MySQL训练营-深度剖析Innodb存储引擎

本内容属于动脑学院Steven老师首创,如需转载,请注明出处!

Steven QQ : 750954883

动脑学院系列课程咨询: QQ 2729772006

特性总览

Feature	MyISAM	Memory	InnoDB	Archive
B-tree indexes	Yes	Yes	Yes	No
Clustered indexes	No	No	Yes	No
Compressed data	Yes (note 2)	No	Yes	Yes
Data caches	No	N/A	Yes	No
Foreign key support	No	No	Yes	No
Geospatial data type support	Yes	No	Yes	Yes
Geospatial indexing support	Yes	No	Yes (note 6)	No
Hash indexes	No	Yes	No (note 7)	No
Index caches	Yes	N/A	Yes	No
Locking granularity	Table	Table	Row	Row
MVCC	No	No	Yes	No
Replication support (note 1)	Yes	Limited (note 8)	Yes	Yes
Storage limits	256TB	RAM	64TB	None
Transactions	No	No	Yes	No

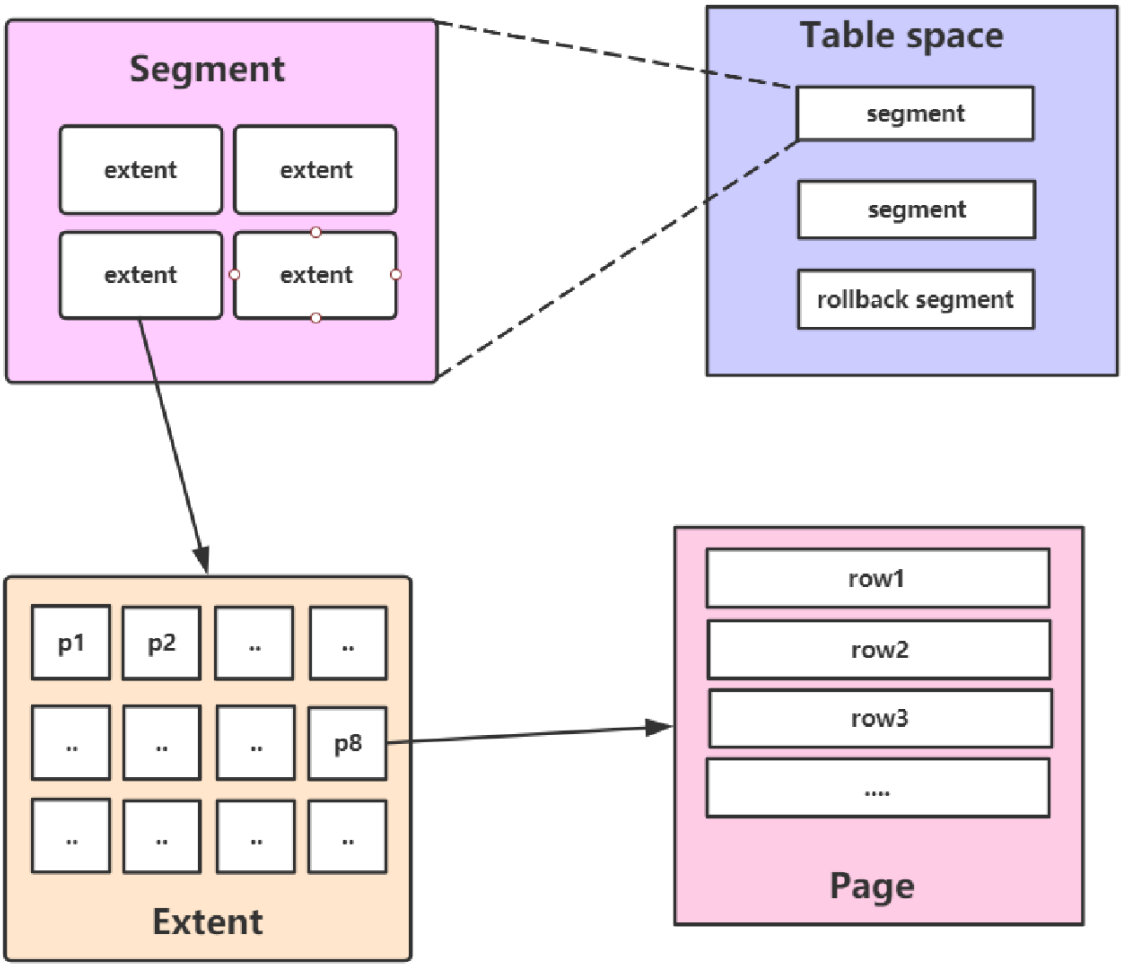
特性总结

- Clustered Indexed 聚集索引,在索引的课中,我们已经分析过聚集索引到底是什么.
- Foreign Key Support 外键支持,Innodb的外键是保证数据完整性的一种强制约束,但是在正式的生产系统设计过程中,我们并不建立建立强制的主外键关系.而是在程序逻辑上,标志逻辑上的主外键关系.

接下来这章节的内容,我们会针对 `Data Caches`,`Index Caches`,`Row Locking granularity`,`MVCC`,`Transaction`等特性做详细的分析和了解.

MySQL存储的逻辑结构

在正式深入Innodb存储引擎特性之前,我们先了解一波MySQL的逻辑存储结构



在MySQL官方文档中.https://dev.mysql.com/doc/refman/5.7/en/innodb-file-space.html

- Table Space 表空间
  - System TableSpace 系统表空间,也叫共享表空间.映射文件 ibdata1.所有的表共享一个表空间，这个

    文件会越来越大，而且它的空间不会收缩
  - - Innodb Data Dictionary

      Innodb引擎的系统表,比如记录执行计划的表,索引信息的表等等.
    - Double Write Buffer

      Doublewrite Buffer是开在共享(系统)表空间的物理文件的buffer,其大小是2MB,

      刷脏时,脏页数据备份的位置.

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

      是Innodb buffer pool中 ChangeBuffer开在共享(系统)表空间的物理映射位置
    - Undo Logs

      Undo Logs 开在共享(系统)表空间的物理映射位置.
  - file_per_table tablespace 独占表空间

    包含单个 InnoDB 表的数据和索引，并存储在文件系统中自己的数据文件中（tbl_name.ibd）

    参数 `innodb_file_per_table=on` 默认值也是on.
  - general tablespace 自定义的表空间

```
create tablespace stevents add datafile 'stevents.ibd'
file_block_size=16K engine=innodb;  -- 创建自定义的表空间

create table test(id integer) tablespace stevents;  --新建表指定表空间

drop tablespace stevents;
```

- temproary tablespace 临时表空间

  存储临时表的数据，对应数据目录下的 ibtmp1 文件,服务停止,文件失效.

  存储用户创建的临时表.

```
CREATE TEMPORARY TABLE temporaryTable (
   id integer
);
```

  程序计算过程中产生的临时表信息.

- Page 页

  一 个Extent中有 64 个连续Page.为1M大小.

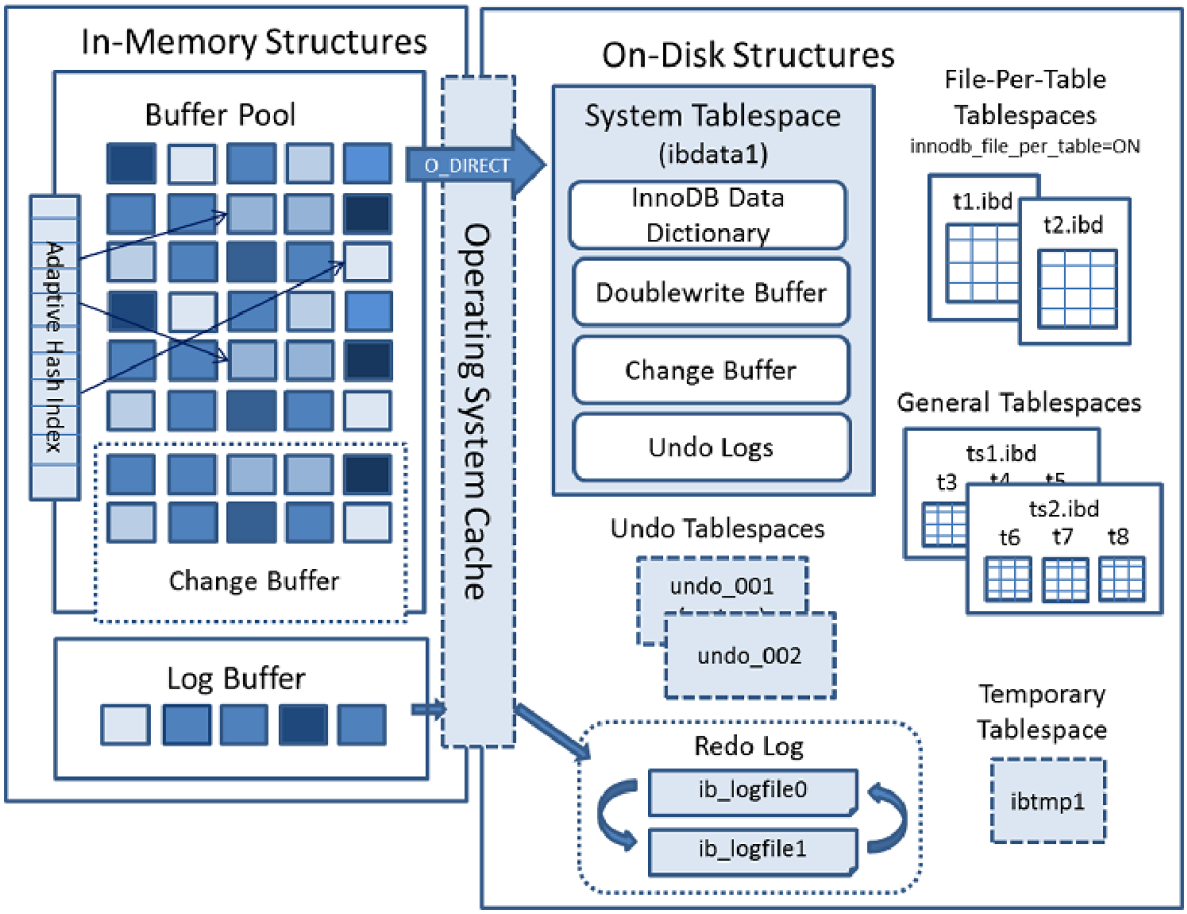
  一个Extent中的页一定是从物理和逻辑上都是连续的.

  在MySQL中Page的默认大小是16K.

```
SHOW VARIABLES LIKE 'innodb_page_size';
```

  另外,在索引内容中所讲的磁盘块就是MySQL的page概念.

Innodb的架构

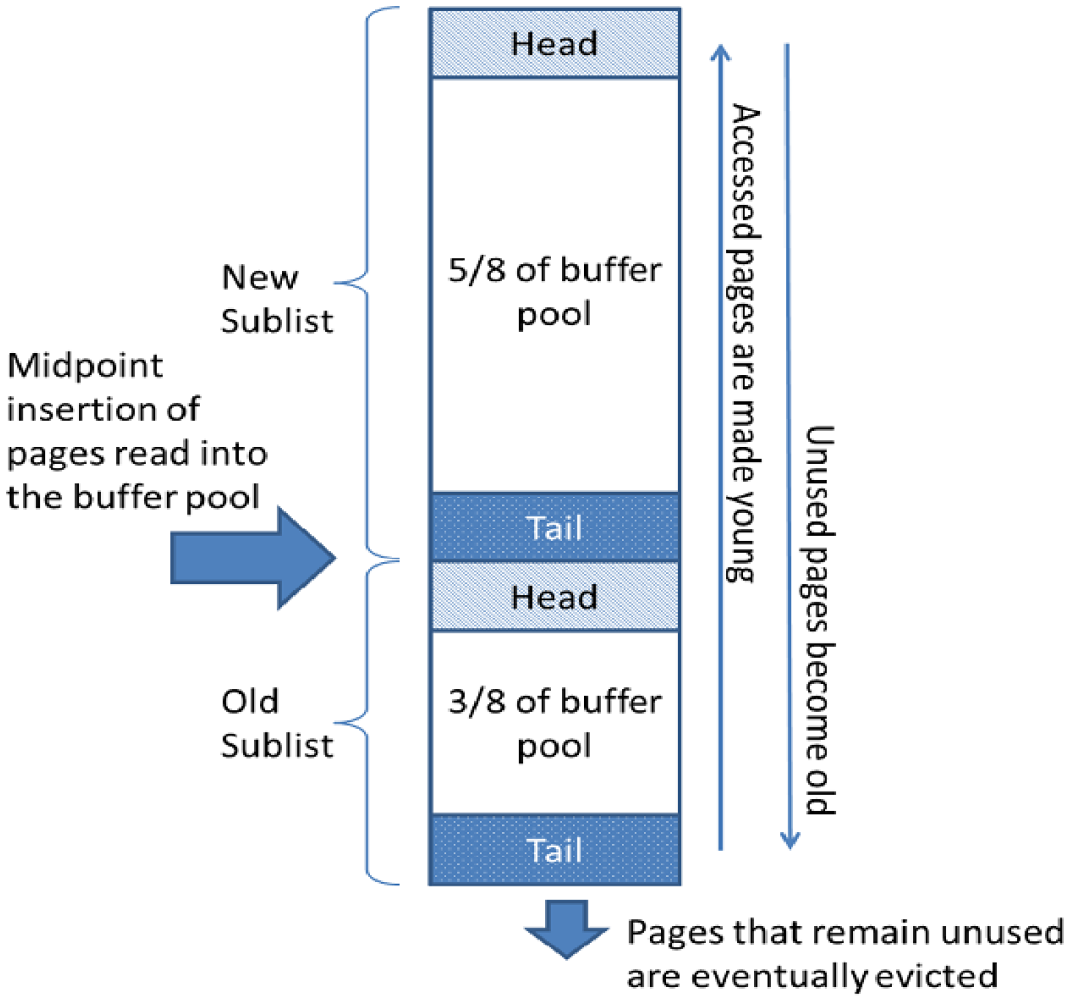


内存架构

- Buffer Pool

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

  InnoDB在访问表和索引数据时将其缓存在Buffer Pool中.



Lua的算法进行数据页面的淘汰.

Data Caches，Index Caches 就是通过Buffer Pool实现的.亦可这么认为,数据的缓存和索引的缓存都是缓存在Buffer Pool中.

- Change Buffer

  MySQL5.5版本中叫Insert Buffer.

  在对**非唯一普通索引页**进行数据的操作.此时该数据不在Buffer Pool中,此时MySQL为了降低磁盘的IO.

  并不会立刻将磁盘页加载到缓冲池，而仅仅记录缓冲变更至Change Buffer中,等未来这一段数据被load时，再将load的数据合并(merge)Change Buffer的变更 恢复到缓冲池中的技术

  - 为什么是非唯一性索引

    因为唯一性索引必须加载数据在内存中进行唯一性验证

  - 设计的初衷

    降低非唯一性索引的Update操作的磁盘IO，提升数据库性能

- Log Buffer

  为Redo Log的日志缓冲区.

  默认大小为16MB.`innodb_Log_buffer_size`参数可以进行相关调节

  此外,该缓冲区的提交策略设定

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

- Adaptive Hash Index

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

索引的课程中有讲过.

磁盘架构

- Doublewrite Buffer

  Doublewrite Buffer是开在共享(系统)表空间的物理文件的buffer,其大小是2MB,

  刷脏时,脏页数据备份的位置.

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

  Undo Log 开在共享表空间的物理文件的buffer.
- Redo Log

  Redo Log落地的磁盘文件对应是2个文件ib_logfile0 和 ib_logfile1

Innodb Transaction

在Innodb的主要的特性中,有一项很关键的特性是对事务的支持.

事务的支持

事务指的是数据库操作的最小工作单元，是作为单个逻辑工作单元执行的一系列操作,事务是一组不可再分割的操作集合（工作逻辑单元）.

经典的事务场景:

转账业务

```
update user_account set balance = balance - 1000 where userID = 3;

update user_account set balance = balance +1000 where userID = 1;
```

但是在MySQL的默认SQL语句执行过程,针对一个SQL语句就是一个单独的事务.是因为在MySQL的参数设定中存在一个`autocommit`参数, 它的默认值为on.这就导致了每一条独立的SQL执行都是一个单独的事务运行.

MySQL如何开启事务

begin / start transaction -- 手工 开启事务. commit / rollback -- 事务提交或回滚 set session autocommit = on/off; -- 设定事务是否自动开启

事务的四大特性及实现

- 原子性（Atomicity） 事务是数据库操作的最小工作单元，整个工作单元要么一起提交成功，要么全部失败回滚

- 一致性（Consistency） 事务中操作的数据及状态改变是一致的，即写入资料的结果必须完全符合预设的规则,不会因为出现系统意外等原因导致状态的不一致
- 隔离性（Isolation） 一个事务所操作的数据在提交之前，对其他事务的可见性设定（一般设定为不可见）
- 持久性（Durability） 事务所做的修改就会永久保存，不会因为系统意外导致数据的丢失

原子性保证

Innodb事务的原子性保证,包含事务的提交机制和事务的回滚机制.

提交机制我们不做讨论.

在innodb中事务的回滚机制是依托Undo Log进行回滚数据的保证的.

在Innodb 磁盘架构中.Undo Log 是开在共享(系统)表空间的物理文件的Buffer.

Undo Log记录的是逻辑日志.记录的是事务过程中每条数据的变化版本和情况.

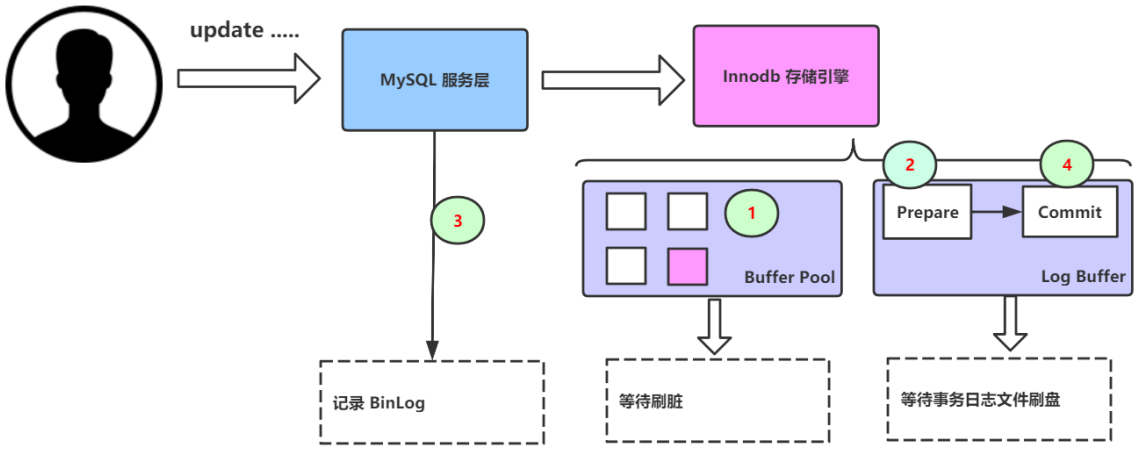
在事务异常中断,或者主动(Rollback)回滚的过程中,Innodb基于Undo Log 进行数据撤销回滚,保证数据回归至事务开始状态.

由于Undo Log记录的是数据变更的各个版本.在Innodb引擎,用Undo Log实现数据的多版本控制.MVCC

Undo Log实现MVCC的基本原理 -> 事务未提交之前，Undo保存了未提交之前的版本数据，Undo 中的数据可作为数据旧版本快照供其他并发事务进行快照读

持久性保证

事务所做的修改就会永久保存，不会因为系统意外导致数据的丢失.



从上诉的update流程中,我们已经知道.在事务数据变更的阶段,已经通过XA的两阶段提交将变更的数据记录在Log Buffer中.

基于上面的流程有可能出现三种场景.

- 数据 刷脏 正常.一切正常提交

  Redo Log 循环记录.数据成功落盘.持久性得以保证

- 数据 刷脏 的过程中出现的系统意外导致 页断裂 现象

  针对 页断裂 情况,采用Double write 机制进行保证 页断裂 数据的恢复.

Double write机制详解

Doublewrite Buffer是开在共享(系统)表空间的物理文件的buffer,其大小是2MB.

- 刷脏操作开始之时,先进行脏页'备份'操作.将脏页数据写入Doublewrite Buffer.
- 将Doublewrite Buffer(顺序IO)写入磁盘文件中(共享表空间)
- 进行刷脏操作.(绝大多数是随机IO)
- Double Write机制其核心思想是: 在刷脏之前,建立脏页数据的副本.系统意外宕机造成页断裂的情况可通过脏页数据副本(DoubleWrite Buffer)进行恢复.

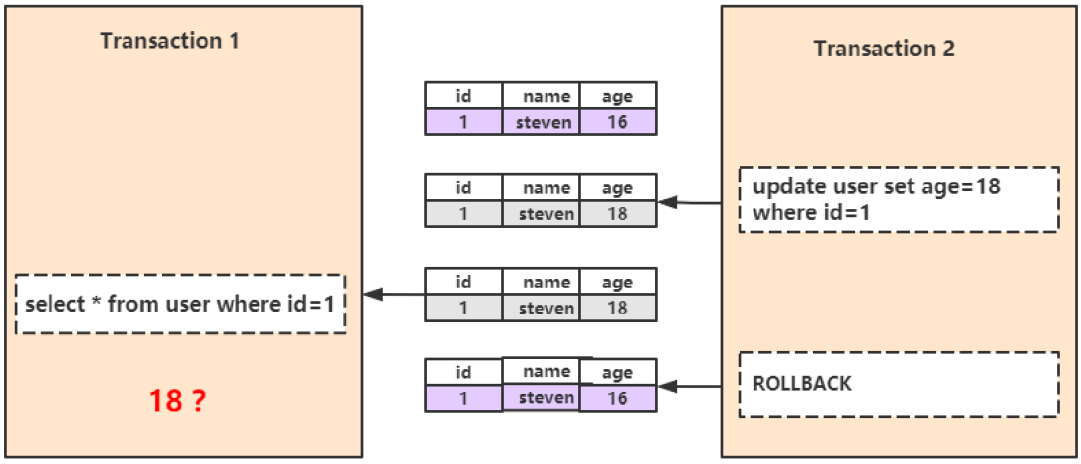
- 数据未出现 `页断裂` 现象,也没有刷脏成功

  MySQL通过Redo Log 进行数据的持久化即可.

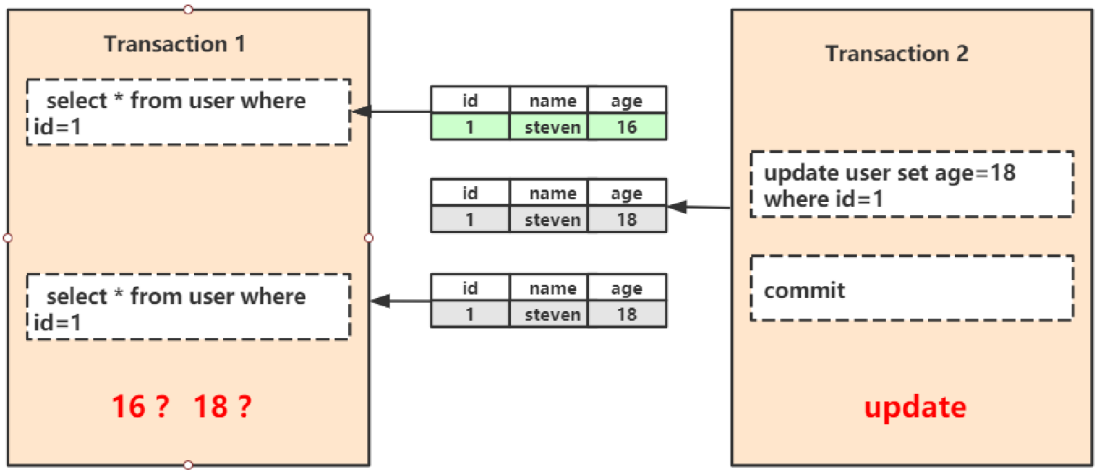
隔离性保证

隔离性问题分析

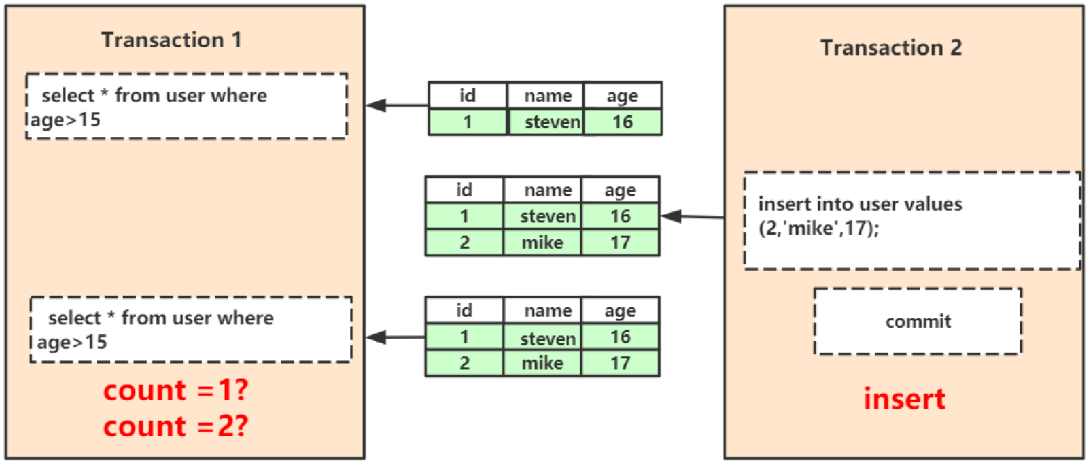
- 脏读



- 不可重复度



- 幻读



SQL92标准隔离级别定义

- Read Uncommitted（未提交读） --未解决并发问题 事务未提交对其他事务也是可见的，脏读（dirty read）
- Read Committed（提交读） --解决脏读问题 一个事务开始之后，只能看到自己提交的事务所做的修改，可以重复读（nonrepeatable read）
- Repeatable Read (可重复读) --解决不可重复读问题 在同一个事务中多次读取同样的数据结果是一样的，这种隔离级别未定义解决幻读的问题
- Serializable（串行化） --解决所有问题 最高的隔离级别，通过强制事务的串行执行

Innodb隔离级别的实现

事务隔离级别	脏读	不可重复读	幻读
未提交读（Read Uncommitted）	可能	可能	可能
已提交读（Read Committed）	不可能	可能	可能
可重复读（Repeatable Read）	不可能	不可能	对 InnoDB 不可能
串行化（Serializable）	不可能	不可能	不可能

隔离级别的实现主要两大技术

- LBCC

  Lock Based Concurrency Control(LBCC),事务开始操作数据前，对其加锁，阻止其他事务对数据进行修改

  针对SQL的操作,使用`当前读`解决并发读写的隔离性问题

- MVCC

  Multi Version Concurrency Control（MVCC）,事务开始操作数据前,将数据在当下时间点进行一份数据快照（Snapshot)的备份，并用这个快照来提供给其他事务进行一致性读取 并发访问(读或写)数据库时，对正在事务内处理的数据做多版本的管理。避免写操作的堵塞，从而引发读操作的并发阻塞问题,使用`快照读`解决并发读写的隔离性问题

Innodb锁机制详解

Innodb的行锁

对比表级别锁行级别的锁具有以下优势:

锁定粒度：表锁 > 行锁 加锁效率：表锁 > 行锁 冲突概率：表锁 > 行锁 并发性能：表锁 < 行锁

Innodb的锁分类

- 共享锁（行锁）： Shared Locks

  读锁(S锁),多个事务对于同一数据可以共享访问,不能操作修改

  使用方法:

  SELECT * FROM table WHERE id=1 LOCK IN SHARE MODE -- 加锁 COMMIT/ROLLBACK -- 释锁

- 排他锁（行锁）： Exclusive Locks

  写锁(X锁)，互斥锁/独占锁,事务获取了一个数据的X锁，其他事务就不能再获取该行的锁（S锁、X锁），只有该获取了排他锁的事务是可以对数据行进行读取和修改.

  使用方法:

  DELETE/ UPDATE/ INSERT -- 加锁 SELECT * FROM table WHERE ... FOR UPDATE -- 加锁 COMMIT/ROLLBACK -- 释锁

- 意向共享锁（表锁）： Intention Shared Locks & 意向排它锁（表锁）： Intention Exclusive Locks

  意向共享锁(IS) 一个数据行加共享锁前必须先取得该表的IS锁，意向共享锁之间是可以相互兼容的 意向排它锁(IX) 一个数据行加排他锁前必须先取得该表的IX锁，意向排它锁之间是可以相互兼容的 意向锁(IS、IX)是InnoDB引擎操作数据之前自动加的，不需要用户干预 意义： 当事务操作需要锁表时，只需判断意向锁是否存在，存在时则可快速返回该表不能启用表锁

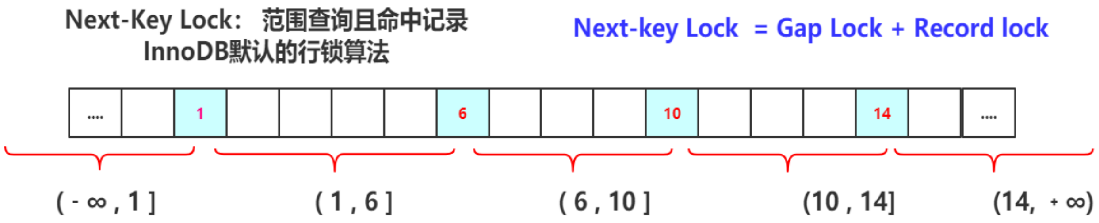
Innodb的行锁实现

- InnoDB的行锁是通过给索引上的索引项加锁来实现的
- Innodb按照辅助索引进行数据操作时,辅助索引和主键索引都将锁定指定的索引项
- 通过索引进行数据检索,InnoDB才使用行级锁,否则InnoDB将使用表锁

Innodb的行锁实现具体的算法

- 临键锁 Next-key Locks
  - 当SQL执行按照索引进行数据的检索时
  - 且查询条件为范围查找（between and、<、>等）[执行计划 type = range]
  - 有数据命中时,该SQL语句事务操作加上的行锁为Next-key locks

    具体实现 : 锁住命中记录区间+下一个区间（区间:左开右闭）



select * from t where id>5 and id<9 for update; ⇨ 锁住: (1,6] (6,10]

select * from t where id>5 and id<11 for update; ⇨ 锁住: (1,6] (6,10] (10,14]

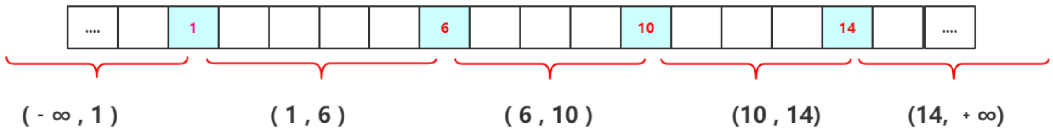
- 间隙锁 Gap Locks
  - 当SQL执行按照索引进行数据的检索时
  - 查询条件的数据不存在时

  具体实现 : 锁住数据不存在的区间（区间:左开右开）

Gap Lock：范围查询或等值查询且记录不存在

Gap锁之间不冲突

当记录不存在，临键锁退化成Gap锁



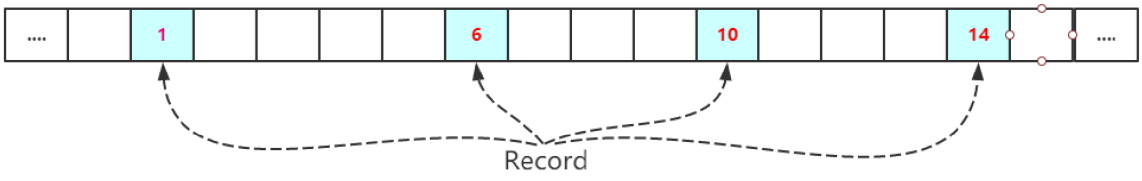
select * from t where id >2 and id <4 for update;
select * from t where id =5 for update; ⇨ 锁住：(1 , 6)

select * from t where id >20 for update; ⇨ 锁住：(14, + ∞)

- 记录锁 Record Locks
  - 当SQL执行按照唯一性（Primary key、Unique key）索引进行数据的检索
  - 且查询条件等值匹配且查询的数据命中存在

  具体实现 : 锁住具体索引的索引项

Record Lock：唯一性（主键/唯一）索引，条件为精准匹配,且命中数据，退化成Record锁

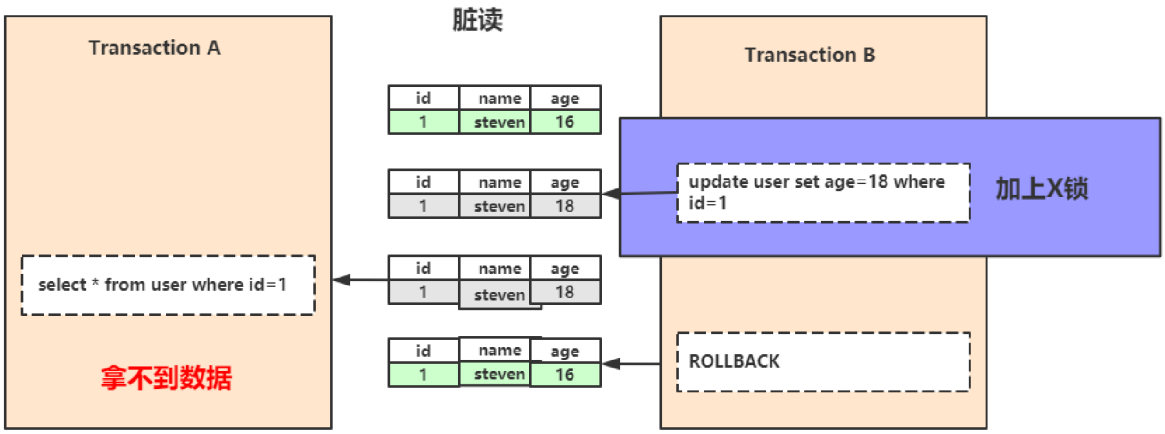


select * from t where id =6 for update; ⇨ 锁住：id=6

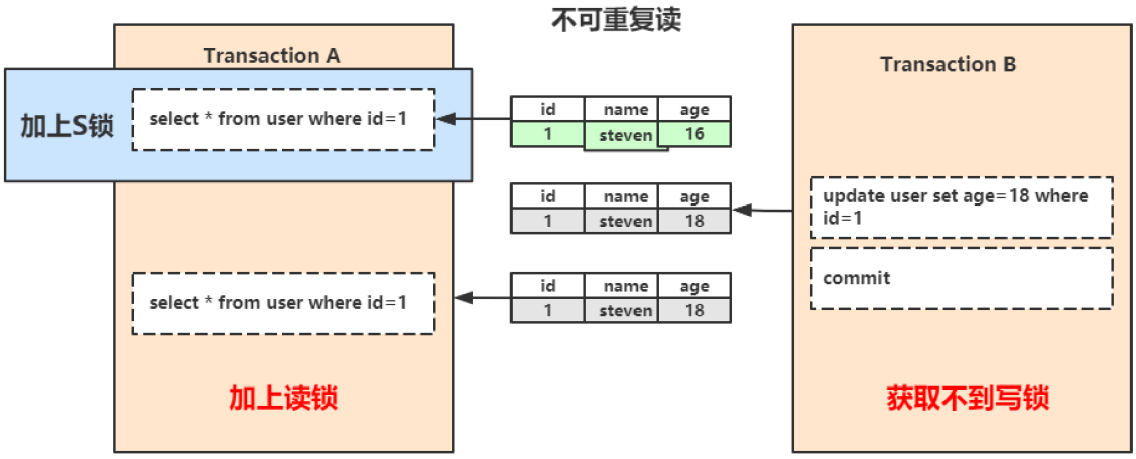
select * from t where id =10 for update; ⇨ 锁住：id=10

锁解决隔离性带来的问题

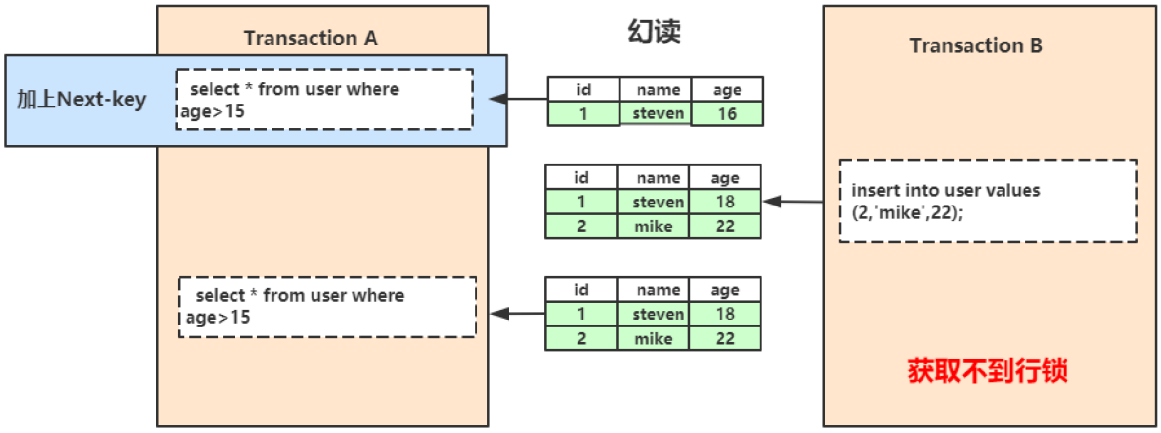
脏读解决



不可重复度解决



幻读解决



死锁问题

产生原因:

- 多个并发事务（2个或者以上）
- 每个事务都持有锁（或者是已经在等待锁）
- 每个事务都需要再继续持有锁
- 事务之间产生加锁的循环等待，形成死锁

避免死锁和排查:

- 发生死锁使用 show engine innodb status 或者 更改配置项 innodb_print_all_deadlocks进行死锁日志排查

- 类似的业务逻辑以固定的顺序访问表和行
- 大事务拆小。大事务更倾向于死锁，如果业务允许，将大事务拆小
- 在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁概率
- 降低隔离级别，如果业务允许，将隔离级别调低也是较好的选择
- 为表添加合理的索引,不走索引的SQL将会启用表锁.

MVCC 多版本控制

在事务的数据操作过程中,为了让事务保证原子性操作.我们在事务的操作之后,会将数据的历史修改版本存储在Undo Log.

所以在RDBMS系统的并发读场景下.如果对于数据的最新要求不高的请求.我们可将历史的数据版本返回给请求端.

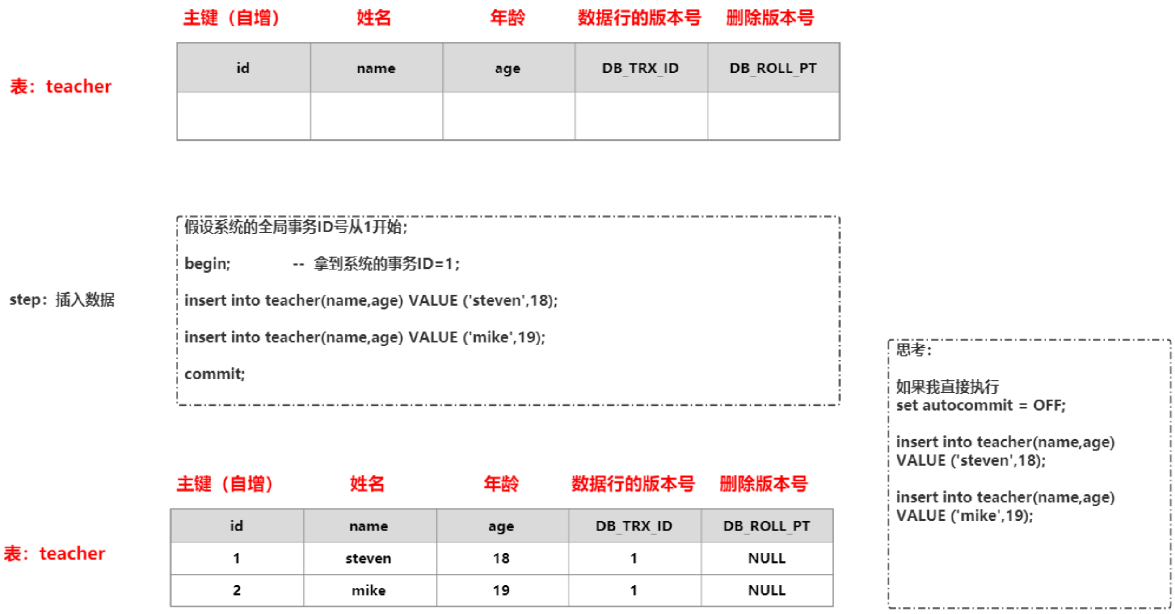
那具体针对这种多版本的控制是怎么做到的呢

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

在官方的介绍中,我们知道每一行数据都存在有隐含的列.

```
A 6-byte DB_TRX_ID field    -- 当前的事务ID

a 7-byte DB_ROLL_PTR field called the roll pointer    --事务回滚点(删除版本号)
```



隐藏字段说明

数据插入说明

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	steven	18	1	NULL
2	mike	19	1	NULL

step：数据的删除

```
假设系统的全局事务ID号目前到了22
begin;          -- 拿到系统的事务ID=22;
delete teacher where id =2 ;
commit;
```

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	steven	18	1	NULL
2	mike	19	1	*22*

数据删除过程

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	*name*	*age*	*DB_TRX_ID*	*DB_ROLL_PT*
1	*steven*	*18*	*1*	*NULL*
2	*mike*	*19*	*1*	*22*

step：修改操作

```
假设系统的全局事务ID号目前到了33
begin;          -- 拿到系统的事务ID=33;
update teacher set age = 19 where id =1;
commit;
```

修改操作是先做命中的数据行的copy，将原行数据的删除版本号的值设置为当前事务ID(33)

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	steven	18	1	*33*
2	mike	19	1	22
1	steven	19	33	NULL

copy

数据修改过程

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	steven	18	1	*33*
2	mike	19	1	22
1	steven	19	33	NULL

step：查询操作

```
假设系统的全局事务ID号目前到了44
begin;          -- 拿到系统的事务ID=44;
select * from users ;
commit;
```

数据查询规则

1.查找数据行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

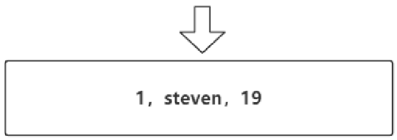
2.查找删除版本号要么为null，要么大于当前事务版本号的记录.

确保取出来的行记录在事务开启之前没有被删除

主键（自增） 姓名 年龄 数据行的版本号 删除版本号

表：teacher

id	name	age	DB_TRX_ID	DB_ROLL_PT
1	steven	18	1	33
2	mike	19	1	22
1	steven	19	33	NULL



数据的查询规则

关于快照读与当前读

Lock Based Concurrency Control(LBCC) 事务开始操作数据前，对其加锁，阻止其他事务对数据进行修改 ----->当前读是lbcc具体的实现

Multi Version Concurrency Control （MVCC） 事务开始操作数据前,将数据在当下时间点进行一份数据快照（Snapshot)的备份，并用这个快照来提供给其他事务进行一致性读取 并发访问(读或写)数据库时，对正在事务内处理的数据做多版本的管理。避免写操作的堵塞，从而引发读操作的并发阻塞问题. ------>快照读是mvcc具体的实现

快照读： SQL读取的数据是快照版本，也就是历史版本.普通的SELECT就是快照读 SELECT * FROM user WHERE id > 1 and id< 100

当前读： SQL读取的数据是最新版本。通过锁机制来保证读取的数据无法通过其他事务进行修改 UPDATE、DELETE、INSERT、SELECT ... LOCK IN SHARE MODE、SELECT ... FOR UPDATE都是当前读