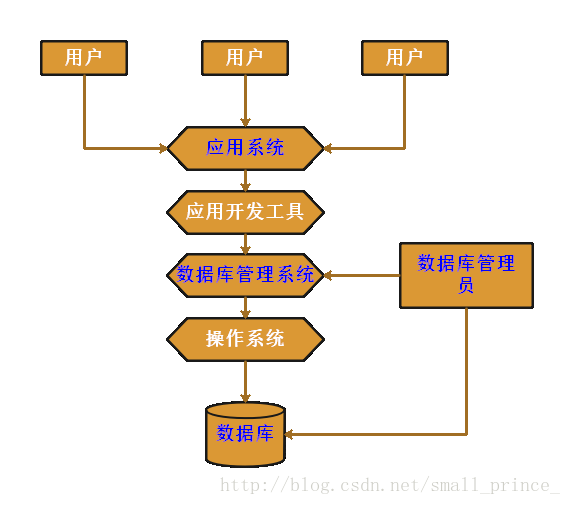
数据库

## 数据库介绍

### 概念

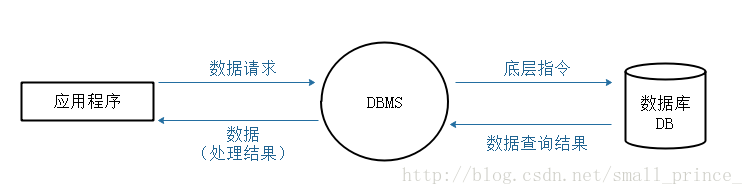
Database：简而言之可视为[电子化](https://baike.baidu.com/item/%E7%94%B5%E5%AD%90%E5%8C%96" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)的[文件柜](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E6%9F%9C" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)——存储电子[文件](https://baike.baidu.com/item/%E6%96%87%E4%BB%B6" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)的处所，用户可以对文件中的数据进行新增、截取、更新、删除等操作。所谓“数据库”是以一定方式储存在一起、能与多个用户[共享](https://baike.baidu.com/item/%E5%85%B1%E4%BA%AB" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)、具有尽可能小的[冗余度](https://baike.baidu.com/item/%E5%86%97%E4%BD%99%E5%BA%A6" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)、与应用程序彼此独立的数据[集合](https://baike.baidu.com/item/%E9%9B%86%E5%90%88" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93/_blank)，这也是与文件系统的本质区别。

数据库中的几个术语：数据，数据库（DB），数据库管理系统（DBMS），数据库系统（DBS），数据库管理员（DBA）。



DBMS：位于用户应用程序与操作系统之间的一层数据管理软件，是数据库系统的核心组成部分。为用户或应用程序提供访问数据库的方法：包括数据库的建立、查询、更新以及各种数据控制。

数据库管理系统是为了科学地组织和存储数据、高效地获取和维护数据。



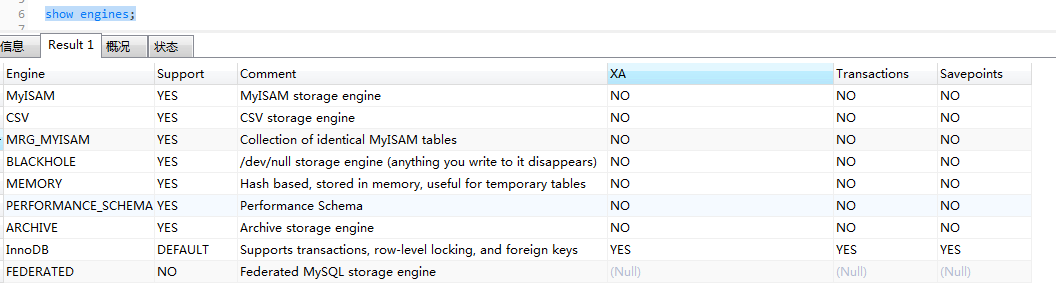
### 类型

关系数据库（sql）：mysql、oracle...

非关系型数据库（nosql）：redis、mongodb...

### 引擎

数据库引擎是用于存储、处理和保护数据的核心服务。当你访问数据库时，不管是手工访问，还是程序访问，都不是直接读写数据库文件，而是通过数据库引擎去访问数据库文件



Support:是否安装和默认，xa：允许分布式事务

Myisam和innodb的区别，新版本5.5后默认innodb引擎，可知综合性能上要高于其他引擎



## 数据库索引

### 概念

[索引](https://baike.baidu.com/item/%E7%B4%A2%E5%BC%95" \t "https://baike.baidu.com/item/_blank)（index）是对数据库表中一个或多个列（组合索引）的值进行排序的数据结构。

### 作用

好比是一本书前面的目录，加快数据库的查询速度，如某列加上了唯一索引，那么查找到了一条数据就结束了，假如没有索引的话，即使只有一条数据查找了也会继续全表数据查找等全部找完了再结束，这样就浪费了很多时间，明明就一条找到返回就可以了，即使是普通索引找到一条后那么其他数据在这条数据的附近，那么找小范围的数据就可以了也不必全表找。

### 原理

二分查找、二叉树查找等，添加索引的话，首先去索引列表中查询，而我们的索引列表是B类树的数据结构，查询的时间复杂度为O(log2N)，什么b+tree太复杂，简单了解即可。

### 类型

主要的类型（不同版本可能有很多）

Normal：普通索引，允许出现相同的索引内容

UNIQU：唯一索引,不可以出现相同的值,可以有NULL值

FULLTEXT :全文索引, FULLTEXT 用于搜索很长一篇文章的时候，效果最好

PRIMARY :主键索引.主键不允许空值，唯一索引允许空值，主键只允许一个，唯一索引允许多个主键产生唯一的聚集索引，唯一索引产生唯一的非聚集索引  
注：聚集索引确定表中数据的物理顺序，所以是主键是唯一的（聚集就是整理数据的意思）

组合索引与前缀索引，这两种称呼是对建立索引技巧的一种称呼,并非索引的类型

组合索引：指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀集合，例如索引index1:(a,b,c)，只会走a、a,b、a,b,c 三种类型的查询，a,c不走，因为只走a字段索引，不会走c字段，如果有c作为条件那么还是会全表查询等于没有走索引。

前缀索引：如果索引列长度过长,这种列索引时将会产生很大的索引文件,不便于操作,可以使用前缀索引方式进行索引，如ALTER TABLE `user` ADD INDEX `uname`(title(10)); 增加前缀索引SQL,将人名的索引建立在10,这样可以减少索引文件大小,加快索引查询速度

### 缺点

* 在数据库建立过程中，需花费较多的时间去建立并维护索引，特别是随着数据总量的增加，所花费的时间将不断递增。
* 在数据库中创建的索引需要占用一定的物理存储空间，这其中就包括数据表所占的数据空间以及所创建的每一个索引所占用的物理空间，如果有必要建立起聚簇索引，所占用的空间还将进一步的增加
* 在对表中的数据进行修改时，例如对其进行增加、删除或者是修改操作时，索引还需要进行动态的维护，对索引文件进行重新排序或更新。

### 适合加索引的列

* 维度高的列创建索引，如user表中性别列就维度低不适合加索引，即使有很多数据但维度很低，加索引也是全表查询
* 对 where,on,group by,order by 中出现的列使用索引
* 对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
* 对于那些定义为text, image和[bit数据类型](https://baike.baidu.com/item/bit%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B" \t "https://baike.baidu.com/item/_blank)的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少,不利于使用索引。

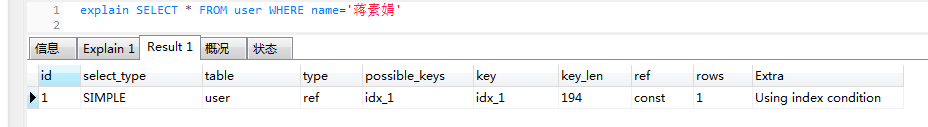
### 不走索引的sql

* 条件字段选择性弱，查出的结果集较大，不走索引；
* 索引列上有函数运算，导致不走索引，如SELECT \* FROM T WHERE FUN(Y) = XXX
* 隐式转换导致不走索引，如SELECT \* FROM T WHERE Y = 5但是Y列是VARCHAR2会隐式的转换SELECT \* FROM T WHERE TO\_NUMBER(Y) = 5；
* 使用不等于操作符如：<>、!= 等不走索引；
* like 后面的字符当首位为通配符时不走索引；
* 对于count(\*)当索引字段有not null约束时走索引，否则不走索引；

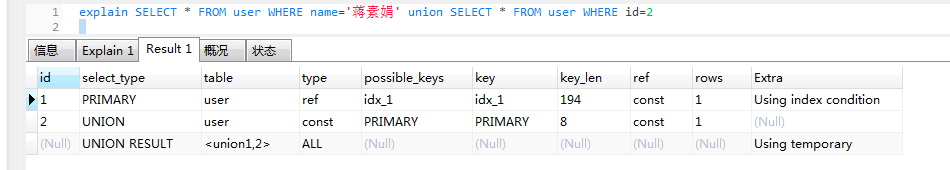
### mysql最佳索引攻略

Explain 执行计划：sql执行调优，查看执行具体情况，可以看到是否用到了索引。在语句前加上explain即可，比较重要的属性type、key、rows。如下示例

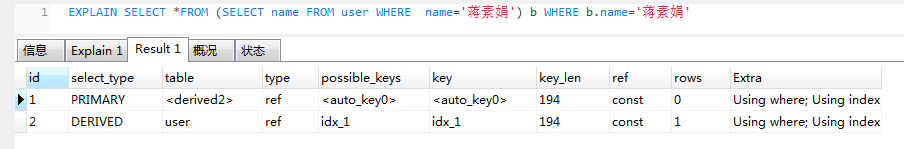
简单查询



Union查询



含有子查询



|  |  |
| --- | --- |
| **id** | SELECT识别符。这是SELECT的查询序列号，如图1代表第一个select、2代表第二个select |
| **select\_type** | SELECT类型,可以为以下任何一种:   * **SIMPLE**:简单SELECT(不使用UNION或子查询) * **PRIMARY**:最外面的SELECT * **UNION**:UNION中的第二个或后面的SELECT语句 * **DEPENDENT UNION**:UNION中的第二个或后面的SELECT语句,取决于外面的查询 * **UNION RESULT**:UNION 的结果 * **SUBQUERY**:子查询中的第一个SELECT * **DEPENDENT SUBQUERY**:子查询中的第一个SELECT,取决于外面的查询 * **DERIVED**:导出表的SELECT(FROM子句的子查询) |
| **table** | 输出的行所引用的表，不一定是实表，可能子查询或联合查询的结果虚表，如上示例。 |
| **type** | 连接类型，一种数据库引擎查找表的一种方式。按照从最佳类型到最坏类型进行排序   * system:表仅有一行(系统表)。这是const联接类型的一个特例。 * const:表最多有一个匹配行，如列上加了唯一索引或主键，那么可以确定只有一条数据,因为仅有一行,在这行的列值可被优化器剩余部分认为是常数。const表很快,因为它们只读取一次! * eq\_ref:对于每个来自于前面的表的行组合,从该表中读取一行。这可能是最好的联接类型,除了const类型。 * ref:对于每个来自于前面的表的行组合,所有有匹配索引值的行将从这张表中读取。 * ref\_or\_null:该联接类型如同ref,但是添加了MySQL可以专门搜索包含NULL值的行。 * index\_merge:该联接类型表示使用了索引合并优化方法。 * unique\_subquery:该类型替换了下面形式的IN子查询的ref: value IN (SELECT primary\_key FROM single\_table WHERE some\_expr) unique\_subquery是一个索引查找函数,可以完全替换子查询,效率更高。 * index\_subquery:该联接类型类似于unique\_subquery。可以替换IN子查询,但只适合下列形式的子查询中的非唯一索引: value IN (SELECT key\_column FROM single\_table WHERE some\_expr) * range:只检索给定范围的行,使用一个索引来选择行。 * index:该联接类型与ALL相同,除了只有索引树被扫描。这通常比ALL快,因为索引文件通常比数据文件小。 * ALL:对于每个来自于先前的表的行组合,进行完整的表扫描。 |
| **possible\_keys** | 指出MySQL能使用哪个索引在该表中找到行 |
| **key** | 显示MySQL实际决定使用的键(索引)。如果没有选择索引,键是NULL。 |
| **key\_len** | 索引长度，索引添加列类型长度，如name1 varchar(225),name2 varchar(225),组合索引index（name1，name2），那么长度为225\*2，建表是不同引擎支持的最大长度不同，不能超过最大值。如果键是NULL,则长度为NULL。 |
| **ref** | 显示使用哪个列或常数与key一起从表中选择行。 |
| **rows** | 显示MySQL认为它执行查询时必须检查的行数。多行之间的数据相乘可以估算要处理的行数。 |
| **filtered** | 显示了通过条件过滤出的行数的百分比估计值。 |
| **Extra** | 该列包含MySQL解决查询的详细信息   * **Distinct**:MySQL发现第1个匹配行后,停止为当前的行组合搜索更多的行。 * **Not exists**:MySQL能够对查询进行LEFT JOIN优化,发现1个匹配LEFT JOIN标准的行后,不再为前面的的行组合在该表内检查更多的行。 * **range checked for each record (index map: #)**:MySQL没有发现好的可以使用的索引,但发现如果来自前面的表的列值已知,可能部分索引可以使用。 * **Using filesort**:MySQL需要额外的一次传递,以找出如何按排序顺序检索行。 * **Using index**:从只使用索引树中的信息而不需要进一步搜索读取实际的行来检索表中的列信息。 * **Using temporary**:为了解决查询,MySQL需要创建一个临时表来容纳结果。 * **Using where**:WHERE 子句用于限制哪一个行匹配下一个表或发送到客户。 * **Using sort\_union(...), Using union(...), Using intersect(...)**:这些函数说明如何为index\_merge联接类型合并索引扫描。 * **Using index for group-by**:类似于访问表的Using index方式,Using index for group-by表示MySQL发现了一个索引,可以用来查 询GROUP BY或DISTINCT查询的所有列,而不要额外搜索硬盘访问实际的表。 |

## 数据库事务

### 概念

事务（Transaction），一般是指要做的或所做的事情。在计算机[术语](https://baike.baidu.com/item/%E6%9C%AF%E8%AF%AD" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)中是指访问并可能更新数据库中各种[数据项](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E9%A1%B9/3227309" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)的一个程序执行单元(unit)。事务通常由[高级数据库](https://baike.baidu.com/item/%E9%AB%98%E7%BA%A7%E6%95%B0%E6%8D%AE%E5%BA%93/1439366" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)操纵语言或编程语言（如SQL，C++或Java）书写的[用户程序](https://baike.baidu.com/item/%E7%94%A8%E6%88%B7%E7%A8%8B%E5%BA%8F/7450916" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)的执行所引起，并用形如begin transaction和end transaction语句（或[函数调用](https://baike.baidu.com/item/%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8/4127405" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)）来界定。事务由事务开始(begin transaction)和事务结束(end transaction)之间执行的全体操作组成。在[关系数据库](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E6%95%B0%E6%8D%AE%E5%BA%93" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)中，一个事务可以是一条SQL语句，一组SQL语句或整个程序。

### ACID特性

[事务](https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/5945882" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)是恢复和[并发控制](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91%E6%8E%A7%E5%88%B6" \t "https://baike.baidu.com/item/%E4%BA%8B%E5%8A%A1/_blank)的基本单位。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。

原子性（atomicity）:一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。

一致性（consistency）:事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

隔离性（isolation）:一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。在同一个事务中，数据需要保证一致性，也就是说，在当前的事务中，删除了就是看不到了，插入了就是多了一条数据，但这些操作都是在当前事务内部的。对于外界是隔离的。如果我们开启事务插入一条数据，还没有提交，这时新开一个事务，去查询数据，是发现不了未提交数据的

持久性（durability）:持久性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

### 事务隔离性

当多个线程都开启事务操作数据库中的数据时，数据库系统要能进行隔离操作，以保证各个线程获取数据的准确性，在介绍数据库提供的各种隔离级别之前，我们先看看如果不考虑事务的隔离性，会发生的几种问题

#### 脏读

脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户A向用户B转账100元，对应SQL命令如下

update account set money=money+100 where name=’B’; (此时A通知B)

update account set money=money - 100 where name=’A’;

当只执行第一条SQL时，A通知B查看账户，B发现确实钱已到账（此时即发生了脏读），而之后无论第二条SQL是否执行，只要该事务不提交，则所有操作都将回滚，那么当B以后再次查看账户时就会发现钱其实并没有转。

#### 不可重复读

不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。例如事务T1在读取某一数据，而事务T2立马修改了这个数据并且提交事务给数据库，事务T1再次读取该数据就得到了不同的结果，发送了不可重复读。

#### 虚读(幻读)

幻读是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。

MySQL数据库为我们提供的四种隔离级别：

① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

③ Read committed (读已提交)：可避免脏读的发生。

④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

以上四种隔离级别最高的是Serializable级别，最低的是Read uncommitted级别，当然级别越高，执行效率就越低。像Serializable这样的级别，就是以锁表的方式(类似于Java多线程中的锁)使得其他的线程只能在锁外等待，所以平时选用何种隔离级别应该根据实际情况

在MySQL数据库中查看当前事务的隔离级别：select @@tx\_isolation;

设置事务的隔离级别：set [glogal | session] transaction isolation level 隔离级别名称;

### 分布式事务

#### 概念

分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性，如果我们做了垂直分库或者水平分库以后,就必然会涉及到跨库执行SQL的问题,这样就引发了互联网界的老大难问题-"分布式事务"。

#### [产生的原因](https://www.jianshu.com/p/b1f881a4e8d8" \t "https://www.jianshu.com/p/_blank)

本质相同，都是因为要操作的数据库变多了！

**[1、数据库分库分表](" \t "_blank)**

当数据库单表一年产生的数据超过1000W，那么就要考虑分库分表，具体分库分表的原理在此不做解释，以后有空详细说，简单的说就是原来的一个数据库变成了多个数据库。这时候，如果一个操作既访问01库，又访问02库，而且要保证数据的一致性，那么就要用到分布式事务。

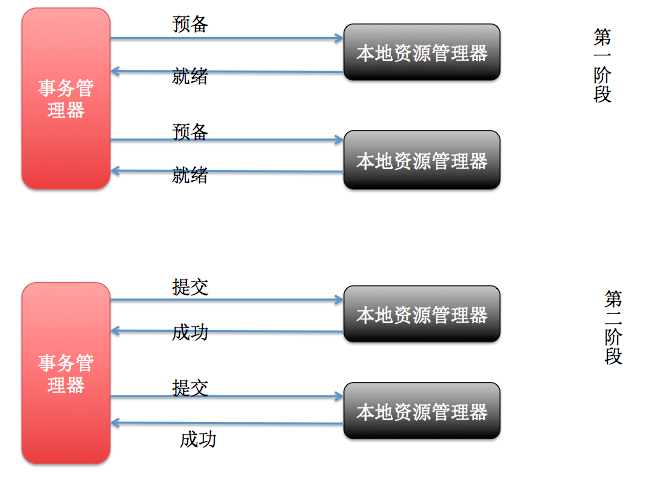
**[2、应用SOA化](" \t "_blank)**

所谓的SOA化，就是业务的服务化。比如原来单机支撑了整个电商网站，现在对整个网站进行拆解，分离出了订单中心、用户中心、库存中心。对于订单中心，有专门的数据库存储订单信息，用户中心也有专门的数据库存储用户信息，库存中心也会有专门的数据库存储库存信息。这时候如果要同时对订单和库存进行操作，那么就会涉及到订单数据库和库存数据库，为了保证数据一致性，就需要用到分布式事务。

#### 解决方案

**1、基于XA协议的两阶段提交**

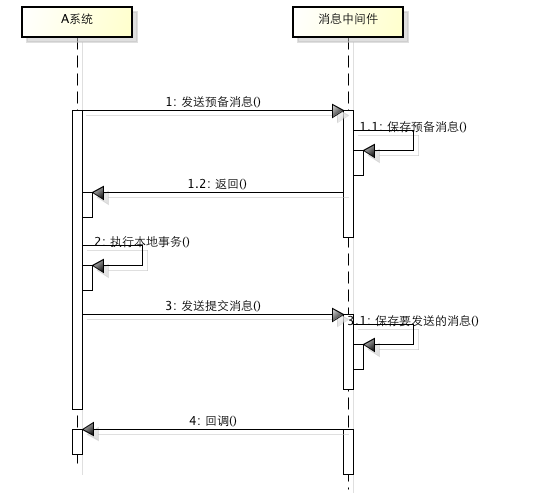
XA是一个分布式事务协议，由Tuxedo提出。XA中大致分为两部分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如Oracle、DB2这些商业数据库都实现了XA接口，而事务管理器作为全局的调度者，负责各个本地资源的提交和回滚。XA实现分布式事务的原理如下：



总的来说，XA协议比较简单，而且一旦商业数据库实现了XA协议，使用分布式事务的成本也比较低。但是，XA也有致命的缺点，那就是性能不理想，特别是在交易下单链路，往往并发量很高，XA无法满足高并发场景。XA目前在商业数据库支持的比较理想，在mysql数据库中支持的不太理想，mysql的XA实现，没有记录prepare阶段日志，主备切换回导致主库与备库数据不一致。许多nosql也没有支持XA，这让XA的应用场景变得非常狭隘。

**2、消息事务+最终一致性**

所谓的消息事务就是基于消息中间件的两阶段提交，本质上是对消息中间件的一种特殊利用，它是将本地事务和发消息放在了一个分布式事务里，保证要么本地操作成功成功并且对外发消息成功，要么两者都失败，开源的RocketMQ就支持这一特性，具体原理如下：



1、A系统向消息中间件发送一条预备消息  
2、消息中间件保存预备消息并返回成功  
3、A执行本地事务  
4、A发送提交消息给消息中间件

通过以上4步完成了一个消息事务。对于以上的4个步骤，每个步骤都可能产生错误，下面一一分析：

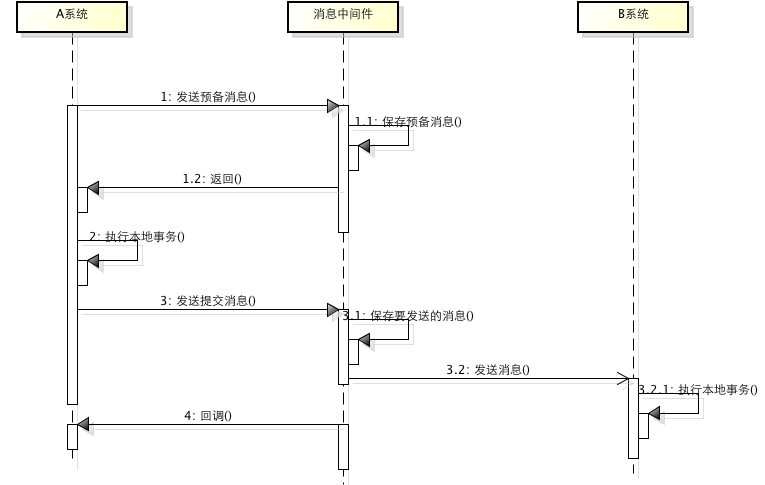
步骤一出错，则整个事务失败，不会执行A的本地操作

步骤二出错，则整个事务失败，不会执行A的本地操作

步骤三出错，这时候需要回滚预备消息，怎么回滚？答案是A系统实现一个消息中间件的回调接口，消息中间件会去不断执行回调接口，检查A事务执行是否执行成功，如果失败则回滚预备消息

步骤四出错，这时候A的本地事务是成功的，那么消息中间件要回滚A吗？答案是不需要，其实通过回调接口，消息中间件能够检查到A执行成功了，这时候其实不需要A发提交消息了，消息中间件可以自己对消息进行提交，从而完成整个消息事务

基于消息中间件的两阶段提交往往用在高并发场景下，将一个分布式事务拆成一个消息事务（A系统的本地操作+发消息）+B系统的本地操作，其中B系统的操作由消息驱动，只要消息事务成功，那么A操作一定成功，消息也一定发出来了，这时候B会收到消息去执行本地操作，如果本地操作失败，消息会重投，直到B操作成功，这样就变相地实现了A与B的分布式事务。原理如下：



虽然上面的方案能够完成A和B的操作，但是A和B并不是严格一致的，而是最终一致的，我们在这里牺牲了一致性，换来了性能的大幅度提升。当然，这种玩法也是有风险的，如果B一直执行不成功，那么一致性会被破坏，具体要不要玩，还是得看业务能够承担多少风险。

**3、TCC编程模式**

所谓的TCC编程模式，也是两阶段提交的一个变种。TCC提供了一个编程框架，将整个业务逻辑分为三块：Try、Confirm和Cancel三个操作。以在线下单为例，Try阶段会去扣库存，Confirm阶段则是去更新订单状态，如果更新订单失败，则进入Cancel阶段，会去恢复库存。总之，TCC就是通过代码人为实现了两阶段提交，不同的业务场景所写的代码都不一样，复杂度也不一样，因此，这种模式并不能很好地被复用。

总结

分布式事务，本质上是对多个数据库的事务进行统一控制，按照控制力度可以分为：不控制、部分控制和完全控制。不控制就是不引入分布式事务，部分控制就是各种变种的两阶段提交，包括上面提到的消息事务+最终一致性、TCC模式，而完全控制就是完全实现两阶段提交。部分控制的好处是并发量和性能很好，缺点是数据一致性减弱了，完全控制则是牺牲了性能，保障了一致性，具体用哪种方式，最终还是取决于业务场景。作为技术人员，一定不能忘了技术是为业务服务的，不要为了技术而技术，针对不同业务进行技术选型也是一种很重要的能力！

## 数据库锁

### 概念

锁是网络数据库中的一个非常重要的概念，当多个用户同时对数据库并发操作时，会带来数据不一致的问题，所以，锁主要用于多用户环境下保证数据库完整性和一致性。

### 目的

处理并发问题

### 技术手段

乐观锁、悲观锁和时间戳。

### 锁分类

从数据库系统角度分为三种：排他锁、共享锁、更新锁。

从程序员角度分为两种：一种是悲观锁，一种乐观锁。

### 悲观锁（Pessimistic Lock）

#### 概念

顾名思义，很悲观，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，这样别人拿这个数据就会block（阻塞），直到它拿锁。

正如其名，具有强烈的独占和排他特性。它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

传统的关系数据库里用到了很多这种锁机制，比如行锁、表锁、读锁、写锁等，都是在操作之前先上锁。

#### 性质分类

##### 共享锁（Share Lock）

S锁，也叫读锁，用于所有的只读数据操作。共享锁是非独占的，允许多个并发事务读取其锁定的资源。

特点

* 多个事务可封锁同一个共享页；
* 任何事务都不能修改该页；
* 通常是该页被读取完毕，S锁立即被释放。

##### 排他锁（Exclusive Lock）

X锁，也叫写锁，表示对数据进行写操作。如果一个事务对对象加了排他锁，其他事务就不能再给它加任何锁了。（某个顾客把试衣间从里面反锁了，其他顾客想要使用这个试衣间，就只有等待锁从里面打开了。）

特点

* 仅允许一个事务封锁此页；
* 其他任何事务必须等到X锁被释放才能对该页进行访问；
* X锁一直到事务结束才能被释放。

产生排他锁的SQL语句如下：select \* from ad\_plan for update;

##### 更新锁

U锁，在修改操作的初始化阶段用来锁定可能要被修改的资源，这样可以避免使用共享锁造成的死锁现象。

因为当使用共享锁时，修改数据的操作分为两步：

1. 首先获得一个共享锁，读取数据，

2. 然后将共享锁升级为排他锁，再执行修改操作。

这样如果有两个或多个事务同时对一个事务申请了共享锁，在修改数据时，这些事务都要将共享锁升级为排他锁。这时，这些事务都不会释放共享锁，而是一直等待对方释放，这样就造成了死锁。

如果一个数据在修改前直接申请更新锁，在数据修改时再升级为排他锁，就可以避免死锁。

特点

1. 用来预定要对此页施加X锁，它允许其他事务读，但不允许再施加U锁或X锁；

2. 当被读取的页要被更新时，则升级为X锁；

3. U锁一直到事务结束时才能被释放。

#### 范围分类

##### 行锁

行锁，锁的作用范围是行级别。

##### 表锁

锁的作用范围是整张表。

##### 锁的选择

数据库能够确定那些行需要锁的情况下使用行锁，如果不知道会影响哪些行的时候就会使用表锁。

举个例子，一个用户表user，有主键id和用户生日birthday。 当你使用update … where id=?这样的语句时，数据库明确知道会影响哪一行，它就会使用行锁；

当你使用update … where birthday=?这样的的语句时，因为事先不知道会影响哪些行就可能会使用表锁。

### 乐观锁（Optimistic Lock）

#### 概念

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受，而乐观锁是在应用程序实现的锁机制，在一定程度上解决了这个问题。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

#### 实现方式

##### 版本号（version）

版本号（记为version）：就是给数据增加一个版本标识，在数据库上就是表中增加一个version字段，每次更新把这个字段加1，读取数据的时候把version读出来，更新的时候比较version，如果还是开始读取的version就可以更新了，如果现在的version比老的version大，说明有其他事务更新了该数据，并增加了版本号，这时候得到一个无法更新的通知，用户自行根据这个通知来决定怎么处理，比如重新开始一遍。这里的关键是判断version和更新两个动作需要作为一个原子单元执行，否则在你判断可以更新以后正式更新之前有别的事务修改了version，这个时候你再去更新就可能会覆盖前一个事务做的更新，造成第二类丢失更新，所以你可以使用update … where … and version=”old version”这样的语句，根据返回结果是0还是非0来得到通知，如果是0说明更新没有成功，因为version被改了，如果返回非0说明更新成功。

##### 时间戳

时间戳（timestamp）：和版本号基本一样，只是通过时间戳来判断而已，注意时间戳要使用数据库服务器的时间戳不能是业务系统的时间。

##### 待更新字段

待更新字段：和版本号方式相似，只是不增加额外字段，直接使用有效数据字段做版本控制信息，因为有时候我们可能无法改变旧系统的数据库表结构。假设有个待更新字段叫count,先去读取这个count,更新的时候去比较数据库中count的值是不是我期望的值（即开始读的值），如果是就把我修改的count的值更新到该字段，否则更新失败。java的基本类型的原子类型对象如AtomicInteger就是这种思想。

##### 几种方式的区别

新系统设计可以使用version方式和timestamp方式，需要增加字段，应用范围是整条数据，不论那个字段修改都会更新version,也就是说两个事务更新同一条记录的两个不相关字段也是互斥的，不能同步进行。旧系统不能修改数据库表结构的时候使用数据字段作为版本控制信息，不需要新增字段，待更新字段方式只要其他事务修改的字段和当前事务修改的字段没有重叠就可以同步进行，并发性更高。

### 锁机制产生的问题

并发控制会造成活锁和死锁，就像操作系统那样，会因为互相等待而导致。

#### 活锁

##### 定义

活锁的情况：如果事务T1封锁了数据R，事务T2又请求封锁R，于是T2等待；T3也请求封锁R，当T1释放了R上的封锁之后系统首先批准了T3的请求，T2仍然等待；然后T4又请求封锁R，当T3释放了R上的封锁之后系统又批准了T4的请求…T2有可能永远等待。

##### 解决方法

避免活锁的简单方法是采用先来先服务的策略。当多个事务请求封锁同一数据对象时，封锁子系统按请求封锁的先后次序对事务排队，数据对象上的锁一旦释放就批准申请队列中第一个事务获得所。

#### 死锁

##### 定义

死锁的情形：如果事务T1封锁了数据R1，T2封锁了数据R2，然后T1又请求封锁R2，因T2已封锁了R1，T2也只能等待T1释放R1上的锁。这样就出现了T1在等待T2，而T2又在等待T1的局面，T1和T2两个事务永远不能结束，死锁这种没有完全解决的方法，只能尽量预防。

##### 预防方法

采取一定措施来预防死锁的发生。在数据库中，产生死锁的原因是两个或多个事务都已封锁了一些数据对象，然后又都请求对已被其他事务封锁的数据对象加锁，从而出现死等待。防止死锁的发生其实就是要破坏产生死锁的条件

1. 一次封锁法：要求每个事务必须将所有要使用的数据全部加锁，否则就不能继续执行。这种方法虽然可以有效地预防死锁的方式，但是一次就将以后要用到的全部数据加锁，势必扩大了封锁的范围，从而降低了系统的并发度，而且，数据库是不断变化的，原来不要求封锁的数据在执行过程中可能会变成封锁对象，所以很难事先精确地确定每个事务所要封锁的数据对象，为此只能扩大封锁范围，将事务在执行过程中可能要鞥说的数据对象全部加锁，这就进一步降低了并发度。
2. 顺序封锁法：顺序封锁法是预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实施封锁。例如在B树结构的索引中，可规定封锁的顺序必须是从根节点开始，然后是下一级的子节点，逐级封锁。顺序封锁法可以有效地防止死锁，但是数据库系统中封锁的数据对象极多，并且随数据的插入、删除等操作而不断变化，要维护这样的组员的封锁顺序非常困难，成本很高，而且事务的封锁请求可以随着事务的执行而动态地决定，很难实现确定每一个事务要封锁哪些对象，因此也就很难规定的顺序去施加封锁。

##### 死锁的诊断与解除

超时法：如果某个事物的等待时间超过指定时限，则判定为出现死锁；

等待图法：如果事务等待图中出现了回路，则判断出现了死锁。

对于解决死锁的方法，只能是撤销一个处理死锁代价最小的事务，释放此事务持有的所有锁，同时对撤销的事务所执行的数据修改操作必须加以恢复。

## 数据库设计

### 三范式

范式（Database Normalization）

设计关系[数据库](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)时，遵从不同的规范[要求](https://baike.baidu.com/item/%E8%A6%81%E6%B1%82/3598753" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)，设计出合理的关系型数据库，这些不同的规范要求被称为不同的范式，各种范式呈递次规范，越高的范式数据库冗余越小。

#### 第一范式（1NF）

数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。即实体中的某个属性有多个值时，必须拆分为不同的属性



应该设计为



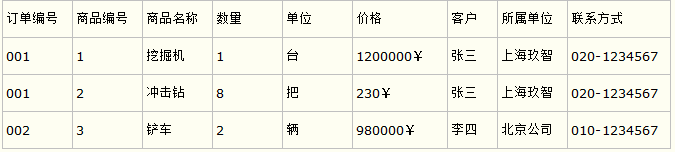
#### 第二范式（2NF）

表中每列与主键相关，而不能只与主键的某部分相关（主要针对联合主键），主键列与非主键列遵循完全函数依赖关系，也就是一个表只描述一件事情。

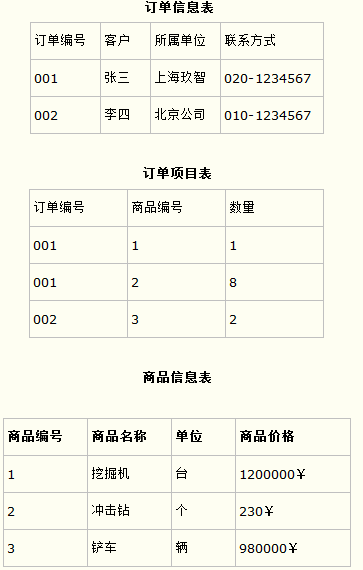
例子引入

需求描述：设计一个订单信息表，订单有多种商品，将订单编号和商品编号作为联合主键。

反范式设计



范式设计



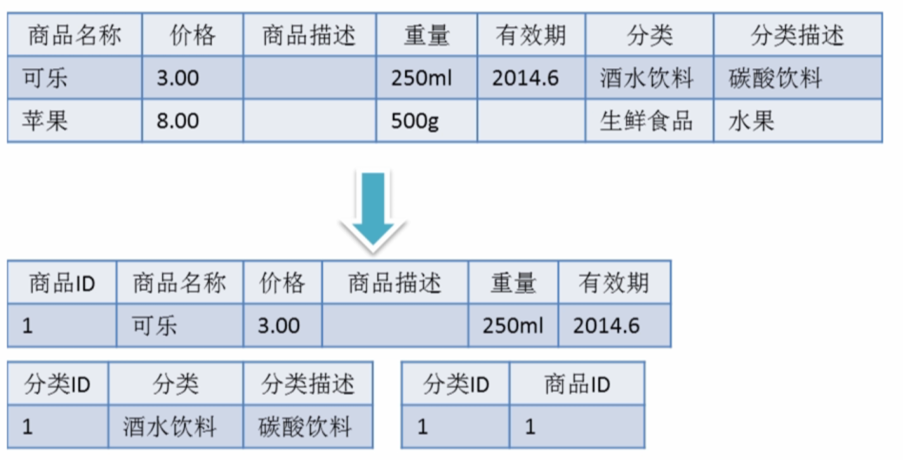
分析

第一种表设计不满足第二范式 ，订单编号和商品编号作为联合主键，由于商品名称，单位，价格这几列只与商品编号有关，与订单编号无关，因此与主键（联合主键）无关，违反范式第二原则；

第二种表设计满足第二范式，把第一种设计表进行拆分，把商品信息分离到另一个表中，把订单项目表也分离到另一个表中。

#### 第三范式（3NF）

在2NF基础上，任何非主[属性](https://baike.baidu.com/item/%E5%B1%9E%E6%80%A7" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)不依赖于其它非主属性（在2NF基础上消除传递依赖），简而言之，要求一个数据库表中不包含已在其它表中已包含的非主键字段。表的信息，如果能够被推导出来，就不应该单独的设计一个字段来存放(能尽量外键join就用外键join)。很多时候，我们为了满足第三范式往往会把一张表分成多张表



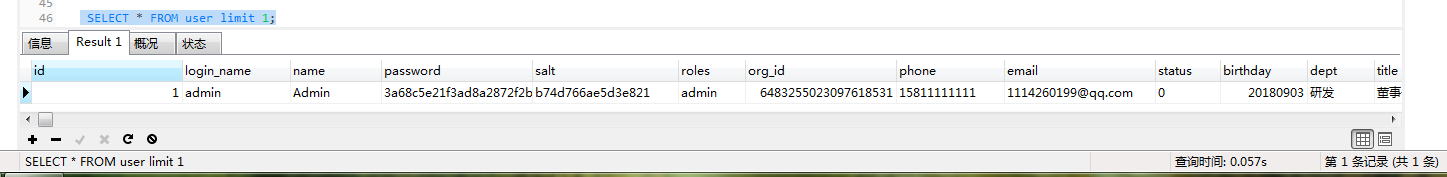
### 反三范式

查询记录时，范式模式往往要进行多表连接，而反范式只需在同一张表中查询，当数据量很大的时候，适当保留冗余数据，显然反范式的效率会更好降低范式就是增加字段，减少了查询时的关联，提高查询效率。反范式有很多重复的数据，会占用更多的内存，查询时可能会较多地使用DROUP BY或DISTINCT等耗时耗性能的关键字。当要修改更新数据时，范式更灵活，而反范式要修改全部的数据，且易出错。

## Mysql优化

目的减少查询时间

### Sql执行时间

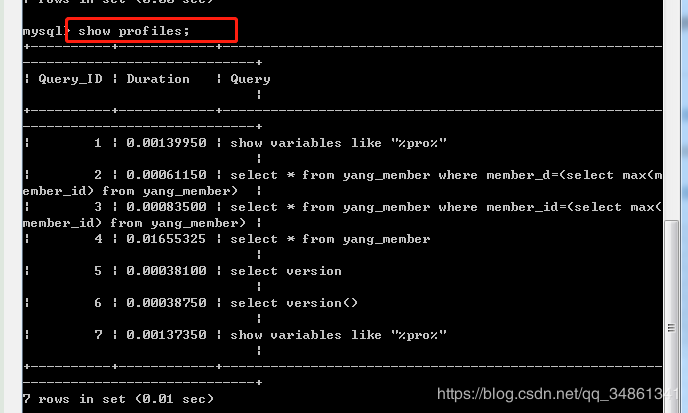


显示的查询时间包括了sql执行时间、io网络传输时间等，一般优化主要看这个时间长短就可以了，如果要知道具体的sql执行时间，数据传输因网路快慢不同，要知道具体的sql执行时间则需要使用show profile，查看是否支持profile和是否开启，数据库默认是不开启的。变量profiling是用户变量，每次都得重新启用。

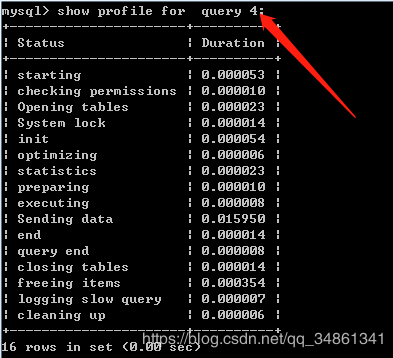
查看方法： show variables like "%pro%";

设置开启方法： set profiling = 1;

可以开始执行一些想要分析的sql语句了，执行完后，show profiles；即可查看所有sql的总的执行时间。



show profile for query 4 即可查看第1个sql语句的执行的各个操作的耗时详情。（注意 4 就是上面的query\_ID）



show profile cpu, block io, memory,swaps,context switches,source for query 6;可以查看出一条SQL语句执行的各种资源消耗情况，比如CPU，IO等， show profile all for query 6 查看第6条语句的所有的执行信息。

### 优化方向

1. 表的设计合理化(符合3NF)
2. 添加适当索引(index) [四种: 普通索引、主键索引、唯一索引unique、全文索引]
3. SQL语句优化
4. 分表技术(水平分割、垂直分割)
5. 读写[写: update/delete/add]分离
6. 存储过程 [模块化编程，可以提高速度]
7. 对mysql配置优化 [配置最大并发数my.ini, 调整缓存大小 ]
8. mysql服务器硬件升级
9. 定时的去清除不需要的数据,定时进行碎片整理(MyISAM)

### Sql语句优化规则

1、表连接数

* 连接的表越多，性能越差
* 可能的话，将连接拆分成若干个过程逐一执行
* 优先执行可显著减少数据量的连接，既降低了复杂度，也能够容易按照预期执行
* 如果不可避免多表连接，很可能是设计缺陷
* 外链接效果差，因为必须对左右表进行表扫描
* 尽量使用inner join查询

2、使用临时表

如果不可避免，可以考虑使用临时表或表变量存放中间结果。

3、少用子查询

4、视图嵌套

不要过深,一般视图嵌套不要超过2个为宜。

5、NULL列

Null列使用索引没有意义，任何包含null值的列都不会被包含在索引中。因此where语句中的is null或is not null的语句优化器是不允许使用索引的

6、concat或||

concat或||是mysql和oracle的字符串连接操作，如果对列进行该函数操作，那么也开会忽略索引的使用。比较下面的查询语句：

-- 忽律索引select ... from .. where first\_name || '' || last\_name = 'bill gates' ;

-- 使用索引select ... from .. where first\_name = 'bill' and last\_name = 'bill gates' ;

7、like

通配符出现在首位，无法使用索引，反之可以。

-- 无法使用索引select .. from .. where name like '%t%' ;

-- 可以使用索引select .. from .. where name like 't%' ;

8、order by

order by子句中不要使用非索引列或嵌套表达式，这样都会导致性能降低。

9、Not运算

not运算无法使用索引，可以改成其他能够使用索引的操作。如下：

-- 索引无效select .. from .. where sal != 3000 ;

-- 索引生效select .. from .. where sal < 3000 or sal > 3000;

10、where与having

select .. from .. on .. where .. group by .. having .. order by .. limit ..，以上是sql语句的语法结构，其中on、where和having是有过滤行为的，过滤行为越能提前完成就越可以减少传递给下一个阶段的数据量，因此如果在having中的过滤行为能够在where中完成，则应该优先考虑where来实现。

11、exists替代in

not in是最低效的，因为要对子查询的表进行全表扫描。可以考虑使用外链接或not exists。如下：

-- 正确SELECT \*FROM EMP WHERE EMPNO > 0 AND EXISTS (SELECT ‘X' FROM DEPT WHERE DEPT.DEPTNO = EMP.DEPTNO AND LOC = ‘MELB')

-- 错误SELECT \* FROM EMP WHERE EMPNO > 0 AND DEPTNO IN(SELECT DEPTNO FROM DEPT WHERE LOC = ‘MELB')

12、>与>=

-- 直接定位到4的记录(推荐)select .. from .. where SAL >= 4 ;

-- 先定位到3，再向后找1个(不推荐)select .. from .. where SAL > 3 ;

13、union代替or

在索引列上，可以使用union替换or操作。索引列上的or操作会造成全表扫描。

-- 高效: SELECT LOC\_ID , LOC\_DESC , REGION FROM LOCATION WHERE LOC\_ID = 10 UNION SELECT LOC\_ID , LOC\_DESC , REGION FROM LOCATION WHERE REGION = 'MELBOURNE'

-- 低效: SELECT LOC\_ID ,LOC\_DESC ,REGION FROM LOCATION WHERE LOC\_ID=10 OR REGION ='MELBOURNE'

14、is null & is not null

如果列可空，避免使用索引。对于多个列使用的索引，起码保证至少有个列不为空。对于多列索引，只有访问了第一个列才会启用索引，如果访问后面的列则使用的是全表扫描。

-- 低效: (索引失效) SELECT .. FROM DEPARTMENT WHERE DEPT\_CODE IS NOT NULL;

-- 高效: (索引有效) SELECT .. FROM DEPARTMENT WHERE DEPT\_CODE >=0;

15、 union & union all

union具有去重的操作，增加了计算时间。union all不需要去重，但会包含相同记录。同样功能下，首选union all操作。

### 慢查询

MySQL默认10秒内没有响应SQL结果,则为慢查询，可以去修改MySQL慢查询默认时间，当Mysql性能下降时，通过开启慢查询来获得哪条SQL语句造成的响应过慢，进行分析处理。当然开启慢查询会带来CPU损耗与日志记录的IO开销，所以我们要间断性的打开慢查询日志来查看Mysql运行状态。慢查询能记录下所有执行超过long\_query\_time时间的SQL语句, 用于找到执行慢的SQL, 方便我们对这些SQL进行优化，从一个大项目中，迅速的定位执行速度慢的语句



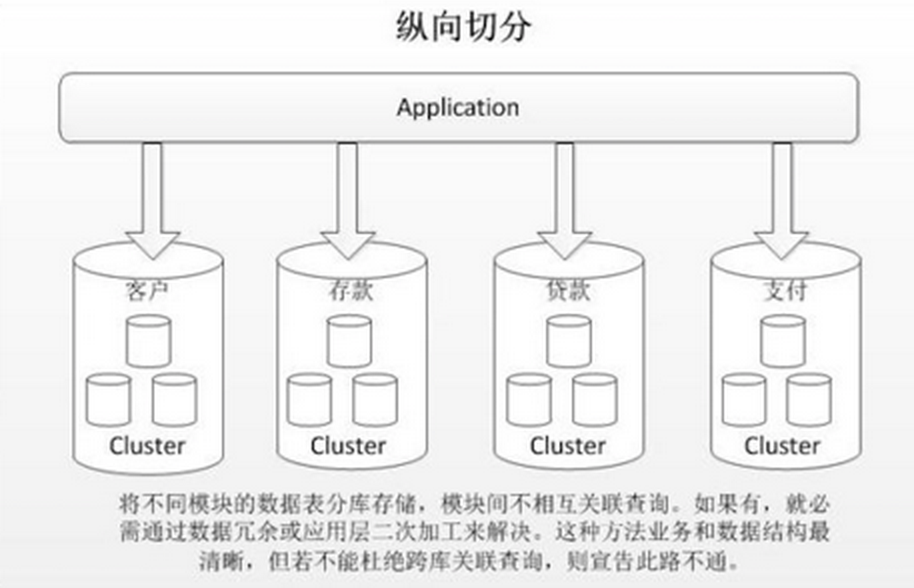
开启日志后定位到慢查询的语句，就可以用explain分析慢的原因一步一步解决问题。

### 分表分库

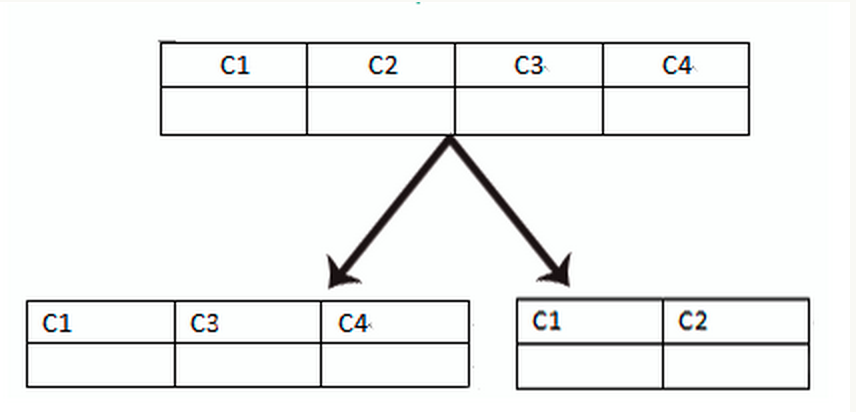
关系型数据库本身比较容易成为系统瓶颈，单机存储容量、连接数、处理能力都有限。当单表的数据量达到1000W或100G以后，由于查询维度较多，即使添加从库、优化索引，做很多操作时性能仍下降严重。此时就要考虑对其进行切分了，切分的目的就在于减少数据库的负担，缩短查询时间。数据库分布式核心内容无非就是数据切分（Sharding），以及切分后对数据的定位、整合。数据切分就是将数据分散存储到多个数据库中，使得单一数据库中的数据量变小，通过扩充主机的数量缓解单一数据库的性能问题，从而达到提升数据库操作性能的目的。数据切分根据其切分类型，可以分为两种方式：垂直（纵向）切分和水平（横向）切分。

#### 垂直（纵向）切分

垂直切分常见有垂直分库和垂直分表两种。垂直分库就是根据业务耦合性，将关联度低的不同表存储在不同的数据库。做法与大系统拆分为多个小系统类似，按业务分类进行独立划分。与"微服务治理"的做法相似，每个微服务使用单独的一个数据库。如图：



垂直分表是基于数据库中的"列"进行，某个表字段较多，可以新建一张扩展表，将不经常用或字段长度较大的字段拆分出去到扩展表中。在字段很多的情况下（例如一个大表有100多个字段），通过"大表拆小表"，更便于开发与维护，也能避免跨页问题，MySQL底层是通过数据页存储的，一条记录占用空间过大会导致跨页，造成额外的性能开销。另外数据库以行为单位将数据加载到内存中，这样表中字段长度较短且访问频率较高，内存能加载更多的数据，命中率更高，减少了磁盘IO，从而提升了数据库性能。



垂直切分的优点：

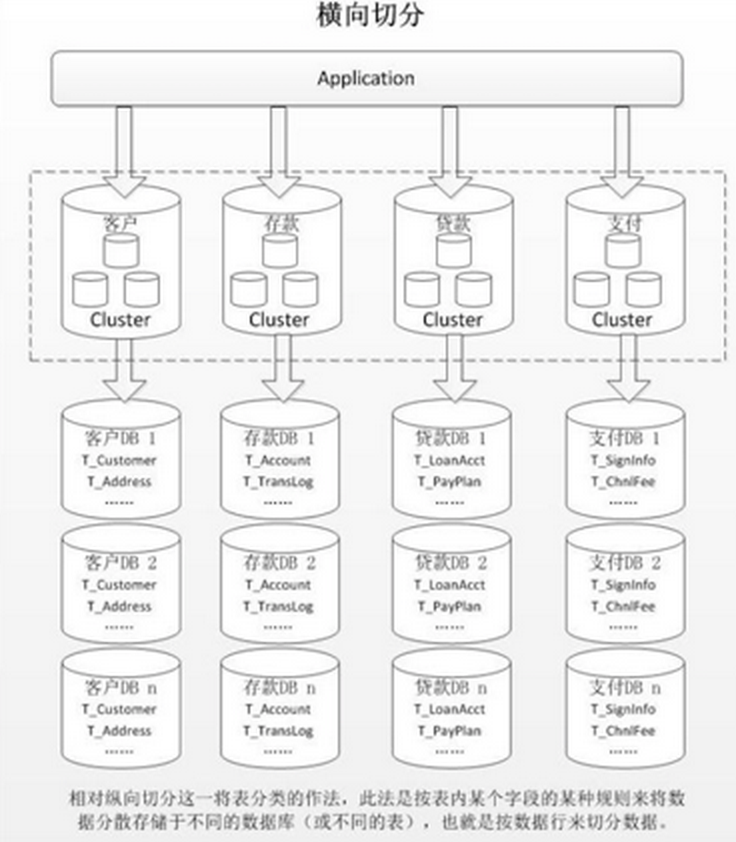
解决业务系统层面的耦合，业务清晰，与微服务的治理类似，也能对不同业务的数据进行分级管理、维护、监控、扩展等，高并发场景下，垂直切分一定程度的提升IO、数据库连接数、单机硬件资源的瓶颈

缺点：

部分表无法join，只能通过接口聚合方式解决，提升了开发的复杂度，分布式事务处理复杂，依然存在单表数据量过大的问题（需要水平切分）

#### 水平（横向）切分

当一个应用难以再细粒度的垂直切分，或切分后数据量行数巨大，存在单库读写、存储性能瓶颈，这时候就需要进行水平切分了。水平切分分为库内分表和分库分表，是根据表内数据内在的逻辑关系，将同一个表按不同的条件分散到多个数据库或多个表中，每个表中只包含一部分数据，从而使得单个表的数据量变小，达到分布式的效果。如图所示：



库内分表只解决了单一表数据量过大的问题，但没有将表分布到不同机器的库上，因此对于减轻MySQL数据库的压力来说，帮助不是很大，大家还是竞争同一个物理机的CPU、内存、网络IO，最好通过分库分表来解决。

水平切分的优点：

不存在单库数据量过大、高并发的性能瓶颈，提升系统稳定性和负载能力，应用端改造较小，不需要拆分业务模块

缺点：

跨分片的事务一致性难以保证，跨库的join关联查询性能较差，数据多次扩展难度和维护量极大。

水平切分后同一张表会出现在多个数据库/表中，每个库/表的内容不同。几种典型的数据分片规则为：

1、根据数值范围

按照时间区间或ID区间来切分。例如：按日期将不同月甚至是日的数据分散到不同的库中；将userId为1~9999的记录分到第一个库，10000~20000的分到第二个库，以此类推。某种意义上，某些系统中使用的"冷热数据分离"，将一些使用较少的历史数据迁移到其他库中，业务功能上只提供热点数据的查询，也是类似的实践。

这样的优点在于：

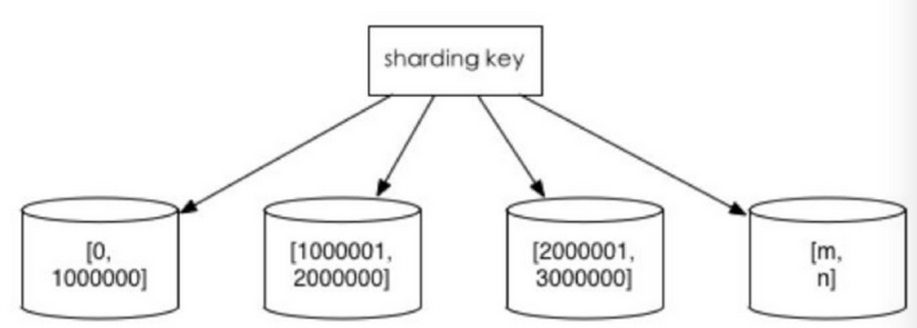
单表大小可控

天然便于水平扩展，后期如果想对整个分片集群扩容时，只需要添加节点即可，无需对其他分片的数据进行迁移

使用分片字段进行范围查找时，连续分片可快速定位分片进行快速查询，有效避免跨分片查询的问题。

缺点：

热点数据成为性能瓶颈。连续分片可能存在数据热点，例如按时间字段分片，有些分片存储最近时间段内的数据，可能会被频繁的读写，而有些分片存储的历史数据，则很少被查询



2、根据数值取模

一般采用hash取模mod的切分方式，例如：将 Customer 表根据 cusno 字段切分到4个库中，余数为0的放到第一个库，余数为1的放到第二个库，以此类推。这样同一个用户的数据会分散到同一个库中，如果查询条件带有cusno字段，则可明确定位到相应库去查询。

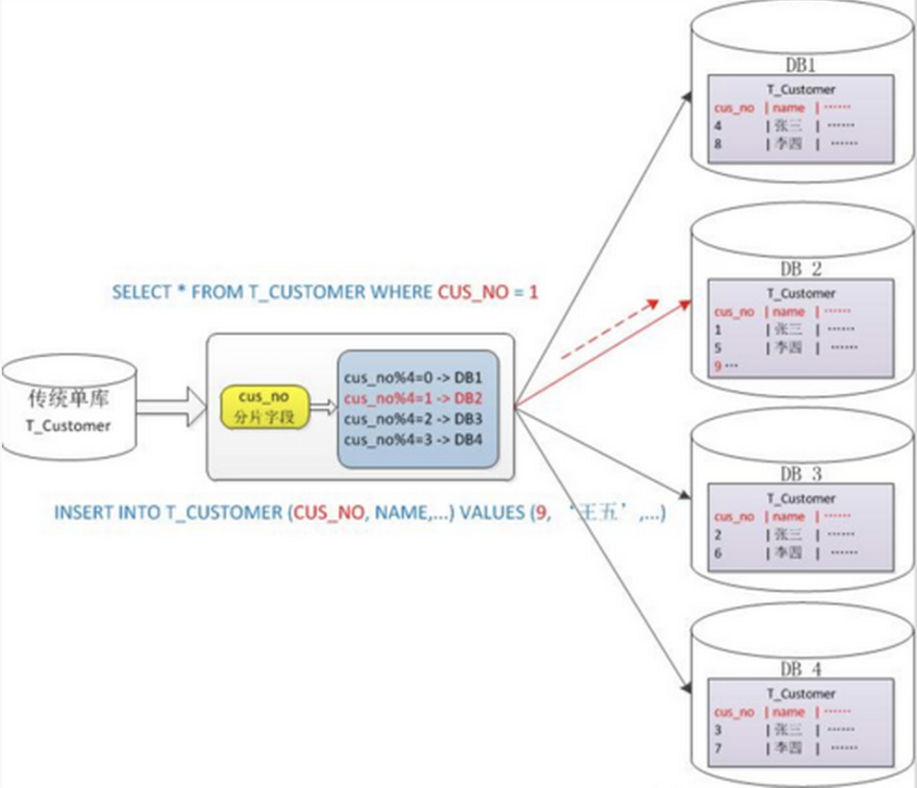
优点：

数据分片相对比较均匀，不容易出现热点和并发访问的瓶颈

缺点：

后期分片集群扩容时，需要迁移旧的数据（使用一致性hash算法能较好的避免这个问题）

容易面临跨分片查询的复杂问题。比如上例中，如果频繁用到的查询条件中不带cusno时，将会导致无法定位数据库，从而需要同时向4个库发起查询，再在内存中合并数据，取最小集返回给应用，分库反而成为拖累。



#### 分库分表后引入的问题

##### 分布式事务问题

分布式事务产生的原因之一就是分库分表

##### 跨库join问题

分库分表后表之间的关联操作将受到限制，我们无法join位于不同分库的表，也无法join分表粒度不同的表，结果原本一次查询能够完成的业务，可能需要多次查询才能完成。

解决方法

* 全局表：基础数据，所有库都拷贝一份。
* 字段冗余：这样有些字段就不用join去查询了。
* 系统层组装：分别查询出所有，然后组装起来，较复杂。

##### 横向扩容问题

当我们使用HASH取模做分表的时候,针对数据量的递增,可能需要动态的增加表,此时就需要考虑因为reHash导致数据迁移的问题。

##### 结果集合并、排序问题

因为我们是将数据分散存储到不同的库、表里的,当我们查询指定数据列表时,数据来源于不同的子库或者子表,就必然会引发结果集合并、排序的问题。如果每次查询都需要排序、合并等操作,性能肯定会受非常大的影响。走缓存可能一条路!

### 主从复制



#### 概念

影响MySQL-A数据库的操作，在数据库执行后，都会写入本地的日志系统A中。假设，实时的将变化了的日志系统中的数据库事件操作，在MYSQL-A的3306端口，通过网络发给MYSQL-B。MYSQL-B收到后，写入本地日志系统B，然后一条条的将数据库事件在数据库中完成。那么，MYSQL-A的变化，MYSQL-B也会变化，这样就是所谓的MYSQL的复制，即MYSQL replication。在上面的模型中，MYSQL-A就是主服务器，即master，MYSQL-B就是从服务器，即slave。日志系统A，其实它是MYSQL的日志类型中的二进制日志，也就是专门用来保存修改数据库表的所有动作，即bin log。【注意MYSQL会在执行语句之后，释放锁之前，写入二进制日志，确保事务安全】日志系统B，并不是二进制日志，由于它是从MYSQL-A的二进制日志复制过来的，并不是自己的数据库变化产生的，有点接力的感觉，称为中继日志，即relay log。可以发现，通过上面的机制，可以保证MYSQL-A和MYSQL-B的数据库数据一致，但是时间上肯定有延迟，即MYSQL-B的数据是滞后的。【即便不考虑什么网络的因素，MYSQL-A的数据库操作是可以并发的执行的，但是MYSQL-B只能从relay log中读一条，执行下。因此MYSQL-A的写操作很频繁，MYSQL-B很可能跟不上。】

#### 环境搭建

1. 准备环境

两台windows操作系统 ip分别为: 172.27.185.1(主)、172.27.185.2(从)

1. 连接到主服务(172.27.185.1)服务器上，给从节点分配账号权限。

GRANT REPLICATION SLAVE ON \*.\* TO 'root'@'172.27.185.2' IDENTIFIED BY 'root';

1. 在主服务my.ini文件新增

|  |
| --- |
| server-id=200  log-bin=mysql-bin  relay-log=relay-bin  relay-log-index=relay-bin-index |

重启mysql服务

1. 在从服务my.ini文件新增

|  |
| --- |
| server-id = 210  replicate-do-db =itmayiedu #需要同步数据库 |

重启mysql服务

1. 从服务同步主数据库

|  |
| --- |
| stop slave;  change  master to master\_host='172.27.185.1',master\_user='root',master\_password='root';  start slave;  show slave status; |

#### 读写分离

在数据库集群架构中，让主库负责处理事务性查询，而从库只负责处理select查询，让两者分工明确达到提高数据库整体读写性能。当然，主数据库另外一个功能就是负责将事务性查询导致的数据变更同步到从库中，也就是写操作。

好处

* 分摊服务器压力，提高机器的系统处理效率，读写分离适用于读远比写的场景，如果有一台服务器，当select很多时，update和delete会被这些select访问中的数据堵塞，等待select结束，并发性能并不高，而主从只负责各自的写和读，极大程度的缓解X锁和S锁争用。
* 增加冗余，提高服务高可用，当一台数据库服务器宕(dang)机(死机)后可以调整另外一台从库以最快速度恢复服务，这是从库变为主库，类似集群一台宕机了还有其他服务器，这就是高可用。

### 中间件

MyCat

是一个开源的分布式数据库系统，但是因为数据库一般都有自己的数据库引擎，而Mycat并没有属于自己的独有数据库引擎，所有严格意义上说并不能算是一个完整的数据库系统，只能说是一个在应用和数据库之间起桥梁作用的中间件。在Mycat中间件出现之前，MySQL主从复制集群，如果要实现读写分离，一般是在程序段实现，这样就带来了一个问题，即数据段和程序的耦合度太高，如果数据库的地址发生了改变，那么我的程序也要进行相应的修改，如果数据库不小心挂掉了，则同时也意味着程序的不可用，而对于很多应用来说，并不能接受；引入Mycat中间件能很好地对程序和数据库进行解耦，这样，程序只需关注数据库中间件的地址，而无需知晓底层数据库是如何提供服务的，大量的通用数据聚合、事务、数据源切换等工作都由中间件来处理；Mycat中间件的原理是对数据进行分片处理，从原有的一个库，被切分为多个分片数据库，所有的分片数据库集群构成完成的数据库存储，有点类似磁盘阵列中的RAID0.处理主从数据库分发类似于集群的nginx的负载均衡功能。Sharding-JDBC当当网开发的简单易用、轻量级的中间件。此外还有淘宝的TDDL,支付宝的OneProxy,360的Atlas等。

安装使用

Todo

## 数据库连接

## Redis