Mysql面试

✓ 数据类型

整形

1、int占4个字节，bigint占8个字节。

2、如果需要设置无符号，则需要加上unsigned关键字，例如 `id int unsigned`

3、如果需要设置显示的长度，在类型后指定，并且加上zerofill关键字。例如 `id int(5) zerofill` **左侧填充0，且此字段变成无符号**。

小数

浮点数float(M, D)、double(M, D)，定点数decimal(M, D)。

- M：总精度，整数加小数部分，1 <= M <= 65, 默认M = 10
- D：小数部分精度，0 <= D <= 30 且D <= M, 默认D = 0

如果是浮点数，则会根据插入的数据决定精度。decimal的精度较高，货币运算优先考虑。

> decimal的存储是将整数跟小数分开存储的。为了节省空间，MYSQL采用4字节来存储9位数位，即两位一个字节，最后三位共用一个字节。
>
> | 数位 | 字节 |
> | --- | --- |
> | 1-2 | 1 |
> | 3-4 | 2 |
> | 5-6 | 3 |
> | 7-9 | 4 |

> 举个例子，decimal(18,9)的整数部分和小数部分各有9位，所以两边各需要4字节来存储。decimal(20,6)有14位整数，6位小数，整数部分先用4字节表示9位，余下5位仍然需要3字节，所以整数部分共7个字节，小数部分则需要3字节。

日期

- datetime：**8**个字节，可以表示日期跟时间，**不受时区影响**，插入什么时间就显示什么时间。
- timestamp：**4**个字节，可以表示日期跟时间，**受时区影响**，会根据时区自动转换。

✓ char(10) 跟 varchar(10)的区别

CHAR(10)会**固定分配10个字符**（无论英文或汉字）的空间。即若输入数据的长度小于10B，则系统自动在其后添加空格来填满设定好的空间；

而**VARCHAR(10)**的存储长度为**实际数值的长度**，但最多会存储10个字符。

> **字节**(Byte)是计量单位，表示数据量多少，是计算机信息技术用于计量存储容量的一种计量单位，通常情况下一字节等于八位。
>
> **字符**(Character)计算机中使用的字母、数字、字和符号，比如'A'、'B'、'$'、'&'等。
>
> - ASCII 码中，一个英文字母为一个字节，一个中文汉字为**两个**字节。
> - Unicode 编码中，一个英文为一个字节，一个中文为**两个**字节。
> - UTF-8 编码中，一个英文字为一个字节，一个中文为**三个**字节。

比如：UTF-8编码中

- 存储英文‘a’：`char(1)` 跟 `carchar(1)` 都占用1个字节
- 存储中文‘我’：`char(1)` 跟 `carchar(1)` 都占用3个字节

✓ 存储引擎

存储引擎：mysql存储表的机制。

在建表的时候可以指定存储引擎和字符集。如 `create table s (id int) ENGINE=InnoDB DEFAULT CHARSET=UTF8;`

mysql默认使用的存储引擎是 `InnoDB` ，默认字符集是 `UTF8` 。

mysql常见存储引擎：

- `MyISAM`
    - 采用三个文件来组织一个表
        - **`.frm` 文件（存储表的结构）**
        - **`.MYD` 文件（存储表的数据）**
        - **`.MYI` 文件（存储表的索引）**
    - 特点： 不支持事务，但每次查询都是原子的。支持表级锁。只会缓存索引。
- `InnoDB`
    - 表的结构存放在 **`.frm` 文件**
    - 数据跟索引存储在 **`.idb` 文件**
    - 优点：支持事务、行级锁、外键等，安全性好，可自动恢复。既缓存索引也缓存数据。
    - 缺点：无法被压缩，无法转换成只读。
- `MEMORY`
    - 每个表结构放在 `.frm` 文件中
    - 数据跟索引放在内存中
    - 优点：**查询速度最快**。
    - 缺点：不支持事务，容易丢失数据。

✓ 事务

> 一个事务是一个完整的业务逻辑，不可再分。

> 事务的存在使数据更加的安全，完整。

例如：银行账户转账，从A账户向B账户转账1000，需要执行两条DML语句 `update t_act set money=money-1000 where actno='A';` 跟 `update t_act set money=money+1000 where actno='B';` 。此时需要保证这两条更新语句必须同时成功或同时失败。

tips： 如果要让多条DML语句同时执行成功或执行失败，则需要使用 `事务` 。

事务执行流程：

1. 开启事务机制
2. 执行一条或多条DML语句（在缓存中执行，不直接影响文件）
3. 提交或回滚事务
    - `commit;` 提交事务。缓存区的内容更新到文件，清空缓存区
    - `rollback;` 回滚事务。清空缓存区。

事务的特性：ACID

- 原子性（Atomicity）：事务是最小的工作单元，不可再分，保证多条DML语句同时成功或失败。
- 一致性（Consistency）：事务前后数据的完整性必须保持一致。例如A+3，B-3
- 隔离性（Isolation）：排除其他事务对本次事务的影响。
- 持久性（Durability）：事务结束后的数据不随着外界原因导致数据丢失。

事物的 `隔离性` 分为 `4` 个等级：

1. **读未提交**（read uncommitted）
    - A事务还没有提交，B事务就可以读取A事务未提交时的数据。
    - 缺点：存在 脏读现象，即随时会读到不确定的数据。
2. **读已提交**（read committed）
    - B事务提交之后，A事务才可以读取到B事务提交之后的数据。
    - 缺点：不可重复读，即只要对方一提交事务，数据立马变化。
3. **可重复读**（repeatable）
    - 只要不退出当前事务，则数据永远是进入事务前的数据。不会随着别的事务的提交而发生数据改变。
    - 缺点：读取到的数据是假象。

4. **序列化读**（serialize）

    - 只有等当前事务结束时，另一个事务才可以执行。
    - 缺点：效率低，需要事务排队。

tips:

- `oracle` 默认使用的隔离级别是：`读已提交`
- `mysql` 默认使用的隔离级别是：`可重复读`
- mysql的事务默认是自动提交的，只要执行一条DML就提交一次。
- mysql使用 `start transaction;` 可关闭自动提交机制。

✓ 三大范式

第一范式(1NF)：每个字段不可再分

↓

↓ 消除**非主属性**对码的**部分函数依赖**，即拆成多张表

↓

第二范式(2NF)：非主属性完全依赖于码

↓

↓ 消除**非主属性**对码的**传递函数依赖**，即拆成多张表

↓

第三范式(3NF)：属性直接依赖于主键

↓

↓ 消除**主属性**与码的**部分函数依赖和传递函数依赖**

↓

BC范式(BCNF)：(一张表只能有一个老大(码)，老大(码)可以是一个属性组)

1. 所有的非主属性对每一个码都是完全函数依赖。
2. 所有主属性对每一个不包含它的码也是完全依赖。
3. 所有属性对非主属性都不能存在函数依赖。

> 所有的主属性（候选码）中选出一个作为码，码才是唯一标识记录的。

✓ exists 跟 in 的区别

`exists` 返回true或false，它常常和子查询配合使用，用法如下：

```
select a.* from A a where exists(select 1 from B b where a.id=b.id)
等价于：
for select a.* from A a
    for select 1 from B b where a.id=b.id
```

exists语句会拿着A表的记录去B表匹配，匹配成功就加入到结果集。exists并不会去缓存子查询的结果集，因为这个结果集并不重要，只需要返回真假即可。

而 `in()` 语句只会执行一次，它查出B表中的所有id字段并且缓存起来，之后，拿着B表的记录去A表匹配，匹配成功则加入结果集。

```
select a.* from A a where a.id in (select id from B)
等价于：
for select id from B
    select a.* from A a where a.id=b.id
```

根据**小表驱动大表**的原则，最外层表的数据越小，则效率越高。所以，有以下结论：

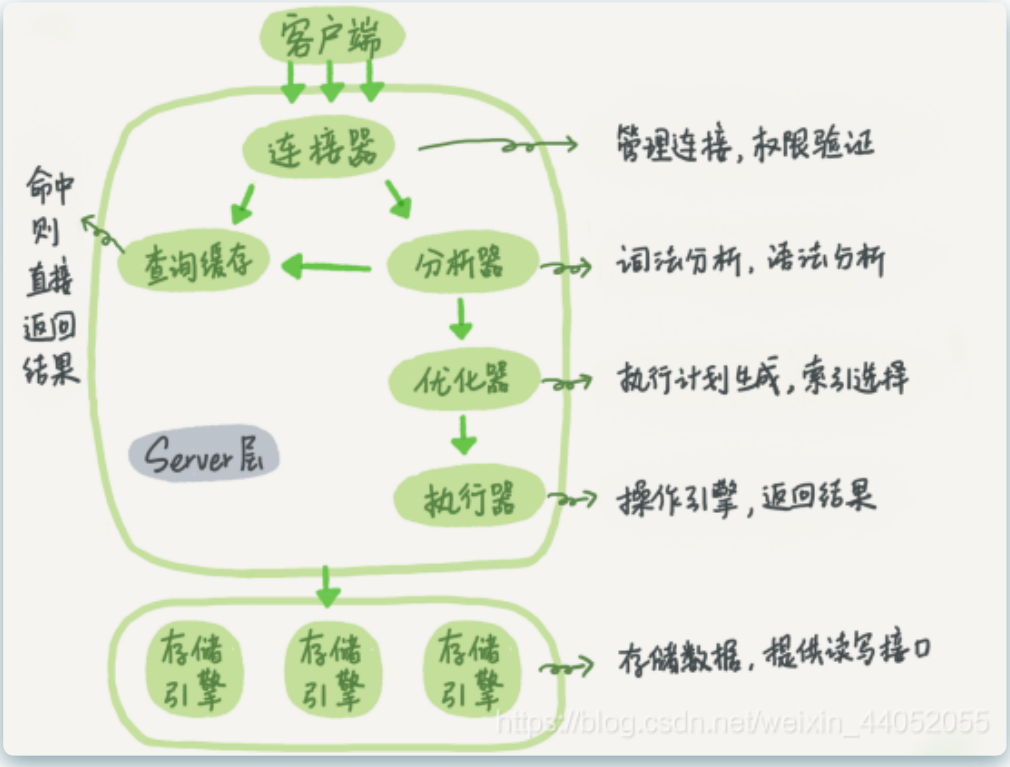
1. in()适合B表比A表数据量小的情况
2. exists()适合B表比A表数据量大的情况
3. 当A表数据与B表数据一样时，in与exists效率差不多，可任选一个使用.

✓ not in 和not exists

如果字段中**没有**null值，则两者效率差不多。

如果字段中**存在**null值，则使用 `not in` 时，索引失效，内外表都进行全表扫描；而 `not exists` 的子查询依然能用到表上的索引。

✓ Mysql基础架构



优化器：在表里面有多个索引的时候，决定**使用哪个索引**或者**表的连接顺序**。

执行器：具体执行sql，执行之前会判断一下你对这个表有没有**执行查询的权限**。

✓ 二阶段提交

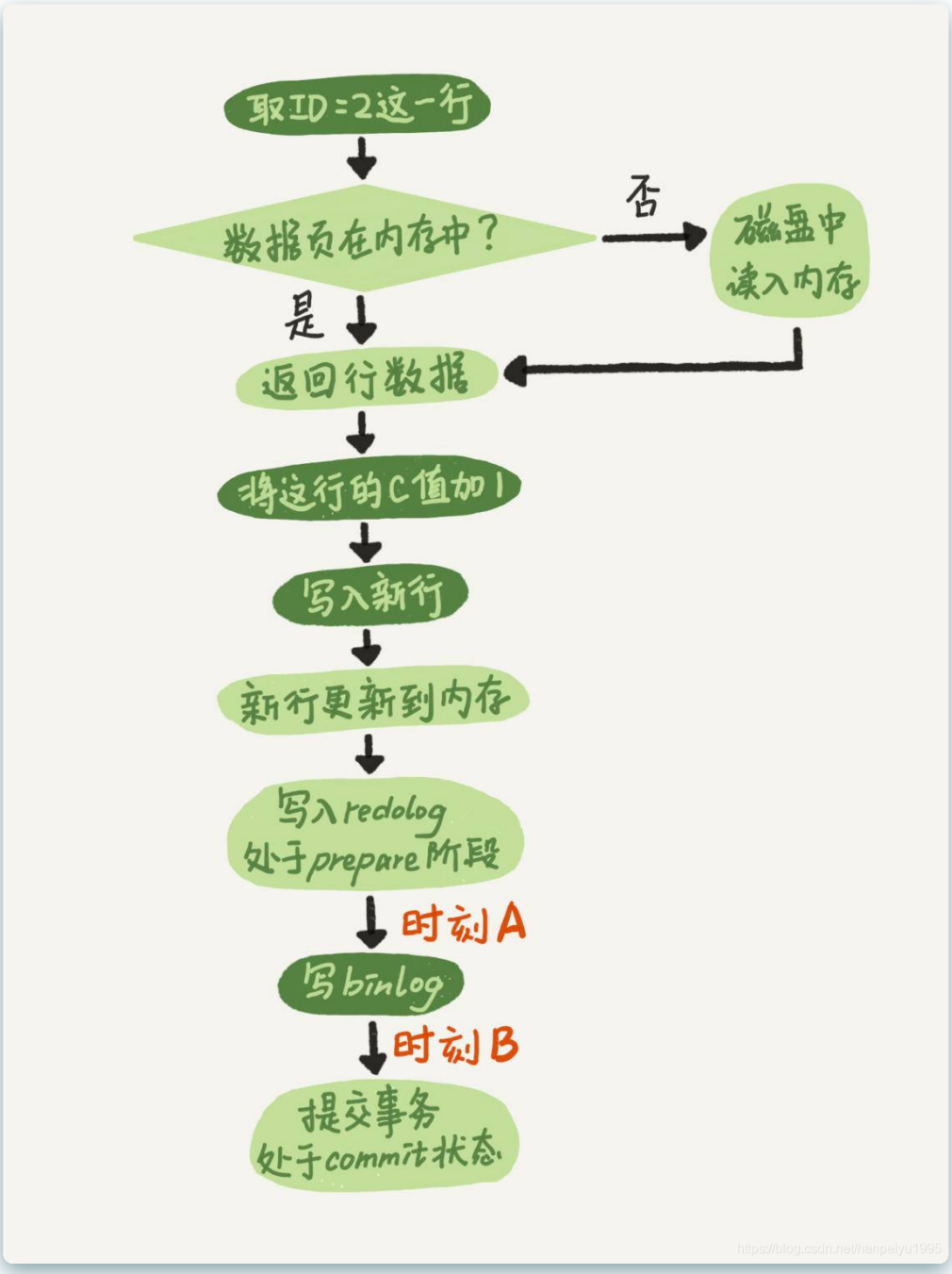
redo log 和 binlog

1. redolog 是**InnoDB引擎**特有的； binlog 是MySQL的 **Server层**实现的，所有引擎都可以使用。
2. redolog 是**物理日志**，记录的是 在某个数据页上做了什么修改；binlog 是**逻辑日志**，记录的是语句的原始逻辑，比如“给ID=2的c字段加1”。
3. redolog的大小是**固定的** (4G)，满了之后就擦除一些记录。binlog是**追加写入**的，即一个文件满了就切换到下一个文件。

为了让两份日志之间的逻辑一致， redolog 的写入拆成了两个步骤： **prepare 和 commit** ，在这两个步骤之间会将记录写入binlog中，这就是 " **两阶段提交** "。用来保证数据的一致性。

两阶段提交步骤：

1. 事务提交之前，将 记录 写入redolog中，此时redolog处于prepare状态
2. 将记录写入binlog
3. 提交事务，将redolog的状态改为commit



为什么要二阶段

假设时刻A，主机挂掉了，数据库进行数据恢复的时候会查看redolog的状态跟binlog的记录。如果redolog 处于prepare状态，并且binlog中没有这个记录，说明该记录还没有写入到binlog中。

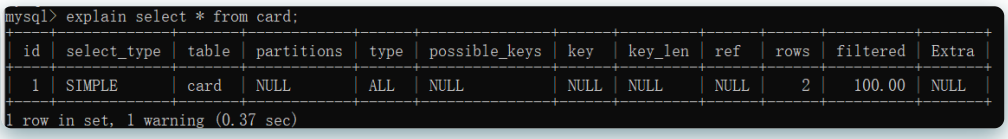
如果时刻B，主机挂掉了，此时binlog中有记录，redolog 处于prepare状态，说明该记录还没有真正的提交。

✓ 执行计划

执行计划，就是 一条SQL语句在数据库中实际执行的步骤。也就是我们用 `EXPLAIN` 分析一条SQL语句时展示出来的那些信息。

`EXPLAIN` 命令是查看 查询优化器 是如何执行查询的，从它的查询结果中可以知道一条SQL语句每一步是如何执行的，都经历了些什么，有没有用到索引，哪些字段用到了什么样的索引，这些信息都是我们SQL优化的依据。

语法： `explain select语句`



执行计划中的列

explain 显示的每个列都有不同的含义：

列名	含义
id	表示查询中执行select子句的顺序。id值大的先执行，若id相等则从上往下优先执行。
select_type	查询的类型，主要是用于区分普通查询、联合查询、子查询等复杂的查询。
table	表明对应行正在访问的是哪个表。
partitions	查询涉及到的分区。
type	查询数据时使用的索引类型。
possible_keys	查询可以使用的索引。如果使用的是覆盖索引，则不会显示；
key	实际使用的索引，如果为NULL，则没有使用索引。

列名	含义
key_len	查询中使用的索引的字节数（最大可能长度），并非实际使用长度，理论上长度越短越好。
ref	显示该表的索引字段关联的字段。
rows	大致估算出找到所需行所需要读取的行数。
filtered	返回结果的行数占读取行数的百分比，值越大越好。
Extra	额外信息，十分重要。

type：查询数据时使用的索引类型，性能由高到底排列如下：

- system，表中只有一行记录，比如系统表；
- const，通过索引一次命中，匹配一行数据；例如 主键置于where列表中：
  - `select * from stu where id = 1;`
- 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；
  - `select A.* from A,B where A.id=B.id`
- ref，非唯一性索引扫描，返回匹配某个单独值的所有行，用于=、<或>操作符带索引的列；
  - `select * from stu where teacher_id = 123;`
- range，只检索给定范围的行，使用一个索引来选择行，一般用于between、<、>；
  - `select * from stu where id between 1 and 10`
- index，只遍历索引树；跟all差不多，只不过index查的字段是索引。
  - `select id from stu;`
- all，全表扫描；
  - `select * from stu;`

extra常见的值如下：

- using filesort，MySQL会对数据使用一个外部索引排序，而不是按照表内索引顺序进行读取，常见于order by。若出现该值，则应优化SQL语句；

- using temporary，使用临时表缓存中间结果，常见于order by和group by。若出现该值，则应优化SQL；
- using index，表示select操作使用了覆盖索引，避免了访问表的数据行；
- using where，使用了where过滤；
- using join buffer，使用连接缓存；
- distinct，发现第一个匹配后，停止搜索更多的行；

✓ 索引数据结构的选择

可以存储数据的数据结构有很多，mysql为什么只选择B+树作为索引的数据结构呢?

- **hash表**。优点：等值查询速度较快。缺点：比较耗费空间，无法进行范围查询
- **二叉树 / 红黑树**。缺点：因为每个节点只有两个子节点，所以数据量大的时候，树的高度会很高，查询的时间复杂度也变高且IO操作的次数也会变多。
- **B树**：每个节点可以有多个子节点，每个节点都带着数据。每个节点的大小可以是4K的倍数，因为IO操作时会读取连续的空间(磁盘页)，就会把连续的索引读到缓存中，可以减少IO操作。**适合一次查询**，比如非关系型数据库Mongodb。
- **B+树**：每个节点都有多个子节点，每个节点不存放数据只存放索引，叶子节点才存放数据，且叶子节点之间用指针相连，**适合做范围查询**。无论如何都会从根节点走到叶子节点

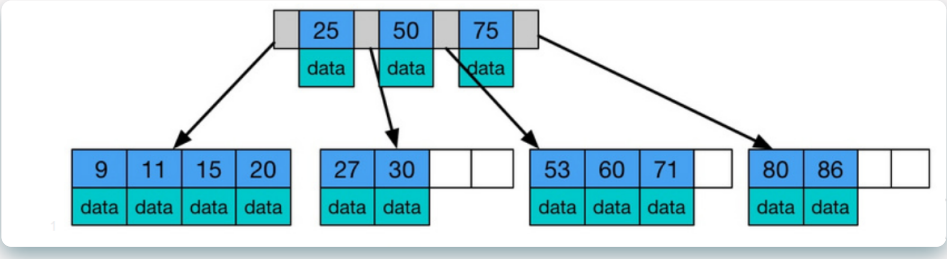
✓ 索引

索引相当于书的目录，可以提高sql的查询效率。

在InnoDB和MyISAM的引擎下，索引和实际的数据都是存储在磁盘的，只不过进行数据读取的时候会**优先把索引加载到内存中**。而对于memory引擎，数据跟索引都是放在内存中。

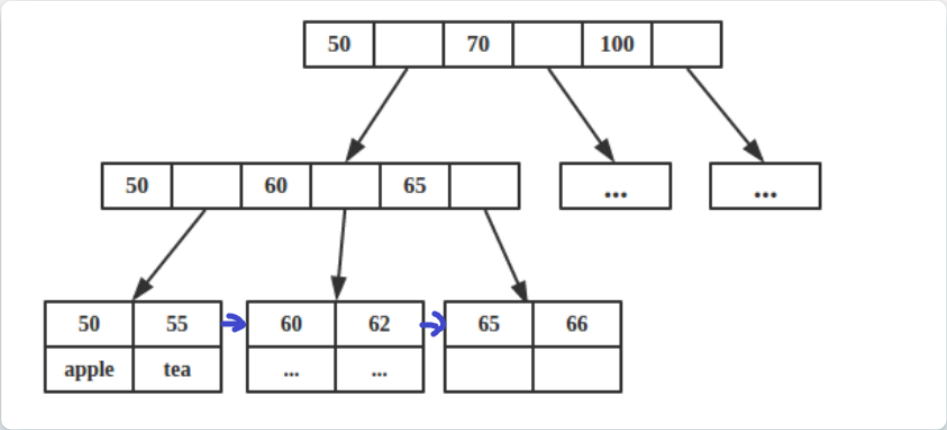
索引底层采用的**数据结构**：`B+树`。3~4层的B+树足以支持千万级别的数据量存储。因为innodb中，B+树的每个节点默认是16K。

B树：类似普通的平衡二叉树，不同的一点是B树允许每个节点有更多的子节点。



B+树是B-树的变体，也是一种多路搜索树, 它与 B- 树的不同之处在于:

1. 所有数据存储在叶子节点，非叶子节点并不存储真正的data，在mysql中非叶子节点存储的都是索引
2. 为所有叶子结点增加了一个双向指针



使用B+树的好处：

1. 因为非叶子节点没有存数据，每个节点能索引的范围更大更精确，可以减少树的高度。
2. B+树的叶子节点两两相连大大增加了区间访问性，可很好的进行范围查询等

索引的设计原则

- **不要盲目的建索引**。索引需要额外的磁盘空间，而维护索引的成本增加。
- **使索引尽可能的小的占用内存空间**。选int还是varchar？因为varchar占用的内存是可变的，而int是固定占用4B，所以当varchar大于4B的时候用int，小于4B的时候用varchar。
- **经常做查询或排序的字段应该建索引**。
- **索引字段应该少做更新删除操作**。因为要维护索引。
- **外键字段最好建索引**。在多表联查中，索引要建立在副表中。

前缀索引

当索引是很长的字符序列时，这个索引将会很占内存，而且会很慢，这时候就会用到前缀索引。前缀索引就是选择索引的前面几个字符作为索引，但是要尽量降低索引的重复率。

1. 首先查看不使用前缀索引时，不重复索引对整个记录的占比。

```
mysql> select 1.0*count(distinct name)/count(*) from test;
+----------------------------------+
| 1.0*count(distinct name)/count(*) |
+----------------------------------+
|                          1.00000 |
+----------------------------------+
1 row in set (0.00 sec)
```

2. 查看截取前**两个**字符作为索引时，不重复索引对整个记录的占比。

```
mysql> select 1.0*count(distinct left(name,2))/count(*) from
test;
+-----------------------------------------+
| 1.0*count(distinct left(name,2))/count(*) |
+-----------------------------------------+
|                                 0.75000 |
+-----------------------------------------+
1 row in set (0.00 sec)
```

3. 查看截取前**三个**字符作为索引时，不重复索引对整个记录的占比。

```
mysql> select 1.0*count(distinct left(name,3))/count(*) from
test;
+-----------------------------------------+
| 1.0*count(distinct left(name,3))/count(*) |
+-----------------------------------------+
|                                 0.75000 |
+-----------------------------------------+
1 row in set (0.00 sec)
```

4. 查看截取前**四个**字符作为索引时，不重复索引对整个记录的占比。

```
mysql> select 1.0*count(distinct left(name,4))/count(*) from test;
+-----------------------------------------+
| 1.0*count(distinct left(name,4))/count(*) |
+-----------------------------------------+
|                                 1.00000 |
+-----------------------------------------+
1 row in set (0.00 sec)
```

5. 查看截取前**五个**字符作为索引时，不重复索引对整个记录的占比。

```
mysql> select 1.0*count(distinct left(name,5))/count(*) from test;
+-----------------------------------------+
| 1.0*count(distinct left(name,5))/count(*) |
+-----------------------------------------+
|                                 1.00000 |
+-----------------------------------------+
1 row in set (0.00 sec)
```

可以看到截取前4个字符的时候就达到了1的占比，所以可以截取前四个字符作为索引。

left 为 字符串截取函数。

创建前缀索引：`alter table 表名 add index(字段名(n));` n为要截取的字符长度

聚簇索引和非聚簇索引

数据和索引存储在一起的叫做聚簇索引，分开存储的叫非聚簇索引。

聚簇索引的叶子节点存放的是数据，而非聚簇索引的叶子节点存放的是聚簇索引的值。（非聚簇索引最后还是要找回聚簇索引）

非聚簇索引的叶子节点不存放数据的好处（为什么要有非聚簇索引）？更新聚簇索引字段的代价很高，当一个数据发生改变时，只维护聚簇索引中的数据即可，而无需更新非聚簇索引。

> innodb存储引擎在进行数据插入的时候，数据必须跟某个索引字段存储在一起，这个字段可以是主键，如果没有主键，选择唯一键，如果没有唯一键，选择6B的隐藏id进行存储。

innodb既有聚簇索引又有非聚簇索引。myisam中只有非聚簇索引。

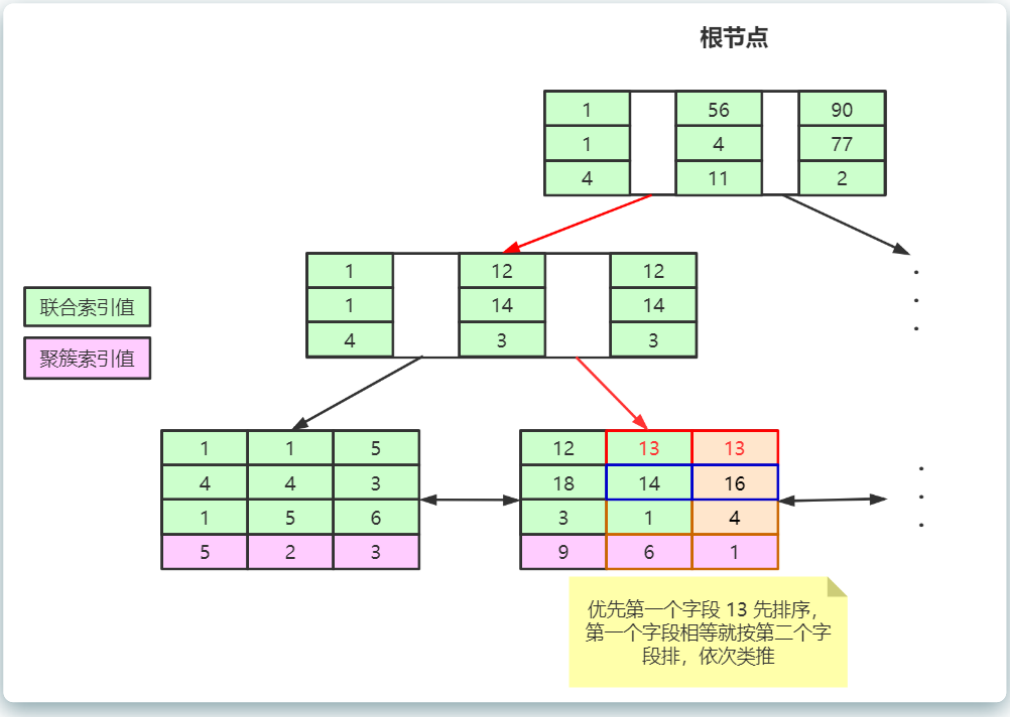
主键索引和非主键索引的区别

1. 主键索引的叶子节点存放着数据，非主键索引的叶子节点存放着主键索引的值（这是在innodb中。如果myisam中，主键索引和普通索引是没有区别的都是直接索引着数据）
2. 进行**主键**查询的话可以**一次性**取到数据，而进行**非主键**查询的话需要进行**回表**才能拿到数据。

组合索引在B+树中是如何存储的

多个列形成一个索引，放入节点中。进行索引查找的时候先对第一个列进行索引查找，然后拿查找出来的索引数据 从 第二列开始往后逐个匹配。

例：`select * from table where b = 13 and c = 16 and d = 4;`



利用第一个索引字段b，找到b=13的索引，然后从找出的索引中继续匹配c=16的索引，最后匹配d=4的索引数据，于是找到该组合索引下的**聚簇索引为1**，再从聚簇索引树上找到最终数据。可以看到只有**第一个索引字段**b是排好序的，剩余的字段都是在前一个字段排好序且相等的情况下再进行排序。

> 组合索引也通常用来**排序**，因为可以省去查出数据后引擎再进行排序的时间。

名词解释

- 回表：先根据非聚簇索引找到聚簇索引的值，然后根据聚簇索引找到数据。
  - id主键，name为普通索引
  - `select * from table where name = ‘zhangsan’;`
- 索引覆盖：要查询的字段全都是索引
  - `select id，name from table where name = ‘zhangsan’;`
- 最左匹配：针对组合索引，只要查询条件与组合索引**从左到右部分字段的顺序**相匹配，该次查询就可以使用组合索引进行搜索。
  - 例如现有联合索引（x,y,z）
  - `WHERE x=1 AND y=2 AND z=3;` 符合最左匹配原则。
  - `WHERE x=1 AND y>2 AND z=3;` 不符合最左匹配原则，但会使用索引x跟y。
  - `WHERE x=1 AND y>2;` 符合最左匹配原则。
  - `WHERE y>2 AND x=1;` 符合最左匹配原则。**优化器为了更好的使用索引，会动态调整字段的顺序**。
  - `WHERE y>2 AND z=3;` 不符合最左匹配原则，无法使用组合索引
  - `WHERE x=1 AND z=3;` 不符合最左匹配原则，无法使用组合索引。但**会先使用索引x找到x=1的数据**，然后再在找到的数据中逐个匹配z=3的数据。
  - `WHERE x=1 AND z=3 order by y;` 不符合最左匹配原则，因为y不是用于查找。
  - `where x=1 and y like ‘%k’ and z=3` 不符合最左匹配原则,但会使用索引x，此时查询的类型是ref
  - `where x=1 and y like ‘k%k’ and z=3` 符合最左匹配原则，此时查询的类型是range
  - `where x=1 and y like ‘kk%’ and z=3` 符合最左匹配原则，此时查询的类型是range

    - `where x=1 and y like ‘kk%’ and z in (3, 5)` 符合最左匹配原则，此时查询的类型是range

- 索引下推：利用出现在where中的组合索引字段都筛选完之后才返回到server层。

    - `select * from user where name like '张%' and age=18 and ismale=1;` 其中(name, age)是组合索引。
    - 使用索引下推时，会先找到 name以张开头的数据，然后再匹配age=18的数据，最后将最后匹配到的数据返回到server，在server层中再筛选出 ismale=1的数据
    - 不使用索引下推时，找到 name以张开头 的数据后直接返回到server层，然后再在server层做数据的筛选，这样会将冗余的数据都拷贝到server层。

> server层是一个进程。

索引失效

1. 模糊查询以%开头
2. 索引列上使用 is null 或is not null
3. 对索引列进行运算
4. 表中数据量很小

✓ 慢查询日志

慢查询日志：是一个日志记录。当一个sql的查询时间超过了一个阈值 `long_query_time` 时，该sql语句会被记录到慢查询日志中。**long_query_time默认为10秒**。

查看慢查询日志状态：`show variables like ‘%slow_query_log%’;` （默认是关闭的）

打开慢查询日志：`set global slow_query_log = 1;` （只针对当前数据库生效，重启mysql后又自动关闭了）

查看阈值：`show variables like ‘%long_query_time%’;`

设置阈值：`set global long_query_time = n;` 需要重新打开窗口登录mysql生效。

✓ 数据库调优

平时在项目中有做过一些优化，首先是数据库设计的时候会先考虑到优化问题，比如表的字段类型、长度、各表之间的关系，以及创建合适的索引。其次是在开发过程中出现sql问题后的优化，一般是查询时间慢，在这个阶段需要查看慢查询日志以及explain的信息，然后调整sql语句，比如索引的创建或修改。

比如 `select id, name, sex where name = 'zhangsan' ;` 可将name跟sex组成联合索引，来避免回表。

✓ MVCC

MVCC(Mutil-Version Concurrency Control)，就是多版本并发控制。MVCC是一种并发控制的方法，实现对数据库的并发访问。主要适用于Mysql的RC, RR隔离级别。

好处：提高数据库并发性能，使用不加锁，非阻塞方式去处理读-写冲突。

> MVCC解决了不可重复读。MVCC+临建锁解决了幻读。

特点：

1. 每行数据都存在一个版本，每次数据更新时都会更新版本号。
2. 修改时Copy出当前版本到当前事务，各个事务之间无干扰。
3. 保存时比较版本号，如果成功，则覆盖原记录（commit）；失败则放弃覆盖（rollback）

当前读和快照读

当前读：读取的一定是数据的最新版本，读取时还要保证其他并发事务不能修改当前数据，会对读取的数据进行加锁。

- 像 `select ... lock in share mode` (共享锁)、 `select ... for update` 、 `update` 、 `insert` 、 `delete` (排他锁) 这些操作都是当前读

快照读：读取的不一定是数据的最新版本，可能是历史版本的记录。比如：不加锁的select操作。

- 快照读前提是隔离级别小于串行级别，串行级别下的快照读会退化成当前读；

实现原理

它的实现原理主要是依赖记录中的 **3个隐藏字段**，**undolog** ， **ReadView** 来通过保存数据在某个时间点的快照。

隐藏字段

每行记录上都会包含几个用户不可见的字段：**DB_TEX_ID**、**DB_ROW_ID**、**DB_ROLL_PTR**。

- DB_TRX_ID：创建/最后一次修改该记录的事务ID
- DB_ROW_ID：隐含的自增ID，如果数据表没有主键，InnoDB会自动以DB_ROW_ID产生一个聚簇索引。
- DB_ROLL_PTR：回滚指针，指向这条记录的上一个版本。配合undolog使用。

undolog

存放每行数据的版本链。链首是最新的历史记录。

执行流程如下：

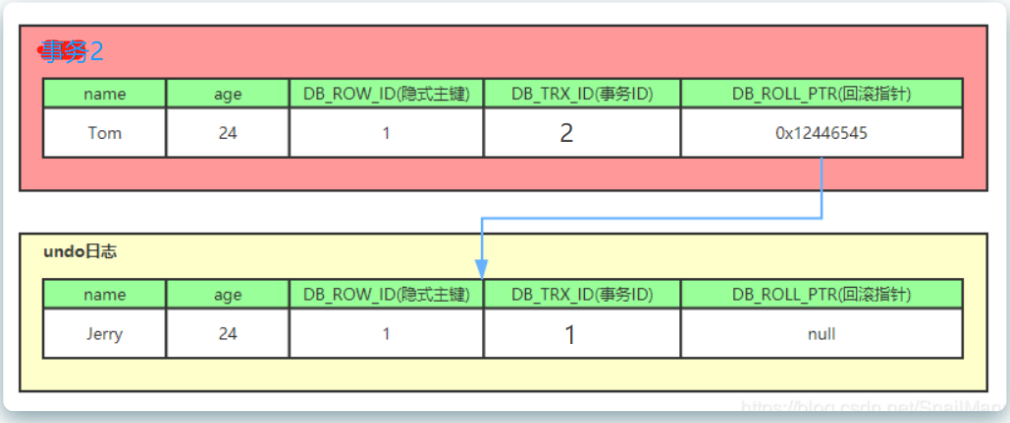
【1】有个事务1在persion表插入了一条新记录：name为Jerry, age为24岁，隐式主键是1，事务ID假设为1，回滚指针因为是第一条新记录所以为NULL。

person表的某条记录

name	age	DB_ROW_ID(隐式主键)	DB_TRX_ID(事务ID)	DB_ROLL_PTR(回滚指针)
Jerry	24	1	1	null

【2】现在来了一个事务2对该记录的name做出了修改，改为Tom

- 在事务2修改该行数据时，数据库会先对该行加排他锁。
- 然后把该行数据拷贝到undolog中作为旧记录，即在undolog中有当前行的拷贝副本
- 拷贝完毕后，修改该行name为Tom，并且修改隐藏字段的事务ID为当前事务2的ID, 假设为2。将该行的回滚指针指向undolog中的最新的副本记录。
- 事务提交后，释放锁



> undolog里面的内容是二进制的，这里为了方便理解写成记录的形式。
>
> 并且undolog不仅可以配合MVCC使用，其本身还会作为逻辑日志，保证原子性。

Read View

Read View就是事务进行快照读操作的时候生成的读视图(Read View)，在该事务执行快照读的那一刻，会生成数据库系统当前的一个快照，记录并维护系统当前活跃事务的ID(当每个事务开启时，都会被分配一个ID, 这个ID是递增的，所以最新的事务，ID值越大)

当我们某个事务执行快照读的时候，对该记录创建一个Read View读视图，用它来判断当前事务能够看到哪个版本的数据。即可能是当前最新的数据，也有可能是该行记录的undolog里面的某个版本的数据。

Read View可简化成下面三部分：

- **trx_list**：Read View生成时系统正活跃的事务ID的集合
- **up_limit_id**：记录trx_list列表中事务最小的ID
- **low_limit_id**：ReadView生成时刻系统尚未分配的下一个事务ID，也就是目前已出现过的事务ID的最大值+1

如何知道ReadView展示的是哪个版本的数据呢？需要遵循一个可见性算法。

1. 首先比较 `DB_TRX_ID < up_limit_id`，如果小于，则当前事务能看到DB_TRX_ID所在的记录，如果大于等于进入下一个判断
2. 接下来判断 `DB_TRX_ID ≥ low_limit_id` ，如果大于等于则代表DB_TRX_ID 所在的记录是在Read View生成后才出现的，那对当前事务肯定不可见。如果小于则进入下一个判断
3. 判断 `DB_TRX_ID 是否在活跃事务之中` ，如果在，则代表Read View生成时刻，这个事务还在活跃，还没有Commit，即修改的数据当前事务也是看不见的；如果不在，则说明，这个事务在Read View生成之前就已经Commit了，即修改的结果当前事务是能看见的

把要被修改的数据的最新记录中的DB_TRX_ID（当前事务ID）取出来，与系统当前其他活跃事务的ID去对比（由Read View维护），如果DB_TRX_ID跟Read View的属性做了某些比较，不符合可见性，那就通过DB_ROLL_PTR回滚指针去取出UndoLog中的DB_TRX_ID再比较，即遍历UndoLog的DB_TRX_ID，直到找到满足特定条件的DB_TRX_ID, 那么这个DB_TRX_ID所在的旧记录就是当前事务能看见的内容。

什么时候会进行会形成读视图？每次快照读都会形成一个读视图吗？

答：RC隔离级别下，每次快照读都会生成一个readview；RR隔离级别下，第一次快照读时会生成一个readview，之后的读操作都是使用这个readview，直到当前事务结束。但是，如果在事务的中途执行了当前读（增删改或加锁的读），则会出现幻读，即之前的readview失效。

✓ 事务的四大特性是如何实现的?

原子性：通过undolog实现，事务执行失败之后会通过undolog回滚到之前的状态。例如delete一条记录，undolog就记录一条insert。

一致性：其他三个特性如果实现了，该特性也就实现了。

隔离性：利用MVCC和锁 机制。

持久性：通过redolog实现，redolog会记录数据库的每一个写操作。当数据库空闲的时候会将redolog的内容更新到数据库中，然后擦除redolog的内容。

✓ 锁的类别

基于锁的属性分类：共享锁（读锁）、排他锁（写锁）。

基于锁的粒度分类：表锁、行锁、记录锁、间隙锁、临建锁。

基于锁的状态分类：意向共享锁、意向排他锁。

- **共享锁**（Share Lock）：共享锁也叫读锁，简称S锁。当一个事务为数据加上读锁之后，其他事务只能对该数据加读锁，而不能对其加写锁。

```
select ... lock in share mode
```

- **排他锁**（exclusive Lock）：排他锁又叫写锁，简称X锁。当一个事务为数据加上写锁后，其他事务不能加任何的锁。innoDB引擎中默认对update,delete,insert加了排他锁，select语句默认不加锁。

```
select ... for update
```

- **表锁**：对整个表的数据进行上锁。
- **行锁**：对某一行或多行记录进行上锁。
  - for update是InnoDB默认的行级别的锁。
- **记录锁**：行锁的特殊化，即只锁一行。
  - 例如使用主键或唯一索引来搜索并给这一行记录加锁。

```
SELECT * FROM child WHERE id = 100 FOR UPDATE;
```

- **间隙锁**：锁住的是一个区间内的记录，即使不存在。只出现在RR级别中。

```
会锁住(5,7)的记录。
SELECT * FROM `test` WHERE `id` BETWEEN 5 AND 7 FOR UPDATE
```

- **临建锁**（Next-Key Lock）：是记录锁与间隙锁组合，是innodb行锁的默认算法。它锁住的范围是左开右闭，并且查询的条件等于右边界的时候会将下一个区间也锁住。可以防止幻读 。
- **意向共享锁 / 意向排他锁**：都是表级的锁，在进行 共享/排他锁 之前会获取这个表的意向共享锁 / 意向排他锁，来告诉其他事务此表已经被加锁了。

✓ RC和RR级别下锁的情况

RC：只会对查询出来的记录加写锁，即无法对这些记录进行查询或update，但还可以进行insert操作。

RR：为了解决幻读，引入了间隙锁。

- 根据**非索引**查询数据时，会锁住表里所有的记录及其之间的间隙。（并非表锁）
- 根据**非唯一索引**查询数据时，只会锁住查出来的记录以及之间的间隙。
- 根据**唯一索引**查询数据时，会对该行记录加记录锁。

✓ 不可重复读和幻读

不可重复读：当前事务正在执行的时候，其他事务**修改或新增**了一条数据，当前事务就会查询到修改或新增的记录。

- 解决：InnoDB 中，RR隔离级别使用 MVCC 来解决不可重复读问题。

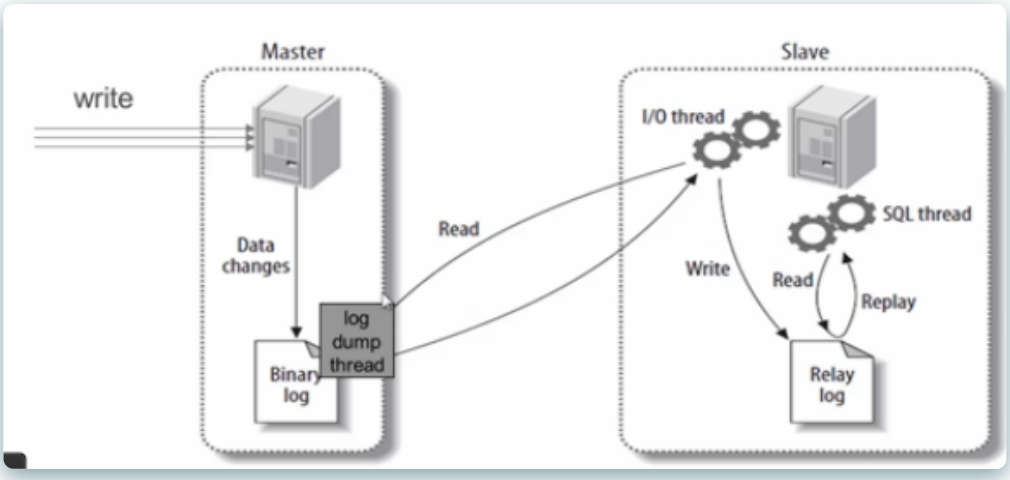
幻读：当前事务正在执行的时候，其他事务**新增**了一条数据，然后当前事务进行查询的时候就会查询到新增的记录。

- 解决：对于快照读，MVCC可以解决幻读。对于当前读来说，**MVCC+临建锁**才能解决幻读。
- 因为mysql中默认开启了临建锁，所以mysql的RR级别下可以解决幻读。

✓ 主从复制

主机master数据更新后根据配置和策略自动同步到从机slaver，**Master**以**写**为主，**Slave**以**读**为主。mysql默认采用异步复制的方式，这样从节点就不用一直访问主服务器来更新自己的数据。

原理如下：



- master将数据的增删改的sql语句 **记录在二进制文件日志binlog**中
- slave会在一定时间间隔内对master的binlog进行探测，看其是否发送改变。如果发生改变则**使用一个IO线程**去获取binlog。同时master为**每个从机的IO线程开启一个dump线程**，用于传输binlog。
- slave获取到binlog后会将其内容写到本地的 **relay log** 中，然后**使用一个sql线程** 读取relay log中的内容并执行，使本地数据跟 master的数据保持一致。
- 最后slave的 IO线程 跟 sql线程 进入睡眠状态，等待下一次被唤醒。

✓ 百万级别或以上的数据如何删除

因为索引的维护很耗费时间，所以可以先删除索引，然后再删除数据。如果先删数据的话，则每次删一条就要维护一次索引，这样会增加不必要的时间。

✓ 为什么要使用视图？什么是视图？

视图是一个虚拟表，是动态生成的，它只存放视图的定义，不存放数据。因此可以**提高复杂sql语句的复用性和数据的安全性**。

特点：

1. 视图的建立和删除不影响基本表
2. 对视图内容的更新直接影响基本表
3. 当视图来自多个基本表时，不允许添加和删除数据。