# 关系型数据库优势

## 关系型数据库为什么能活这么久？

直观：就是个表格；

使用简单：一般都只需要简单的SQL便可查出；

对数据完整性的支持很好；

我支持事务；

范式：想要使用我们关系型数据库，必须得遵守一定的规则，这些规则就是“范式”。

第一范式是基本要求，即每个列都是不分割的数据项， 如果连这个都满足不了，还是洗洗睡吧。

第二范式要求实体属性要完全依赖主键，不能依赖部分主键。

第三范式就是一个表中不能包含其它表中已包含的非主关键字信息。不严谨地说就是这个表只包含其他表的ID。

做“数据的集成”

存在问题：高并发，大流量存在瓶颈。对分布式系统支持不好, 难于组成集群。

# 事务

Java事务的类型有三种：JDBC事务、容器事务、JTA(Java Transaction API)事务。

不用事务的编程框架来管理事务，直接使用资源管理器来控制事务。典型的就是java.sql.Connection 中的 setAutoCommit、commit、rollback方法。本地事务比较简单，基本原理就是数据库的事务原理

**本地事务的优点**

支持严格的ACID属性

可靠

高效

状态可以只在资源管理器中维护

应用编程模型简单

**本地事务的局限**

不具备分布式事务处理能力

隔离的最小单位由资源管理器决定，如数据库中的一条记录

全局事务的定义：全局事务就是一个标准的分布式事务。全局事务是由资源管理器管理和协调的事务。

全局事务是一个DTP模型的事务，所谓DTP模型指的是X/Open DTP(X/Open Distributed Transaction Processing Reference Model)，是X/Open 这个组织定义的一套分布式事务的标准，也就是了定义了规范和API接口，由这个厂商进行具体的实现。

X/Open DTP 定义了三个组件：AP，TM，RM 和两个协议：XA、TX

AP(Application Program)：也就是应用程序，可以理解为使用DTP的程序

RM(Resource Manager)：资源管理器，这里可以理解为一个DBMS系统，或者消息服务器管理系统，应用程序通过资源管理器对资源进行控制。

TM(Transaction Manager)：事务管理器，负责协调和管理事务，提供给AP应用程序编程接口以及管理资源管理器。

XA协议：应用或应用服务器与事务管理之前通信的接口

TX协议：全局事务管理器与资源管理器之间通信的接口

事务管理器控制着全局事务，管理事务生命周期，并协调资源。资源管理器负责控制和管理实际资源

JTA(Java Transaction API):面向应用、应用服务器与资 源管理器的高层事务接口。

JTS(Java Transaction Service):JTA事务管理器的实现标 准,向上支持JTA,向下通过CORBA OTS实现跨事务域的互 操作性。

EJB:基于组件的应用编程模型,通过声明式事务管理进一步 简化事务应用的编程。

原文：https://mp.weixin.qq.com/s/QToXYvPzIBW36vAPqp6Ciw

# MySQL中事物ACID的原理

**Mysql怎么保证一致性的**

ACID嘛，原子性(Atomicity/,ætə'mɪsɪtɪ/)、一致性(Consistency/kən'sɪst(ə)nsɪ/)、隔离性(Isolation/aɪsə'leɪʃ(ə)n/)、持久性(Durability/djuərə'biləti/)

ACID四大特性之中，C(一致性)是目的，A(原子性)、I(隔离性)、D(持久性)是手段，是为了保证一致性，数据库提供的手段。数据库必须要实现AID三大特性，才有可能实现一致性。

**Mysql怎么保证原子性的**

OK，是利用Innodb的undo log

undo log名为回滚日志，是实现原子性的关键，当事务回滚时能够撤销所有已经成功执行的sql语句，他需要记录你要回滚的相应日志信息。

undo log记录了这些回滚需要的信息，当事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。

**Mysql怎么保证持久性的**

OK，是利用Innodb的redo log。

正如之前说的，Mysql是先把磁盘上的数据加载到内存中，在内存中对数据进行修改，再刷回磁盘上。如果此时突然宕机，内存中的数据就会丢失。

怎么解决这个问题？简单啊，事务提交前直接把数据写入磁盘就行啊。但是这样太浪费资源了。于是，决定采用redo log解决上面的问题。当做数据修改的时候，不仅在内存中操作，还会在redo log中记录这次操作。当事务提交的时候，会将redo log日志进行刷盘(redo log一部分在内存中，一部分在磁盘上)。当数据库宕机重启的时候，会将redo log中的内容恢复到数据库中，再根据undo log和binlog内容决定回滚数据还是提交数据。

采用redo log的好处是进行刷盘比对数据页刷盘效率高，具体表现如下：redo log体积小，毕竟只记录了哪一页修改了啥，因此体积小，刷盘快。redo log是一直往末尾进行追加，属于顺序IO。效率显然比随机IO来的快。

**Mysql怎么保证隔离性的**

OK,利用的是锁和MVCC机制。

至于MVCC,即多版本并发控制(Multi Version Concurrency Control),一个行记录数据有多个版本对快照数据，这些快照数据在undo log中。

如果一个事务读取的行正在做DELELE或者UPDATE操作，读取操作不会等行上的锁释放，而是读取该行的快照版本。

由于MVCC机制在可重复读(Repeateable Read)和读已提交(Read Commited)的MVCC表现形式不同，就不赘述了。但是有一点说明一下，在事务隔离级别为读已提交(Read Commited)时，一个事务能够读到另一个事务已经提交的数据，是不满足隔离性的。但是当事务隔离级别为可重复读(Repeateable Read)中，是满足隔离性的。

# 优化现有MySQL数据库

**1.数据库设计和表创建时就要考虑性能**

简言之就是使用合适的数据类型,选择合适的索引

设计表时要注意的东西：表字段避免null值出现，null值很难查询优化且占用额外的索引空间；尽量使用INT而非BIGINT；使用枚举或整数代替字符串类型；单表不要有太多字段；

索引：要根据查询有针对性的创建，考虑在WHERE和ORDERBY命令上涉及的列建立索引；应尽量避免在WHERE子句中对字段进行NULL值判断；字符字段只建前缀索引；字符字段最好不要做主键；不用外键，由程序保证约束；使用多列索引时主意顺序和查询条件保持一致，同时删除不必要的单列索引；

**2.SQL的编写需要注意优化**

使用limit对查询结果的记录进行限定

避免select \*，将需要查找的字段列出来

使用连接（join）来代替子查询

拆分大的delete或insert语句

可通过开启慢查询日志来找出较慢的SQL

不做列运算：SELECT id WHERE age + 1 = 10，任何对列的操作都将导致表扫描，它包括数据库教程函数、计算表达式等等，查询时要尽可能将操作移至等号右边

SQL语句尽可能简单：一条SQL只能在一个cpu运算；大语句拆小语句，减少锁时间；一条大SQL可以堵死整个库

OR改写成IN：OR的效率是n级别，IN的效率是log(n)级别，in的个数建议控制在200以内

不用函数和触发器，在应用程序实现

避免%xxx式查询

少用JOIN

使用同类型进行比较，比如用'123'和'123'比，123和123比

尽量避免在WHERE子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描

对于连续数值，使用BETWEEN不用IN：SELECT id FROM t WHERE num BETWEEN 1 AND 5

列表数据不要拿全表，要使用LIMIT来分页，每页数量也不要太大

**3.分区**

MySQL在5.1版引入的分区是一种简单的水平拆分，用户需要在建表的时候加上分区参数，对应用是透明的无需修改代码

MySQL实现分区的方式也意味着索引也是按照分区的子表定义，没有全局索引

用户的SQL语句是需要针对分区表做优化，SQL条件中要带上分区条件的列，从而使查询定位到少量的分区上，否则就会扫描全部分区，可以通过EXPLAINPARTITIONS来查看某条SQL语句会落在那些分区上，从而进行SQL优化

分区的好处：

可以让单表存储更多的数据

分区表的数据更容易维护，可以通过清楚整个分区批量删除大量数据，也可以增加新的分区来支持新插入的数据。另外，还可以对一个独立分区进行优化、检查、修复等操作

部分查询能够从查询条件确定只落在少数分区上，速度会很快

分区表的数据还可以分布在不同的物理设备上，从而搞笑利用多个硬件设备

可以使用分区表赖避免某些特殊瓶颈，例如InnoDB单个索引的互斥访问、ext3文件系统的inode锁竞争

可以备份和恢复单个分区

分区的限制和缺点

一个表最多只能有1024个分区

如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来

分区表无法使用外键约束

NULL值会使分区过滤无效

所有分区必须使用相同的存储引擎

**4.分表**

分表分为垂直拆分和水平拆分，通常以某个字段做拆分项。

**5.分库**

把一个数据库分成多个，建议做个读写分离就行了，真正的做分库也会带来大量的开发成本