MySQL

事务的四大特性

事务特性ACID：

- **原子性**（ `Atomicity` ）：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。
- **一致性**（ `Consistency` ）：指一个事务执行之前和执行之后 都必须处于一致性状态。比如a与b账户共有1000块，两人之间转账之后无论成功还是失败，它们的账户总和还是1000。
- **隔离性**（ `Isolation` ）：跟隔离级别相关，如 `read committed` ，一个事务只能读到已经提交的修改。
- **持久性**（ `Durability` ）：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。

数据库的三大范式

第一范式 1NF

确保数据库表字段的原子性。

比如字段 `userInfo：广东省 10086` ，依照第一范式必须拆分成 `userInfo：广东省` `userTel：10086` 两个字段。

第二范式 2NF

首先要满足第一范式，另外包含两部分内容，一是表必须有一个主键；二是非主键列必须完全依赖于主键，而不能只依赖于主键的一部分。

举个栗子：
假定选课关系表为 `student_course`（student_no, student_name, age, course_name, grade, credit),主键为(student_no, course_name)。其中学分完全依赖于课程名称，姓名年龄完全依赖学号，不符合第二范式，会导致数据冗余（学生选 n 门课，姓名年龄有 n 条记录）、插入异常（插入一门新课，因为没有学号，无法保存新课记录）等问题。

应该拆分成三个表：
学生表（ `student`（student_no, student_name, age)）
课程表（ `course`（course_name, credit)）
选课关系表（ `student_course`（student_no, course_name, grade)）

第三范式 **3NF**

首先要满足第二范式，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

栗子：
假定学生关系表为 `student` (student_no, student_name, age, academy_id, academy_telephone)，主键为“学号”，其中学院id依赖于学号，而学院地点和学院电话依赖于学院id，存在传递依赖，不符合第三范式。

可以把学生关系表分为如下两个表：
学生表（ `student` (student_no, student_name, age, academy_id)）
学院表（ `academy` (academy_id, academy_telephont)）

事务隔离级别有哪些

需知：脏读、不可重复读、幻读。

- **脏读**：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。
- **不可重复读**：在对于数据库中的某行记录，一个事务范围内多次查询去返回了不同的数据值，这是由于在查询间隔，另一个事务修改了数据并提交了。
- **幻读**：当某个事务内在读取操作的结论不能支撑之后业务的执行。假设事务要新增一条记录，主键为id，在新增之前执行了select，没有发现id为xxx的记录，但插入时出现主键冲突，这就属于幻读，读取不到记录却发现主键冲突是因为记录实际上已经被其他的事务插入了，但当前事务不可见。

不可重复读和脏读的区别：脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

事务隔离就是为了解决上面提到的脏读、不可重复读、幻读这几个问题。

MySQL数据库为我们提供的四种隔离级别：

- **Serializable**（串行化）：通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。
- **Pepeatable read**（可重复读）：MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行，解决了不可重复读的问题。
- **Read committed**：（读已提交）：一个事务只能看见已经提交事务所做的改变。可避免脏读的发生。
- **Read uncommitted**（读未提交）：所有事务都可以看到其他未提交事务的执行结果。

查看隔离级别：

```
select @@transaction_isolation;
```

设置隔离级别：

```
set session transaction isolation level read uncommitted;
```

生产环境数据库一般用的隔离级别

生产环境大多使用 RC。

为什么不是 PR 呢？

> 可重复读（Repeatable Read），简称为 PR 读已提交（Read Committed），简称为 RC

缘由一：在RR隔离级别下，存在间隙锁，导致出现死锁的几率比RC大的多！
缘由二：在RR隔离级别下，条件列未命中索引会锁表！而在RC隔离级别下，只锁行！

也就是说，RC 的并发性高于 RR。

并且大部分场景下，不可重复读问题是可以接受的。毕竟数据都已经提交了，读出来本身就没有太大问题！

> **问**：互联网项目中MySQL应该选什么事务隔离级别？
> **答**：*读已提交（Read Committed）*。
> MySQL默认的事务隔离级别是**可重复读(Repeatable Read)**
> **思考**：Oracle, SqlServer中都是选择 读已提交（Read Committed）作为默认的隔离级别，而MySQL为什么选择 可重复读(Repeatable Read)作为默认隔离级别呢？
> 这里要说说到 **主从复制** ：基于 binlog 复制的
> binlog是一个记录数据库更改的文件，它有三种格式：
>
> - statement：记录的是修改SQL语句
> - row：记录的是每行实际数据的变更
> - mixed：statement和row模式的混合
>
> MySQL在5.0这个版本以前，binlog只支持 `STATEMENT` 这种格式。而这种格式在**读已提交（Read Committed）**这个隔离级别下 主从复制 是有bug的，所有MySQL将 **可重复读(Repeatable Read)**作为默认的隔离级别。那这个 bug 是什么呢？
> 在主（master），执行下列语句
>
> ```
> select * from test;
> ```
>
> 输出如下：

```
+----+
| id |
+----+
| 10 |
+----+
1 row in set (0.03 sec)
```

在 从（slave）上执行该语句，则输出结果如下：

```
Empty set
```

这里就出现了 主从 不一致性的问题！原因很简单，就是在master上执行的顺序为**先删后插**，这个时候 binlog 是 `STATEMENT` 格式，它记录的顺序是**先插后删** 。从(slave)同步的是 binlog，因此从机执行的顺序和主机不一致！就会出现主从不一致。这里解决方案有两种：

- 隔离级别设为**可重复读(Repeatable Read)**，在该隔离级别下引入间隙锁（*为了解决幻读*）。当 `Session 1` 执行 deleted 语句时，会锁住间隙，则 `Session 2` 执行插入语句就会阻塞住。
- 将binlog的格式修改为 row 格式，此时是基于行的复制，自然就不会出现sql执行顺序不一样的问题。不过这个格式在 mysql 5.1版开始才引入。

由于上面历史原因，mysql将默认的隔离级别设为**可重复读(Repeatable Read)**，保证主从复制不出问题。

项目不用**读未提交(Read UnCommitted)和串行化(Serializable)**两个隔离级别，原因有二：

- 采用**读未提交(Read UnCommitted)**，一个事务读到另一个事务未提交读数据。
- 采用**串行化(Serializable)**，每个次读操作都会加锁，快照读失效，一般是使用mysql自带分布式事务功能时才使用该隔离级别。
- mysql自带的功能是XA事务，是强一致性事务，性能不佳。互联网的分布式方案多采用最终一致性的事务解决方案。

为什么选 `读已提交(Read Committed)` 作为事务隔离级别？
栗子:
创建一个表 test

```
CREATE TABLE `test` (`id` int(11) NOT NULL,`name` varchar(20) NOT NULL,PRIMARY KEY (`id`)) ENGINE=InnoDB;
```

数据如下：

```
+----+------+
| id | name |
+----+------+
|  1 | yyy  |
|  2 | xxx  |
|  5 | yyy  |
|  7 | xxx  |
+----+------+
4 rows in set (0.03 sec)
```

缘由一：在*可重复读(Repeatable Read)*隔离级别下，存在间隙锁，导致出现死锁的几率比*读已提交(Read Committed)*大的多，此时执行语句

```
select * from test where id<3 for update;
```

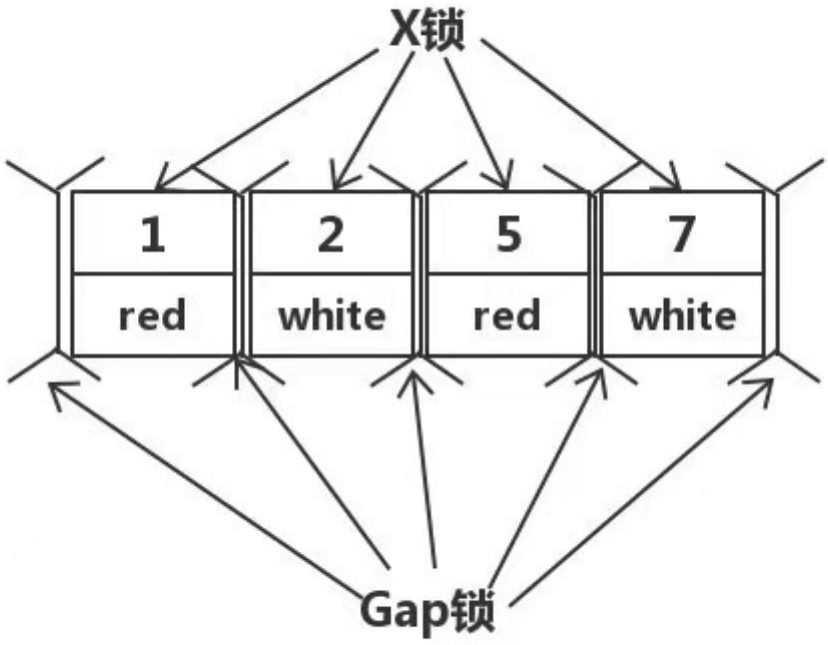
在*可重复读(Repeatable Read)*级别下，存在间隙锁，可以锁住(2,3)这个间隙，防止其他事务插入数据。而在*读已提交(Read Committed)*隔离级别下，不存在间隙锁，其他事务是可以插入数据。

ps：在*读已提交(Read Committed)*隔离级别下并不是不会出现死锁，只是出现几率比*可重复读(Repeatable Read)*低。

缘由二：在*可重复读(Repeatable Read)*隔离级别下，条件列未命中索引会锁表。而在*读已提交(Read Committed)*隔离级别下，只锁行此时执行语句

```
update test set name = 'qqq' where color = 'xxx';
```

在*读已提交(Read Committed)*隔离级别下，其先走聚簇索引，进行全部扫描(1,2,5,7)都有加锁；但在实际中，MySQL做了优化，在MySQL Server过滤条件，发现不满足后，会调用 `unlock_row` 方法，把不满足条件的记录放锁，实际就只会锁(2和7)。
而在 *可重复读(Repeatable Read)* 隔离级别下，走聚簇索引，进行全部扫描，最后会将束个表锁上，如图：



缘由三：在*读已提交(Read Committed)*隔离级别下，半一致性读(semi-consistent)特性增加了update操作的并发性！在5.1.15的时候，innodb引入一个概念叫做“semi-consistent”，减少了更新同一行记录时的冲突，减少锁等待。所谓半一致性：一个update语句，如果读到一行已经加锁的记录，此时InnoDB返回记录最近提交的版本，由MySQL上层判断版本是否满足update的where条件。若满足（需要更新），则MySQL会重新发现一次读操作，此时读取行的最新版本（并加锁）。

栗子：此时有两个Session，Session1和Session2
Session1执行

```
update test set name = 'yyyy' where name = 'yyy'
```

先不Commit事务，与此同时Session2执行

```
update test set name= 'xxxx' where name = 'xxx'
```

session 2尝试加锁的时候，发现行上已经存在锁，InnoDB会开启semi-consistent read，返回最新的committed版本(1,yyy),(2,xxx),(3,yyy),(4,xxx)。MySQL会重新发起一次读操作，此时会读取行的最新版本（并加锁），而*可重复读(Repeatable Read)*隔离级别下，Session2只有等待。

> 其中有两个疑问
> *在 读已提交(Read Committed)级别下，不可重复读问题需要解决么？*
> *不用解决，这个问题是可以接受。因为数据已经提交，读出来本身就没有太大问题。*
> *Oracle的默认隔离级别就是读已提交(Read Committed)*。
>
> 在*读已提交(Read Committed)*级别下，主从复制用什么binlog格式？
> 用得是 row 格式，是基于行的复制。InnoDB的创始人也是建议binlog使用该格式。
>
> 详细说明见原文：https://zhuanlan.zhihu.com/p/59061106

编码和字符集的关系

我们平时可以在编辑器上输入各种中文英文字母，但这些都是给人读的，不是给计算机读的，其实计算机真正保存和传输数据都是以**二进制010101**的格式进行的。

那么就需要有一个规则，把中文和英文字母转化为二进制。其中d对应十六进制下的64，它可以转换为01二进制的格式。于是字母和数字就这样一一对应起来了，这就是ASCII编码格式。

它用**一个字节**，也就是 8位 来标识字符，基础符号有128个，扩展符号也是128个。也就只能表示下英文字母和数字。

这明显不够用。于是，为了标识**中文**，出现了**GB2312**的编码格式。为了标识**希腊语**，出现了**greek**编码格式，为了标识**俄语**，整个**cp866**编码格式。

为了统一它们，于是出现了**Unicode编码格式**，它用了2~4个字节来表示字符，这样理论上所有符号都能被收录进去，并且它还完全兼容ASCII的编码，也就是说，同样的字母d，在ASCII用64表示，在Unicode里还是用64来表示。

在不同的地方是ASCII编码用1个字节来表示，而Unicode用则两个字节来表示。

同样都是字母d，Unicode比ASCII多使用了一个字节，如下：

```
D ASCII:                01100100
D Unicode:  00000000    01100100
```

可以看到，上面的unicode编码，前面的都是0，其实用不上，但还是占了字节，有点浪费。如果我们能做到该隐藏时隐藏，这样就能省下不少空间，按这个思路，就有了UTF-8编码。

总结：

- 编码：按照一定规则把符号和二进制码对应起来
- 字符集：把n多这种已经编码的字符聚在一起

utf8和utf8mb4的区别

mysql支持的字符集中有 utf8 和 utf8mb4。

utf8mb4编码中的 `mb4` 是 `most bytes 4` 的意思。它最大支持用**4个字节**来表示字符，它几乎可以用来表示目前已知的所有的字符。

mysql字符集里的**utf8**，它是数据库的默认字符集。这里utf8最多支持用**3个字节**去表示字符，按utf8mb4的命名方式，可以叫它**utf8mb3**。

`utf8` 就像是阉割版的utf8mb4，只支持部分字符。比如 `emoji` 表情，它就不支持。

```
mysql> show charset;
+----------+---------------------------------+---------------------+--------+
| Charset  | Description                     | Default collation   | Maxlen |
+----------+---------------------------------+---------------------+--------+
| armscii8 | ARMSCII-8 Armenian              | armscii8_general_ci |      1 |
| ascii    | US ASCII                        | ascii_general_ci    |      1 |
| big5     | Big5 Traditional Chinese        | big5_chinese_ci     |      2 |
| binary   | Binary pseudo charset           | binary              |      1 |
| cp1250   | Windows Central European        | cp1250_general_ci   |      1 |
| cp1251   | Windows Cyrillic                | cp1251_general_ci   |      1 |
| cp1256   | Windows Arabic                  | cp1256_general_ci   |      1 |
| cp1257   | Windows Baltic                  | cp1257_general_ci   |      1 |
| cp850    | DOS West European               | cp850_general_ci    |      1 |
| cp852    | DOS Central European            | cp852_general_ci    |      1 |
| cp866    | DOS Russian                     | cp866_general_ci    |      1 |
| cp932    | SJIS for Windows Japanese       | cp932_japanese_ci   |      2 |
| dec8     | DEC West European               | dec8_swedish_ci     |      1 |
| eucjpms  | UJIS for Windows Japanese       | eucjpms_japanese_ci |      3 |
| euckr    | EUC-KR Korean                   | euckr_korean_ci     |      2 |
| gb18030  | China National Standard GB18030 | gb18030_chinese_ci  |      4 |
| gb2312   | GB2312 Simplified Chinese       | gb2312_chinese_ci   |      2 |
| gbk      | GBK Simplified Chinese          | gbk_chinese_ci      |      2 |
| geostd8  | GEOSTD8 Georgian                | geostd8_general_ci  |      1 |
| greek    | ISO 8859-7 Greek                | greek_general_ci    |      1 |
| hebrew   | ISO 8859-8 Hebrew               | hebrew_general_ci   |      1 |
| hp8      | HP West European                | hp8_english_ci      |      1 |
| keybcs2  | DOS Kamenicky Czech-Slovak      | keybcs2_general_ci  |      1 |
| koi8r    | KOI8-R Relcom Russian           | koi8r_general_ci    |      1 |
| koi8u    | KOI8-U Ukrainian                | koi8u_general_ci    |      1 |
| latin1   | cp1252 West European            | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European     | latin2_general_ci   |      1 |
| latin5   | ISO 8859-9 Turkish              | latin5_turkish_ci   |      1 |
| latin7   | ISO 8859-13 Baltic              | latin7_general_ci   |      1 |
| macce    | Mac Central European            | macce_general_ci    |      1 |
| macroman | Mac West European               | macroman_general_ci |      1 |
| sjis     | Shift-JIS Japanese              | sjis_japanese_ci    |      2 |
| swe7     | 7bit Swedish                    | swe7_swedish_ci     |      1 |
| tis620   | TIS620 Thai                     | tis620_thai_ci      |      1 |
| ucs2     | UCS-2 Unicode                   | ucs2_general_ci     |      2 |
| ujis     | EUC-JP Japanese                 | ujis_japanese_ci    |      3 |
| utf16    | UTF-16 Unicode                  | utf16_general_ci    |      4 |
| utf16le  | UTF-16LE Unicode                | utf16le_general_ci  |      4 |
| utf32    | UTF-32 Unicode                  | utf32_general_ci    |      4 |
| utf8     | UTF-8 Unicode                   | utf8_general_ci     |      3 |
| utf8mb4  | UTF-8 Unicode                   | utf8mb4_0900_ai_ci  |      4 |
+----------+---------------------------------+---------------------+--------+
41 rows in set (0.08 sec)
```

而mysql支持的字符集里(上图)，第三列 **collation**：字符集的比较规则。

比如，“debug”和“Debug”是同一个单词，但它们大小写不同，该不该判为不同单词呢，这里就需要用到 **collation**:

通过 `SHOW COLLATION WHERE Charset = 'utf8mb4';` 可以查看到 `utf8mb4` 下支持什么比较规则。

```
mysql> SHOW COLLATION WHERE Charset = 'utf8mb4';
+----------------------------+---------+-----+---------+----------+---------+---------------+
| Collation                  | Charset | Id  | Default | Compiled | Sortlen | Pad_attribute |
+----------------------------+---------+-----+---------+----------+---------+---------------+
| utf8mb4_0900_ai_ci         | utf8mb4 | 255 | Yes     | Yes      |       0 | NO PAD        |
| utf8mb4_0900_as_ci         | utf8mb4 | 305 |         | Yes      |       0 | NO PAD        |
| utf8mb4_0900_as_cs         | utf8mb4 | 278 |         | Yes      |       0 | NO PAD        |
| utf8mb4_0900_bin           | utf8mb4 | 309 |         | Yes      |       1 | NO PAD        |
| utf8mb4_bin                | utf8mb4 |  46 |         | Yes      |       1 | PAD SPACE     |
| utf8mb4_croatian_ci        | utf8mb4 | 245 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_cs_0900_ai_ci      | utf8mb4 | 266 |         | Yes      |       0 | NO PAD        |
| utf8mb4_cs_0900_as_cs      | utf8mb4 | 289 |         | Yes      |       0 | NO PAD        |
| utf8mb4_czech_ci           | utf8mb4 | 234 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_danish_ci          | utf8mb4 | 235 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_da_0900_ai_ci      | utf8mb4 | 267 |         | Yes      |       0 | NO PAD        |
| utf8mb4_da_0900_as_cs      | utf8mb4 | 290 |         | Yes      |       0 | NO PAD        |
| utf8mb4_de_pb_0900_ai_ci   | utf8mb4 | 256 |         | Yes      |       0 | NO PAD        |
| utf8mb4_de_pb_0900_as_cs   | utf8mb4 | 279 |         | Yes      |       0 | NO PAD        |
| utf8mb4_eo_0900_ai_ci      | utf8mb4 | 273 |         | Yes      |       0 | NO PAD        |
| utf8mb4_eo_0900_as_cs      | utf8mb4 | 296 |         | Yes      |       0 | NO PAD        |
| utf8mb4_esperanto_ci       | utf8mb4 | 241 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_estonian_ci        | utf8mb4 | 230 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_es_0900_ai_ci      | utf8mb4 | 263 |         | Yes      |       0 | NO PAD        |
| utf8mb4_es_0900_as_cs      | utf8mb4 | 286 |         | Yes      |       0 | NO PAD        |
| utf8mb4_es_trad_0900_ai_ci | utf8mb4 | 270 |         | Yes      |       0 | NO PAD        |
| utf8mb4_es_trad_0900_as_cs | utf8mb4 | 293 |         | Yes      |       0 | NO PAD        |
| utf8mb4_et_0900_ai_ci      | utf8mb4 | 262 |         | Yes      |       0 | NO PAD        |
| utf8mb4_et_0900_as_cs      | utf8mb4 | 285 |         | Yes      |       0 | NO PAD        |
| utf8mb4_general_ci         | utf8mb4 |  45 |         | Yes      |       1 | PAD SPACE     |
| utf8mb4_german2_ci         | utf8mb4 | 244 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_hr_0900_ai_ci      | utf8mb4 | 275 |         | Yes      |       0 | NO PAD        |
| utf8mb4_hr_0900_as_cs      | utf8mb4 | 298 |         | Yes      |       0 | NO PAD        |
| utf8mb4_hungarian_ci       | utf8mb4 | 242 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_hu_0900_ai_ci      | utf8mb4 | 274 |         | Yes      |       0 | NO PAD        |
| utf8mb4_hu_0900_as_cs      | utf8mb4 | 297 |         | Yes      |       0 | NO PAD        |
| utf8mb4_icelandic_ci       | utf8mb4 | 225 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_is_0900_ai_ci      | utf8mb4 | 257 |         | Yes      |       0 | NO PAD        |
| utf8mb4_is_0900_as_cs      | utf8mb4 | 280 |         | Yes      |       0 | NO PAD        |
| utf8mb4_ja_0900_as_cs      | utf8mb4 | 303 |         | Yes      |       0 | NO PAD        |
| utf8mb4_ja_0900_as_cs_ks   | utf8mb4 | 304 |         | Yes      |      24 | NO PAD        |
| utf8mb4_latvian_ci         | utf8mb4 | 226 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_la_0900_ai_ci      | utf8mb4 | 271 |         | Yes      |       0 | NO PAD        |
| utf8mb4_la_0900_as_cs      | utf8mb4 | 294 |         | Yes      |       0 | NO PAD        |
| utf8mb4_lithuanian_ci      | utf8mb4 | 236 |         | Yes      |       8 | PAD SPACE     |
| utf8mb4_lt_0900_ai_ci      | utf8mb4 | 268 |         | Yes      |       0 | NO PAD        |
| utf8mb4_lt_0900_as_cs      | utf8mb4 | 291 |         | Yes      |       0 | NO PAD        |
| utf8mb4_lv_0900_ai_ci      | utf8mb4 | 258 |         | Yes      |       0 | NO PAD        |
```

如果 `collation = utf8mb4_general_ci` 是指使用 utf8mb4字符集的前提下，**挨个字符进行比较**（ `general` ），并且不区分大小写（ `_ci, case insensitice` ）。

这种情况下，“debug”和“Debug”是同一个单词。

如果改成 `collation=utf8mb4_bin` ，就是**挨个比较二进制位大小**。

于是“debug”和“Debug”就是不同一个单词。

那 utf8mb4 对比 utf8 有什么劣势吗？

在数据库表中，字段类型如果是 `char(2)` 的话，里面的 `2` 是指**字符个数**，也就是说**不管这张表用的是什么编码字符集**，都能放上2个字符。

而char又是**固定长度**，为了能放下2个utf8mb4的字符，char会默认保留 `2*4 (maxlen=4) = 8` 个字节的空间。

如果utf8mb3，则会默认保留 `2*3 (maxlen=3) = 6` 个字节的空间。也就是说，在这种情况下，utf8mb4会比utf8mb3多使用一些空间。

索引

什么是索引？

索引是存引擎用于提高数据库表的访问速度的一种**数据结构**。

索引的优缺点？

优点：

- **加快数据查找的速度**
- 为用来排序或者是分组的字段添加索引，可以加快分组的排序的速度
- 加快表与表之间的连接

缺点：

- 建立索引需要**占用物理空间**
- 会降低表的增删改的效率，因为每次对表记录进行增删改，需要进行**动态维护索引**，导致增删改时间变长

索引的作用？

数据是存储在磁盘上的，查询数据时，如果没有索引，会加载所有的数据到内存，依次进行检索，读取磁盘次数较多。有了索引，就不需要加载所有数据，因为B+树的高度一般在2-4层，最多只需要读取2-4

次磁盘，查询速度大大提升。

什么情况下需要建索引？

- 经常用于查询的字段
- 经常用于连接的字段建立索引，可以加快连接的速度
- 经常需要排序的字段建立索引，因为索引已经排好序，可以加快排序查询速度

什么情况下不建索引？

- `where` 条件中用不到的字段不适合建立索引
- 表记录较少。比如只有几百条数据，没必要加索引
- 需要经常增删改。需要评估是否适合加索引
- **参与列计算**的列不适合建索引
- **区分度不高**的字段不适合建立索引，如性别，只有男、女、未知三个值。加了索引，查询效率也不会提高。

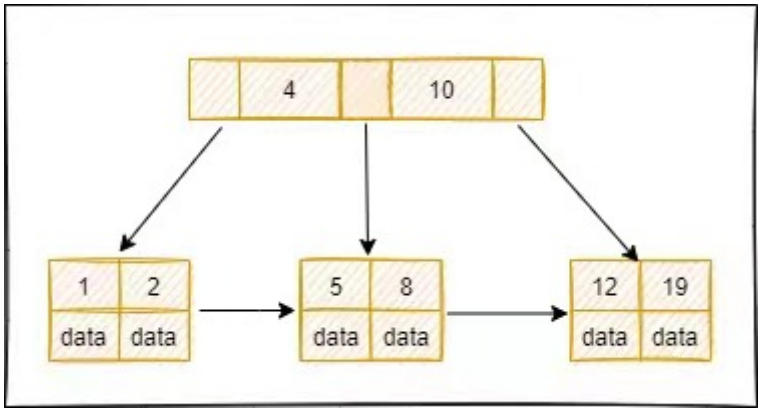
索引的数据结构

索引的数据结构主要有 B+树和哈希表，对应的索引分别为 B+树索引和哈希索引。InnoDB引擎的索引类型有 B+树索引和哈希索引，默认的索引类型为 B+树索引。

B+树索引

B+树是基于B树和叶子节点顺序访问指针进行实现，它具有B树的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在 B+树中，节点中的 `key` 从左到右递增排列，如果某个指针的左右相邻 `key` 分别是 key_i 和 key_{i+1}，则该指针指向节点的所有 `key` 大于等于 key_i 且小于等于 key_{i+1}。



进行查找操作时，首先在根节点进行二分查找，找到 `key` 所在的指针，然后递归地在指针指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 `key` 所对应的数据项。

MySQL数据库使用最多的索引类型的 `BTREE` 索引，底层基于 B+树数据结构来实现。

```
mysql> show index from blog;
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
| blog  |          0 | PRIMARY  |            1 | id          | A         |           0 | NULL     | NULL   |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
1 row in set (0.44 sec)
```

哈希索引

哈希索引是基于哈希表实现的，对于每一行数据，存储引擎会对索引列进行哈希计算得到哈希码，并且哈希算法要尽量保证不同的列值计算出的哈希码值是不同的，将哈希码的值作为哈希表的key值，将指向数据行的指针作为哈希表的value值。这样查找一个数据的时间复杂度就是O(1)，一般多用于精确查找。

Hash索引和B+树索引的区别？

- 哈希索引**不支持排序**，因为哈希表是无序的。
- 哈希索引**不支持范围查找**。
- 哈希索引**不支持模糊查询**及列索引的最左前缀匹配。
- 因为哈希表中会**存在哈希冲突**，所以哈希索引的性能是不稳定的，而B+树索引的性能是相对稳定的，每次查询都是从根节点到叶子节点。

为什么B+树比B树更适合实现数据库索引？

- 由于B+树的数据都存储在叶子结点中，叶子结点均为索引，方便归库，只需要扫一遍叶子结点即可，但是B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在区间查询的情况，而在数据库中基于范围的查询是非常频繁的，所以通常B+树用于数据索引。
- B+树的节点只存储索引key值，具体信息的地址存在于叶子节点的地址中，这就使以页为单位的索引中可以存放更多的节点。减少更多的I/O支出。
- B+树的查询效率更加稳定，任何关键字的查找必须走一条从根节点到叶子节点的路。所以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。

索引有什么分类？

1、**主键索引**：名为primary的唯一非空索引，不允许有空值。

2、**唯一索引**：索引列中的值必须是唯一的，但是允许为空值。唯一索引和主键索引的区别是：唯一索引字段可以为null，且可以存在多个null值，而主键索引字段不可以为null。唯一索引的用途：唯一标识数据库表中的每条记录，主要是用来防止数据重复插入。创建唯一索引的SQL语句如下：

```
ALTER TABLE table_name ADD CONSTRAINT constraint_name UNIQUE KEY(column_1, column_2, ...);
```

3、**组合索引**：在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时需遵循最左前缀原则。

4、**全文索引**：只能在 `CHAR` 、 `VARCHAR` 和 `TEXT` 类型字段上使用全文索引。

5、**普通索引**：普通索引是最基本的索引，它没有任何限制，值可以为空。

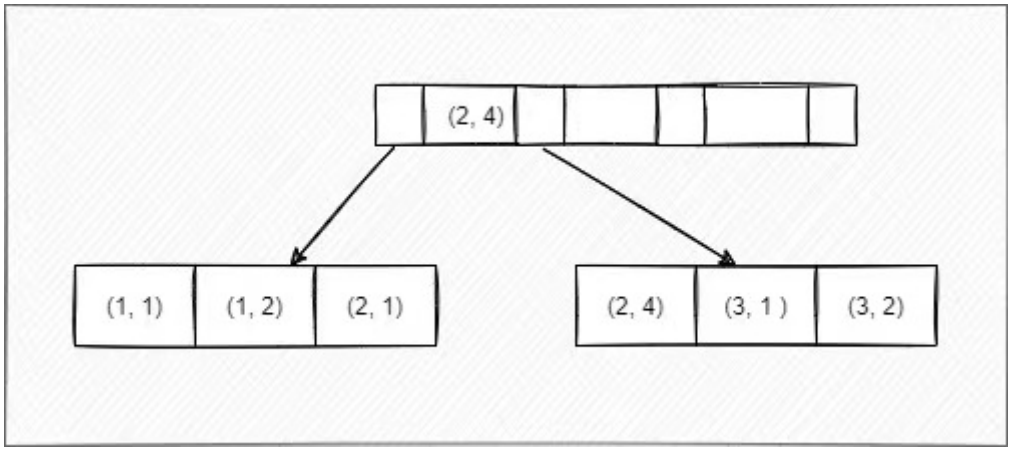
什么是最左匹配原则？

在SQL语句中用到了组合索引中的最左边的索引，那么这条SQL语句就可以利用这个组合索引去进行匹配。当遇到范围查询(`>` 、 `<` 、 `between` 、 `like`)就会停止匹配，后面的字段不会用到索引。

对 `(a,b,c)` 建立索引，查询条件使用 `a/ab/abc` 会走索引，使用 `bc` 不会走索引。

对 `(a,b,c,d)` 建立索引，查询条件为 `a = 1 and b = 2 and c > 3 and d = 4` ，那么 a、b 和 c 三个字段能用到索引，而 d 无法使用索引。因为遇到了范围查询。

如下图，对(a,b)建立索引，a 在索引树中是全局有序的，而 b 是全局无序，局部有序（当 a 相等时，会根据 b 进行排序）。直接执行 `b = 2` 这种查询条件无法使用索引。



当a的值确定的时候，b是有序的。例如 `a = 1` 时，b值为1，2是有序的状态。当 `a = 2` 时候，b的值为1，4也是有序状态。当执行 `a = 1 and b = 2` 时 a 和 b 字段能用到索引。而执行 `a > 1 and b = 2` 时，a 字段能用到索引，b字段用不到索引。因为a的值此时是一个范围，不是固定的，在这个范围内b值不是有序的，因此b字段无法使用索引。

什么是聚集索引？

InnoDB使用表的主键构造主键索引树，同时叶子节点中存放的即为整张表的记录数据。聚集索引叶子节点的存储是连加上连续的，使用双向链表连接，叶子节点按照主键的顺序排序，因此对于主键的排序查找和范围查找速度比较快。

聚集索引的叶子节点就是整张表的行记录，InooDB主键使用的是聚簇索引。聚集索引要比非聚集索引查询效率高很多。

对于InooDB来说，聚集索引一般是表中的主键索引，如果表中没有显示指定主键，则会选择表中的第一个允许为 NULL 的唯一索引。如果没有主键也没合适的唯一索引，那么 InnoDB 内部会生成一个隐藏的主键作为聚集索引，这个隐藏的主键长度为6个字节，它的值会随着数据的插入自增。

什么是覆盖索引？

select 的数据列只用从索引中就能够取得，不需要回表进行二次查询，也就是说查询列要被所使用的索引覆盖。对于 InooDB 表的二级索引，如果索引能覆盖到查询的列，那么就可以避免对主键索引的二次查询。

不是所有类型的索引都可以成为覆盖索引。覆盖索引要存储索引的值，而哈希索引、全文索引不存储索引列的值，所以MySQL使用B+树索引做覆盖索引。

对于使用了覆盖索引的查询，在查询前面使用 explain ，输出的extra列会显示为 using index 。

比如 user_like 用户点赞表，组合索引为 (user_id, blog_id) ， user_id 和 blog_id 都不为null。

```
mysql> create table user_like (
    -> user_id int(11) not null,
    -> blog_id int(11) not null,
    -> KEY index_user_blog_id(user_id,blog_id)
    -> )
    -> ;
```

```
explain select blog_id from user_like where user_id = 13;
```

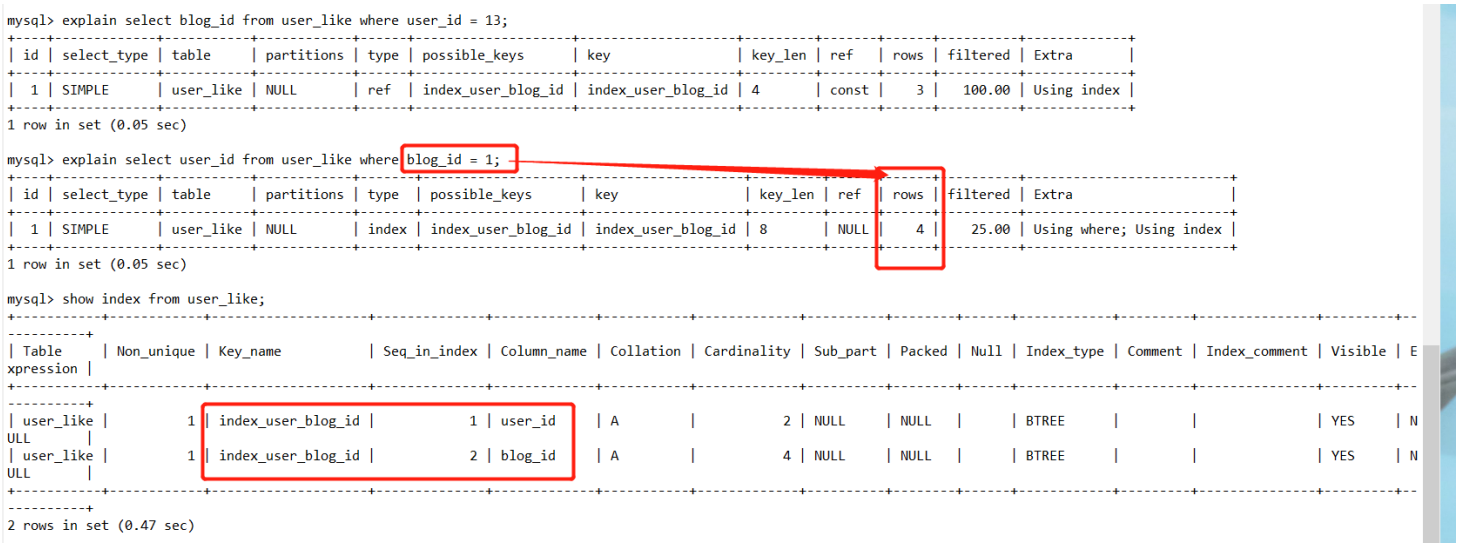
explain 结果的 Extra 列为 Using index ，查询的列被索引覆盖，并且where筛选条件符合最左前缀原则，通过索引查找就能直接找到符合条件的数据，不需要回表查询数据。

```
explain select user_id from user_like where blog_id = 1;
```

explain 结果的 Extra 列为 Using where; Using index ，查询的列被索引覆盖，where筛选条件不符合最左前缀原则，无法通过索引查找到符合条件的数据，但可以通过索引扫描找到符合条件的数据，也不需要回表查询数据。

```
mysql> select * from user_like;
+---------+---------+
| user_id | blog_id |
+---------+---------+
|      13 |       1 |
|      13 |       2 |
|      13 |       4 |
|      14 |       1 |
+---------+---------+
4 rows in set (0.05 sec)
```

图中可以看出， `blog_id` 查询结果与实现不符，因为 `blog_id` ： 非最左前缀，无法通过索引查找。



索引的设计原则？

- 对于经常作为查询条件的字段，应该建立索引，以提高查询速度。
- 为经常需要排序、分组和联合操作的字段建立索引。
- 索引列的区分度越高，索引的效果越好。比如使用性别这种区分很低的列作为索引，效果就会很差。
- 避免给“大字段”建立索引，尽量使用数据量小的字段作为索引。因为MySQL在维护索引的时候是会将字段值一起维护的，那这们必然会导致索引占用更多的空间，另外在排序的时候需要花费更多的时间去对比。
- 尽量使用**短索引**，对于较长的字符串进行索引时应该指定一个较短的前缀长度，因为较小的索引涉及到的磁盘I/O较少，查询速度更快。
- 索引不是越多越好，每个索引都需要额外的物理空间，维护也需要花费时间。
- 频繁增删改的字段不要建立索引。假设某个字段频繁修改，那就意味着需要频繁的重建索引，这必然影响MySQL的性能。
- 利用**最左前缀原则**。

索引什么时候会失效？

导致索引失效的情况：

- 对于组合索引，不是使用组合索引最左边的字段，则不会使用索引。
- 以%开头的like查询为 `%abc` ，无法使用索引；非 % 开头的like查询如 `abc%` ，相当于范围查询，会使用索引。
- 查询条件中列类型是字符串，没有使用引号，可能会因为类型不同发生隐式转换，使用索引失效。
- 判断索引列是否不等于某个值时。
- 对索引列进行运算。
- 查询条件使用 `or` 连接，也会导致索引失效。

什么是前缀索引？

有时需要在很长的字符列上创建索引，这会造成索引特别大且慢。使用前缀索引可以避免这个问题。

前缀索引是指对文本或者字符串的前几个字符建立索引，这样索引的长度更短，查询速度更快。

创建前缀索引的关键字在于选择足够长的前缀以**保证较高的索引选择性**。索引选择性越高查询效率就越建立前缀索引的试：

```
// email列创建前缀索引
ALTER TABLE table_name ADD KEY(column_name(prefix_length));
```

索引下推（Index Condition Pushdown，简称ICP）

参考文：https://mp.weixin.qq.com/s/W1XQYmihtSdbLWQKeNwZvQ

索引下推是MySQL 5.6 新添加的特性，用于优化数据的查询。

在MySQL5.6之前，通过使用非主键索引进行查询的时候，存储引擎通过索引查询数据，然后将结果返回给MySQL server层，在server层判断是否符合条件。

在MySQL5.6及以上版本，可以使用索引下推的特性。当存在索引的列做为判断条件时，MySQL server将这一部分判断条件传递给存储引擎，然后存储引擎会筛选出**符合MySQL server传递条件的索引项**，即在存储引擎层根据索引条件**过滤**不符合条件的索引项，然后回表查询得到查询得到结果，将结果返回给MySQL server。

有了索引下推的优化，在满足一定的条件下，存储引擎层会在回表查询之前对数据进行过滤，可以减少存储引擎回表查询的次数。

栗子

假设有一张用户信息表 `user_info` ，有三个字段 `name, level, weapon(装备)` ，建立联合索引 `(name, level)` ， `user_info` 表初始数据如下：

```
+----+------+-------+--------+
| id | name | level | weapon |
+----+------+-------+--------+
|  1 | 废柴 |     1 | 键盘   |
|  2 | 路飞 |     5 | 橡胶   |
|  3 | 索隆 |     6 | 三刀流 |
|  4 | 废物 |     7 | 摆烂   |
+----+------+-------+--------+
4 rows in set (0.06 sec)
```

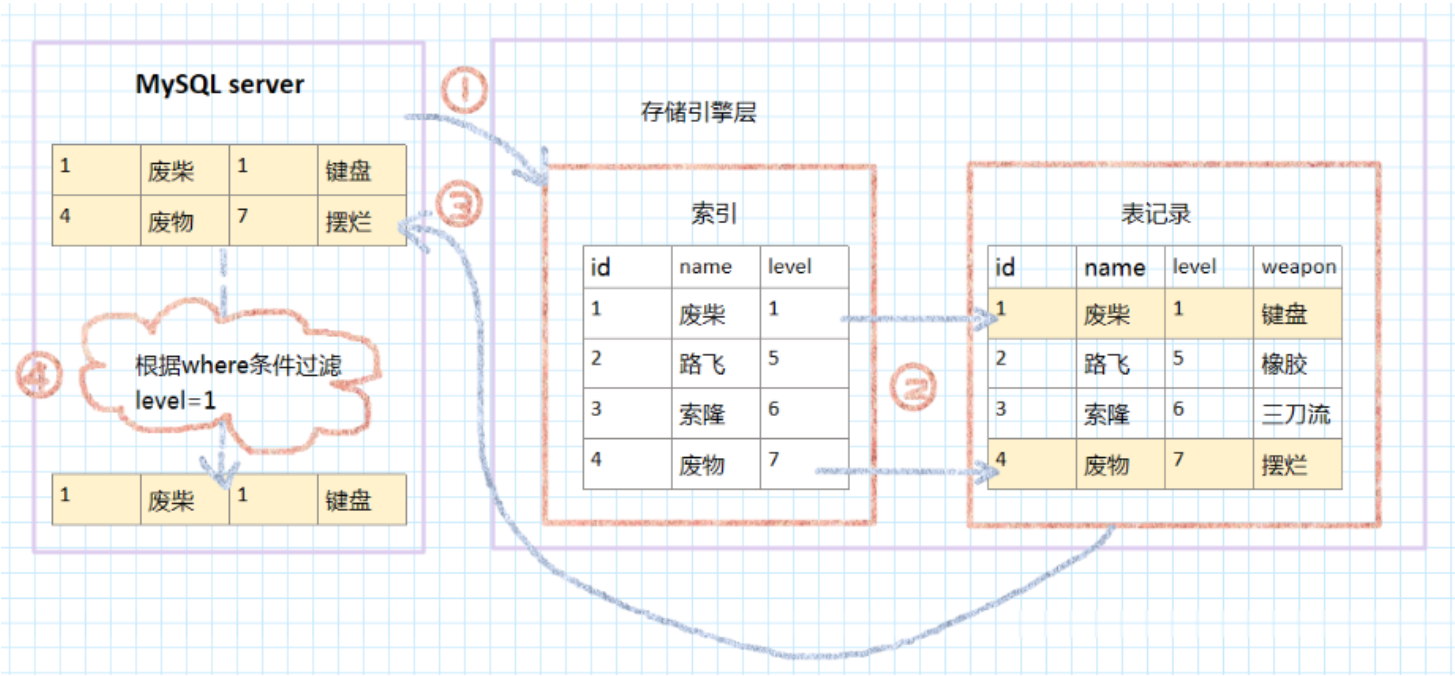
假如需要匹配姓名第一个字为“废”，并且level为1用户，SQL语句如下：

```
SELECT * FROM `user_info` WHERE name LIKE "废%" AND level = 1;
```

这条SQL具体执行情况分析如下：

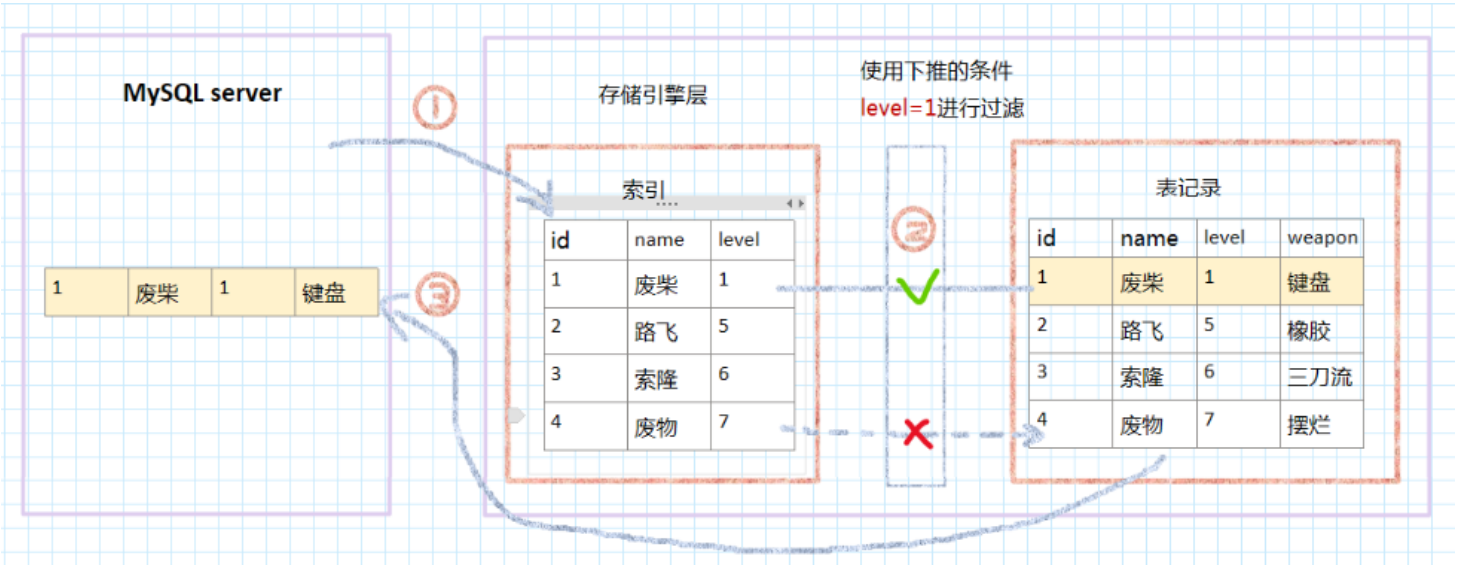
先看一下MySQL 5.6以前的版本

前面提到MySQL 5.6以前的版本没有索引下推，其执行过程如下:



查询条件 `name LIKE "废%"` 不是等值匹配，根据最左匹配原则，在（name,level)索引树上只用到 `name` 去匹配，查找到两条记录（id为1和4），拿到这两条记录的id分别回表查询，然后将结果返回给MySQL server，在MySQL server层进行 `level` 字段的判断。**整个过程需要回表2次。**

**然后看一下MySQL 5.6及以上的版本的执行过程，如图：



相比5.6以前的版本，多了索引下推的优化，在索引遍历过程中，对**索引中的字段**先做判断，过滤掉不符合条件的索引项，**也就是判断level是否等于1**，level不为1则直接跳过。因此在 `(name, level)` 索引树只匹配一个记录（id=1），之后拿着此记录对应的id回表查询全部数据，**整个过程回表1次。**

可以使用explain查看是否使用索引下推，当 `Extra` 列的值为 `Using index condition` ，则表示使用了索引下推。

```
mysql> explain SELECT * FROM `user_info` WHERE name LIKE "废%" AND level = 1;
+----+-------------+-----------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
| id | select_type | table     | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-----------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | user_info | NULL       | range | idx_n_l       | idx_n_l | 1028    | NULL |    1 |   100.00 | Using index condition |
+----+-------------+-----------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------+
1 row in set (0.19 sec)

mysql> SELECT * FROM `user_info` WHERE name LIKE "废%" AND level = 1;
+----+------+-------+--------+
| id | name | level | weapon |
+----+------+-------+--------+
|  1 | 废柴 |     1 | 键盘   |
+----+------+-------+--------+
1 row in set (0.06 sec)
```

从上面的栗子可以看出，使用索引下推在某些场景下可以有效**减少回表次数**，从而提高查询效率。

常见的存储引擎有哪些？

MySQL中常用的有四种存储引擎：**MyISAM**、**InnoDB**、**MEMORY**、**ARCHIVE**。MySQL 5.5版本后默认的存储引擎为**InnoDB**。

InnoDB 存储引擎

InnoDB是MySQL**默认的事务型存储引擎**，使用最广泛，基于聚簇索引建立的。InnoDB内部做了很多优化，如能够自动在内存中创建自适应hash索引，以加速读操作。

优点：支持事务和崩溃修复能力；引入了行级锁和外键约束。
缺点：占用的数据空间相对较大。

适用场景：需要事务支持，并且有较高的并发读写频率。

MyISAM 存储引擎

数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，可以使用MyISAM引擎。MyISAM会将存储在两个文件中，数据文件 `.MYD` 和索引文件 `.MYI` 。

优点：访问速度快。
缺点：MyISAM不支持事务和行级锁，不支持崩溃后的完全恢复，也不支持外键。
适用场景：对事务完整性没有要求；表的数据都会只读的。

MEMORY 存储引擎

MEMORY引擎将数据全部放在内存中，访问速度较快，但是一旦系统崩溃的话，数据都会丢失。

MEMORY引擎默认使用哈希索引，将键的哈希值和指向数据行的指针保存在哈希索引中。

优点：访问速度较快。

缺点：

1. 哈希索引数据不是按照索引值顺序存储，无法于排序。
2. 不支持部分索引匹配查找，因为哈希索引是使用索引列的全部内容来计算哈希值的。
3. 只支持等值比较，不支持范围查询。
4. 当出现哈希冲突时，存储引擎需要遍历链表中所有的行指针，逐行进行比较，直到找到符合条件的行。

ARCHIVE 存储引擎

ARCHIVE存储引擎非常适合存储大量独立的、作为历史记录的数据。ARCHIVE提供了压缩功能，拥有高效的速度，但是这种引擎不支持索引，所以查询性能较差。

MyISAM 和 InnoDB 的区别？

1. **存储结构的区别。** 每个MyISAM在磁盘上存储成三个文件。文件的名字以表的名字开始，扩展名指出文件类型。.frm 文件存储表定义。数据文件的扩展名 .MYD（MYData）。索引文件的扩展名 .MYI（MYIndex）。InnoDB所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为 2GB。
2. **存储空间的区别。** MyISAM支持三种不同的存储格式：静态表（默认，但是注意数据末尾不能有空格，会被去掉）、动态表、压缩表。当表在创建之后并导入数据之后，不会再进行修改操作，可以使用压缩表，极大的减少磁盘的空间占用。InnoDB需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

3. **可移植性、备份及恢复。** MyISAM数据是以文件的形式存储，所有在跨平台的数据转移中会很方便。在备份和恢复时可单独针对某个表进行操作。对于InnoDB，可行的方案是拷贝数据文件、备份binlog，或者用mysqldump，在数据量达到几十G的时候相对麻烦了。
4. **是否支持行级锁。** 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。而InnoDB支持行级锁和表级锁，默认为行级锁。行锁大幅度提高了多用户并发操作的性能。
5. **是否支持事务和崩溃后的安全恢复。** MyISAM不提供事务支持。而InnoDB提供事务支持，具有事务、回滚和崩溃修复能力。
6. **是否支持外键。** MyISAM不支持，而InnoDB支持。
7. **是否支持MVCC。** MyISAM不支持，InnoDB支持。应对高并发事务，MVCC比单纯的加锁更高效。
8. **是否支持聚集索引。** MySISAM不支持聚集索引，InnoDB支持聚集索引 。
9. **全文索引。** MyISAM支持FULLTEXT类型的全文索引。InnoDB不支持FULLTEXT类型的全文索引，但是InnoDB可以使用sphinx插件支持全文索引，并且效果更好。
10. **表主键。** MyISAM允许没有任何索引和主键的表存在，索引都是保存行的地址。对于InnoDB，如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键（用户不可见）。
11. **表的行数。** MyISAM保存有表的总行数，如果 `select count(*) from table;` 会直接取出该值。InnoDB没有保存表的总行数，如果使用 `select count(*) from table;` 就会遍历整个表，消耗相当大，但是在加了where条件后，MyISAM和InnoDB处理的方式都一样。

MySQL 有哪些锁？

按锁粒度分类， 有行级锁、表级锁和页级锁。

1. 行级锁是MySQL中锁定粒度最细的一种锁。表示只针对当前操作的行进行加锁。行级锁能大减少数据库操作的冲突，其加锁粒度最小，但加锁的开销也最大。行级锁的类型主要有三类：
   - Record Lock，记录锁，也就是仅仅把一条记录锁上；
   - Gap Lock，间隙锁，锁定一个范围，但是不包含记录本身；
   - Next-Key Lock：Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。
2. 表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加东风，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常用的MyISAM与InnoDB都支持表级锁定。
3. 页级锁是是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。因此，采取了折中的页级锁，一次锁定相邻的一组记录。

按锁级别分类，有共享锁、排他锁和意向锁。

1. 共享锁又称读锁，是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。

2. 排他锁又称写锁、独占锁，如果事务 T 对数据 A 加上排他锁后，则其他事务不能再对 A 加任何类型的封锁。获准排他锁的事既能读数据，又能修改数据。
3. 意向锁是表级锁，其设计目的主要是为了在一个事务中揭示下一行将要被请求锁的类型。InnoDB中的两个表锁：

意向共享锁（IS）：表示事务准备给数据行加入共享锁，也就是说一个数据行加共享锁前必须先取得该表的IS锁；

意向排他锁（IX）：类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必先取得该表的IX锁。

意向锁是InnoDB自动加的，不需要用户干预。

对于INSERT、UPDATE和DELETE，InnoDB会自动给涉及的数据加排他锁；对于一般的SELECT语句，InnoDB不会加任何锁，事务可以通过以下语句显式加共享锁或排他锁。

共享锁： `SELECT … LOCK IN SHARE MODE;`
排他锁： `SELECT … FOR UPDATE;`