MySQL 基础知识

作者：XinyiZhang
原文链接：https://ld246.com/article/1601264892949
来源网站：链滴

事务

概念

事务是逻辑上的一组操作,要么都执行,要么都不执行;

四大特性(ACID)

- 原子性： 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;
- 一致性： 执行事务前后,数据保持一致,多个事务对同一个数据读取的结果是相同的;
- 隔离性： 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库是独立的;
- 持久性：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有任何影响;

并发事务带来哪些问题?

- 脏读(Dirty read)：当一个事务正在访问数据并且对数据进行了修改,而这种修改还没有提交到数据中,这时另外一个事务也访问了这个数据,然后使用了这个数据.因为这个数据是还没有提交的数据,那么

外一个事务读到的这个数据是"脏数据",依据"脏数据"所做的操作可能是不正确的;

● 丢失修改(Lost to modify)：在一个事务读取一个数据时,另外一个事务也访问了该数据,那么在第一事务中修改了这个数据后,第二个事务也修改了这个数据.这样第一个事务内的修改结果就被丢失,因此为丢失修改;

● 不可重复读(Unrepeatableread)：在一个事务内多次读同一数据.在这个事务还没有结束时,另一个务也访问该数据.那么,在第一个事务中的两次读数据之间,由于第二个事务的修改导致第一个事务两次取的数据可能不太一样.这就发生了在一个事务内两次读到的数据是不一样的情况,因此称为不可重复读;

● 幻读(Phantom read)：幻读与不可重复读类似.它发生在一个事务(T1)读取了几行数据,接着另一个发事务(T2)插入了一些数据时.在随后的查询中,第一个事务(T1)就会发现多了一些原本不存在的记录,好像发生了幻觉一样,所以称为幻读;

事务隔离级别有哪些?

● READ-UNCOMMITTED(读取未提交)：最低的隔离级别,允许读取尚未提交的数据变更,可能会导致读、幻读或不可重复读;

● READ-COMMITTED(读取已提交)：允许读取并发事务已经提交的数据,可以阻止脏读,但是幻读或可重复读仍有可能发生;

● REPEATABLE-READ(可重复读)：对同一字段的多次读取结果都是一致的,除非数据是被本身事务自所修改,可以阻止脏读和不可重复读,但幻读仍有可能发生;

● SERIALIZABLE(可串行化)：最高的隔离级别,完全服从ACID的隔离级别.所有的事务依次逐个执行,这事务之间就完全不可能产生干扰,也就是说,该级别可以防止脏读、不可重复读以及幻读;

InnoDB 存储引擎在 REPEATABLE-READ(可重读)事务隔离级别下使用的是Next-Key Lock 锁算法,此可以避免幻读的产生,这与其他数据库系统(如 SQL Server)是不同的.所以说InnoDB 存储引擎的默支持的隔离级别是 REPEATABLE-READ(可重读)已经可以完全保证事务的隔离性要求,即达到了 SQL准的SERIALIZABLE(可串行化)隔离级别。

InnoDB 存储引擎在分布式事务的情况下一般会用到SERIALIZABLE(可串行化)隔离级别

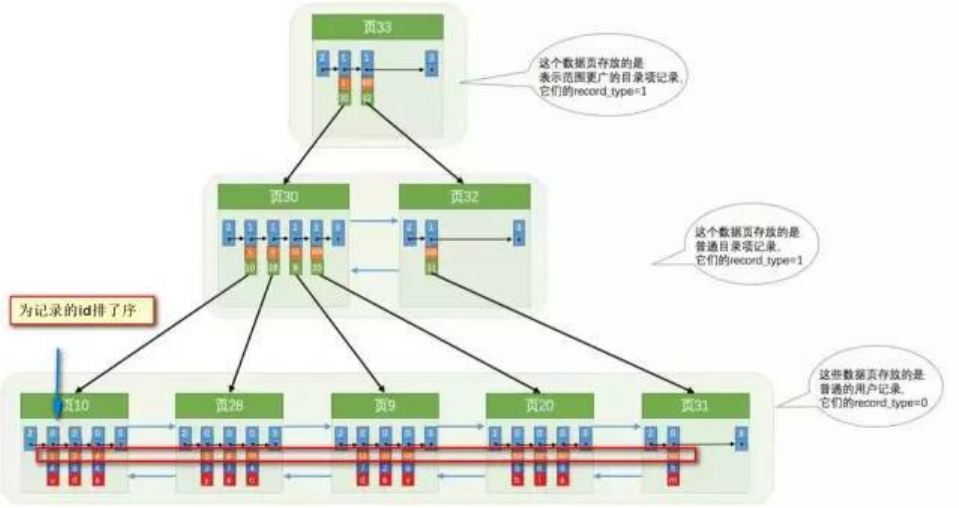
MySQL InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ(可重读)

可以通过 "SELECT @@tx_isolation;" 来查看隔离级别

索引

索引做了什么可以让我们查询加快速度?

将无序的数据变成有序(相对)



二分查找,时间复杂度近似为O(logn),其实底层结构就是B+树,B+树作为树的一种实现,能够让我们很快查找出对应的记录

什么是最左前缀原则?

MySQL中的索引可以以一定顺序引用多列,这种索引叫作联合索引.如User表的name和city加联合索就是(name,city),最左前缀原则指的是,如果查询的时候查询条件精确匹配索引的左边连续一列或几列,此列就可以被用到

由于最左前缀原则,在创建联合索引时,索引字段的顺序需要考虑字段值去重之后的个数,较多的放前面.RDER BY子句也遵循此规则

注意避免冗余索引

冗余索引指的是索引的功能相同,能够命中就肯定能命中,那么就是冗余索引如(name,city)和(name)这个索引就是冗余索引,能够命中后者的查询肯定是能够命中前者的.在大多数情况下,都应该尽量扩展已的索引而不是创建新索引.MySQLS7版本后,可以通过查询sys库的schema_redundant_indexes表来看冗余索引

添加索引

添加PRIMARY KEY(主键索引)

```
ALTER TABLE `table_name` ADD PRIMARY KEY (`column`);
```

添加UNIQUE(唯一索引)

```
ALTER TABLE `table_name` ADD UNIQUE (`column`);
```

添加INDEX(普通索引)

```
ALTER TABLE `table_name` ADD INDEX index_name (`column`);
```

添加FULLTEXT(全文索引)

`ALTER TABLE `table_name` ADD FULLTEXT (`column`);

添加多列索引

ALTER TABLE `table_name` ADD INDEX index_name (`column1`,`column2`,`column3`);

存储引擎

常用命令

查看MySQL提供的所有存储引擎

show engines;

查看MySQL当前默认的存储引擎

show variables like '%storage_engine%';

查看表的存储引擎

show table status like "table_name";

MyISAM和InnoDB区别

是否支持行级锁

MyISAM只有表级锁(table-level locking),而InnoDB支持行级锁(row-level locking)和表级锁,默认行级锁.

是否支持事务和崩溃后的安全恢复

MyISAM强调的是性能,每次查询具有原子性,其执行比InnoDB类型更快,但是不提供事务支持.但是InnDB提供事务支持事务,外部键等高级数据库功能.具有事务(commit)、回滚(rollback)和崩溃修复能力(cash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表.

是否支持外键

MyISAM不支持,而InnoDB支持.

是否支持MVCC

- 仅InnoDB支持.应对高并发事务,MVCC比单纯的加锁更高效;
- MVCC只在READ COMMITTED和REPEATABLE READ两个隔离级别下工作;
- MVCC可以使用乐观(optimistic)锁和悲观(pessimistic)锁来实现;
- 各数据库中MVCC实现并不统一;

乐观锁与悲观锁

悲观锁

总是假设最坏的情况,每次去拿数据的时候都认为别人会修改,所以每次在拿数据的时候都会上锁,这样人想拿这个数据就会阻塞直到它拿到锁(共享资源每次只给一个线程使用,其它线程阻塞,用完后再把资转让给其它线程).传统的关系型数据库里边就用到了很多这种锁机制,比如行锁、表锁、读锁、写锁等,是在做操作之前先上锁.Java中synchronized和ReentrantLock等独占锁就是悲观锁思想的实现.

乐观锁

总是假设最好的情况,每次去拿数据的时候都认为别人不会修改,所以不会上锁,但是在更新的时候会判一下在此期间别人有没有去更新这个数据,可以使用版本号机制和CAS算法实现.乐观锁适用于多读的用类型,这样可以提高吞吐量,像数据库提供的类似于write_condition机制,其实都是提供的乐观锁.在Jav中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的.

两种锁的使用场景

乐观锁适用于写比较少的情况下(多读场景),即冲突真的很少发生的时候,这样可以省去了锁的开销,加了系统的整个吞吐量.但如果是多写的情况,一般会经常产生冲突,这就会导致上层应用会不断的进行retry这样反倒是降低了性能,所以一般多写的场景下用悲观锁就比较合适.

乐观锁常见的两种实现方式

版本号机制

在数据表中加上一个数据版本号version字段,表示数据被修改的次数,当数据被修改时,version值会加一当线程A要更新数据值时,在读取数据的同时也会读取version值,在提交更新时,若刚才读取到的version为当前数据库中的version值相等时才更新,否则重试更新操作,直到更新成功.

CAS算法

即compare and swap(比较与交换),是一种有名的无锁算法.无锁编程,即不使用锁的情况下实现多线之间的变量同步,也就是在没有线程被阻塞的情况下实现变量的同步,所以也叫非阻塞同步(Non-blocki g Synchronization).CAS算法涉及到三个操作数:

- 需要读写的内存值 V
- 进行比较的值 A
- 拟写入的新值 B

当且仅当V的值等于A时,CAS通过原子方式用新值B来更新V的值,否则不会执行任何操作(比较和替换一个原子操作).一般情况下是一个自旋操作,即不断的重试.

乐观锁的缺点

ABA问题

如果一个变量V初次读取的时候是A值,并且在准备赋值的时候检查到它仍然是A值,那我们就能说明它值没有被其他线程修改过了吗?很明显是不能的,因为在这段时间它的值可能被改为其他值,然后又改回A那CAS操作就会误认为它从来没有被修改过.这个问题被称为CAS操作的 "ABA"问题.

JDK 1.5 以后的AtomicStampedReference类就提供了此种能力,其中的compareAndSet方法就是首检查当前引用是否等于预期引用,并且当前标志是否等于预期标志,如果全部相等,则以原子方式将该引和该标志的值设置为给定的更新值.

循环时间长开销大

自旋CAS(也就是不成功就一直循环执行直到成功)如果长时间不成功,会给CPU带来非常大的执行开销.如果JVM能支持处理器提供的pause指令那么效率会有一定的提升,pause指令有两个作用

- 可以延迟流水线执行指令(de-pipeline),使CPU不会消耗过多的执行资源,延迟的时间取决于具体实的版本,在一些处理器上延迟时间是零.
- 可以避免在退出循环的时候因内存顺序冲突(memory order violation)而引起CPU流水线被清空(CP pipeline flush),从而提高CPU的执行效率.

只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效,当操作涉及跨多个共享变量时CAS无效.但是从JDK1.5开始,提供了Atomiceference类来保证引用对象之间的原子性,你可以把多个变量放在一个对象里来进行 CAS 操作.所以我可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作.

锁机制与InnoDB锁算法

MyISAM和InnoDB存储引擎使用的锁

- MyISAM -- 采用表级锁(table-level locking)
- InnoDB -- 支持行级锁(row-level locking)和表级锁,默认为行级锁

表级锁和行级锁对比

- 表级锁： Mysql中锁定粒度最大的一种锁,对当前操作的整张表加锁,实现简单,资源消耗也比较少,加快,不会出现死锁.其锁定粒度最大,触发锁冲突的概率最高,并发度最低,MyISAM和InnoDB引擎都支持级锁;
- 行级锁： Mysql中锁定粒度最小的一种锁,只针对当前操作的行进行加锁.行级锁能大大减少数据库作的冲突.其加锁粒度最小,并发度高,但加锁的开销也最大,加锁慢,会出现死锁;

InnoDB存储引擎的锁的算法

- Record lock：单个行记录上的锁;
- Gap lock：间隙锁,锁定一个范围,不包括记录本身;
- Next-key lock：record+gap锁定一个范围,包含记录本身;

相关知识点

- innodb对于行的查询使用next-key lock;
- Next-locking keying为了解决Phantom Problem幻读问题;
- 当查询的索引含有唯一属性时,将next-key lock降级为record key;
- Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内,而这会导致幻读问题的产生;
- 有两种方式显式关闭gap锁：(除了外键约束和唯一性检查外,其余情况仅使用record lock)
  - 将事务隔离级别设置为RCB;
  - 将参数innodb_locks_unsafe_for_binlog设置为1;

大表优化

限定数据的范围

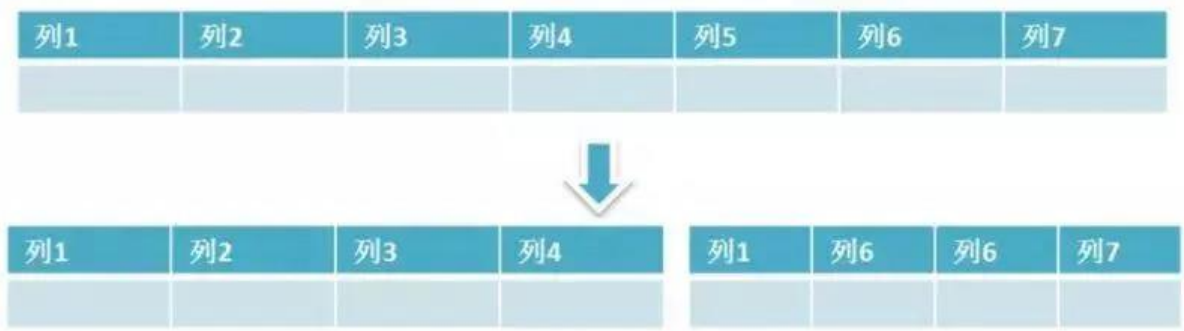
务必禁止不带任何限制数据范围条件的查询语句.

读/写分离

经典的数据库拆分方案,主库负责写,从库负责读;

垂直分区

- 根据数据库里面数据表的相关性进行拆分.
- 简单来说垂直拆分是指数据表列的拆分,把一张列比较多的表拆分为多张表.

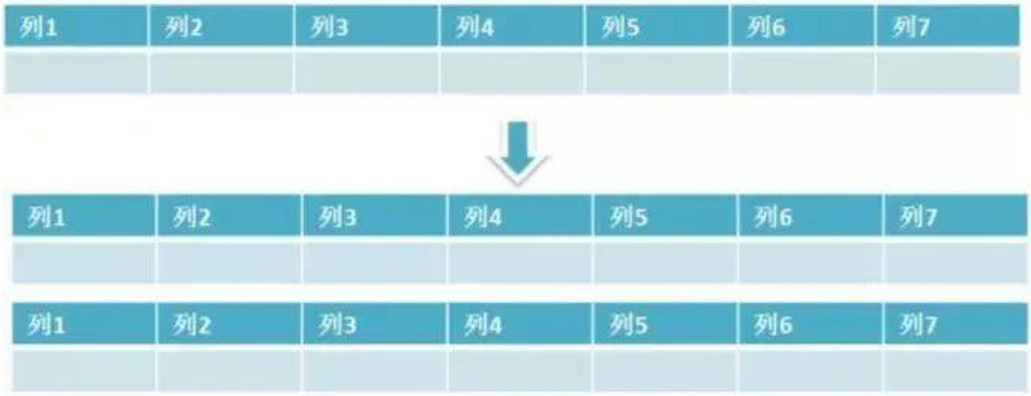


- 垂直拆分的优点：可以使得列数据变小,在查询时减少读取的Block数,减少I/O次数.此外,垂直分区可简化表的结构,易于维护.
- 垂直拆分的缺点：主键会出现冗余,需要管理冗余列,并会引起Join操作,可以通过在应用层进行Join解决.此外,垂直分区会让事务变得更加复杂;

水平分区

- 保持数据表结构不变,通过某种策略存储数据分片.这样每一片数据分散到不同的表或者库中,达到了布式的目的.
- 水平拆分可以支撑非常大的数据量.
- 水平拆分是指数据表行的拆分,表的行数超过200万行时,就会变慢,这时可以把一张的表的数据拆成

张表来存放.



● 水平拆分可以支持非常大的数据量.需要注意的一点是：分表仅仅是解决了单一表数据过大的问题,由于表的数据还是在同一台机器上,其实对于提升MySQL并发能力没有什么意义,所以水平拆分最好分库.

● 水平拆分能够支持非常大的数据量存储,应用端改造也少,但分片事务难以解决,跨节点Join性能较差,辑复杂.《Java工程师修炼之道》的作者推荐尽量不要对数据进行分片,因为拆分会带来逻辑、部署、维的各种复杂度,一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的.如果实要分片,尽量选择客户端分片架构,这样可以减少一次和中间件的网络I/O.

数据库分片的两种常见方案

● 客户端代理：分片逻辑在应用端,封装在jar包中,通过修改或者封装JDBC层来实现. 当当网的Sharding JDBC 、阿里的TDDL是两种比较常用的实现;

● 中间件代理：在应用和数据中间加了一个代理层.分片逻辑统一维护在中间件服务中.我们现在谈的Mcat、360的Atlas、网易的DDB等等都是这种架构的实现;