Mysql

InnoDB体系架构

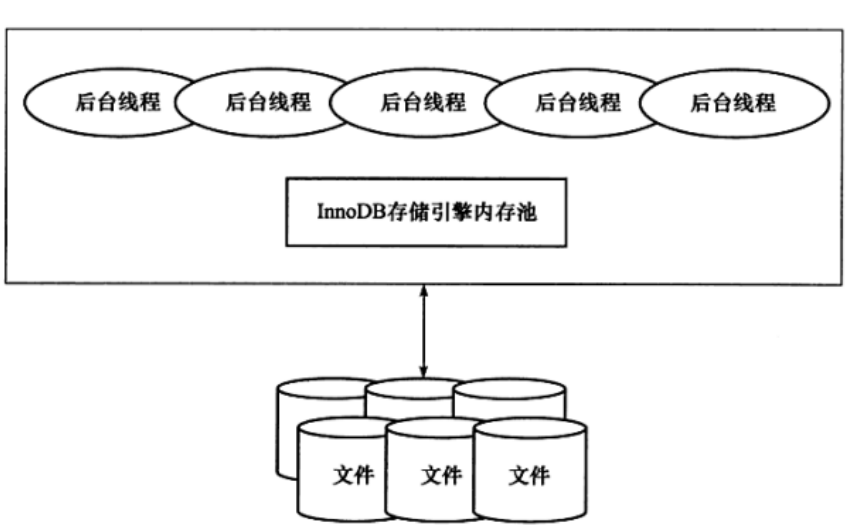


图 2-1　InnoDB 存储引擎体系架构

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，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。

- Main thread（核心的后台线程）：

  即使事务还没有提交，innoDB仍然会每秒将重做日志缓存中的内容刷新到重做日志文件，这也解释了为什么再大的事务commit的时间也是很短的。

InnoDB关键特性

- 插入缓冲 insert buffer

  解决非聚集索引的离散读取性能问题（b+树的问题）

  使用需要满足两个条件：1. 索引是辅助索引（secondary index）2. 索引不是唯一（unique）的（否则插入缓冲时数据库会查找索引页来判断插入的记录的唯一性，这样肯定又会发生离散读取的情况，导致insert buffer失去意义）

- 双写 double write

  解决正在写入页到表中发生宕机的问题。在应用重做日志前，需要一个页的副本，当写入失效发生时，先通过页的副本来还原该页，再进行重做。

  在MySQL写数据page时，会写两遍到磁盘上，第一遍是写到doublewrite buffer，第二遍是从doublewrite buffer写到真正的数据文件中。如果发生了极端情况（断电），InnoDB再次启动后，发现了一个page数据已经损坏，那么此时就可以从doublewrite buffer中进行数据恢复了。

- 自适应哈希

  InnoDB会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升，则建立哈希索引，称为自适应哈希索引（Adaptive Hash Index）注意只能用于等值查询。

InnoDB逻辑储存结构

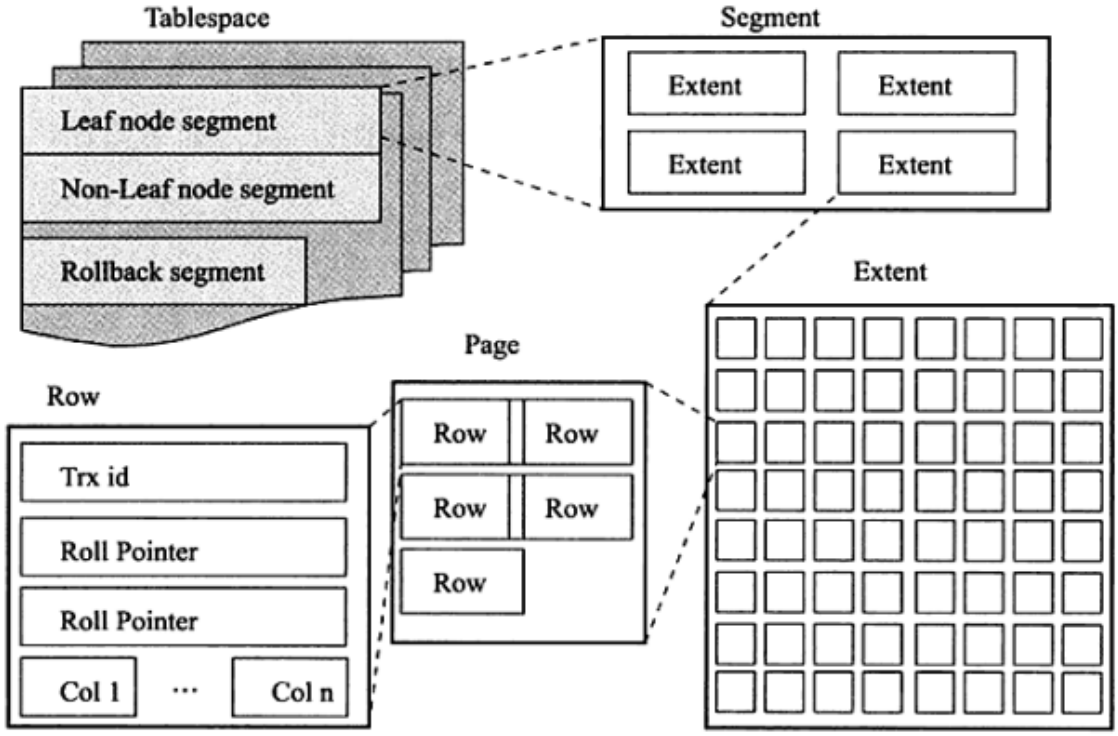


图 4-1　InnoDB 逻辑存储结构

1. 表空间

   InnoDB存储引擎逻辑结构最高层

2. 段

   数据段，索引段，回滚段.....

   数据段：B+树叶子节点

   索引段：B+树非叶子节点

3. 区

   由连续页组成的空间

4. 页

   InnoDB磁盘管理的最小单位

索引

1. B+树索引

   了解b+树的插入和删除（fan out和fill factor）

   - 聚集索引（clustered index）：

     按照每张表的主键构造一棵B+树，同时叶子节点中存放的即为整张表的行记录数据。能够快速地访问针对范围值的查询。每张表只有一个。

   - 辅助索引（secondary index）：

     叶子节点并不包含行记录的全部数据。叶节点除了包含键值外，每个叶节点中的索引行还包含了一个书签，用来告诉InnoDB哪里可以找到与索引相对应的行数据。每张表可以有多个。

2. 哈希算法

   对于字典类型的查找非常快速，对于范围查找无能为力

3. 全文检索
    - 倒排索引（inverted index）

      在辅助表中存储了单词与单词自身在一个或多个文档中所在位置的映射

锁

1. Lock与Latch

表 6-1　lock 与 latch 的比较

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过 waits-for graph、time out 等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（lock leveling）保证无死锁的情况发生
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

2. 锁的类型
    - InnoDB实现了两种标准的行级锁：
        - 共享锁（S Lock），允许事务读一行数据。

          排他锁（X Lock），允许事务删除或更新一行数据。

          X锁与任何的锁都不兼容（需要等待释放），S锁仅与S锁兼容
        - 额外的锁方式：意向锁（Intention Lock）

          如果将上锁的对象看成一棵树，那么对最细粒度的对象进行上锁，首先要对粗粒度的对象上意向锁。如果其中任何一个部分导致等待，那么该操作需要等待粗粒度锁的完成

          意向共享锁（IS Lock），事务想要获得一张表中某几行的共享锁

          意向排他锁（IX Lock），事务想要获得一张表中某几行的排他锁

          意向锁不会堵塞除全表扫描以外的任何请求

表 6-4　InnoDB 存储引擎中锁的兼容性

	IS	IX	S	X
IS	兼容	兼容	兼容	不兼容
IX	兼容	兼容	不兼容	不兼容
S	兼容	不兼容	兼容	不兼容
X	不兼容	不兼容	不兼容	不兼容

    - 一致性非锁定读（默认）

      指InnoDB通过多版本控制的方式来读取当前执行时间数据库中行的数据。如果读取的行正在执行delete或update操作，这时读取操作不会因此去等待行上锁的释放，而是去读取行的一个快照数据。

      一个行可能有不止一个快照数据，一般称为行多版本技术。由此带来的并发控制，称为多版本并发控制（Multi Version Concurrency Control，MVCC）

      READ_COMMITTED时非一致性读会读取被锁定行的最新一份快照数据

      REPEATABLE READ会读取事务开始时的行数据版本
    - 一致性锁定读

      SELECT...FOR UPDATE 对读取的行记录加一个X锁，其他事务不能对已锁定的行加上任何锁。

      SELECT...LOCK IN SHARE MODE 对读取的行记录加一个S锁，其他事务可以向被锁定的行加S锁，但X锁会被堵塞

- 自增长与锁

  自增长列的插入实现方式称为AUTO-INC Locking，为了提高插入的性能，锁不是在事务完成后才释放，而是在完成对自增长值插入的SQL语句后立即释放。

- 外键和锁

  对于外键值的插入或更新，首先需要查询父表中的记录（select 父表），这时不时使用一致性非锁定读的方式（会发生数据不一致的问题），而是使用SELECT...LOCK IN SHARE MODE。

3. 锁的算法

   行锁的三种算法：

   Record Lock：记录锁。单个行记录上的锁。

   Gap Lock：间隙锁。锁定一个范围，但不包含记录本身。

   Next-Key Lock：临键锁 Gap Lock + Record Lock，锁定一个范围，并且锁定记录本身。

   - **索引上的等值查询，给唯一索引加锁的时候，next-key lock 退化为行锁。**
   - **索引上的等值查询，向右遍历时且最后一个值不满足等值条件的时候，next-key lock 退化为间隙锁**

   例子：

```
CREATE TABLE `t` (
  `id` int(11) NOT NULL,
  `c` int(11) DEFAULT NULL,
  `d` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `c` (`c`)
) ENGINE=InnoDB;

insert into t values(0,0,0),(5,5,5),
(10,10,10),(15,15,15),(20,20,20),(25,25,25);
```

   执行操作

```
select id from t where c = 5 lock in share mode;
```

   这个sql的查询过程大致 如下，通过c索引，找到5这个节点，那么会加行锁和间隙锁，范围是(0,5]，但是c索引不是唯一索引，索引innodb会继续往下遍历，找到c索引的下一个值，就是10，满足了规则2，即访问到的节点都需要加锁，所以会加(5,10]，但是这个是最后一个节点，且10不等于等值查询条件（c = 5）,所以这个next-key lock就退化成间隙锁(5,10)，所以这个语句一共会加以下两个间隙锁和一个行锁

4. 死锁

   例子：两个事务都使用了SELECT...LOCK IN SHARE MODE，于是两个事务都无法对同一记录执行insert操作，互相等待对方释放资源。

   解决方案：

   - 超时。当两个事务相互等待时，当一个等待超过阈值时，其中一个事务进行回滚，另一个事务就能继续进行。
   - 等待图（wait-for graph）：保存锁的信息链表，事务等待链表。这样可以构造一个图，若图中存在回路说明有死锁。这是使用dfs实现的。

事务

1. ACID：原子性，一致性，隔离性，持久性

2. 分类：
    - **扁平事务（Flat Transactions）**

      其中的操作要么全部执行，要么全部回滚。
    - **带有保存点的扁平事务（Flat Transactions with Savepoint）**

      允许在事务执行过程中回滚到同一事务较早的保存点。
    - **链事务（Chained Transaction）**

      保存点模式的变种。在提交一个事务时，释放不需要的数据对象，将必要的处理上下文隐式地传给下一个要开始的事务（提交事务操作和开始下一个事务操作将合并为一个原子操作）

      与保存点扁平事务不同的是，保存点扁平事务能回滚到任意正确的保存点，而链事务中的回滚仅限于当前事务。
    - **嵌套事务（Nested Transaction）**

      由一个顶层事务控制各个层次的事务（子事务）。子事务既可以提交也可以回滚，但提交操作不会马上生效，必须在顶层事务提交后才真正提交。树中任意一个事务回滚会引起它的所有子事务一同回滚。
    - **分布式事务（Distributed Transactions）**

      在一个分布式环境下运行的扁平事务，需要根据数据所在位置访问网络中的不同节点。
3. 事务的实现：

   Redo Log（恢复提交事务修改的页操作） 和 Undo Log（回滚行记录到某个特定版本）

   Redo： 实现事务的持久性，由两部分组成。
    - 内存中的重做日志缓冲
    - 重做日志文件
    - redo log的写入拆成了两个步骤：prepare和commit，这就是"**两阶段提交** "

   Undo： 存放在数据库内部的一个特殊段中，称为undo段，位于共享表空间内。

   Purge: delete 和 update操作可能并不直接删除原有的数据，真正删除记录的操作可能被延时，最终在purge中完成。
4. Redo Log（记录物理修改），Undo Log（记录逻辑修改）顺序：

   在InnoDB内存中，一般的顺序如下：
    - 写undo的redo
    - 写undo
    - 修改数据页
    - 写Redo

MVCC实现

1. ## 事务版本号：

   每次事务开启前都会从数据库获得一个自增长的事务ID，可以从事务ID判断事务的执行先后顺序。
2. ## 表格的隐藏列：

   DB_TRX_ID: 记录操作该数据事务的事务ID；

   DB_ROLL_PTR：指向上一个版本数据在undo log 里的位置指针；

   DB_ROW_ID: 隐藏ID ， 当创建表没有合适的索引作为聚集索引时，会用该隐藏ID创建聚集索引;

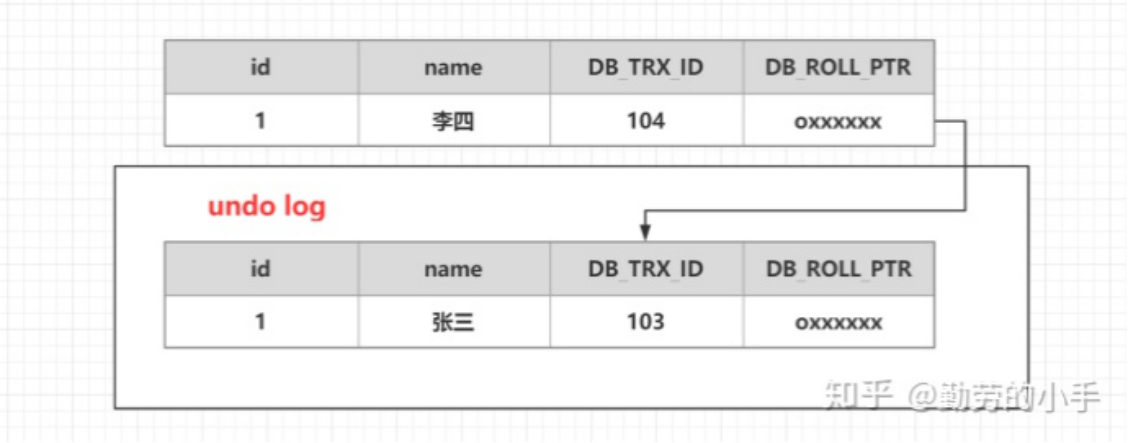
3. Undo log：

Undo log 主要用于记录数据被修改之前的日志，在表信息修改之前先会把数据拷贝到undo log 里，当事务进行回滚时可以通过undo log 里的日志进行数据还原。

Undo log 的用途

（1）保证事务进行rollback时的原子性和一致性，当事务进行回滚的时候可以用undo log的数据进行恢复。

（2）用于MVCC快照读的数据，在MVCC多版本控制中，通过读取undo log的历史版本数据可以实现不同事务版本号都拥有自己独立的快照数据版本。



4. Read view：

在innodb 中每个SQL语句执行前都会得到一个read_view。副本主要保存了当前数据库系统中正处于活跃（没有commit）的事务的ID号，其实简单的说这个副本中保存的是系统中当前不应该被本事务看到的其他事务id列表。

5. Read view 的几个重要属性

trx_ids: 当前系统活跃(未提交)事务版本号集合。

low_limit_id: 创建当前read view 时“当前系统最大**事务版本号**+1”。

up_limit_id: 创建当前read view 时“系统正处于**活跃事务**最小版本号”

creator_trx_id: 创建当前read view的事务版本号；

Read view 匹配条件

（1）数据事务ID <up_limit_id 则显示

如果数据事务ID小于read view中的最小活跃事务ID，则可以肯定该数据是在当前事务启之前就已经存在了的,所以可以显示。

（2）数据事务ID>=low_limit_id 则不显示

如果数据事务ID大于read view 中的当前系统的最大事务ID，则说明该数据是在当前read view 创建之后才产生的，所以数据不予显示。

（3） up_limit_id <数据事务ID<**low_limit_id 则与活跃事务集合**trx_ids**里匹配**

如果数据的事务ID大于最小的活跃事务ID,同时又小于等于系统最大的事务ID，这种情况就说明这个数据有可能是在当前事务开始的时候还没有提交的。

所以这时候我们需要把数据的事务ID与当前read view 中的活跃事务集合trx_ids 匹配:

情况1: 如果事务ID不存在于trx_ids 集合（则说明read view产生的时候事务已经commit了），这种情况数据则可以显示。

情况2： 如果事务ID存在trx_ids则说明read view产生的时候数据还没有提交，但是如果数据的事务ID等于creator_trx_id ，那么说明这个数据就是当前事务自己生成的，自己生成的数据自己当然能看见，所以这种情况下此数据也是可以显示的。

情况3： 如果事务ID既存在trx_ids而且又不等于creator_trx_id那就说明read view产生的时候数据还没有提交，又不是自己生成的，所以这种情况下此数据不能显示。

（4）不满足read view条件时候，从undo log里面获取数据

当数据的事务ID不满足read view条件时候，从undo log里面获取数据的历史版本，然后数据历史版本事务号回头再来和read view 条件匹配 ，直到找到一条满足条件的历史数据，或者找不到则返回空结果；

Mysql实战45讲：

https://github.com/meiqinggao/interview-docs/blob/master/docs/2019/MySQL%E5%AE%9E%E6%88%9845%E8%AE%B2%E7%AC%94%E8%AE%B0.md