# 数据库通用问题

数据库的关系有哪些？

一对一：例如学校名称和学校地址

一对多：例如班级和班内同学

多对多：例如学生和学生的选修课

数据库关系怎么实现？

一对一：

唯一外键（在A表和B表中任意一个表，添加对方的主键／唯一键作为外键）；

主键关联（A和B使用相同的主键）；

中间关系表（创建新的表作为中间关系表，分别记录列A和列B，并且每一列的取值中，值均不能重复）

一对多：

在多的一方添加外键（在学生表添加班级信息列，取值为班级表的主键）；

建立中间关系表（创建新的表，至少有两列，分别记录A列和B列。多的一方要设为外键＋唯一键。例如学生作为唯一外键，班级列可以重复）

多对多：

中间关系表（只有这一种方式。建立新表，一列记录学生，一列记录课程，两列均为外键。学生列取值为学生的唯一标识列，课程取值为课程的唯一标识列。虽然有各自有单行重复，但是组和唯一。）

# 写sql

group by和having

左连接（leetcode P183）

# redis

redis的基础数据类型和具体应用场景

redis的set求交集的时间复杂度

缓存雪崩和缓存击穿

降级策略？（不确定是不是redis的知识点）

zset的底层结构

哈希表

happen-before规则

# Ｍysql

## 数据类型

### 有哪些

数值类型

Mysql支持所有标准SQL数值数据类型

严格数据类型

INT：TINYINT(1 byte);SMALLINT(2 byte);MEDIUMINT(3 byte);INT或INTEGER(4 Byte);BIGINT(8 byte)

【注】使用int(10)，表示显示宽度，对数据的存取范围没有约束力

近似数据类型

FLOAT(4 byte)；DOUBLE(8 byte)

存储精确数值：

DECIMAL(P,D)，p表示有效数字的精度，范围是1~65；d表示小数点后的位数，0~30；要求d<=p

日期和时间

DATE：3 byte；1000-01-01~9999-12-31，YYYY-MM-DD格式的日期值

TIME：3 byte；HH:MM:SS 格式的时间值或者持续时间

YEAR：1 byte；YYYY格式的年份值

DATETIME：8 byte；YYYY-MM-DD HH:MM:SS 格式的混合时间和日期

TIMESTAMP：4 byte；1970-01-01 00:00:00~2038年，用毫秒表示的时间戳

字符串

CHAR：定长字符串，0~255byte

VARCHAR：变长字符串，0~65535字节

BLOB：二进制形式的长文本数据；可以用来存储图像

TINYBLOB(0~255byte);BLOB(0~64KB);MEDIUMBLOB(0~16MB);LONGBLOB(0~4GB)

TEXT：字符形式的长文本数据

TINYTEXT(0~255byte);TEXT(0~64KB); MEDIUMTEXT(0~16MB);LONGTEXT (0~4GB)

### 有什么区别

char(n)和varchar(n)：

都用来表示字符串；n都表示多少个字符（mysql4之前表示多少字节）

char定长（0~255字节）；

varchar变长（0~65535字节），同时varchar还需要额外的空间用于记录实际长度，长度为0~255的时候需要一个字节用来记录，超过255需要两个，此外还需要一个字节表示接受，因此varchar的实际最大可用于存储字符串的长度只有65532字节；

输入长度小于n时：char右侧补齐空格，在查询时去掉空格；varchar不需要补齐，查询的时候也不需要去掉空格

输入右侧有空格时：char先自动修剪/填充到n，查询时去掉右侧全部空格；varchar把空格当作正常字符操作

输入长度大于n时：char在严格模式下会报错，非严格模式会截断；varchar会阶段并且生成警告

适用条件：定长或者长度变化很小的字符串用char（比如记录身份证号），特别短的也用char（varchar记录长度会浪费1byte空间）。

性能：char的性能比varchar高

建立索引：char可以建立索引；varchar建立索引时长度不能超过768字节，否则只在索引中保留前768字节

blob和text：

blob是二进制，可以支持文本与非文本形式的内容，没有字符集的概念（因此大写和小写的二进制表示不同，排序也就不同）

text是文本文件（无视大小写）

## 索引

### 底层结构

【官方定义】索引（Index）是帮助MySQL高效获取数据的数据结构

索引是排好序的快速查找的数据结构，这些数据结构一某种方式引用（指向）数据，这样就可以在这些数据结构的基础上实现高级查找算法，这种数据结构就是索引

一是排序（影响order by）

二是查找（影响where）

### 【4】B+树和B树的区别，为什么用B+树，为什么不用红黑树

B+树和B树

B树的非叶节点既可以存放节点，也可以存放数据

B+树的非叶节点全部存储索引，非叶节点全部存储数据，因此查询时都是从根节点到叶节点，性能稳定；此外叶节点还存储了指向下一个叶节点的指针，构成了一个叶节点的链表，有利于进行区间查找

B+树和红黑树

红黑树的出度为最大为2，因此树高容易变得很高，影响查找效率

### 让你设计数据库的索引你会注意什么

一张表上最多建立16个索引

聚簇索引设置在主键上，没有主键使用其他唯一键，没有唯一键InnoDB会自动生成一个隐式主键用于创建聚簇索引

索引的选择性：索引列中不同值的数目与总行数的比值。越接近1代表选择性越好，越适合作为索引

列的读取需求：对需要频繁读取的列创建索引，如果有较为固定的多个列，可以考虑使用覆盖索引，减少回表

列的组合顺序：当需要多个列进行组和时，把选择性好的列放在前面，选择性差的放在后面；如果单个索引列有重复，可以考虑使用组合唯一的多个列作为索引

列的数据类型和长度：blob和text不适合作为索引；varchar长度超过768字节后，使用前768字节创建索引，后面的部分不会放入索引内（相当于局部数据作为了索引，此时覆盖索引的效果没有实现，也有可能因此导致了索引选择性的降低）；char在使用utf8mb4字符集后也可能出现长度超过768字节的情况，此时会先转成变长字符串

列的更新：列的更新可能会导致索引重建（尤其时随机变化），因此要避免在频繁更新的列上创建索引

【一般性建议】：

对单值索引，尽量选择针对当前查询过滤性更好的索引

在选择组合索引的时候，当前查询中过滤性最好的字段在索引字段顺序中，位置越靠前越好

在选择组合索引的时候，尽量选择可以能够包含当前查询中where子句中更多的字段

### 【2】为什么不对每个列创建索引

有的列读取的需求较少：只有频繁进行查找排序的列才有创建索引的需要

有的列索引选择性差：主要发生在一对多或者多对多的情况下，单列可能重复性很高

有的列频繁更新：频繁更新的列不适合创建索引，因为会导致索引的频繁重建

一张表上索引的个数有限制：16个

### 聚簇索引和非聚簇索引的区别

聚簇索引（也叫做聚集索引，还叫做主键索引，目的是为了找到目标列，并且获取该列的全部数据（除了过长的变长字符串，InnoDB把它们放在页外））

聚簇索引一张表只有一个，体积超过表体积本身（120%左右）。当表存在主键时使用主键创建聚簇索引，主键不存在时选择一个不为空的唯一列作为索引，如果还不存在，会生成一个六个字节长度的隐藏主键列（用户不可访问），在上面建立聚簇索引。

通过聚簇索引可以直接访问到目标列的全部数据

非聚簇索引叶节点中存放是主键的值，查找到主键值后，需要根据主键值再去聚簇索引中再查找一次，以获得对应的其他列值。这个动作称为回表。

一张表上可以有多个非聚簇索引。非聚簇索引的体积一般比较小。允许使用非唯一的列建立非聚簇索引。

索引覆盖：

当所查找的列是构建非聚簇索引使用的列的子集时，列的数据已经完全包括在了索引内，此使索引覆盖了列的内容，称为覆盖索引。

当出现覆盖索引时，不需要通过回表来获取数据，只需要进行一次查找，对效率有所提升。

如果出现了使用覆盖索引的情况，explain查看语句执行的时候，extra字段会出现using index

### 主键索引和二级索引

当主键存在时，mysql会默认在主键上创建聚集索引，这就是主键索引。如果主键不存在，那么也会使用别的列来创建聚集索引。

二级索引即是非聚集索引

### 回表

当使用非聚集索引（二级索引）查找时，在索引叶节点找到的内容为主键列的值，需要根据该值再去聚集索引中进行查找以获得列的完整结果。这个二次查找的动作称为回表。

### 联合索引

有别于单列索引，是指使用一个以上的列来创建索引。

当使用联合索引时，需要遵循最左前缀的原则。

## 引擎

### MyISAM和InnoDB的区别

最主要区别：

InnoDB支持事务，MyISAM不支持；

InnoDB支持行锁，MyISAM只支持表锁（因此InnoDB适合频繁写的情况）

其他区别：

InnoDB不仅缓存索引，还缓存真实数据；MyISAM只缓存索引

InnoDB支持外键，MyISAM不支持

InnoDB支持聚簇索引，所以必须要有主键（一定会保证存在一个非空唯一列）；MyISAM索引和数据分离，只支持非聚集索引

InnoDB不保存行数，count(\*)要全表扫描；MyISAM保留行数，可以直接读取

总结：

InnoDB更关注事务（并发度）

MyISAM更关注读取速度

## 事务

### 特性（ACID）

原子性(Atomicity)：事务是一个原子操作单元，其对事务的修改要么全部执行要么全部不执行

一致性(Consistency)：事务开始和完成时，数据都必须保持一致状态（数据只能从一个一致状态变到另一个一致状态，这里的一致状态是指符合应用语义的，比如银行转账时，转出方的余额不能低于转出金额），意味着所有相关的数据规则都必须应用于事务的修改，以保持数据的完整性；事务结束时，所有的内部数据结构（B+树索引或者双向链表）也都必须是正确的。

隔离性(Isolation)：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理的中间过程对外部不可见。反之亦然。

持久性(Durable)：事务完成后，它对数据的修改是永久性的，即使出现系统故障也会保持

### 为什么要有一致性

### AID不是已经保证一致性了吗

### 并发事务带来的问题

更新丢失：事务A读取数据并修改，但是在取数和修改之间，有事务B修改了数据，事务A再次修改后，数据B的修改被覆盖

脏读，即读到其他事务修改的数据，然后其他事务回滚，导致读到的数据实际不存在

不可重复读，事务A读取一条数据，事务B对该条数据修改并提交，事务A再次读取的时候会读取到新数据，和上次读取的结果不一致

幻读，即按范围检索一批数据，然后新的事务再范围中增加新数据，导致再次读取的时候出现了新数据

解决方案：使用锁机制

### 事务的隔离级别

读未提交：可能会出现“脏读”

读已提交：会出现“不可重复读”的问题

可重复度：会出现“幻读”的问题

串行化：解决了“幻读”的问题，但是效率低下

mysql默认的隔离级别是可重复读

网商银行OceanBase的隔离级别是可重复读

### MVCC机制

MVCC，多版本并发控制，是一种实现数据库并发访问的方式。

MVCC只在读已提交和可重复读两个隔离级别下工作。

通常使用锁来实现隔离性。最原始的悲观锁不适合读多写少的状态；后来引入了读写锁，用于支持读与读的并发；再进一步可以使用MVCC实现读与写的并发。

MVCC的核心思想是不同事务保有属于自己的“快照”，后续的读写操作基于快照进行，实现读写并发。

MVCC依赖的内容：redo log、undo log、bin log

此外，对于一张表，InnoDB维护了几个隐藏列。其中包括trx\_id（表示最近修改的事务id），db\_roll\_ptr（用于指向行之前的版本，构成undo log），删除标记位

MVCC的实现：

事务id：对于每一个事务，当事务中出现了增删改等写操作的时候，会给事务分配一个事务id，事务id是递增的，也即id越大，事务生成的越晚。

ReadView：同时，每一个事务生成的时候都会生成一个ReadView，ReadView的生命周期就是它所属的事务的生命周期，一旦生成，不再修改，在事务提交后销毁。每一个事务都有自己的ReadView。

ReadView是一个列表，存放了事务id，通过这个列表来判断哪一个版本的数据对当前事务是可见的。ReadView中的事务id有两类，一类是已提交的，一类是未提交的。并且id最小的事务一定是活跃的（未提交的）。当id最小的事务提交后（记为idA）， 再生成的新事务的新的ReadView就不含有这个idA，也就是说，新ReadView记录的最小事务id已经发生了变化。此时可能还存在一些未提交事务，他们的ReadView依然含有idA（因为ReadView是不可变的）。

ReadView用up\_limit\_id来表示它所持有的最小事务id，用low\_limit\_id表示他所持有的最大事务id，用一个trx\_ids来记录这个ReadView生成时还活跃的id（肯定介于up\_limit\_id和low\_limit\_id之间，并且up\_limit\_id对应的事务应该是活跃的）。

数据列的存储：InnoDB在对数据进行写（增删改）操作时，不是原地操作，而是生成一个新的行。新的行中记录了新的数据，同时记录了该操作所对应的事务id，还维护了一个指向上一个版本的指针。如果是删除操作，那么也并不会真正的删除数据，而是设置一个已删除的标记位。

MVCC的过程

假设当前有事务A，id=100；事务B，id=101；事务C，id=105这三个未提交事务，且事务C是最新的事务。

再新建一个事务进行查询（称为查询事务1），由于事务中没有出现写操作，查询事务1不会被分配事务id，但是它会在事务创建的时候生成一个ReadView。

ReadView中此时的up\_limit\_id = 100;low\_limit\_id = 105；trx\_idx = [100,101,105]；同时ReadView中还记录了103和104两个事务id，这两个作为已提交事务，但是之前还有比他们id更小的未提交事务排在前面。

此时查询事务1查询一条数据1，发现修改该数据的数据id = 98，小于当前ReadView中的up\_limit\_id，则该版本有效，直接获得该数据。

查询事务1接着查询数据2，发现修改该数据的id=103，存在于自己的ReadView中。查询ReadView知道这是一个已提交的事务，因此该版本也有效。

查询事务1接着查询数据3，发现修改该数据的id=100，ReadView中。查询ReadView知道这是一个未提交的事务，因此该版本无效。根据数据列中的指针寻找上一个版本，直到找到事务版本号低于100，或者是ReadView中的已提交事务为止。

此时事务D生成，id=106，修改数据2并提交。

查询事务1再次查询数据2，发现修改该数据的事务id变成了106，106超过了查询事务1的ReadView中的low\_limit\_id，此时该数据版本视为无效，寻找到上一个版本103，视为有效。（这里实现了可重复读）。

此时事务A提交。

查询事务1再次查询数据3，发现修改该数据的事务id为100。虽然id=100对应的事务A已提交，但是由于查询事务1的ReadView不会更新，因此依然视为无效，需要去寻找更早的版本。（这里实现了可重复读）

此时生成查询事务2，依然不分配事务id，但是生成ReadView2，ReadView2中不再含有100，但是含有id=106。因此查询事务2在查询数据3和数据4时，查询到的版本和查询事务1查到的版本不一致。

事务B修改数据2；

事务B查询数据2。事务B自身也持有一个ReadView，并且其中的low\_limit\_id就等于自身101。此时查询到数据2，发现修改版本的事务id存在于ReadView中并且存在于trx\_id中，是个未提交事务。但是由于和本身事务id相同，所以视为有效。

此时事务B删除数据1，并提交。

数据1因此复制出来一个新的版本，该版本的事务id=101，被标记为已删除。查询事务1和查询事务2在查询到此条内容时发现各自的ReadView中都含有id101的事务B，且未提交，因此去寻找之前的版本，所以依然可以找到数据1。

【注】如果工作在读已提交的隔离级别下，则每一次查询的开始都会更新/生成新的ReadView，使得已提交事务的版本生效。

MVCC总结

一个事务持有ReadView，记录了该ReadView生成时，所有尚未提交的事务id，其中最小的事务id标记为up\_limit\_id。同时记录了部分已提交的事务id，要求id必须大于up\_limit\_id。尚未提交的id又组成了称为trx\_ids的集合。ReadView中所有的事务id（含已提交和未提交的）最大的那个标记位low\_limit\_id。

查询时

如果修改数据的事务id小于up\_limit\_id，则视为有效；

如果修改数据的事务id大于low\_limit\_id，则视为无效，向前寻找

如果修改数据的事务id在ReadView中可以找到，并且已提交，则视为有效。

如果修改数据的事务id在ReadView中可以找到，并且未提交

是自己修改的，则视为有效

不是自己修改的，视为无效，向前寻找。

### ACID的实现原理

原子性：

redolog和binlog的两阶段提交

隔离性：

加锁

采用MVCC实现可重复读和读已提交

持久性：

redo log在事务提交前进行刷盘

一致性：

## 锁

### update时什么锁（范围：next-key-lock，唯一性索引时降级为行锁）

## 调优

### explain中rows、type、key、extra的含义

rows：可能需要扫描的行数

type：包括all, range, ref, eq\_ref, const, system几个不同的级别，表示了索引的使用情况

key：实际使用的索引

extra：包括了一些不适合放在其他字段中的信息。例如using index表示使用了覆盖索引；using filesort表示使用了外部文件排序，没有依赖已有索引，对性能有较大负面影响等等

### count(1),count(\*) 和 count(列值)的区别

### 大表Join小表优化

始终采用小表驱动大表的思路，用小的结果集去驱动大的结果集。例如

A LEFT JOIN B，此时A是主表，索引优先建在B上，使B称为小表，用来驱动大表

A RIGHT JOIN B，此时B是主表，索引优先建在A上

当出现子查询的时候

子查询数据集大于主表，使用exist

SELECT … FROM A WHERE EXIST (subquery)

子查询数据集小于主表

SELECT … FROM A WHERE list IN (subquery)