录

### Free Space

* + - 1. 链表数据结构，当某行记录被删除，就会被加入到空闲链表待重新分配

### Page Directory

<https://dev.mysql.com/doc/internals/en/innodb-page-directory.html>

* + - 1. 保存了记录指针，这里的记录指针有时候会被称为slot或directory slot
      2. 并不是页中每个记录都有slot，因此采用了sparse directory实现 ，
      3. 一个slot中可能包含多个记录，记录按索引键顺序存放，可以通过二叉查找快速找到记录的指针
      4. 但二叉查找的结果只是个大致结果，需要通过recorder header的next\_record继续查找目标记录
      5. 一个B tree实际查找过程



### File Trailer

<https://dev.mysql.com/doc/internals/en/innodb-fil-trailer.html>

* + - 1. 用于检测页是否完整地写入磁盘
      2. 占用8byte，通过
         1. 将前4byte(checksum)跟header的FIL\_PAGE\_SPACE\_OR\_CHKSUM用checksum函数比较
         2. 后4byte跟header的FIL\_PAGE\_LSN比较，以此保证页完整
      3. checksum函数默认为crc32，可以通过innodb-checksum-algorithm参数指定具体算法

## Row Format

<https://dev.mysql.com/doc/refman/5.7/en/innodb-row-format.html>

### 概述

* + - 1. 行格式(row format): 决定了表的每行物理存储格式，直接影响到查询和DML操作的性能
      2. 硬盘上单页中存储的行更多，性能越好
      3. 行记录数据格式有四种

| 格式 | Compact Storage Characteristics | Enhanced Variable-Length Column Storage | Large Index Key Prefix Support | 支持压缩 | 表空间类型 | 文件格式 |
| --- | --- | --- | --- | --- | --- | --- |
| REDUNDANT | No | No | No | No | system  file-per-table, general | Antelope  Barracuda |
| COMPACT | Yes | No | No | No | system  file-per-table  general | Antelope  Barracuda |
| DYNAMIC | Yes | Yes | Yes | No | system  file-per-table  general | Barracuda |
| COMPRESSED | Yes | Yes | Yes | Yes | file-per-table  general | Barracuda |

* + - 1. Variable-length columns:
         1. 一种存储在B树索引节点的特例
         2. 如果值长度（byte）超过b-tree page大小，超过部分以singly-linked list形式存储在overflow pages上
         3. 这些列也称为off-page columns
      2. Variable-length columns实际大小根据character set确定，比如utf8mb4，单个字符串最长4btyes，如果是char(N)，则最大长度为N\*4 byte

### REDUNDANT

* + - 1. 老版本Mysql采用的数据结构，5.7版本为兼容老版本而依旧存在
      2. 支持Antelope和Barracuda文件
      3. 对于variable-length column，值的前768 bytes的内容存储在B树节点的索引中，其余的存储在overflow pages中
      4. 对于Fixed-length columns，如果值实际长度大于等于768 bytes ，会被转为variable-length columns形式存储
      5. 当列的值实际长度小于768 bytes，不会存储在overflow page，都存储在B树节点上，从而节省I/O开支
      6. 特征
         1. 每个索引记录包含6 byte(48bit)的头，用于将连续的记录连在一起，以及行级锁
         2. clustered index record包含:所有用户定义的列，一个6 byte的事务ID字段，一个7 byte的滚动指针(roll pointer)字段
         3. 如果用户没有定义表的主键，clustered index record还包括一个6 byte的row id字段
         4. 每个Secondary Index包含clustered indexd中的所有不在辅助索引中的主键字段
         5. 每个记录都包含一个指向所有字段的指针:

如果所有字段总长度小于128byte，则指针大小为1byte

如果所有字段总长度大于等于128byte，则指针大小为2byte

这类指针组被称为记录目录(The array of pointers is called the record directory)

指针指向的空间，是记录的数据部分

* + - * 1. 对于数据NULL值

无论可变长度列或固定长度列，都会在record directory预留1byte或2byte

对于variable-length column，NULL值不会在数据区预留空间

对于Fixed-length columns，NULL值也会预留固定长度的空间，目的是为了字段改为非NULL时不产生碎片

* + - 1. 对于采用可变长度字符集的char(N)列，占用最大空间=最大字符长度\*N，比如utf8mb4，单个字符串最长4btyes，如果是char(N)，则最大长度为N\*4 byte
      2. 存储结构示例



* + - 1. 记录头信息(总共48bit)

|  |  |  |
| --- | --- | --- |
| 名称 | size(bit) | 说明 |
| () | 1 | 未知 |
| () | 1 | 未知 |
| deleted\_flag | 1 | 是否删除 |
| min\_rec\_flag | 1 | 为1，如果该记录是被定义为最小的记录 |
| n\_owned | 4 | 记录数 |
| heap\_no | 13 | 该记录在索引堆中的排序记录 |
| n\_fields | 10 | 列数量 |
| 1byte\_offs\_flg | 1 | 偏移列表为1字节还是2字节 |
| next\_record | 16 | 页中下条记录相对位置 |

### COMPACT

* + - 1. 相对于REDUNDANT格式，以增加CPU消耗为代价，减少了20%存储空间
      2. 支持Antelope和Barracuda文件
      3. 在I/O密集型环境中，性能会有一定提升；在计算密集型，性能反而下降
      4. 对于variable-length column，值的前768 bytes的内容存储在B树节点的索引中，其余的存储在overflow page中
      5. 对于fixed-length columns，如果值实际长度大于等于768 bytes ，会被转为variable-length column方式存储
      6. 当列的值实际长度小于768 bytes，不会存储在overflow page，都存储在B树节点上，从而减少I/O消耗
      7. 特征
         1. 每个索引记录包含5 byte(40 bit)头，用于将连续的记录连在一起，以及行级锁，在其前面可能还有个可变长度的头部
         2. 记录头的可变长部分(The variable-length part of the record header)

包含 1个 位矢量(bit vector)用于表明有NULL列，

如果NULL列数量为N，则这个位矢量占CEILING(N/8) bytes

除了这个位矢量, NULL值不占用其他空间

也包含各个variable-length columns的长度，根据variable-length columns的最大长度，占1byte或2byte，逆序表示

如果所有列都是固定长度且NOT NULL，记录头就没有可变长部分

* + - * 1. 对于每个 非NULL的variable-length columns

记录头用1byte或2byte表示列长度

仅当当前记录需要存储在overflow page或字段最大长度超过255byte且实际长度超过127byte时，占用2byte

对于2中的字段，2 byte 表示了内部存储部分长度加上20 byte 指向外部存储的部分的指针 的长度；例如：内部数据长度时768 bytes，所以 2byte表示 768 + 20

* + - 1. 记录头后面是内容非NULL列的数据
      2. 聚集索引的记录包含所有用户定义的列：
         1. 一个6 byte的事务ID字段，一个7 byte的roll pointer字段
         2. 如果用户没有定义表的主键，clustered index还包括一个6 byte的row id字段
      3. 对于每个Secondary Index
         1. 包含clustered index中的所有不在secondary index中的主键字段
         2. 如果部分主键列是可变长度的，每个Secondary Index的头部都会有一个记录主键列长度的可变长度部分，即使辅助索引字列是固定长度
      4. 在内部，对于固定长度字符集(nonvariable-length character sets)
         1. 固定长度字符列采用固定长度格式存储
         2. 尾部的空格不会从varchar列截掉
      5. 在内部，对于可变长度字符集(utf8mb3、utf8mb4)
         1. 针对char(N)

先尝试通过截断尾部空格，达到用N bytes空间存储的目的

如果实际值仍超过了N bytes，尾部空格将被截取到列值的最小字节长度

char(N)最大长度 = 单字符最大长度 \* N

至少要预留N bytes空间，用于保证列完成更新并不产生碎片

* + - 1. 格式样例



* + - 1. 记录头信息(总共40bit)

|  |  |  |
| --- | --- | --- |
| 名称 | size(bit) | 说明 |
| () | 1 | 未知 |
| () | 1 | 未知 |
| deleted\_flag | 1 | 是否删除 |
| min\_rec\_flag | 1 | 为1，如果该记录是被定义为最小的记录 |
| n\_owned | 4 | 记录数 |
| heap\_no | 13 | 该记录在索引堆中的排序记录 |
| record\_type | 3 | 记录类型：  000 普通类型  001 B+节点指针  010 Infimum  011 Suprenum  1xx 保留 |
| next\_record | 16 | 页中下条记录相对位置 |

### DYNAMIC

* + - 1. 整体特征跟COMPACT一样，这里仅指出与COMPACT不同点
      2. 增加了对超长可变长度列(long variable-length columns)的支持
      3. 增加了对大索引key前缀(large index key prefixes)的支持
      4. 支持Barracuda文件
      5. 当建表时指定ROW\_FORMAT=DYNAMIC，long variable-length column的数据将完全存储在overflow page，而clustered index记录仅包含20byte的指向overflow page的指针。
      6. 列是否存储在off-page 依赖于页大小和行的总大小，当行太长时，最大的列将存在off-page，直到聚簇索引记录大小不超过B树页
      7. 当TEXT 和 BLOB 列长度小于等于40byte时，存储在同行中
      8. DYNAMIC格式规避了大数据量的列填满B树节点的问题
      9. 短的列将被保留在B树节点，从而最小化每行的 overflow page数量
      10. 支持索引key前缀3072 byte，需要通过innodb\_large\_prefix配置

### COMPRESSED

* + - 1. 整体特征跟COMPACT和 DYNAMIC一样，这里仅指出不同点
      2. 增加了支持对表和索引数据的压缩
      3. 支持Barracuda文件
      4. 采用了于DYNAMIC类似的内部细节存储off-page，同时也考虑到了因为表和索引数据被压缩以及用更小的页所带来的额外存储和性能消耗
      5. 不支持系统表空间
      6. 不支持同个表空间中同时存在压缩和不压缩的表

# 索引

## 概述

https://dev.mysql.com/doc/refman/5.7/en/innodb-indexes.html

* + - 1. 除了spatial index外，其他innodb index 都是用B-tree实现
      2. Spatial index 使用R-tree实现
      3. Index records都存于B-tree或R-tree的 leaf pages
      4. 如果innodb index page 的填充因子小于MERGE\_THRESHOLD(默认 50%)
         1. innodb会尝试通过收缩index tree来释放page
         2. MERGE\_THRESHOLD值应用于B-tree 和R-tree

## Sorted Index Builds

https://dev.mysql.com/doc/refman/5.7/en/sorted-index-builds.html

* + - 1. Innodb通过 bulkload执行B-tree index创建或rebuilding：
         1. 这个过程也称为sorted index build
         2. 不适用于spatial index
         3. 支持 fulltext index
      2. 采用 “bottom-up”方法执行Index build，有三段步骤：
         1. 第一段：

扫描clustered index，生成index entry 并将其加入sort buffer

当 sort buffer 满了，将里面的entry排序并写到临时中间文件中

这段过程也称为 run

* + - * 1. 第二段：将整合并排序文件中entry
        2. 第三段：将sorted entries插入到B-tree
      1. 基于“bottom-up”方法，B-tree的所有level都存有指向right-most leaf page 的引用
         1. right-most leaf page被分配在B-tree最深的一层，并根据sorted order 插入entries
         2. 当leaf page 满了：

将节点指针追加到父page，并在为下次insert分配一个sibling leaf page

重复操作，直到所有entries插入完成

* + - * 1. 当sibling leaf page分配后：

释放先前pinned leaf page的引用

新分配的 leaf页成为right-most leaf page以及默认插入

* + - 1. 当执行sorted index build 时，Redo logging会被禁用：
         1. 但会生成一个checkpoint确保index build过程不受奔溃或失败影响
         2. Checkpoint 会强制将所有dirty page写入到disk
         3. 在build期间，当clean page 数量低于指定阈值时，调用page cleaner 线程flush dirty page，确保checkpoint 操作效率

## Clustered index

https://dev.mysql.com/doc/refman/5.7/en/innodb-index-types.html

https://dev.mysql.com/doc/refman/5.7/en/innodb-physical-structure.html

### 概述

* + - 1. Clustered index 不等同于 primary key，index search只能查找到page层面
      2. 通常情况下：
         1. 当表存在primary key 时，innodb将此key作为clustered index
         2. 如果没有primary key，innodb 将第一个唯一非空索引，作为clustered index
         3. 如果连唯一非空索引都没有，则产生一个synthetic column作为clustered index：6 byte, 名为GEN\_CLUST\_INDEX, 包含row\_id
      3. 当一条新记录插入Innodb clustered index
         1. 首先会尝试预留page的1/16空间用于之后index 记录的插入和更新
         2. 如果index记录插入是有序的，则每一页都会填满15/16容量数据
         3. 如果index记录插入是无序的，则每页随机填满1/2至15/16容量的数据

## Secondary index

https://dev.mysql.com/doc/refman/5.7/en/innodb-index-types.html

### 概述

* + - 1. 所有非Clustered index都叫 Secondary index
      2. 每个secondary index 中记录都包含行的primary key字段

## Adaptive Hash Index

https://dev.mysql.com/doc/refman/5.7/en/innodb-adaptive-hash.html

* + - 1. 默认时开启，可以通过innodb\_adaptive\_hash\_index 参数配置是否启用或者启动时带上--skip-innodb-adaptive-hash-index参数禁用
      2. 根据搜索模式，innodb内部为经常访问的页，创建根据索引键前缀创建哈希索引:
         1. 索引键前缀长度不确定
         2. 仅Mysql内部创建并使用，用户无法干预
      3. 在5.7版本，每个index会绑定到指定的partition：
         1. 每个partition被一个separate latch保护
         2. 可以通过innodb\_adaptive\_hash\_index\_parts控制partition数量
      4. 可以通过SHOW ENGINE INNODB STATUS 语句查看状态
      5. 通过哈希算法实现，冲突机制采用链表方式，哈希函数采用除法散列方式



## Full-Text Index

https://dev.mysql.com/doc/refman/5.7/en/innodb-fulltext-index.html

### 概述

* + - 1. 提高text-based字段(CHAR、VARCHAR、TEXT等字段)的查询效率，
      2. 搜索方法参考：[这里](https://dev.mysql.com/doc/refman/5.7/en/fulltext-search.html)
      3. 通过full inverted index 实现

### 倒排索引(Inverted Index)

* + - 1. 存储了一组候选词，以及各个候选词出现的记录主键索引，同时为了支持proximity search，也存储了每个词在各个文档中的字节偏移量
      2. 建立一个全文索引，会生成一组辅助表，可以在INFORMATION\_SCHEMA.INNODB\_SYS\_TABLES表中查询到：

六张auxiliary index tables(FTS\_${table\_id}\_${index\_id}\_INDEX\_${1-6})、FTS\_${table\_id}\_BEING\_DELETED、FTS\_${table\_id}\_BEING\_DELETED\_CACHE、FTS\_${table\_id}\_DELETED、FTS\_${table\_id}\_DELETED\_CACHE、FTS\_${table\_id}\_CONFIG

例子: SELECT table\_id, name, space from INFORMATION\_SCHEMA.INNODB\_SYS\_TABLES WHERE name LIKE 'test/%';

|  |  |  |
| --- | --- | --- |
| table\_id | name | space |
| 333 | test/FTS\_0000000000000147\_00000000000001c9\_INDEX\_1 | 289 |
| 334 | test/FTS\_0000000000000147\_00000000000001c9\_INDEX\_2 | 290 |
| 335 | test/FTS\_0000000000000147\_00000000000001c9\_INDEX\_3 | 291 |
| 336 | test/FTS\_0000000000000147\_00000000000001c9\_INDEX\_4 | 292 |
| 337 | test/FTS\_0000000000000147\_00000000000001c9\_INDEX\_5 | 293 |
| 338 | test/FTS\_0000000000000147\_00000000000001c9\_INDEX\_6 | 294 |
| 330 | test/FTS\_0000000000000147\_BEING\_DELETED | 286 |
| 331 | test/FTS\_0000000000000147\_BEING\_DELETED\_CACHE | 287 |
| 332 | test/FTS\_0000000000000147\_CONFIG | 288 |
| 328 | test/FTS\_0000000000000147\_DELETED | 284 |
| 329 | test/FTS\_0000000000000147\_DELETED\_CACHE | 285 |
| 327 | test/opening\_lines | 283 |

* + - 1. 、如果主表创建在file-per-table tablespace，则index tables也存储在各自的tablespace

### Auxiliary index tables(FTS\_${table\_id}\_${index\_id}\_INDEX\_${1-6})

https://dev.mysql.com/doc/refman/5.7/en/innodb-fulltext-index.html#innodb-fulltext-index-tables

* + - 1. 当插入数据被tokenized时，当中每个关键词会带着位置信息和分配到的Document ID(DOC\_ID)，根据首个字符在编码集中的权重，插入其中一张auxiliary index tables
      2. 通过分区插入 auxiliary index tables实现并发索引创建，默认情况下，用两个线程执行tokenize、sort、insert words、关联数据到index table等流程操作，可以通过配置innodb\_ft\_sort\_pll\_degree参数修改执行线程数
      3. auxiliary index tables命名规则：FTS\_${table\_id}\_${index\_id}\_INDEX\_${1-6}
         1. table\_id：INFORMATION\_SCHEMA.INNODB\_SYS\_TABLES 表的table\_id，16进制表示，查表信息时，需要转为10进制
         2. index\_id：INFORMATION\_SCHEMA.INNODB\_SYS\_INDEXES 表的index\_id，16进制表示，查索引信息时，需要转为10进制

# InnoDB Buffer

## 概述

### InnoDB In-Memory Structures



## Buffer Pool

<https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool.html>

* + - 1. 实现算法是LUR算法的一个变种，默认操作流程：
         1. 将缓冲池分为两部分：

年轻子链表(new/young sublist)：占5/8缓存池容量，最新访问的页

年老子链表(old sublist)：占3/8缓存池容量，非最近访问的页，淘汰页时候选页

* + - * 1. 设置一个边界midpoint：在年轻子链表的表尾链接年老子链的表头位置
        2. 当因为用户查询操作或作为innodb自动执行的预读操作的一部分，需要读取一个新页进入缓冲池时，将新页插入到midpoint位置
        3. 当缓冲池中的一个页被存取

如果是用户操作引起，在第一次存取时，立即将此页放入到年轻子链表的头部

如果是因为innodb自动执行的预读操作引起，在第一次存取时，不会将此页放入到年轻子链表的头部，可能直到此页被淘汰，都不会被放入到年轻子链表的头部

* + - * 1. 随着数据库操作，缓冲池中的页会被逐步移向缓冲池尾部(年老子链表尾部)
        2. 每个页随着其他页被移到年轻子链表的头部而逐渐老化
        3. 年老子链中的页，同时也因为新页插入到midpoint而加速老化
        4. 最后，缓冲池中始终未被使用的页因为到达了年老子链表的尾部而被淘汰



## Change Buffer

<https://dev.mysql.com/doc/refman/5.7/en/innodb-change-buffer.html>

<https://dev.mysql.com/doc/refman/5.7/en/faqs-innodb-change-buffer.html>

* + - 1. 这部分用于缓存不在缓冲池中的secondary index 页的变更
      2. 这部分缓存会根据相关的读操作，定期合并到Buffer Pool中去，避免频繁的磁盘随机I/O
      3. 当系统有大量空闲时间或处于slow shutdown时，会对这部分缓冲执行清理操作，将数据索引页更新序列化地写入到磁盘
      4. 当涉及很多行数据和secondary index更新，合并操作可能持续几小时。
      5. 当事务提交后也可能会触发合并操作
      6. 在memory中，Change Buffer占用buffer pool一部分空间；在disk中，Change Buffer是系统表空间的一部分
      7. Change Buffer不支持包含降序索引列的索引

## Log Buffer

https://dev.mysql.com/doc/refman/5.7/en/innodb-redo-log-buffer.html

* + - 1. 用于缓冲日志数据，并定期写入到disk
      2. 通过innodb\_log\_buffer\_size 配置，默认16MB
      3. 大的Log Buffer可以使执行大事务而不需要在事务提交前将redo日志数据写入到disk中，减少disk I/O
      4. 参数innodb\_flush\_log\_at\_trx\_commit 控制log buffer 写disk方式，参数innodb\_flush\_log\_at\_timeout控制写频率

## Doublewrite Buffer

https://dev.mysql.com/doc/refman/5.7/en/innodb-doublewrite-buffer.html

https://dev.mysql.com/doc/refman/5.7/en/innodb-disk-io.html#innodb-disk-io-doublewrite

* + - 1. innodb在将page从buffer pool写到数据文件中的指定位置之前，会先将数据写到一个连续的tablespace空间，即doublewrite buffer pool
      2. 只有当数据成功写到doublewrite buffer pool后，innodb才会将数据写到data文件
      3. 如果system tablespace 文件存储在Fusion-io设备上，则自动禁用，否则默认为开启；可以通过innodb\_doublewrite参数开启
      4. 优势：
         1. 可以增加数据库崩溃或主机发生意外后安全恢复
         2. 如果没有设置innodb\_flush\_method为O\_DIRECT\_NO\_FSYNC，可以减少Unix上fsync()操作，提高性能

# Lock

## 概述

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html

https://dev.mysql.com/doc/refman/5.7/en/innodb-locks-set.html

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking-reads.html

https://dev.mysql.com/doc/refman/5.7/en/innodb-consistent-read.html

* + - 1. Innodb支持行locks和表 locks并存的多粒度锁
      2. Innodb 支持row-level locks，实际上可视为index-record locks
      3. 事务请求lock
         1. 如果跟现有lock不冲突，则可以获取，
         2. 反之则不能获取，并阻塞等待，直到现有的锁被释放
         3. 如果现有lock 一直不被释放，会产生deadlock

## Shared and Exclusive Locks

* + - 1. shared lock：允许事务读取一行数据， 简称S
      2. exclusive lock：允许事务更新/删除一行，简称 X
      3. 当事务T1获取R行一个shared lock后，事务T2也想操作R行：
         1. 如果T2也是请求一个shared lock，则同时获取，即T1和T2同时持有R行的shared lock
         2. 如果T2请求一个exclusive lock，请求将被阻塞，直到T1释放shared lock
      4. 如果事务T1 获取到R行的exclusive lock后，事务T2的任何请求都将被阻塞，直到T1释放exclusive lock

## Intention Locks

* + - 1. 是一种table-level locks，具体有两种：
         1. intention shared locks：当一个事务想在一张表中获取某行的shared locks，需要先获取此锁或更强的锁，简称IS
         2. intention exclusive lock：当一个事务想在一张表中获取某行的exclusive locks，需要先获取此锁，简称IX
      2. Intention Locks主要目的是用于哪些人锁定了某行，除了全表扫描请求，不会阻塞请求
      3. lock冲突简表：

|  | X | IX | S | IS |
| --- | --- | --- | --- | --- |
| X | 不兼容 | 不兼容 | 不兼容 | 不兼容 |
| IX | 不兼容 | 兼容 | 不兼容 | 兼容 |
| S | 不兼容 | 不兼容 | 兼容 | 兼容 |
| IS | 不兼容 | 兼容 | 兼容 | 兼容 |

## Record Locks

* + - 1. 锁住索引记录，阻止其他事务对锁定的行执行inserting、updating、deleting操作；比如(SELECT \* FROM t WHERE a=1 FOR UPDATE)
      2. Record Locks 一定是锁索引记录，如果表没有建立任何索引，默认锁建表时默认创建的clustered index

## Gap Locks

* + - 1. 锁在两个索引记录之间或者在第一个索引记录之前或最后个索引记录之后，用于阻止其他事务插入数据到Gap中
      2. 使用场景例子
         1. SELECT c1 FROM t WHERE c1 BETWEEN 10 AND 20 FOR UPDATE;
         2. 会产生一个Gap Lock，会锁住10-20之间所有记录
         3. 即阻止其他事务插入 t.c1=15 的记录，即使15值已经存在
      3. Gap lock 的跨度可以是N个索引值(N >= 0)
      4. 但搜索条件是单列时，
         1. 如果通过唯一索引搜索行，则不会产生Gap Lock，
         2. 如果通过非索引或非唯一索引搜索行，则会在行前加个Gap Lock
      5. 如果搜索条件是多列的组合唯一索引，则会产生Gap Lock
      6. 相同Gap位置，允许多个事务的Gap Locks共存，如果从索引中清除一条记录时，则必须合并记录上由不同事务持有的Gap Lock
      7. 可以通过将隔离级别设置为READ\_COMMITTED或者启用 innodb\_locks\_unsafe\_for\_binlog 配置，在搜索和索引扫描操作中禁用Gap lock，使Gap Lock仅在外键约束检查和唯一键检查时产生

## Next-Key Locks

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html#innodb-next-key-locks

* + - 1. 是一种Record Locks和在索引记录之前的Gap Lock的组合Locks
      2. 如果一个请求持有索引中记录R的shared或exclusive lock，其他请求就不能在索引序列中的记录R前插入新的索引记录，即在最大索引值之后的Gap位置设置lock
      3. 在innodb的默认事务隔离级别(REPEATABLE\_READ)下，next-key lock可以防止phantom rows出现

## Insert Intention Locks

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html#innodb-insert-intention-locks

* + - 1. 是一种Gap lock，在insert操作执行前设置
      2. 当多个事务插入数据到同一个Gap的不同位置的时，这个锁用于表明事务插入操作不需要阻塞
         1. 假设索引记录值4和7，两个事务分别尝试插入5和6，这时都会在获得各自对应的exclusive lock前锁住4和7之间的Insert intention Locks, 但因为插入行不冲突，所以，两个事务不会互相阻塞
         2. 还有个例子建议看官方文档

## AUTO-INC Locks

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html#innodb-auto-inc-locks

### 概述

* + - 1. 是一种特别的table-level lock，用于事务将数据插入到有自增字段的表中
      2. 当一个事务插入数据时，其他事务必须等待，以便当前事务获得连续的字段值
      3. 可用通过配置innodb\_autoinc\_lock\_mode参数指定此锁的实现算法

### AUTO\_INCREMENT

https://dev.mysql.com/doc/refman/5.7/en/innodb-auto-increment-handling.html

## Predicate Locks

https://dev.mysql.com/doc/refman/5.7/en/innodb-locking.html#innodb-predicate-locks

* + - 1. 用于支持对spatial字段索引
      2. 对于spatial index，由于在多维度数据中，没有绝对的序列观念，所以next-key lock无法很好支持REPEATABLE READ 或者 SERIALIZABLE 事务隔离级别，因为无法清楚获取”下一个”key
      3. 每个spatial index 包含一组minimum bounding rectangle(MBR)值，innodb强制对匹配查询条件的记录设置predicate lock，阻塞非当前事务修改/插入查询条件匹配的行

# 日志(Log)

## redo log

https://dev.mysql.com/doc/refman/5.7/en/innodb-redo-log.html

https://dev.mysql.com/doc/refman/5.7/en/innodb-recovery.html

### 概述

* + - 1. 用于事务执行异常时恢复数据
      2. 一般情况下，用于记录通过sql statement 或 low-level api修改表数据的结果
      3. 如果修改后的结果未在系统崩溃前写入数据文件，重启后会在初始化阶段，接收新请求之前，自动重新执行
      4. 默认情况下redo log循环写入ib\_logfile0 文件和 ib\_logfile1文件，

## undo log

https://dev.mysql.com/doc/refman/5.7/en/innodb-undo-logs.html

### 概述

* + - 1. Undo logs包含了一个读写事务的undo log record集合，存在于包含在rollback segment里的undo log segments里



* + - 1. Rollback segment 可以存在于 System tablespace、undo tablespace、temporary tablespace
      2. Undo log record记录了如何撤销一个事务对clustered index record的修改等信息
      3. 如果一个事务可以从undo log读一条被其他事务占用的记录的原始数据
      4. 在temporary tablespace中的undo log 用于事务对用户临时表的修改，这里的unlog跟redo log 不同：
         1. 不能用于灾难恢复
         2. 只用系统正常运行期间回滚数据
      5. 每个rollback segment中undo slot数量=innodb page size/16，根据innodb\_rollback\_segments配置：
         1. 支持最大128个rollback segment
         2. 其中固定32段分配给temporary tablespace
         3. 如果innodb\_rollback\_segments值大于32，剩余都分配给常规表的事务；
         4. 如果innodb\_rollback\_segments 值小于等于32，固定只分配一个rollback segment给常规表的事务
      6. 一个事务根据需要最多分配4个undo log slot：用户表的插入操作、用户表的更新/删除操作、用户临时表的插入、用户临时表的更新/删除操作
      7. 当一个undo log 被分配给一个事务，将会一直跟此事务绑定、复用，直到事务结束

## Binary Log

https://dev.mysql.com/doc/refman/5.7/en/binary-log.html

* + - 1. 包含：
         1. 表创建或表数据修改等数据库更改的事件信息
         2. 如果不是基于行日志记录，还可能包含修改数据的statement事件
         3. 每个statement更新数据的长度信息
      2. 用途：
         1. 用于master 向slave发送数据变更记录
         2. 用于数据恢复；在备份恢复后，会重新执行binary log中的记录，使数据库从备份点开始处于最新状态
      3. 不记录类似SELECT或SHOW等不改动数据的Statement的操作记录
      4. 仅当事务成功后，锁释放前才会记录Binary log
      5. 可以通过启动mysql时添加 --binlog-format参数指定binary log格式：
         1. --binlog-format=STATEMENT：基于sql statement记录和发送binary log
         2. --binlog-format=ROW：通过主键标识行如何被影响记录binary log，但有些DDL语句如CREATE TABLE,AlTER TABLE或DROP TABLE 等仍以STATEMENT格式记录
         3. --binlog-format=MIXED：混合模式，默认情况下基于sql statement记录，在以下情况中，会自动转为基于ROW记录binary log：

https://dev.mysql.com/doc/refman/5.7/en/binary-log-mixed.html

因为复制statement存在不确定因素， 当Mysql无法保证正确

执行STATEMENT中存在UUID()

当某表的AUTO\_INCREMENT字段被更新触发了一个触发器或存储过程

当创建View需要基于ROW

涉及对UDF调用

当执行一个基于ROW的statement涉及一些临时表，随后操作，除了访问操作，无论临时表实际是否被记录日志，都基于ROW，直到Session中所有临时表被drop；但临时表不能基于ROW记录日志，因此这时候所有后续statement都是不安全的，直到session清理了所有临时表

使用 FUND\_ROWS()、ROW\_COUNT()、USER()、CURRENT\_USER()、CURRENT\_USER、LOAD\_FILE()等内置

当statement引用系统参数时

当涉及的表有一个时系统日志表

* + - 1. 也可以在运行时通过SET GLOBAL/SESSION binlog\_format改变全局/当前会话的binary模式，但在mysql进行主从复制时可能会出现问题：
         1. 当改为STATEMENT时，binlog\_format 新值不会被复制
         2. 当改为MIX或ROW时，slave会忽略变更
      2. 当master实例发送了ROW格式的binlog时：
         1. slave实例不能将收到的binlog从ROW转为STATEMENT，因此slave的binlog\_format值必须为MIXED或ROW
         2. 当slave为STATEMENT时，会报Error executing row event: 'Cannot execute statement: impossible to write to binary log since statement is in row format and BINLOG\_FORMAT = STATEMENT.
      3. 当使用innodb时，事务隔离级别为READ\_COMMITTED或READ\_UNCOMMITTED：
         1. 只能使用ROW模式
         2. 如果被改为STATEMENT，很有可能因为Innodb不能执行insert操作而出错
      4. 存储引擎对binlog的支持：

| Storage Engine | Row Logging Supported | Statement Logging Supported |
| --- | --- | --- |
| ARCHIVE | Yes | Yes |
| BLACKHOLE | Yes | Yes |
| CSV | Yes | Yes |
| EXAMPLE | Yes | No |
| FEDERATED | Yes | Yes |
| HEAP | Yes | Yes |
| InnoDB | Yes | Yes when the transaction isolation level is [REPEATABLE READ](https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html#isolevel_repeatable-read) or [SERIALIZABLE](https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html#isolevel_serializable); No otherwise. |
| MyISAM | Yes | Yes |
| MERGE | Yes | Yes |
| [NDB](https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster.html) | Yes | No |

## Slow Query Log

https://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html

* + - 1. 记录了执行时间超过long\_query\_time参数配置的时间(单位s)和涉及行数超过min\_examined\_row\_limit配置的sql statement
      2. 可以通过mysqldumpslow命令总结日志内容
      3. 默认情况下slow query log是禁用的：
         1. 可以通过启动参数--slow\_query\_log参数是否启用
         2. 在运行时通过 全局系统参数 slow\_query\_log启用
      4. 执行时间不包括初始获取锁阶段，仅根据statement实际执行并释放所有锁所消耗的时间，记录日志，所以导致日志记录顺序可能跟实际执行顺序有所不同
      5. 默认情况下不记录Administrative statement(如ALTER TABLE, ANALYZE TABLE, CHECK TABLE, CREATE INDEX, DROP INDEX, OPTIMIZE TABLE, REPAIR TABLE等)，可以通过log\_slow\_admin\_statements参数配置
      6. 也不记录没有用到索引的查询，可以通过log\_queries\_not\_using\_indexes参数配置，但即使启用，也不会记录涉及少于2行的查询
      7. 对于slave实例：
         1. 默认情况下不会将复制查询写到slow query log，可以通过log\_slow\_slave\_statements参数启用
         2. 当log\_slow\_slave\_statements启用时：
         3. 如果binlog\_format=ROW，则也不记录
         4. 如果 binlog\_format=STATEMENT，则会记录慢查询
         5. 如果binlog\_format=MIXED，则只会记录statement部分
      8. 日志文件内容，看这里：[Slow Query Log Contents](https://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html#slow-query-log-contents)

## Error Log

https://dev.mysql.com/doc/refman/5.7/en/error-log.html

* + - 1. 记录实例启动关闭时间和运行期间error、warning等诊断信息

## DDL Log

https://dev.mysql.com/doc/refman/5.7/en/ddl-log.html

* + - 1. 也叫 metadata log，记录因为data definition statement(如 DROP TABLE、ALTER TABLE) 产生的metadata操作
      2. 用于恢复metadata操作崩溃
      3. 存储在名为ddl\_log.log的二进制文件中，无法尝试修改内容
      4. 因为这个日志文件仅在需要记录metadata操作创建，并会在mysqld启动成功后自动删除，所以可能不会在实例中被发现
      5. 文件最大支持4GB(即1048573个记录)，如果超过，必须重命名或移除当前文件
      6. 针对这个日志文件，没有任何可自定义的配置

## General Query Log

https://dev.mysql.com/doc/refman/5.7/en/query-log.html

* + - 1. 记录客户端连接/断开，以及收到的所有客户端发过来的SQL statement
      2. 当客户端链接时，日志包含连接协议connection\_type：TCP/IP、SSL/TLS、Socket、Named Pipe、Shared Memory
      3. mysql 实例按照接收的顺序记录日志，可能会跟实际执行顺序有所不同
      4. 当master实例binlog为：
         1. 基于statement，slaves实例会记录所有这个日志里的内容
         2. 基于row，则slaves不会记录这类日志
         3. 基于mixed，仅记录statement部分
      5. 默认情况下General Query log 是禁用的：
         1. 可以通过启动时参数--general\_log启用。
         2. 运行时也可以通过全局参数 general\_log启用
      6. 服务器重启和刷新日志并不会重建日志文件，只能手动处理
      7. 如果仅针对当前session关闭/开启此日志记录，可以通过sql\_log\_off参数设置

# 事务(Transaction)

https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-model.html

## 概述

* + - 1. ACID 模型是一组强调数据库可靠性的设计原则, innodb 严格遵守 此模型
      2. Innodb的事务模型，目标是结合多版本数据库的最佳特性与传统的两阶段锁
      3. 所有用户行为都在一个事务中，
      4. 默认情况下，autocommit是启用的：

每个SQL statement都各自形成自己的事务，默认情况下，针对每个新链接：

* + - * 1. 如果SQL statement执行没有抛出错误 ，在执行之后自动执行一次commit
        2. 如果SQL statement执行抛出错误，根据错误决定提交当前事务或回滚
        3. 也可以通过在逻辑开始处指定START\_TRANSACTION 或 BEGIN statement，并在逻辑结束时执行 COMMIT或ROLLBACK statement ，来实现同个事务中执行多个statement
      1. 如果autocommit禁用时：
         1. 当前session的事务将一直开放，直到执行COMMIT或ROLLBACK statement 结束当前事务
         2. 如果当前session结束前没有以commit statement 结束最后的事务，将会回滚最后一个事务
      2. 有些statement会在执行前，产生一个隐式的提交操作以结束当前事务：

* + - 1. COMMIT 和 ROLLBACK statement：
         1. COMMIT：表明将当前事务对数据的修改持久化并对其他session可见
         2. ROLLBACK：表明要取消当前事务所作的所有修改
         3. 两者都会释放所有当前事务所获得的Locks
      2. Consistent read

## ACID

https://en.wikipedia.org/wiki/ACID

https://dev.mysql.com/doc/refman/5.7/en/mysql-acid.html

### A(atomicity)

* + - 1. 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用
      2. 涉及功能点：自动提交、commit声明、rollback声明、操作INFORMATION\_SCHEMA 表的数据

### C(cosistency)

* + - 1. 在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
      2. 涉及功能点：doublewrite buffer、crash recovery

### I(isolation)

* + - 1. 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据是独立的
      2. 涉及功能点：自动提交、配置隔离等级、lock的low-level detail

### D(durability)

* + - 1. 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。
      2. 涉及功能：
         1. doublewrite buffer开关
         2. 配置项：innodb\_flush\_log\_at\_trx\_commit、sync\_binlog、innodb\_file\_per\_table
         3. write buffer
         4. Battery-backed cache
         5. 操作系统是否支持fsync()
         6. UPS
         7. 备份策略

## Transaction Isolation

https://en.wikipedia.org/wiki/Isolation\_(database\_systems)

https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html

### 概述

* + - 1. ISO和ANIS SQL 标准制定了四种事务隔离级别(SLQ:1992)从低到高：READ\_UNCOMMITTED、READ\_COMMITTED、REPEATABLE\_READ(2.9999。)、SERIALIZABLE(3。)
         1. 标准的REPEATABLE\_READ没有幻读保护，所以是2.9999。隔离
         2. SQL和SQL2标准的默认事务隔离级别是SERIALIZABLE
      2. 不是所有数据库都遵循这个标准
      3. Innodb通过不同的lock策略支持了所有4种事务隔离级别，默认隔离级别是REPEATABLE\_READ

### READ UNCOMMITED

https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html#isolevel\_read-uncommitted

* + - 1. SELECT statement 采用无锁方式执行，很可能获取记录的早期版本
      2. 无法保证读一致性，会产生 dirty read
      3. 其他方面，跟READ\_COMMITED一致

### READ COMMITTED

https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html#isolevel\_read-committed

* + - 1. 同个事务中，通过第一次读取数据时建立的快照，实现读一致性
      2. 对于 Locking read(SELECT \* FROM t FOR UPDATE 或者 LOCK IN SHARE MODE)、UPDATE以及DELETE statements，只锁索引记录，不锁gap，因此允许在被锁定的记录旁边插入新记录，可能会导致phantom problems
      3. 仅在需要检查外键约束和唯一键约束时，才使用Gap locks
      4. 此隔离级别只支持基于行的二进制日志，如果配置binlog\_format=MIXED，系统会自动的转为基于行的日志
      5. 对于UPDATE 或 DETELE statements，只会保持锁定条件匹配的行，不匹配的行会被释放，会大程度的减少deadlock(但仍可能发生deadlock)
      6. 对于 UPDATE statement：
         1. 如果指定的行R已经被锁定，Innodb执行一个”semi-consistent” read，即返回R最后一次提交的版本，用于判断R是否匹配WHERE条件
         2. 如果R匹配，则重新读取R最新记录，并尝试锁定R或者阻塞等待

### REPEATABLE READ

https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html#isolevel\_repeatable-read

* + - 1. Innodb 默认事务隔离等级，与SQL标准不同的是，Innodb在此隔离级别下，使用了next-key lock 避免了幻读的产生，达到了SQL标准的SERIALIZABLE隔离级别
      2. 同个事务中，通过第一次读取数据时建立的快照，实现读一致性
      3. 对于 Locking read(SELECT \* FROM t FOR UPDATE 或者 LOCK IN SHARE MODE)、UPDATE以及DELETE statements的lock策略 取决于：
         1. 对于带唯一索引字段的搜索条件，仅锁定发现索引记录，不锁记录前的gap
         2. 对于其他搜索条件，锁定扫描到的索引范围，通过gap lock或 next-key locks 阻塞其他事务在范围内的插入数据请求

### SERIALIZABLE

https://dev.mysql.com/doc/refman/5.7/en/innodb-transaction-isolation-levels.html#isolevel\_serializable

* + - 1. 在innodb中，类似于REPEATABLE\_READ
      2. 如果禁用了autocommit，Innodb会自动在每个SELECT statements后面加上LOCK IN SHARE MODE
      3. 如果启用了autocommit，则SELECT会有自己的事务
      4. 不支持一致性的非锁定读