数据库面试知识点汇总

一、基本概念

1.主键、外键、超键、候选键

超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。
候选键：是最小超键，即没有冗余元素的超键。
主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。
外键：在一个表中存在的另一个表的主键称此表的外键。

2.为什么用自增列作为主键

如果我们定义了主键(PRIMARY KEY)，那么 InnoDB 会选择主键作为聚集索引、
如果没有显式定义主键，则 InnoDB 会选择第一个不包含有 NULL 值的唯一索引作为主键索引、
如果也没有这样的唯一索引，则 InnoDB 会选择内置 6 字节长的 ROWID 作为隐含的聚集索引(ROWID 随着行记录的写入而主键递增，这个 ROWID 不像 ORACLE 的 ROWID 那样可引用，是隐含的)。
数据记录本身被存于主索引（一颗 B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放，因此每当有一条新的记录插入时，MySQL 会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB 默认为 15/16），则开辟一个新的页（节点）
如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页
如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置，此时 MySQL 不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过 OPTIMIZE TABLE 来重建表并优化填充页面。

3.触发器的作用？

触发器是一种特殊的存储过程，主要是通过事件来触发而被执行的。它可以强化约束，来维护数据的完整性和一致性，可以跟踪数据库内的操作从而不允许未经许可的更新和变化。可以联级运算。如，某表上的触发器上包含对另一个表的数据操作，而该操作又会导致该表触发器被触发。

4.什么是存储过程？用什么来调用？

存储过程是一个预编译的 SQL 语句，优点是允许模块化的设计，就是说只需创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次 SQL，使用存储过程比单纯 SQL 语句执行要快。
调用：
1）可以用一个命令对象来调用存储过程。

2）可以供外部程序调用，比如：java 程序。

5.存储过程的优缺点？

优点：
1）存储过程是预编译过的，执行效率高。
2）存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯。
3）安全性高，执行存储过程需要有一定权限的用户。
4）存储过程可以重复使用，可减少数据库开发人员的工作量。
缺点：移植性差

6.存储过程与函数的区别

1）一般来说，存储过程实现的功能要复杂一点，而函数的实现的功能针对性比较强。
2）对于存储过程来说可以返回参数，而函数只能返回值或者表对象。
3）存储过程一般是作为一个独立的部分来执行，而函数可以作为查询语句的一个部分来调用，由于函数可以返回一个表对象，因此它可以在查询语句中位于 FROM 关键字的后面。
4）当存储过程和函数被执行的时候，SQL Manager 会到 procedure cache 中去取相应的查询语句，如果在 procedure cache 里没有相应的查询语句，SQL Manager 就会对存储过程和函数进行编译。
Procedure cache 中保存的是执行计划（execution plan），当编译好之后就执行 procedure cache 中的 execution plan,之后 SQL SERVER 会根据每个 execution plan 的实际情况来考虑是否要在 cache 中保存这个 plan，评判的标准一个是这个 execution plan 可能被使用的频率;其次是生成这个 plan 的代价，也就是编译的耗时。保存在 cache 中的 plan 在下次执行时就不用再编译了

7.什么叫视图？游标是什么？

视图：
是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图通常是有一个表或者多个表的行或列的子集。对视图的修改会影响基本表。它使得我们获取数据更容易，相比多表查询。
游标：
是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

8.视图的优缺点

优点：
1 对数据库的访问，因为视图可以有选择性的选取数据库里的一部分。
2)用户通过简单的查询可以从复杂查询中得到结果。
3)维护数据的独立性，试图可从多个表检索数据。
4)对于相同的数据可产生不同的视图。
缺点：

性能：查询视图时，必须把视图的查询转化成对基本表的查询，如果这个视图是由一个复杂的多表查询所定义，那么，那么就无法更改数据

9.drop、truncate、 delete 区别

最基本：

- drop 直接删掉表。
- truncate 删除表中数据，再插入时自增长 id 又从 1 开始。
- delete 删除表中数据，可以加 where 字句。

（1） DELETE 语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

（2） 表和索引所占空间。当表被 TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而 DELETE 操作不会减少表或索引所占用的空间。drop 语句将表所占用的空间全释放掉。

（3） 一般而言，drop > truncate > delete

（4） 应用范围。TRUNCATE 只能对 TABLE；DELETE 可以是 table 和 view

（5） TRUNCATE 和 DELETE 只删除数据，而 DROP 则删除整个表（结构和数据）。

（6） truncate 与不带 where 的 delete ：只删除数据，而不删除表的结构（定义）drop 语句将删除表的结构被依赖的约束（constrain),触发器（trigger)索引（index);依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

（7） delete 语句为 DML（data maintain Language),这个操作会被放到 rollback segment 中,事务提交后才生效。如果有相应的 tigger,执行的时候将被触发。

（8） truncate、drop 是 DLL（data define language),操作立即生效，原数据不放到 rollback segment 中，不能回滚。

（9） 在没有备份情况下，谨慎使用 drop 与 truncate。要删除部分数据行采用 delete 且注意结合 where 来约束影响范围。回滚段要足够大。要删除表用 drop;若想保留表而将表中数据删除，如果于事务无关，用 truncate 即可实现。如果和事务有关，或老师想触发 trigger,还是用 delete。

（10） Truncate table 表名 速度快,而且效率高,因为:?truncate table 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据,并且只在事务日志中记录页的释放。

（11） TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。

（12） 对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

10.什么是临时表，临时表什么时候删除?

临时表可以手动删除：

```
DROP TEMPORARY TABLE IF EXISTS temp_tb;
```

临时表只在当前连接可见，当关闭连接时，MySQL 会自动删除表并释放所有空间。因此在不同的连接中可以创建同名的临时表，并且操作属于本连接的临时表。创建临时表的语法与创建表语法类似，不同之处是增加关键字 TEMPORARY，如：

```
CREATE TEMPORARY TABLE tmp_table (
NAME VARCHAR (10) NOT NULL,
time date NOT NULL
);
select * from tmp_table;
```

11.非关系型数据库和关系型数据库区别，优势比较?

非关系型数据库的优势：

- 性能：NOSQL 是基于键值对的，可以想象成表中的主键和值的对应关系，而且不需要经过 SQL 层的解析，所以性能非常高。
- 可扩展性：同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。

关系型数据库的优势：

- 复杂查询：可以用 SQL 语句方便的在一个表以及多个表之间做非常复杂的数据查询。
- 事务支持：使得对于安全性能很高的数据访问要求得以实现。

其他：

1.对于这两类数据库，对方的优势就是自己的弱势，反之亦然。

2.NOSQL 数据库慢慢开始具备 SQL 数据库的一些复杂查询功能，比如 MongoDB。

3.对于事务的支持也可以用一些系统级的原子操作来实现例如乐观锁之类的方法来曲线救国，比如 Redis set nx。

12.数据库范式，根据某个场景设计数据表?

- **第一范式**:(确保每列保持原子性)所有字段值都是不可分解的原子值。

第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。

第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到“地址”这个属性，本来直接将“地址”属性设计成一个数据库表的字段就

行。但是如果系统经常会访问“地址”属性中的“城市”部分，那么就非要将“地址”这个属性重新拆分为省份、城市、详细地址等多个部分进行存储，这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式，如下表所示。
上表所示的用户信息遵循了第一范式的要求，这样在对用户使用城市进行分类的时候就非常方便，也提高了数据库的性能。

- **第二范式:**(确保表中的每列都和主键相关)在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。
比如要设计一个订单信息表，因为订单中可能会有多种商品，所以要将订单编号和商品编号作为数据库表的联合主键。

- **第三范式:**(确保每列都和主键列直接相关,而不是间接相关) 数据表中的每一列数据都和主键直接相关，而不能间接相关。

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。
BCNF:符合 3NF，并且，主属性不依赖于主属性。
若关系模式属于第二范式，且每个属性都不传递依赖于键码，则 R 属于 BC 范式。通常 BC 范式的条件有多种等价的表述：每个非平凡依赖的左边必须包含键码；每个决定因素必须包含键码。
BC 范式既检查非主属性，又检查主属性。当只检查非主属性时，就成了第三范式。满足 BC 范式的关系都必然满足第三范式。
还可以这么说：若一个关系达到了第三范式，并且它只有一个候选码，或者它的每个候选码都是单属性，则该关系自然达到 BC 范式。
一般，一个数据库设计符合 3NF 或 BCNF 就可以了。

- **第四范式:**要求把同一表内的多对多关系删除。
- **第五范式:**从最终结构重新建立原始结构。

13.什么是 内连接、外连接、交叉连接、笛卡尔积等?

- **内连接**: 只连接匹配的行
- **左外连接**: 包含左边表的全部行（不管右边的表中是否存在与它们匹配的行），以及右边表中全部匹配的行
- **右外连接**: 包含右边表的全部行（不管左边的表中是否存在与它们匹配的行），以及左边表中全部匹配的行

  例如 1：

  ```
  SELECT a.,b. FROM luntan LEFT JOIN usertable as b ON
  a.username=b.username
  ```

  例如 2：

  ```
  SELECT a.,b. FROM city as a FULL OUTER JOIN user as b ON
  a.username=b.username
  ```

- **全外连接**：包含左、右两个表的全部行，不管另外一边的表中是否存在与它们匹配的行。
- **交叉连接**：生成笛卡尔积－它不使用任何匹配或者选取条件，而是直接将一个数据源中的每个行与另一个数据源的每个行都一一匹配
  例如：
  ```
  SELECT type,pub_name FROM titles CROSS JOIN publishers ORDER BY type
  ```

注意：

很多公司都只是考察是否知道其概念，但是也有很多公司需要不仅仅知道概念，还需要动手写 sql,一般都是简单的连接查询，具体关于连接查询的 sql 练习，参见以下链接：

牛客网数据库 SQL 实战

leetcode 中文网站数据库练习

我的另一篇文章，常用 sql 练习 50 题

14.varchar 和 char 的使用场景?

- **char** 的长度是不可变的，而 **varchar** 的长度是可变的。

定义一个 **char[10]**和 **varchar[10]**。

如果存进去的是**‘csdn’**,那么 **char** 所占的长度依然为 10,除了字符**‘csdn’** 外，后面跟六个空格，**varchar** 就立马把长度变为 4 了，取数据的时候，char 类型的要用 **trim()**去掉多余的空格，而 **varchar** 是不需要的。

- **char** 的存取数度还是要比 **varchar** 要快得多，因为其长度固定，方便程序的存储与查找。**char** 也为此付出的是空间的代价，因为其长度固定，所以难免会有多余的空格占位符占据空间，可谓是以空间换取时间效率。
  varchar 是以空间效率为首位。
- **char** 的存储方式是：对英文字符（**ASCII**）占用 1 个字节，对一个汉字占用两个字节。
  varchar 的存储方式是：对每个英文字符占用 2 个字节，汉字也占用 2 个字节。
- 两者的存储数据都非 **unicode** 的字符数据。

15.SQL 语言分类

SQL 语言共分为四大类：

- 数据查询语言 DQL
- 数据操纵语言 DML
- 数据定义语言 DDL
- 数据控制语言 DCL。

（1）数据查询语言 DQL

数据查询语言 DQL 基本结构是由 SELECT 子句，FROM 子句，WHERE 子句组成的查询块：

SELECT

FROM

WHERE

（2）数据操纵语言 DML

数据操纵语言 DML 主要有三种形式：

1）插入：INSERT

2）更新：UPDATE

3）删除：DELETE

（3）数据定义语言 DDL

数据定义语言 DDL 用来创建数据库中的各种对象-----表、视图、索引、同义词、聚簇等如：

CREATE TABLE/VIEW/INDEX/SYN/CLUSTER

表 视图 索引 同义词 簇

DDL 操作是隐性提交的！不能 rollback

（4）数据控制语言 DCL

数据控制语言 DCL 用来授予或回收访问数据库的某种特权，并控制数据库操纵事务发生的时间及效果，对数据库实行监视等。如：

1）GRANT：授权。

2）ROLLBACK [WORK] TO [SAVEPOINT]：回退到某一点。回滚---ROLLBACK；回滚命令使数据库状态回到上次最后提交的状态。其格式为：

SQL>ROLLBACK;

3）COMMIT [WORK]：提交。

在数据库的插入、删除和修改操作时，只有当事务在提交到数据库时才算完成。在事务提交前，只有操作数据库的这个人才能有权看到所做的事情，别人只有在最后提交完成后才可以看到。

提交数据有三种类型：显式提交、隐式提交及自动提交。下面分别说明这三种类型。

- ✓ **显式提交**

  用 COMMIT 命令直接完成的提交为显式提交。其格式为：

  SQL>COMMIT;

- ✓ **隐式提交**

  用 SQL 命令间接完成的提交为隐式提交。这些命令是：

  ALTER，AUDIT，COMMENT，CONNECT，CREATE，DISCONNECT，DROP，EXIT，GRANT，NOAUDIT，QUIT，REVOKE，RENAME。

- ✓ **自动提交**

  若把 AUTOCOMMIT 设置为 ON，则在插入、修改、删除语句执行后，系统将自动进行提交，这就是自动提交。其格式为：

  SQL>SET AUTOCOMMIT ON;

参考文章：
https://www.cnblogs.com/study-s/p/5287529.html

16.like %和-的区别

通配符的分类:

%百分号通配符:表示任何字符出现任意次数(可以是 0 次).

_下划线通配符:表示只能匹配单个字符,不能多也不能少,就是一个字符.

like 操作符: LIKE 作用是指示 mysql 后面的搜索模式是利用通配符而不是直接相等匹配进行比较.

注意: 如果在使用 like 操作符时,后面的没有使用通用匹配符效果是和=一致的,SELECT * FROM products WHERE products.prod_name like '1000';
只能匹配的结果为 1000,而不能匹配像 JetPack 1000 这样的结果.

- %通配符使用: 匹配以"yves"开头的记录:(包括记录"yves")
  SELECT *FROM products WHERE products.prod_name like 'yves%';*
  匹配包含"yves"的记录(包括记录"yves") SELECT FROM products WHERE products.prod_name like '%yves%';
  匹配以"yves"结尾的记录(包括记录"yves",不包括记录"yves ",也就是 yves 后面有空格的记录,这里需要注意) SELECT * FROM products WHERE products.prod_name like '%yves';

- *通配符使用: SELECT FROM products WHERE products.prod_name like '_yves'; 匹配结果为: 像"yyves"这样记录.*
  *SELECT FROM products WHERE products.prod*name like 'yves*';*
  匹配结果为: 像"yvesHe"这样的记录.(一个下划线只能匹配一个字符,不能多也不能少)

注意事项:

- 注意大小写,在使用模糊匹配时,也就是匹配文本时,mysql 是可能区分大小的,也可能是不区分大小写的,这个结果是取决于用户对 MySQL 的配置方式.如果是区分大小写,那么像 YvesHe 这样记录是不能被"yves__"这样的匹配条件匹配的.
- 注意尾部空格,"%yves"是不能匹配"heyves "这样的记录的.
- 注意 NULL,%通配符可以匹配任意字符,但是不能匹配 NULL,也就是说 SELECT * FROM products WHERE products.prod_name like '%;是匹配不到 products.prod_name 为 NULL 的的记录.

技巧与建议:

正如所见， MySQL 的通配符很有用。但这种功能是有代价的：通配符搜索的处理一般要比前面讨论的其他搜索所花时间更长。这里给出一些使用通配符要记住的技巧。

- 不要过度使用通配符。如果其他操作符能达到相同的目的，应该 使用其他操作符。
- 在确实需要使用通配符时，除非绝对有必要，否则不要把它们用 在搜索模式的开始处。把通配符置于搜索模式的开始处，搜索起 来是最慢的。
- 仔细注意通配符的位置。如果放错地方，可能不会返回想要的数.

参考博文：https://blog.csdn.net/u011479200/article/details/78513632

17.count(*)、count(1)、count(column)的区别

- count(*)对行的数目进行计算，包含 NULL
- count(column)对特定的列的值具有的行数进行计算，不包含 NULL 值。
- count()还有一种使用方式，count(1)这个用法和 count(*)的结果是一样的。

性能问题：

1.任何情况下 SELECT COUNT(*) FROM tablename 是最优选择；

2.尽量减少 SELECT COUNT(*) FROM tablename WHERE COL = ‘value’ 这种查询；

3.杜绝 SELECT COUNT(COL) FROM tablename WHERE COL2 = ‘value’ 的出现。

- 如果表没有主键，那么 count(1)比 count(*)快。
- 如果有主键，那么 count(主键，联合主键)比 count(*)快。
- 如果表只有一个字段，count(*)最快。

count(1)跟 count(主键)一样，只扫描主键。count(*)跟 count(非主键)一样，扫描整个表。明显前者更快一些。

18.最左前缀原则

多列索引：

```
ALTER TABLE people ADD INDEX lname_fname_age (lame,fname,age);
```

为了提高搜索效率，我们需要考虑运用多列索引,由于索引文件以 B－Tree 格式保存，所以我们不用扫描任何记录，即可得到最终结果。

注：在 mysql 中执行查询时，只能使用一个索引，如果我们在 lname,fname,age 上分别建索引,执行查询时，只能使用一个索引，mysql 会选择一个最严格(获得结果集记录数最少)的索引。

最左前缀原则：顾名思义，就是最左优先，上例中我们创建了 lname_fname_age 多列索引,相当于创建了(lname)单列索引，(lname,fname)组合索引以及(lname,fname,age)组合索引。

二、索引

1.什么是索引？

何为索引：

数据库索引，是数据库管理系统中一个排序的数据结构，索引的实现通常使用 B 树及其变种 B+树。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用(指向)数据,这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

2.索引的作用？它的优点缺点是什么？

索引作用：

协助快速查询、更新数据库表中数据。

为表设置索引要付出代价的：

- 一是增加了数据库的存储空间
- 二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

3.索引的优缺点？

创建索引可以大大提高系统的性能（优点）：

1.通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

2.可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

3.可以加速表和表之间的连接,特别是在实现数据的参考完整性方面特别有意义。

4.在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

5.通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

增加索引也有许多不利的方面(缺点)：

1.创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

2.索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

3.当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

4.哪些列适合建立索引、哪些不适合建索引？

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。

一般来说，应该在这些列上创建索引：

（1）在经常需要搜索的列上，可以加快搜索的速度；

（2）在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；

（3）在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；

（4）在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；

（5）在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；

（6）在经常使用在 WHERE 子句中的列上面创建索引，加快条件的判断速度。

对于有些列不应该创建索引：

（1）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。

（2）对于那些只有很少数据值的列也不应该增加索引。

这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

（3）对于那些定义为 text, image 和 bit 数据类型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。

(4)当修改性能远远大于检索性能时，不应该创建索引。

这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

5.什么样的字段适合建索引

唯一、不为空、经常被查询的字段

6.MySQL B+Tree 索引和 Hash 索引的区别?

Hash 索引和 B+树索引的特点：

- Hash 索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位;
- B+树索引需要从根节点到枝节点，最后才能访问到页节点这样多次的 IO 访问;

为什么不都用 Hash 索引而使用 B+树索引？

1. Hash 索引仅仅能满足"="，"IN"和""查询，不能使用范围查询,因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和 Hash 运算前完全一样；
1. Hash 索引无法被用来避免数据的排序操作，因为 Hash 值的大小关系并不一定和 Hash 运算前的键值完全一样；
1. Hash 索引不能利用部分索引键查询，对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用；
1. Hash 索引在任何时候都不能避免表扫描，由于不同索引键存在相同 Hash 值，所以即使取满足某个 Hash 键值的数据的记录条数，也无法从 Hash 索引中直接完成查询，还是要回表查询数据；

1. Hash 索引遇到大量 Hash 值相等的情况后性能并不一定就会比 B+树索引高。

补充：

1.MySQL 中，只有 HEAP/MEMORY 引擎才显示支持 Hash 索引。

2.常用的 InnoDB 引擎中默认使用的是 B+树索引，它会实时监控表上索引的使用情况，如果认为建立哈希索引可以提高查询效率，则自动在内存中的“自适应哈希索引缓冲区”建立哈希索引（在 InnoDB 中默认开启自适应哈希索引），通过观察搜索模式，MySQL 会利用 index key 的前缀建立哈希索引，如果一个表几乎大部分都在缓冲池中，那么建立一个哈希索引能够加快等值查询。
B+树索引和哈希索引的明显区别是：

3.如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据；

4.如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；
同理，哈希索引没办法利用索引完成排序，以及 like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；

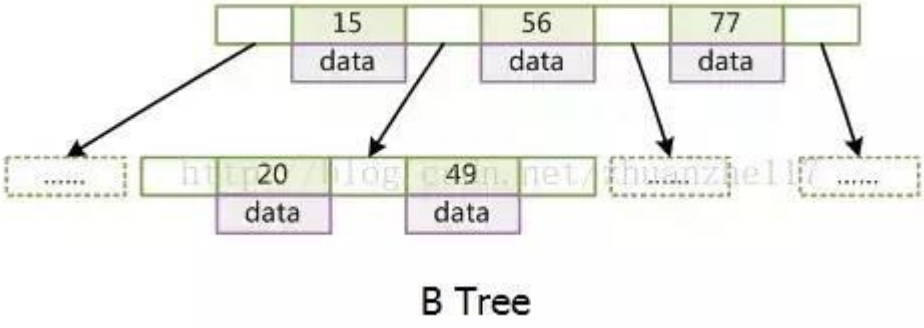
5.哈希索引也不支持多列联合索引的最左匹配规则；

6.B+树索引的关键字检索效率比较平均，不像 B 树那样波动幅度大，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题。

7.在大多数场景下，都会有范围查询、排序、分组等查询特征，用 B+树索引就可以了。

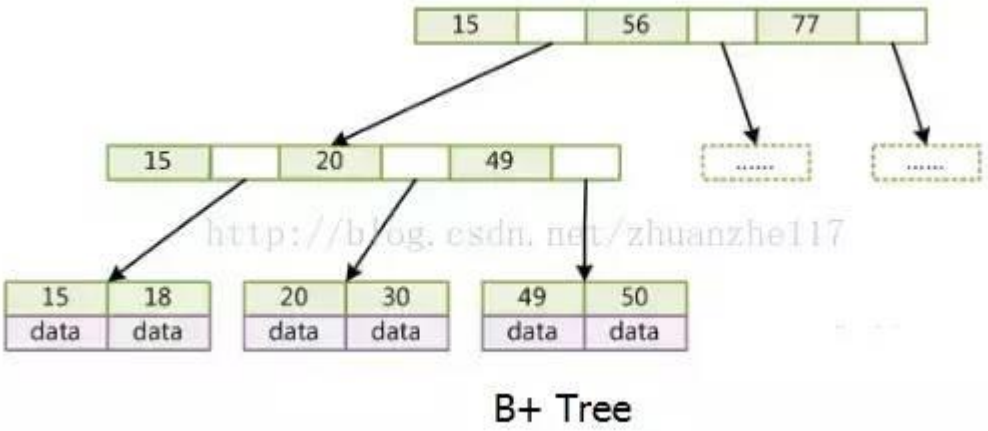
7.B 树和 B+树的区别

1. B 树，每个节点都存储 key 和 data，所有节点组成这棵树，并且叶子节点指针为 nul，叶子结点不包含任何关键字信息。



B Tree

2. B+树，所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接，所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。（而 B 树的非终节点也包含需要查找的有效信息）



B+ Tree

8.为什么说 B+比 B 树更适合实际应用中操作系统的文件索引和数据库索引？

1.B+的磁盘读写代价更低

B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说 IO 读写次数也就降低了。

2.B+tree 的查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

9.聚集索引和非聚集索引区别?

聚合索引(clustered index):

聚集索引**表记录的排列顺序和索引的排列顺序一致，所以查询效率快，**只要找到第一个索引值记录，其余就连续性的记录在物理也一样连续存放。聚集索引对应的缺点就是修改慢，因为为了保证表中记录的物理和索引顺序一致，在记录插入的时候，会对数据页重新排序。
聚集索引类似于新华字典中用拼音去查找汉字，拼音检索表于书记顺序都是按照 a~z 排列的，就像相同的逻辑顺序于物理顺序一样，当你需要查找 a,ai 两个读音的字，或是想一次寻找多个傻(sha)的同音字时，也许向后翻几页，或紧接着下一行就得到结果了。

非聚合索引(nonclustered index):

非聚集索引**指定了表中记录的逻辑顺序，但是记录的物理和索引不一定一致，**两种索引都采用 B+树结构，非聚集索引的叶子层并不和实际数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针方式。非聚集索引层次多，不会造成数据重排。
非聚集索引类似在新华字典上通过偏旁部首来查询汉字，检索表也许是按照横、竖、撇来排列的，但是由于正文中是 a~z 的拼音顺序，所以就类似于逻辑地址于物理地址的不对应。同时适用的情况就在于分组，大数目的不同值，频繁更新的列中，这些情况即不适合聚集索引。

根本区别：

聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致。

三、事务

1.什么是事务?

事务是对数据库中一系列操作进行统一的回滚或者提交的操作，主要用来保证数据的完整性和一致性。

2.事务四大特性（ACID）原子性、一致性、隔离性、持久性?

原子性（Atomicity）：
原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的

操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

一致性（Consistency）：
事务开始前和结束后，数据库的完整性约束没有被破坏。比如 A 向 B 转账，不可能 A 扣了钱，B 却没收到。

隔离性（Isolation）：
隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如 A 正在从一张银行卡中取钱，在 A 取钱的过程结束前，B 不能向这张卡转账。

持久性（Durability）：
持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

3.事务的并发?事务隔离级别，每个级别会引发什么问题，MySQL 默认是哪个级别?

从理论上来说，事务应该彼此完全隔离，以避免并发事务所导致的问题，然而，那样会对性能产生极大的影响，因为事务必须按顺序运行， 在实际开发中，为了提升性能，事务会以较低的隔离级别运行， 事务的隔离级别可以通过隔离事务属性指定。

事务的并发问题

1、脏读：事务 A 读取了事务 B 更新的数据，然后 B 回滚操作，那么 A 读取到的数据是脏数据

2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务 A 多次读取的过程中，对数据作了更新并提交，导致事务 A 多次读取同一数据时，结果因此本事务先后两次读到的数据结果会不一致。

3、幻读：幻读解决了不重复读，保证了同一个事务里，查询的结果都是事务开始时的状态（一致性）。

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，会发现还有跟没有修改一样，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。

小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于

新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表。

事务的隔离级别

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
不可重复读（read-committed）	否	是	是
可重复读（repeatable-read）	否	否	是
串行化（serializable）	否	否	否

读未提交：另一个事务修改了数据，但尚未提交，而本事务中的 SELECT 会读到这些未被提交的数据脏读

不可重复读：事务 A 多次读取同一数据，事务 B 在事务 A 多次读取的过程中，对数据作了更新并提交，导致事务 A 多次读取同一数据时，结果因此本事务先后两次读到的数据结果会不一致。

可重复读：在同一个事务里，SELECT 的结果是事务开始时时间点的状态，因此，同样的 SELECT 操作读到的结果会是一致的。但是，会有幻读现象

串行化：最高的隔离级别，在这个隔离级别下，不会产生任何异常。并发的事务，就像事务是在一个个按照顺序执行一样

特别注意：

MySQL 默认的事务隔离级别为 repeatable-read

MySQL 支持 4 中事务隔离级别.

事务的隔离级别要得到底层数据库引擎的支持，而不是应用程序或者框架的支持.

Oracle 支持的 2 种事务隔离级别：READ_COMMITED , SERIALIZABLE

SQL 规范所规定的标准，不同的数据库具体的实现可能会有些差异

MySQL 中默认事务隔离级别是“可重复读”时并不会锁住读取到的行

事务隔离级别：未提交读时，写数据只会锁住相应的行。

事务隔离级别为：可重复读时，写数据会锁住整张表。

事务隔离级别为：串行化时，读写数据都会锁住整张表。

隔离级别越高,越能保证数据的完整性和一致性,但是对并发性能的影响也越大,鱼和熊掌不可兼得啊。对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为 Read Committed，它能够避免脏读取，而且具有较好的并发性能。尽管它会导致不可重复读、幻读这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。

4.事务传播行为

1.PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

2.PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。

3.PROPAGATION_MANDATORY：支持当前事务,如果当前存在事务,就加入该事务，如果当前不存在事务，就抛出异常。

4.PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

5.PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

6.PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

7.PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与 PROPAGATION_REQUIRED 类似的操作。

5.嵌套事务

什么是嵌套事务？

嵌套是子事务套在父事务中执行,子事务是父事务的一部分,在进入子事务之前，父事务建立一个回滚点，叫 save point，然后执行子事务，这个子事务的执行也算是父事务的一部分，然后子事务执行结束，父事务继续执行。重点就在于那个 save point。看几个问题就明了了：

如果子事务回滚，会发生什么？

父事务会回滚到进入子事务前建立的 save point，然后尝试其他的事务或者其他的业务逻辑，父事务之前的操作不会受到影响，更不会自动回滚。

如果父事务回滚，会发生什么？

父事务回滚，子事务也会跟着回滚！为什么呢，因为父事务结束之前，子事务是不会提交的，我们说子事务是父事务的一部分，正是这个道理。那么：

事务的提交，是什么情况？

是父事务先提交，然后子事务提交，还是子事务先提交，父事务再提交？答案是第二种情况，还是那句话，子事务是父事务的一部分，由父事务统一提交。

参考文章：https://blog.csdn.net/liangxw1/article/details/51197560

四、存储引擎

1.MySQL 常见的三种存储引擎（InnoDB、MyISAM、MEMORY）的区别?

两种存储引擎的大致区别表现在：

1. **InnoDB 支持事务，MyISAM 不支持， 这一点是非常之重要。**事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MyISAM 就不可以了。

2. MyISAM 适合查询以及插入为主的应用。

3. InnoDB 适合频繁修改以及涉及到安全性较高的应用。

4. InnoDB 支持外键，MyISAM 不支持。

5. 从 MySQL5.5.5 以后，InnoDB 是默认引擎。

6. InnoDB 不支持 FULLTEXT 类型的索引。

7. InnoDB 中不保存表的行数，如 select count() *from table 时，InnoDB 需要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count()*语句包含 where 条件时 MyISAM 也需要扫描整个表。

8. 对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引。

9. DELETE FROM table 时，InnoDB 不会重新建立表，而是一行一行的 删除，效率非常慢。MyISAM 则会重建表。

10. InnoDB 支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like '%lee%'。

2.MySQL 存储引擎 MyISAM 与 InnoDB 如何选择

MySQL 有多种存储引擎，每种存储引擎有各自的优缺点，可以择优选择使用：MyISAM、InnoDB、MERGE、MEMORY(HEAP)、BDB(BerkeleyDB)、EXAMPLE、FEDERATED、ARCHIVE、CSV、BLACKHOLE。

虽然 MySQL 里的存储引擎不只是 MyISAM 与 InnoDB 这两个，但常用的就是两个。关于 MySQL 数据库提供的两种存储引擎，MyISAM 与 InnoDB 选择使用：

- 1.INNODB 会支持一些关系数据库的高级功能，如事务功能和行级锁，MyISAM 不支持。
- 2.MyISAM 的性能更优，占用的存储空间少，所以，选择何种存储引擎，视具体应用而定。

如果你的应用程序一定要使用事务，毫无疑问你要选择 INNODB 引擎。但要注意，INNODB 的行级锁是有条件的。在 where 条件没有使用主键时，照样会锁全表。比如 DELETE FROM mytable 这样的删除语句。

如果你的应用程序对查询性能要求较高，就要使用 MyISAM 了。MyISAM 索引和数据是分开的，而且其索引是压缩的，可以更好地利用内存。所以它的查询性能明显优于 INNODB。压缩后的索引也能节约一些磁盘空间。MyISAM 拥有全文索引的功能，这可以极大地优化 LIKE 查询的效率。

有人说 MyISAM 只能用于小型应用，其实这只是一种偏见。如果数据量比较大，这是需要通过升级架构来解决，比如分表分库，而不是单纯地依赖存储引擎。

现在一般都是选用 innodb 了，主要是 MyISAM 的全表锁，读写串行问题，并发效率锁表，效率低，MyISAM 对于读写密集型应用一般是不会去选用的。
MEMORY 存储引擎

MEMORY 是 MySQL 中一类特殊的存储引擎。它使用存储在内存中的内容来创建表，而且数据全部放在内存中。这些特性与前面的两个很不同。
每个基于 MEMORY 存储引擎的表实际对应一个磁盘文件。该文件的文件名与表名相同，类型为 frm 类型。该文件中只存储表的结构。而其数据文件，都是存储在内存中，这样有利于数据的快速处理，提高整个表的效率。值得注意的是，服务器需要有足够的内存来维持 MEMORY 存储引擎的表的使用。如果不需要了，可以释放内存，甚至删除不需要的表。

MEMORY 默认使用哈希索引。速度比使用 B 型树索引快。当然如果你想用 B 型树索引，可以在创建索引时指定。

注意，MEMORY 用到的很少，因为它是把数据存到内存中，如果内存出现异常就会影响数据。如果重启或者关机，所有数据都会消失。因此，基于 MEMORY 的表的生命周期很短，一般是一次性的。

3.MySQL 的 MyISAM 与 InnoDB 两种存储引擎在，事务、锁级别，各自的适用场景?

事务处理上方面

- MyISAM：强调的是性能，每次查询具有原子性,其执行数度比 InnoDB 类型更快，但是不提供事务支持。
- InnoDB：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(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。

锁级别

- MyISAM:只支持表级锁,用户在操作 MyISAM 表时,select,update,delete,insert 语句都会给表自动加锁，如果加锁以后的表满足 insert 并发的情况下，可以在表的尾部插入新的数据。
- InnoDB：支持事务和行级锁，是 innodb 的最大特色。行锁大幅度提高了多用户并发操作的新能。但是 InnoDB 的行锁，只是在 WHERE 的主键是有效的，非主键的 WHERE 都会锁全表的。

关于存储引擎 MyISAM 和 InnoDB 的其他参考资料如下：

MySQL 存储引擎中的 MyISAM 和 InnoDB 区别详解

MySQL 存储引擎之 MyISAM 和 Innodb 总结性梳理

五、优化

1.查询语句不同元素（where、jion、limit、group by、having 等等）执行先后顺序?

- 1.查询中用到的关键词主要包含**六个**，并且他们的顺序依次为 select--from--where--group by--having--order by

其中 select 和 from 是必须的，其他关键词是可选的，这六个关键词的执行顺序 与 sql 语句的书写顺序并不是一样的，而是按照下面的顺序来执行

from:需要从哪个数据表检索数据

where:过滤表中数据的条件

group by:如何将上面过滤出的数据分组

having:对上面已经分组的数据进行过滤的条件

select:查看结果集中的哪个列，或列的计算结果

order by :按照什么样的顺序来查看返回的数据

- 2.from 后面的表关联，是自右向左解析 而 where 条件的解析顺序是自下而上的。

也就是说，在写 SQL 语句的时候，尽量把数据量小的表放在最右边来进行关联(用小表去匹配大表)，而把能筛选出小量数据的条件放在 where 语句的最左边 （用小表去匹配大表）

其他参考资源：
http://www.cnblogs.com/huminxxl/p/3149097.html

2.使用 explain 优化 sql 和索引?

对于复杂、效率低的 sql 语句，我们通常是使用 explain sql 来分析 sql 语句，这个语句可以打印出，语句的执行。这样方便我们分析，进行优化

table：显示这一行的数据是关于哪张表的

type：这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为 const、eq_reg、ref、range、index 和 ALL

all:full table scan ;MySQL 将遍历全表以找到匹配的行；

index: index scan; index 和 all 的区别在于 index 类型只遍历索引；

range：索引范围扫描，对索引的扫描开始于某一点，返回匹配值的行，常见与 between ，等查询；

ref：非唯一性索引扫描，返回匹配某个单独值的所有行，常见于使用非唯一索引即唯一索引的非唯一前缀进行查找；

eq_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配，常用于主键或者唯一索引扫描；

const，system：当 MySQL 对某查询某部分进行优化，并转为一个常量时，使用这些访问类型。如果将主键置于 where 列表中，MySQL 就能将该查询转化为一个常量。

possible_keys：显示可能应用在这张表中的索引。如果为空，没有可能的索引。可以为相关的域从 WHERE 语句中选择一个合适的语句

key： 实际使用的索引。如果为 NULL，则没有使用索引。很少的情况下，MySQL 会选择优化不足的索引。这种情况下，可以在 SELECT 语句中使用 USE INDEX

（indexname）来强制使用一个索引或者用 IGNORE INDEX（indexname）来强制 MySQL 忽略索引

key_len：使用的索引的长度。在不损失精确性的情况下，长度越短越好

ref：显示索引的哪一列被使用了，如果可能的话，是一个常数

rows：MySQL 认为必须检查的用来返回请求数据的行数

Extra：关于 MySQL 如何解析查询的额外信息。将在表 4.3 中讨论，但这里可以看到的坏的例子是 Using temporary 和 Using filesort，意思 MySQL 根本不能使用索引，结果是检索会很慢。

3.MySQL 慢查询怎么解决?

- slow_query_log 慢查询开启状态。
- slow_query_log_file 慢查询日志存放的位置（这个目录需要 MySQL 的运行帐号的可写权限,一般设置为 MySQL 的数据存放目录）。
- long_query_time 查询超过多少秒才记录。

六、数据库锁

1.mysql 都有什么锁，死锁判定原理和具体场景，死锁怎么解决?

MySQL 有三种锁的级别：页级、表级、行级。

- **表级锁**：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。
- **行级锁**：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。
- **页面锁**：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

  什么情况下会造成死锁?

什么是死锁？

死锁：是指两个或两个以上的进程在执行过程中。因争夺资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁,这些永远在互相等竺的进程称为死锁进程。

表级锁不会产生死锁.所以解决死锁主要还是针对于最常用的 InnoDB。

死锁的关键在于：两个(或以上)的 Session 加锁的顺序不一致。

那么对应的解决死锁问题的关键就是：让不同的 session 加锁有次序。

死锁的解决办法?

1.查出的线程杀死 kill
SELECT trx_MySQL_thread_id FROM information_schema.INNODB_TRX;

2.设置锁的超时时间
Innodb 行锁的等待时间，单位秒。可在会话级别设置，RDS 实例该参数的默认值为 50（秒）。

生产环境不推荐使用过大的 innodb_lock_wait_timeout 参数值
该参数支持在会话级别修改,方便应用在会话级别单独设置某些特殊操作的行锁等待超时时间，如下：
set innodb_lock_wait_timeout=1000; —设置当前会话 Innodb 行锁等待超时时间，单位秒。

3.指定获取锁的顺序

2.有哪些锁（乐观锁悲观锁），select 时怎么加排它锁?

悲观锁（Pessimistic Lock）：

悲观锁特点:先获取锁，再进行业务操作。

即“悲观”的认为获取锁是非常有可能失败的,因此要先确保获取锁成功再进行业务操作。通常所说的**“一锁二查三更新”即指的是使用悲观锁。**通常来讲在数据库上的悲观锁需要数据库本身提供支持，即通过常用的 select … for update 操作来实现悲观锁。当数据库执行 select for update 时会获取被 select 中的数据行的行锁，因此其他并发执行的 select for update 如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。select for update 获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。

补充：
不同的数据库对 select for update 的实现和支持都是有所区别的，

- oracle 支持 select for update no wait，表示如果拿不到锁立刻报错，而不是等待，MySQL 就没有 no wait 这个选项。
- MySQL 还有个问题是 select for update 语句执行中所有扫描过的行都会被锁上，这一点很容易造成问题。因此如果在 MySQL 中用悲观锁务必要确定走了索引，而不是全表扫描。

乐观锁（Optimistic Lock）：

1.乐观锁，也叫乐观并发控制，它假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的那部分数据。在提交数据

更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，那么当前正在提交的事务会进行回滚。

2.**乐观锁的特点先进行业务操作，不到万不得已不去拿锁。**即“乐观”的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。
乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。

3.一般的做法是**在需要锁的数据上增加一个版本号，或者时间戳，**

实现方式举例如下：

乐观锁（给表加一个版本号字段） 这个并不是乐观锁的定义，给表加版本号，是**数据库实现乐观锁的一种方式**。

1. SELECT data AS old_data, version AS old_version FROM …;
2. 根据获取的数据进行业务操作，得到 new_data 和 new_version
3. UPDATE SET data = new_data, version = new_version WHERE version = old_version

```
if (updated row > 0) {

// 乐观锁获取成功，操作完成

} else {

// 乐观锁获取失败，回滚并重试

}
```

注意：

- 乐观锁在不发生取锁失败的情况下开销比悲观锁小，但是一旦发生失败回滚开销则比较大，因此适合用在取锁失败概率比较小的场景，可以提升系统并发性能
- 乐观锁还适用于一些比较特殊的场景，例如在业务操作过程中无法和数据库保持连接等悲观锁无法适用的地方。

总结：
悲观锁和乐观锁是数据库用来保证数据并发安全防止更新丢失的两种方法，例子在 select ... for update 前加个事务就可以防止更新丢失。悲观锁和乐观锁大部分场景下差异不大，一些独特场景下有一些差别，一般我们可以从如下几个方面来判断。

- **响应速度：** 如果需要非常高的响应速度，建议采用乐观锁方案，成功就执行，不成功就失败，不需要等待其他并发去释放锁。’

- **冲突频率：** 如果冲突频率非常高，建议采用悲观锁，保证成功率，如果冲突频率大，乐观锁会需要多次重试才能成功，代价比较大。

- **重试代价：** 如果重试代价大，建议采用悲观锁。

七、其他

1.数据库的主从复制

主从复制的几种方式:

同步复制:

所谓的同步复制，意思是 master 的变化，必须等待 slave-1,slave-2,...,slave-n 完成后才能返回。 这样，显然不可取，也不是 MySQL 复制的默认设置。比如，在 WEB 前端页面上，用户增加了条记录，需要等待很长时间。

异步复制:

如同 AJAX 请求一样。master 只需要完成自己的数据库操作即可。至于 slaves 是否收到二进制日志，是否完成操作，不用关心,MySQL 的默认设置。

半同步复制:

master 只保证 slaves 中的一个操作成功，就返回，其他 slave 不管。 这个功能，是由 google 为 MySQL 引入的。

2.数据库主从复制分析的 7 个问题?

问题 1：master 的写操作，slaves 被动的进行一样的操作，保持数据一致性，那么 slave 是否可以主动的进行写操作？

假设 slave 可以主动的进行写操作，slave 又无法通知 master，这样就导致了 master 和 slave 数据不一致了。因此 slave 不应该进行写操作，至少是 slave 上涉及到复制的数据库不可以写。实际上，这里已经揭示了读写分离的概念。

问题 2：主从复制中，可以有 N 个 slave,可是这些 slave 又不能进行写操作，要他们干嘛？

实现数据备份:
类似于高可用的功能，一旦 master 挂了，可以让 slave 顶上去，同时 slave 提升为 master。

异地容灾:比如 master 在北京，地震挂了，那么在上海的 slave 还可以继续。
主要用于实现 scale out,分担负载,可以将读的任务分散到 slaves 上。
【很可能的情况是，一个系统的读操作远远多于写操作，因此写操作发向 master，读操作发向 slaves 进行操作】

问题 3：主从复制中有 master,slave1,slave2,...等等这么多 MySQL 数据库，那比如一个 JAVA WEB 应用到底应该连接哪个数据库?

我们在应用程序中可以这样，insert/delete/update 这些更新数据库的操作，用 connection(for master)进行操作，

select 用 connection(for slaves)进行操作。那我们的应用程序还要完成怎么从 slaves 选择一个来执行 select，例如使用简单的轮循算法。

这样的话，相当于应用程序完成了 SQL 语句的路由，而且与 MySQL 的主从复制架构非常关联，一旦 master 挂了，某些 slave 挂了，那么应用程序就要修改了。
能不能让应用程序与 MySQL 的主从复制架构没有什么太多关系呢?
找一个组件，application program 只需要与它打交道，用它来完成 MySQL 的代理，实现 SQL 语句的路由。
MySQL proxy 并不负责，怎么从众多的 slaves 挑一个？可以交给另一个组件(比如 haproxy)来完成。

这就是所谓的 MySQL READ WRITE SPLITE，MySQL 的读写分离。

问题 4：如果 MySQL proxy ， direct ， master 他们中的某些挂了怎么办?

总统一般都会弄个副总统,以防不测。同样的,可以给这些关键的节点来个备份。

问题 5：当 master 的二进制日志每产生一个事件，都需要发往 slave，如果我们有 N 个 slave,那是发 N 次，还是只发一次？如果只发一次，发给了 slave-1，那 slave-2,slave-3,...它们怎么办?

显 然，应该发 N 次。实际上，在 MySQL master 内部，维护 N 个线程，每一个线程负责将二进制日志文件发往对应的 slave。master 既要负责写操作，还的维护 N 个线程，负担会很重。可以这样，slave-1 是 master 的从，slave-1 又是 slave-2,slave-3,...的主，同时 slave-1 不再负责 select。 slave-1 将 master 的复制线程的负担，转移到自己的身上。这就是所谓的多级复制的概念。

问题 6：当一个 select 发往 MySQL proxy，可能这次由 slave-2 响应，下次由 slave-3 响应，这样的话，就无法利用查询缓存了。

应该找一个共享式的缓存，比如 memcache 来解决。将 slave-2,slave-3,...这些查询的结果都缓存至 mamcache 中。

问题 7:随着应用的日益增长,读操作很多,我们可以扩展 slave,但是如果 master 满足不了写操作了，怎么办呢?

scale on ?更好的服务器? 没有最好的，只有更好的，太贵了。。。
scale out ? 主从复制架构已经满足不了。
可以分库【垂直拆分】，分表【水平拆分】。

3.mysql 高并发环境解决方案?

MySQL 高并发环境解决方案： 分库 分表 分布式 增加二级缓存。。。。。

需求分析：互联网单位 每天大量数据读取，写入，并发性高。

现有解决方式：水平分库分表，由单点分布到多点数据库中，从而降低单点数据库压力。

集群方案：解决 DB 宕机带来的单点 DB 不能访问问题。

读写分离策略：极大限度提高了应用中 Read 数据的速度和并发量。无法解决高写入压力。

4.数据库崩溃时事务的恢复机制（REDO 日志和 UNDO 日志）?

转载：MySQL REDO 日志和 UNDO 日志

Undo Log:

Undo Log 是为了实现事务的原子性，在 MySQL 数据库 InnoDB 存储引擎中，还用了 Undo Log 来实现多版本并发控制(简称：MVCC)。

事务的原子性(Atomicity)事务中的所有操作，要么全部完成，要么不做任何操作，不能只做部分操作。如果在执行的过程中发生了错误，要回滚(Rollback)到事务开始前的状态，就像这个事务从来没有执行过。
原理 Undo Log 的原理很简单，为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为 UndoLog）。然后进行数据的修改。如果出现了错误或者用户执行了 ROLLBACK 语句，系统可以利用 Undo Log 中的备份将数据恢复到事务开始之前的状态。

之所以能同时保证原子性和持久化，是因为以下**特点：**

更新数据前记录 Undo log。
为了保证持久性，必须将数据在事务提交前写到磁盘。只要事务成功提交，数据必然已经持久化。
Undo log 必须先于数据持久化到磁盘。如果在 G,H 之间系统崩溃，undo log 是完整的， 可以用来回滚事务。

如果在 A-F 之间系统崩溃,因为数据没有持久化到磁盘。所以磁盘上的数据还是保持在事务开始前的状态。

缺陷:每个事务提交前将数据和 Undo Log 写入磁盘,这样会导致大量的磁盘 IO,因此性能很低。
如果能够将数据缓存一段时间，就能减少 IO 提高性能。但是这样就会丧失事务的持久性。因此引入了另外一种机制来实现持久化，即 Redo Log。

Redo Log:

原理和 Undo Log 相反，Redo Log 记录的是新数据的备份。在事务提交前，只要将 Redo Log 持久化即可，不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是 Redo Log 已经持久化。系统可以根据 Redo Log 的内容，将所有数据恢复到最新的状态。