第08章_索引的创建与设计原则

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网：http://www.atguigu.com

1.索引的声明与使用

1.1 索引的分类

MySQL的索引包括普通索引、唯一性索引、全文索引、单列索引、多列索引和空间索引等。

- 从`功能逻辑`上说，索引主要有4种，分别是普通索引、唯一索引、主键索引、全文索引。
- 按照`物理实现`方式，索引可以分为2种：聚簇索引和非聚簇索引。
- 按照`作用字段个数`进行划分，分成单列索引和联合索引。

1.普通索引

在创建普通索引时，不附加任何限制条件，只是用于提高查询效率。这类索引可以创建在`任何数据类型`中，其值是否唯一和非空，要由字段本身的完整性约束条件决定。建立索引以后，可以通过索引进行查询。例如，在表`student`的字段`name`上建立一个普通索引，查询记录时就可以根据该索引进行查询。

2.唯一性索引

使用`UNIQUE`参数可以设置索引为唯一性索引，在创建唯一性索引时，限制该索引的值必须是唯一的，但允许有空值。在一张数据表里`可以有多个`唯一索引。

例如，在表`student`的字段`email`中创建唯一性索引，那么字段email的值就必须是唯一的。通过唯一性索引，可以更快速地确定某条记录。

3.主键索引

主键索引就是一种`特殊的唯一性索引`，在唯一索引的基础上增加了不为空的约束，也就是`NOT NULL+UNIQUE`，一张表里`最多只有一个`主键索引。

`why?`这是由主键索引的物理实现方式决定的，因为数据存储在文件中只能按照一种顺序进行存储。

4.单列索引

在表中的单个字段上创建索引。单列索引只根据该字段进行索引。单列索引可以是普通索引，也可以是唯一性索引，还可以是全文索引。只要保证该索引只对应一个字段即可。一个表可以`有多个`单列索引。

5.多列(组合、联合)索引

多列索引是在表的`多个字段组合`上创建一个索引。该索引指向创建时对应的多个字段，可以通过这几个字段进行查询，但是只有查询条件中使用了这些字段中的第一个字段时才会被使用。例如，在表中的字段id、name和gender上建立一个多列索引`idx_id_name_gender`，只有在查询条件中使用了字段id时该索引才会被使用。使用组合索引时遵循`最左前缀集合`。

6.全文索引

全文索引（也称全文检索）是目前`搜索引擎`使用的一种关键技术。它能够利用【`分词技术`】等多种算法智能分析出文本文字中关键词的频率和重要性，然后按照一定的算法规则智能地筛选出我们想要的搜索结果。全文索引非常适合大型数据集，对于小的数据集，它的用处比较小。

使用参数 FULLTEXT 可以设置索引为全文索引。在定义索引的列上支持值的全文查找，允许在这些索引列中插入重复值和空值。全文索引只能创建在 CHAR、VARCHAR 或 TEXT 类型及其系列类型的字段上，**查询数据量较大的字符串类型的字段时，使用全文索引可以提高查询速度。**例如，表 student 的字段 information 是 TEXT 类型，该字段包含了很多文字信息。在字段information上建立全文索引后，可以提高查询字段information的速度。

全文索引典型的有两种类型：自然语言的全文索引 和 布尔全文索引。

- 自然语言搜索引擎将计算每一个文档对象和查询的相关度。这里，相关度是基于匹配的关键词的个数，以及关键词在文档中出现的次数。**在整个索引中出现次数越少的词语，匹配时的相关度就越高。**相反，非常常见的单词将不会被搜索，如果一个词语的在超过50%的记录中都出现了，那么自然语言的搜索将不会搜索这类词语。

MySQL数据库从3.23.23版开始支持全文索引，但MySQL5.6.4以前只有Myisam支持，5.6.4版本以后innodb才支持，但是官方版本不支持中文分词，需要第三方分词插件。在5.7.6版本，MySQL内置了ngram全文解析器，用来支持亚洲语种的分词。测试或使用全文索引时，要先看一下自己的MySQL版本、存储引引擎和数据类型是否支持全文索引。

随着大数据时代的到来，关系型数据库应对全文索引的需求已力不从心，逐渐被 solr、ElasticSearch 等专门的搜索引擎所替代。

7.补充：空间索引

使用参数SPATIAL 可以设置索引为空间索引。空间索引只能建立在空间数据类型上，这样可以提高系统获取空间数据的效率。MySQL中的空间数据类型包括 GEOMETRY、POINT、LINESTRING 和 POLYGON 等。目前只有MyISAM存储引擎支持空间检索，而且索引的字段不能为空值。对于初学者来说，这类索引很少会用到。

小结：不同的存储引擎支持的索引类型也不一样

InnoDB：支持B-tree、Full-text等索引，不支持Hash索引；

MyISAM：支持B-tree、Full-text等索引，不支持Hash索引；

Memory：支持B-tree、Hash等索引，不支持Full-text索引；

NDB：支持Hash索引，不支持B-tree、Full-text等索引；

Archive：不支持B-tree、Hash、Full-text等索引；

1.2 创建索引

MySQL支持多种方法在单个或多个列上创建索引：在创建表的定义语句 CREATE TABLE 中指定索引l列，使用 ALTER TABLE 语句在存在的表上创建索引，或者使用 CREATE INDEX 语句在已存在的表上添加索引。

1. 创建表的时候创建索引

使用CREATE TABLE创建表时，除了可以定义列的数据类型外，还可以定义主键约束、外键约束或者唯一性约束，而不论创建哪种约束，在定义约束的同时相当于在指定列上创建了一个索引。

隐式的索引创建举例：

```
# 1.隐式的添加索引(在添加有主键约束、唯一性约束或者外键约束的字段会自动地添加相关的索引)
CREATE TABLE dept(
    dept_id INT PRIMARY KEY AUTO_INCREMENT,# 创建主键索引
    dept_name VARCHAR(20)
);
CREATE TABLE emp(
    emp_id INT PRIMARY KEY AUTO_INCREMENT,# 主键索引
    emp_name VARCHAR(20) UNIQUE,# 唯一索引
    dept_id INT,
    CONSTRAINT emp_dept_id_fk FOREIGN KEY(dept_id) REFERENCES dept(dept_id)
); # 外键索引
```

但是，如果显式创建表时创建索引的话，基本语法格式如下：

```
CREATE TABLE table_name[col_namedata_type]
[UNIQUE | FULLTEXT | SPATIAL] [INDEX | KEY] [index_name] (col_name[length])
[ASC | DESC]
```

- `UNIQUE`、`FULLTEXT`和`SPATIAL`为可选参数，分别表示唯一索引、全文索引和空间索引；
- `INDEX`与`KEY`为同义词，两者的作用相同，用来指定创建索引；
- `index_name`指定索引的名称，为可选参数，如果不指定，那么MySQL默认col_name为索引名；
- `col_name`为需要创建索引的字段列，该列必须从数据表中定义的多个列中选择；
- `length`为可选参数，表示索引的长度，只有字符串类型的字段才能指定索引长度；
- `ASC`或`DESC`指定升序或者降序的索引值存储。
- 特例：主键索引使用主键约束的方式来创建。

1.创建普通索引

在book表中的year_publication字段上建立普通索引，SQL语句如下：

```
CREATE TABLE book(
    book_id INT,
    book_name VARCHAR(100),
    authors VARCHAR(100),
    info VARCHAR(100),
    comment VARCHAR(100),
    year_publication YEAR,
    INDEX(year_publication)
);
```

使用EXPLAIN语句查看索引是否正在使用

```
EXPLAIN SELECT * FROM book WHERE year_publication='1990';
```

```sql
CREATE TABLE book(
    book_id INT ,
    book_name VARCHAR(100),
    `AUTHORS` VARCHAR(100),
    info VARCHAR(100) ,
    `COMMENT` VARCHAR(100),
    year_publication YEAR,
    # 声明索引
    INDEX idx_bname(book_name)
);
```

通过命令查看索引有没有创建成功

方式1:

```sql
SHOW CREATE TABLE book; # Linux下添加\G参数
```

```
mysql> SHOW CREATE TABLE book\G;
*************************** 1. row ***************************
       Table: book
Create Table: CREATE TABLE `book` (
  `book_id` int DEFAULT NULL,
  `book_name` varchar(100) DEFAULT NULL,
  `authors` varchar(100) DEFAULT NULL,
  `info` varchar(100) DEFAULT NULL,
  `comment` varchar(100) DEFAULT NULL,
  `year_publication` year DEFAULT NULL,
  KEY `idx_bname` (`book_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

ERROR:
No query specified
```

索引列

方式2:

```sql
SHOW INDEX FROM book;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null
book	1	idx_bname	1	book_name	A	0	(NULL)	(NULL)	YES

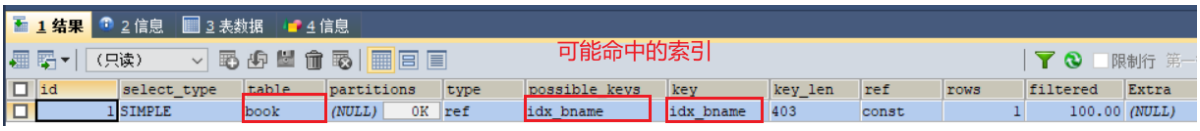
1：true　索引名称　字段名

```
*************************** 1. row ***************************
        Table: book
   Non_unique: 1
     Key_name: idx_bname
  Seq_in_index: 1
  Column_name: book_name
    Collation: A
  Cardinality: 0
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
```

```
      Visible: YES
   Expression: NULL
1 row in set (0.03 sec)
```

性能分析工具：EXPLAIN，查看索引是否正在使用

```
EXPLAIN SELECT * from book where book_name = 'mysql高级';
```




```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: book
   partitions: NULL
         type: ref
possible_keys: idx_bname
          key: idx_bname
      key_len: 403
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

EXPLAIN语句输出结果的各个行我们在下一章讲解，这里主要关注两个字段

- possible_keys行给出了MySQL在搜索数据记录时可选用的各个索引
- key行时MySQL实际选用的索引

可以看到，possible_keys和key值都为year_publication，查询时使用了索引。

2.创建唯一索引

创建唯一索引的目的也是???，尤其是对比较庞大的数据表。与前面的普通索引类似，不同的是，索引列的值必须唯一，但允许有空值。如果是？？，则列值的？？？

举例：

```
CREATE TABLE test1(
    id INT NOT NULL,
    name varchar(30) NOT NULL,
    UNIQUE INDEX uk_idx_id(id)
);
```

该语句执行完毕之后,使用SHOW CREATE TABLE查看表结图

```
SHOW INDEX FROM test1\G
```

其中各个主要参数的含义为

1. Table表示创健索引的表

2. Non_unique表示索引非唯一，1代表非唯一索引，0代表唯一索引
3. Key_name表示索引的名称。
4. Seq_In_Index表示该字段在索引中的位置，单列索引该值为1，组合索引为每个字段在索引定义中的顺序
5. Column_name表示定义索引的列字段。
6. Sub_part表示索引的长度
7. Null表示该字段是否能为空值。
8. Idex_type表示索引类型。

由结果可以看到，id字段上已经成功建立了一个名为uk_idx_id的唯一索引。

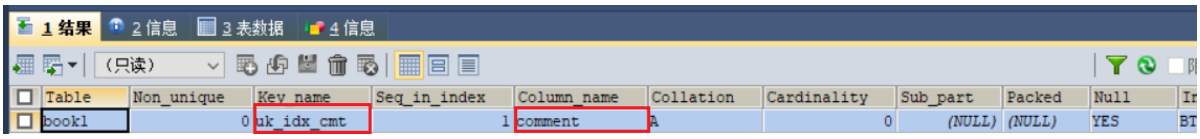
```
# ②创建唯一索引
CREATE TABLE book1 (
  book_id INT,
  book_name VARCHAR (100),
  AUTHORS VARCHAR (100),
  info VARCHAR (100),
  COMMENT VARCHAR (100),
  year_publication YEAR,
  #声明索引
  UNIQUE INDEX uk_idx_cmt (COMMENT)
);
```

该语句执行完毕之后，使用SHOW CREATE TABLE查看表结构：

```
SHOW INDEX FROM test1\G
```

```
# ②创建唯一索引
# 声明有唯一索引的字段，在添加数据时，要保证唯一性，但是可以添加null
CREATE TABLE book1 (
  book_id INT,
  book_name VARCHAR (100),
  AUTHORS VARCHAR (100),
  info VARCHAR (100),
  COMMENT VARCHAR (100),
  year_publication YEAR,
  #声明索引
  UNIQUE INDEX uk_idx_cmt (COMMENT)
);
```

```
show INDEX from book1;# 查看索引
```



3.主键索引

设定为主键后数据库会自动建立索引，innodb为聚簇索引，语法：

- 随表一起建索引：

```
# ③主键索引
# 通过定义主键约束的方式定义主键索引
CREATE TABLE student(
    id INT(10) UNSIGNED AUTO_INCREMENT,
    student_no VARCHAR(200),
    student_name VARCHAR(200),
    PRIMARY KEY(id)
)
create table book2(
    book_id int primary key,
    book_name varchar(100),
    AUTHORS VARCHAR (100),
    info VARCHAR (100),
    COMMENT VARCHAR (100),
    year_publication YEAR
) ;
```

- 删除主键索引：

```
# 通过删除主键约束的方式删除主键索引
ALTER TABLE student drop PRIMARY KEY;
```

- 修改主键索引：必须先删除掉(drop)原索引，再新建(add)索引

4.创建单列索引

单列索引是在数据表中的某一个字段上创建的索引，一个表中可以创建多个单列索引。前面例子中创建的索引都为单列索引。

举例：

```
CREATE TABLE test2(
    id INT NOT NULL,
    name CHAR(50) NULL,
    #INDEX single_idx_name(name)
    INDEX single_idx_name(name(20)) # 前20个字符
);
CREATE TABLE book3(
    book_id INT,
    book_name VARCHAR(100),
    AUTHORS VARCHAR (100),
    info VARCHAR (100),
    COMMENT VARCHAR (100),
    year_publication YEAR,
    UNIQUE INDEX idx_bname(book_name)
);

show index from book3;
```

该语句执行完毕之后，使用SHOW CREATE TABLE查看表结构：

```
SHOW INDEX FROM test2\G
```

由结果可以看到，id字段上已经成功建立了一个名为single_idx_name的单列索引。索引长度为20。

5.创建组合索引

组合索引是在多个段上创建一个索引。

举例：创建表test3，在表中的id、name和age字段上建立组合索引，SQL语句如下：

```
CREATE TABLE test3(
    id INT(11) NOT NULL,
    name CHAR(30) NOT NULL,
    age INT(11) NOT NULL,
    info VARCHAR(255),
    INDEX multi_idx(id,name,age)
);
```

该语句执行完毕之后，使用SHOWINDEX查看：

```
SHOW INDEX FROM test3\G
```

由结果可以看到，id、name和age字段上已经成功建立了一个名为multi_idx的组合索引。

组合索引可起几个索引的作用，但是使用时并不是随便查询哪个字段都可以使用索引，而是遵从“最左前缀”。例如，索引可以搜索的字段组合为：(id,name,age)、(id,name)或者id。而(age)或者(name,age)组合不能使用索引查询。

在test3表中，查询id和name字段，使用EXPLAIN语句查看索引的使用情况：

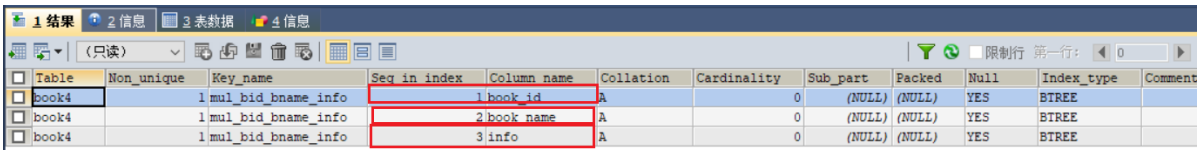
```
EXPLAIN SELECT * FROM test3 WHERE id=1 AND name='songhongkang'\G
```

可以看到，查询id和name字段时，使用了名称为Multildx的索引，如果查询（name,age）组合或者单独查询name和age字段，会发现结果中possible_keys和key值为NULL，并没有使用在t3表中创建的索引进行查询。

举例：创建表 book4，在表中的 book_id、book_name和 info字段上建立组合索引，SQL 语句如下：

```
# ⑤ 创建联合索引
create table book4(
    book_id INT,
    book_name VARCHAR(100),
    AUTHORS VARCHAR (100),
    info VARCHAR (100),
    COMMENT VARCHAR (100),
    year_publication YEAR,
    index mul_bid_bname_info(book_id,book_name,info)
)

SHOW INDEX FROM book4;
```



注意上面三行依次是book_id,book_name,info，与我们创建索引时指定的顺序是严格对应的。在查询时会遵守最左索引原则，先进行book_id条件的比较，然后再进行book_name比较，最后才是info。因此注意把最常用的查询字段放在索引的最左边。

```
# 分析
explain select * from book4 where book_id = 1001 and book_name = 'mysql'; # 会使用到mul_bid_bname_info索引

explain select * from book4 where book_name = 'mysql';# 不会使用到mul_bid_bname_info索引
```

6.创建全文索引

FULLTEXT全文索引可以用于全文搜索，并且只为`CHAR`、`VARCHAR`和`TEXT`列创建索引。索引总是对整个列进行，不支持局部（前缀）索引。

举例1：创建表test4，在表中的info字段上建立全文索引，SQL语句如下：

```
CREATE TABLE test4(
    id INT NOT NULL,
    name CHAR(30) NOT NULL,
    age INT NOT NULL,
    info VARCHAR(255),
    FULLTEXT INDEX futxt_idx_info(info)
    # FULLTEXT INDEX futxt_idx_info(info(50))
)ENGINE=MyISAM;
```

> 在MySQL5.7及之后版本中可以不指定最后的ENGINE了，因为在此版本中InnoDB支持全文索引。

语句执行完毕后，用SHOW CREATE TABLE查看表结构：

```
SHOW INDEX FROM test4\G;
```

```
mysql> SHOW INDEX FROM test4\G;
*************************** 1. row ***************************
        Table: test4
   Non_unique: 1
     Key_name: futxt_idx_info
 Seq_in_index: 1
  Column_name: info
    Collation: NULL
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: FULLTEXT
      Comment:
Index_comment:
      Visible: YES
   Expression: NULL
1 row in set (0.01 sec)
```

由结果可以看到，info字段上已经成功建立了一个名为futxt_idx_info的FULLTEXT索引。

举例2：

```
CREATE TABLE articles(
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(200),
    body TEXT,
    FULLTEXT index(title,body)
)ENGINE=INNODB;
```

创建了一个给title和body字段添加全文索引的表。

举例3：

```
CREATE TABLE `papers`(
    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
    `title` varchar(200) DEFAULT NULL,
    `content` text,
    PRIMARY KEY(`id`),
    FULLTEXT KEY `title` (`title`,`content`)
)ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

不同于like方式的的查询：

```
SELECT * FROM papers WHERE content LIKE '%查询字符串%';
```

全文索引用match+against方式查询：

```
SELECT * FROM papers WHERE MATCH(title,content) AGAINST('查询字符串');
```

明显的提高查询效率

> 注意点：
>
> 1. 使用全文索引前，搞清楚版本支持情况；

> 2. 全文索引比like+%快N倍，但是可能存在精度问题；
>
> 3. 如果需要全文索引的是大量数据，建议先添加数据，再创建索引。

7.创建空间索引

空间索引创建中，要求空间类型的字段必须为`非空`。

举例：创建表test5，在空间类型为GEOMETRY的字段上创建空间索引，SQL语句如下：

```
CREATE TABLE test5(
    geo GEOMETRY NOT NULL,
    SPATIAL INDEX spa_idx_geo(geo)
)ENGINE=MyISAM;
```

该语句执行完毕之后，使用SHOW CREATE TABLE查看表结构：

```
SHOW INDEX FROM test5\G;
```

可以看到，test5的geo字段上创建了名称为spa_idx_geo的空间索引。注意创建时指定空间类型字段值的非空约束，并且表的存储引擎为MyISAM。

2. 在已经存在的表上创建索引

在已经存在的表中创建索引可以使用ALTER TABLE语句或者CREATE INDEX语句。

1.使用ALTER TABLE语句创建索引

ALTER TABLE语句创建索引的基本语法如下：

```
ALTER TABLE table_name ADD [UNIQUE | FULLTEXT | SPATIAL] [INDEX | KEY]
[index_name] (col_name[length],...) [ASC | DESC]
```

与创建表时创建索引的语法不同的是，在这里使用了`ALTER TABLE`和ADD关键字，ADD表示向表中添加索引。

举例1：在book表中的book_name字段上建立名为BkNameIdx的普通索引。

```
ALTER TABLE book ADD INDEX idx_bkname( book_name(30) );
```

使用SHOW INDEX语句查看表中的索引：

```
SHOW INDEX FROM book \G
```

举例2：在book表的book_id字段上建立名称为uk_idx_bid的唯一索引，SQL语句如下：

```
ALTER TABLE book ADD UNIQUE INDEX uk_idx_bid(book_id) ;
```

使用SHOWINDEX语句查看表中的索引引：

```
SHOW INDEX FROM book \G
```

可以看到Non_unique的属性值为0，表示名称为Uniqidldx的索引为唯一索引，创建唯一索引成功。

举例3：在book表的comment字段上建立单列索引，SQL语句如下：

```
ALTER TABLE book ADD INDEX idx_cmt ( comment(50) )
```

举例4：在book表的authors和info字段上建立组合索引，SQL语句如下：

```
ALTER TABLE book ADD INDEX idx_author_info ( authors(30),info(50) )
```

使用SHOW INDEX语句查看表中的索引：

```
SHOW INDEX FROM book \G
```

举例5：给customer2表的id字段声明主键索引：

```
CREATE TABLE customer2(
    id INT(10) UNSIGNED,
    customer_no VARCHAR(200),
    customer_name VARCHAR(200)
);

ALTER TABLE customer2
add PRIMARY KEY customer2(id):
```

2.使用CREATE INDEX创建索引

CREATE INDEX语句可以在已经存在的表上添加索引，在MySQL中，CREATE INDEX被映射到一个ALTER TABLE语句上，基本语法结构为：

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
ON table_name(col_name[length],...) [ASC | DESC]
```

```
CREATE TABLE book(
    book_id INT NOT NULL,
    book_name VARCHAR（255)NOT NULL,
    authors VARCHAR（255)NOT NULL,
    info VARCHAR(255) NULL,
    comment VARCHAR（255）NULL,
    year_publication YEAR NOT NULL
```

练习1：在book表中的book_name字段上建立名为idx_bk_name的普通索引，SQL语句如下：

```
CREATE INDEX idx_bk_name ON book(book_name);
```

练习2：在book表的book_id字段上建立名称为Uniqidldx的唯一索引，SQL语句如下：

```
CREATE UNIQUE INDEX idx_uk_bid ON book ( book_id );
```

练习3：在book表的comment字段上建立单列索引，SQL语句如下：

```
CREATE INDEX idx_cmt ON book(comment(50));
```

练习4：在book表的authors和info字段上建立组合索引，SQL语句如下：

```
CREATE INDEX idx_aut_info ON book (authors(20),info(50));
```

举例1：在book表的comment字段上建立名为idx_cmt的普通索引

```
create index idx_cmt on book(comment);
```

举例2：在book表中的book_id字段上建立名为uk_idx_bid的唯一索引，SQL语句如下：

```
CREATE UNIQUE INDEX uk_idx_bid ON book(book_id);
```

举例3：在book表的book_id、book_name、info字段上建立联合索引，SQL语句如下：

```
CREATE INDEX mul_bid_bname_info ON book(book_id,book_name,info);
```

1.3 删除索引

MySQL中删除索引使用`ALTER TABLE`或`DROP INDEX`语句，两者可实现相同的功能，DROP INDEX语句在内部被映射到一个ALTER TABLE语句中

1.使用ALTER TABLE删除索引

ALTER TABLE删除索引的基本语法格式如下：

```
ALTER TABLE table_name DROP INDEX index_name;
```

练习：删除book表中名称为idx_bk_id的唯一索引

首先查看book表中是否名称为idx_bk_id的索引，输入SHOW语句如下：

```
SHOW INDEX FROM book\G;
```

下面删除该索引，输入删除语句如下：

```
ALTER TABLE book DROP INDEX idx_bk_id;
```

> 提示
>
> 添加AUTO_INCREMENT约束字段的唯一索引不能被删除。

2.使用DROP INDEX语句删除索引

DROP INDEX删除索引的基本语法格式如下：

```
DROP INDEX index_name ON table_name;
```

练习：删除book表中名称为idx_aut_info的组合索引，SQL语句如下：

```
DROP INDEX idx_aut_info ON book;
```

语句执行完毕，使用SHOW查看索引是否删除：

```
SHOW CREATE TABLE book\G;
```

可以看到，book表中已经没有名称为idx_aut_info的组合索引，删除索引成功。

> 提示 删除表中的列时，如果要删除的列为索引的组成部分，则该列也会从索引中删除。如果组成索引的所有列都被删除，则整个索引将被删除。

```
# 测试：删除联合索引中的相关字段，索引的变化
CREATE TABLE test(
    id INT(11) NOT NULL,
    name CHAR(30)NOT NULL,
    age INT(11) NOT NULL,
    info VARCHAR(255),
    INDEX multi_idx(id,name,age)
);
show index from test;
```

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation
1	test	1	multi_idx	1	id	A
2	test	1	multi_idx	2	name	A
3	test	1	multi_idx	3	age	A

```
alter table test drop column name;
show index from test;
```

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation
1	test	1	multi_idx	1	id	A
2	test	1	multi_idx	2	age	A

2.MySQL8.0索引新特性

2.1支持降序索引

降序索引以降序存储键值。虽然在语法上，从MySQL 4版本开始就已经支持降序索引的语法了，但实际上该DESC定义是被忽略的，直到MySQL8.x版本才开始真正支持降序索引（仅限于InnoDB存储引擎）。

MySQL在**8.0版本之前创建的仍然是升序索引，使用时进行反向扫描，这大大降低了数据库的效率。**在某些场景下，降序索引意义重大。例如，如果一个查询，需要对多个列进行排序，且顺序要求不一致，那么使用降序索引将会避免数据库使用额外的文件排序操作，从而提高性能。

举例：分别在MySQL5.7版本和MySQL8.0版本中创建数据表ts1，结果如下：

```
CREATE TABLE ts1(a int,b int,index idx_a_b(a,b desc));
```

在MySQL5.7版本中查看数据表ts1的结构，结果如下：

```
mysql> show create table ts1\G
*************************** 1. row ***************************
       Table: ts1
Create Table: CREATE TABLE `ts1` (
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  KEY `idx_a_b` (`a`,`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.01 sec)
```

```
mysql> SHOW CREATE TABLE ts1\G;
*************************** 1. row ***************************
       Table: ts1
Create Table: CREATE TABLE `ts1` (
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  KEY `idx_a_b` (`a`,`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
1 row in set (0.00 sec)
```

MySQL5.7并不支持降序索引

从结果可以看出，索引仍然是默认的**升序**。

在MySQL8.0版本中查看数据表ts1的结构，结果如下：

```
mysql> show create table ts1\G
*************************** 1. row ***************************
       Table: ts1
Create Table: CREATE TABLE `ts1` (
  `a` int DEFAULT NULL,
  `b` int DEFAULT NULL,
  KEY `idx_a_b` (`a`,`b` DESC)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

```
mysql> SHOW CREATE TABLE ts1\G;
*************************** 1. row ***************************
       Table: ts1
Create Table: CREATE TABLE `ts1` (
  `a` int DEFAULT NULL,
  `b` int DEFAULT NULL,
  KEY `idx_a_b` (`a`,`b` DESC)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

MySQL8.0支持降序索引

从结果可以看出，索引已经是**降序**了。下面继续测试降序索引在执行计划中的表现。

分别在MySQL5.7版本和MySQL8.0版本的数据表ts1中插入800条随机数据，执行语句如下：

```
DELIMITER //
CREATE PROCEDURE ts_insert()
BEGIN
    DECLARE i INT DEFAULT 1;
    WHILE i < 800
    DO
        insert into ts1 select rand()*80000,rand()*80000;
        SET i = i + 1;
    END WHILE;
    commit;
```

```
END //
DELIMITER ;

#调用
CALL ts_insert();
```

在MySQL5.7版本中查看数据表ts1的执行计划，结果如下：

```
# 优化测试
EXPLAIN SELECT * FROM ts1 ORDER BY a,b DESC LIMIT 5;
```

```
mysql> EXPLAIN SELECT * FROM ts1 ORDER BY a,b DESC LIMIT 5;
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra                       |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------------+
|  1 | SIMPLE      | ts1   | NULL       | index | NULL          | idx_a_b | 10      | NULL |  799 |   100.00 | Using index; Using filesort |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------------+
1 row in set, 1 warning (0.00 sec)
```

从结果可以看出，执行计划中扫描数为799，而且使用了Using filesort。

> 提示：Using filesort是MySQL中一种速度比较慢的外部排序，能避免是最好的。多数情况下，管理员可以通过优化索引来尽量避免出现Using filesort，从而提高数据库执行速度。

在MySQL8.0版本中查看数据表ts1的执行计划。

```
mysql> EXPLAIN SELECT * FROM ts1 ORDER BY a,b DESC LIMIT 5;
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | ts1   | NULL       | index | NULL          | idx_a_b | 10      | NULL |    5 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

从结果可以看出，执行计划中扫描数为5，而且没有使用Using filesort。

> 注意：降序索引只对查询中特定的排序顺序有效，如果使用不当，反而查询效率更低。例如，上述查询排序条件改为order by a desc,b desc，MySQL5.7的执行计划要明显好于MySQL8.0。

将排序条件修改为order by a desc,b desc后，下面来对比不同版本中执行计划的效果。

在MySQL5.7版本中查看数据表ts1的执行计划，结果如下：

```
#优化测试
EXPLAIN SELECT * FROM ts1 ORDER BY a DESC,b DESC LIMIT 5;
```

```
mysql> EXPLAIN SELECT * FROM ts1 ORDER BY a DESC,b DESC LIMIT 5;
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | ts1   | NULL       | index | NULL          | idx_a_b | 10      | NULL |    5 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

在MySQL8.0版本中查看数据表ts1的执行计划。

```
mysql> EXPLAIN SELECT * FROM ts1 ORDER BY a DESC,b DESC LIMIT 5;
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra                       |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------------+
|  1 | SIMPLE      | ts1   | NULL       | index | NULL          | idx_a_b | 10      | NULL |  799 |   100.00 | Using index; Using filesort |
+----+-------------+-------+------------+-------+---------------+---------+---------+------+------+----------+-----------------------------+
1 row in set, 1 warning (0.00 sec)
```

从结果可以看出，修改后MySQL5.7的执行计划要明显好于MySQL8.0。

2.2隐藏索引

在MySQL5.7版本及之前，只能通过显式的方式删除索引。此时，如果发现删除索引后出现错误，又只能通过显式创建索引的方式将删除的索引创建回来。如果数据表中的数据量非常大，或者数据表本身比较大，这种操作就会消耗系统过多的资源，操作成本非常高。

MySQL8.x开始支持`隐藏索引（invisibleindexes）`，只需要将待删除的索引设置为隐藏索引，使查询优化器不再使用这个索引（即使使用forceindex（强制使用索引），优化器也不会使用该索引），确认将索引设置为隐藏索引后系统不受任何响应，就可以彻底删除索引。`这种通过先将索引设置为隐藏索引，再删除索引的方式就是软删除。`

同时，如果你想验证某个索引删除之后的`查询性能影响`，就可以暂时先隐藏该索引。

> 注意：
>
> 主键不能被设置为隐藏索引。当表中没有显式主键时，表中第一个唯一非空索引会成为隐式主键，也不能设置为隐藏索引

索引默认是可见的，在使用CREATE TABLE，CREATE INDEX或者ALTER TABLE等语句时可以通过`VISIBLE`或者`INVISIBLE`关键词设置索引的可见性。

1. **创建表时直接创建**
   在MySQL中创建隐藏索引通过SQL语句`INVISIBLE`来实现，其语法形式如下：

```
CREATE TABLE tablename(
    propname1 type1[CONSTRAINT1],
    propname2 type2[CONSTRAINT2],
    ……
    propnamen typen,
    INDEX [indexname](propname1[(length)]) INVISIBLE
);
```

上述语句比普通索引多了一个关键字`INVISIBLE`，用来标记索引为不可见索引。

练习：在创建书籍表book时，在字段`idx_cmt`上创建隐藏索引

```
#① 创建表时，隐藏索引
create table book(
    book_id INT,
    book_name VARCHAR(100),
    AUTHORS VARCHAR (100),
    info VARCHAR (100),
    COMMENT VARCHAR (100),
    year_publication YEAR,
    # 创建不可见的索引
    index idx_cmt(comment) invisible
);
```

通过explain查看发现，优化器并没有使用索引，而是使用的全表扫描

```
explain select * from book7 where comment = 'mysql...';
```

```
mysql> EXPLAIN SELECT * FROM book7 WHERE COMMENT = 'mysql...';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | book7 | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    1 |   100.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

2. **在已经存在的表上创建**

可以为已经存在的表设置隐藏索引，其语法形式如下：

```
CREATE INDEX indexname
ON tablename(propname[(length)]) INVISIBLE;
```

举例：

```
CREATE INDEX idx_year_pub ON book(year_publication) INVISIBLE;
```

3. **通过ALTER TABLE语句创建**

语法形式如下：

```
ALTER TABLE tablename
ADD INDEX indexname (propname [(length)]) INVISIBLE;
```

举例：

```
ALTER TABLE book
ADD UNIQUE INDEX uk_idx_bname(book_name) INVISIBLE;
```

4. **切换索引可见状态**已存在的索引可通过如下语句切换可见状态：

```
 ALTER TABLE tablename ALTER INDEX index_name INVISIBLE; #切换成隐藏索引
 ALTER TABLE tablename ALTER INDEX index_name VISIBLE; #切换成非隐藏索引
```

举例：

```
# 修改索引的可见性
ALTER TABLE book ALTER INDEX idx_year_pub invisible;#可见--->不可见
ALTER TABLE book ALTER INDEX idx_cmt visible;#不可见--->可见
```

如果将idx_cmt 索引切换成可见状态，通过explain查看执行计划，发现优化器选择了idx_cmt 索引。

```
mysql> EXPLAIN SELECT * FROM book7 WHERE COMMENT = 'mysql...';
+----+-------------+-------+------------+------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | book7 | NULL       | ref  | idx_cmt       | idx_cmt | 403     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

> **注意：**当索引被隐藏时，它的内容仍然是和正常索引一样实时更新的。如果一个索引需要长期被隐藏，那么可以将其删除，因为索引的存在会影响插入、更新和删除的性能。

通过设置隐藏索引的可见性可以查看索引对调优的帮助。

5. 使隐藏索引对查询优化器可见

在MySQL8.x版本中，为索引提供了一种新的测试方式，可以通过查询优化器的一个开关（use_invisible_indexes）来打开某个设置，使隐藏索引对查询优化器可见。如果use_invisible_indexes设置为off(默认)，优化器会忽略隐藏索引。如果设置为on，即使隐藏索引不可见，优化器在生成执行计划时仍会考虑使用隐藏索引。

（1）在MySQL命令行执行如下命令查看查询优化器的开关设置。

```
mysql> select @@optimizer_switch \G
```

在输出的结果信息中找到如下属性配置。

```
use_invisible_indexes=off
```

此属性配置值为off，说明隐藏索引默认对查询优化器不可见。

（2）使隐藏索引对查询优化器可见，需要在MySQL命令行执行如下命令：

```
mysql> set session optimizer_switch="use_invisible_indexes=on";
Query OK, 0 rows affected (0.00 sec)
```

SQL语句执行成功，再次查看查询优化器的开关设置。

```
mysql>  select @@optimizer_switch \G
*************************** 1. row ***************************
@@optimizer_switch:
index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,engine_condition_pushdown=on,index_condition_pushdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_access=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,duplicateweedout=on,subquery_materialization_cost_based=on,use_index_extensions=on,condition_fanout_filter=on,derived_merge=on,use_invisible_indexes=on,skip_scan=on,hash_join=on
1 row in set (0.00 sec)
```

此时，在输出结果中可以看到如下属性配置。

```
use_invisible_indexes=on
```

use_invisible_indexes属性的值为on，说明此时隐藏索引对查询优化器可见。

（3）使用EXPLAIN查看以字段invisible_column作为查询条件时的索引使用情况。

```
explain select * from classes where cname = '高一2班';
```

查询优化器会使用隐藏索引来查询数据。

（4）如果需要使隐藏索引对查询优化器不可见，则只需要执行如下命令即可。

```
mysql> set session optimizer_switch="use_invisible_indexes=off";
Query OK, 0 rows affected (0.00 sec)
```

再次查看查询优化器的开关设置。

```
mysql> select @@optimizer_switch \G;
```

此时，use_invisible_indexes属性的值已经被设置为“off”。

3.索引的设计原则

为了使索引的使用效率更高，在创建索引时，必须考虑在哪些字段上创建索引和创建什么类型的索引。**索引设计不合理或者缺少索引都会对数据库和应用程序的性能造成障碍。**高效的索引对于获得良好的性能非常重要。设计索引时，应该考虑相应准则。

3.1数据准备

第1步：创建数据库、创建表

```
CREATE DATABASE atguigudb1;

USE atguigudb1;

#1.创建学生表和课程表
CREATE TABLE `student_info` (
    `id` INT(11) NOT NULL AUTO_INCREMENT,
    `student_id` INT NOT NULL ,
    `name` VARCHAR(20) DEFAULT NULL,
    `course_id` INT NOT NULL ,
    `class_id` INT(11) DEFAULT NULL,
    `create_time` DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
CREATE TABLE `course` (
    `id` INT(11) NOT NULL AUTO_INCREMENT,
    `course_id` INT NOT NULL ,
    `course_name` VARCHAR(40) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

第2步：创建模拟数据必需的存储函数

```
#函数1：创建随机产生字符串函数
DELIMITER //
CREATE FUNCTION rand_string(n INT)
    RETURNS VARCHAR(255) #该函数会返回一个字符串
BEGIN
    DECLARE chars_str VARCHAR(100) DEFAULT
'abcdefghijklmnopqrstuvwxyzABCDEFJHIJKLMNOPQRSTUVWXYZ';
    DECLARE return_str VARCHAR(255) DEFAULT '';
    DECLARE i INT DEFAULT 0;
    WHILE i < n DO
        SET return_str
=CONCAT(return_str,SUBSTRING(chars_str,FLOOR(1+RAND()*52),1));
        SET i = i + 1;
    END WHILE;
    RETURN return_str;
END //
```

```
DELIMITER ;
```

```
#函数2：创建随机数函数
DELIMITER //
CREATE FUNCTION rand_num (from_num INT ,to_num INT) RETURNS INT(11)
BEGIN
    DECLARE i INT DEFAULT 0;
    SET i = FLOOR(from_num +RAND()*(to_num - from_num+1)) ;
    RETURN i;
END //
DELIMITER ;
```

创建函数，假如报错：

```
This function has none of DETERMINISTIC......
```

由于开启过慢查询日志bin-log,我们就必须为我们的function指定一个参数。

主从复制，主机会将写操作记录在bin-log日志中。从机读取bin-log日志，执行语句来同步数据。如果使用函数来操作数据，会导致从机和主键操作时间不一致。所以，默认情况下，mysql不开启创建函数设置。

- 查看mysql是否允许创建函数：

```
show variables like 'log_bin_trust_function_creators';
```

- 命令开启：允许创建函数设置：

```
set global log_bin_trust_function_creators=1; # 不加global只是当前窗口有效。
```

- mysqld重启，上述参数又会消失。永久方法：
  - windows下：my.ini[mysqld]加上：

```
log_bin_trust_function_creators=1
```

  - linux下：/etc/my.cnf下my.cnf[mysqld]加上：

```
log_bin_trust_function_creators=1
```

第3步：创建插入模拟数据的存储过程

```
# 存储过程1：创建插入课程表存储过程
DELIMITER //
CREATE PROCEDURE insert_course( max_num INT )
BEGIN
DECLARE i INT DEFAULT 0;
    SET autocommit = 0; #设置手动提交事务
    REPEAT #循环
    SET i = i + 1; #赋值
    INSERT INTO course (course_id, course_name )
VALUES(rand_num(10000,10100),rand_string(6));
```

```
    UNTIL i = max_num
    END REPEAT;
    COMMIT; #提交事务
END //
DELIMITER ;
```

```
# 存储过程2：创建插入学生信息表存储过程
DELIMITER //
CREATE PROCEDURE insert_stu( max_num INT )
BEGIN
DECLARE i INT DEFAULT 0;
    SET autocommit = 0; #设置手动提交事务
    REPEAT #循环
    SET i = i + 1; #赋值
    INSERT INTO student_info (course_id, class_id ,student_id ,NAME )
VALUES(rand_num(10000,10100),rand_num(10000,10200),rand_num(1,200000),rand_s
tring(6));
    UNTIL i = max_num
    END REPEAT;
    COMMIT; #提交事务
END //
DELIMITER ;
```

第4步：调用存储过程

```
CALL insert_course(100);
CALL insert_stu(1000000);
```

3.2哪些情况适合创建索引

1.字段的数值有唯一性的限制

索引本身可以起到约束的作用，比如唯一索引、主键索引都是可以起到唯一性约束的，因此在我们的数据表中如果`某个字段是唯一性的`，就可以直接`创建唯一性索引`，或者`主键索引`。这样可以更快速地通过该索引来确定某条记录。

例如，学生表中`学号`是具有唯一性的字段，为该字段建立唯一性索引可以很快确定某个学生的信息，如果使用`姓名`的话，可能存在同名现象，从而降低查询速度。

> 业务上具有唯一特性的字段，即使是组合字段，也必须建成唯一索引。（来源：Alibaba）
>
> 说明：不要以为唯一索引影响了 insert 速度，这个速度损耗可以忽略，但提高查找速度是明显的。

2.频繁作为WHERE查询条件的字段

某个字段在SELECT语句的WHERE条件中经常被使用到，那么就需要给这个字段创建索引了。尤其是在数据量大的情况下，创建普通索引就可以大幅提升数据查询的效率。

比如student_info数据表（含100万条数据），假设我们想要查询student_id=123110的用户信息。

如果我们没有对student_id字段创建索引，进行如下查询：

```
SELECT course_id,class_id,name,create_time,student_id
FROM student_info
WHERE student_id = 123110;
```

运行结果：

```
mysql> SELECT course_id,class_id,name,create_time,student_id
    -> FROM student_info
    -> WHERE student_id = 123110;
+-----------+----------+--------+---------------------+------------+
| course_id | class_id | name   | create_time         | student_id |
+-----------+----------+--------+---------------------+------------+
|     10052 |    10087 | OaKzqi | 2024-04-16 05:10:31 |     123110 |
|     10071 |    10030 | JpEaqF | 2024-04-16 05:10:33 |     123110 |
|     10021 |    10050 | qNQRMi | 2024-04-16 05:10:59 |     123110 |
|     10077 |    10076 | WQoZew | 2024-04-16 05:11:13 |     123110 |
|     10075 |    10074 | XXSwJP | 2024-04-16 05:11:14 |     123110 |
|     10015 |    10163 | HsSlCe | 2024-04-16 05:11:38 |     123110 |
+-----------+----------+--------+---------------------+------------+
6 rows in set (0.23 sec)
```

运行时间为0.20s，当我们对student_id字段创建索引之后，运行时间为0.01s，原来查询时间的1/20，效率提升还是明显的。

```
mysql> SELECT course_id,class_id,name,create_time,student_id
    -> FROM student_info
    -> WHERE student_id = 123110;
+-----------+----------+--------+---------------------+------------+
| course_id | class_id | name   | create_time         | student_id |
+-----------+----------+--------+---------------------+------------+
|     10052 |    10087 | OaKzqi | 2024-04-16 05:10:31 |     123110 |
|     10071 |    10030 | JpEaqF | 2024-04-16 05:10:33 |     123110 |
|     10021 |    10050 | qNQRMi | 2024-04-16 05:10:59 |     123110 |
|     10077 |    10076 | WQoZew | 2024-04-16 05:11:13 |     123110 |
|     10075 |    10074 | XXSwJP | 2024-04-16 05:11:14 |     123110 |
|     10015 |    10163 | HsSlCe | 2024-04-16 05:11:38 |     123110 |
+-----------+----------+--------+---------------------+------------+
6 rows in set (0.01 sec)
```

①查看`student_info`表中的索引

```
mysql> SHOW INDEX FROM student_info;
+--------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| Table        | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+--------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| student_info |          0 | PRIMARY  |            1 | id          | A         |      993366 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
+--------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
1 row in set (0.00 sec)
```

可以看出，我们没有对student_id字段创建索引。

②进行如下查询，耗时220ms

```
mysql> SELECT
    ->   course_id,
    ->   class_id,
    ->   NAME,
    ->   create_time,
    ->   student_id
    -> FROM
    ->   student_info
    -> WHERE student_id = 123110 ;
+-----------+----------+--------+---------------------+------------+
| course_id | class_id | NAME   | create_time         | student_id |
+-----------+----------+--------+---------------------+------------+
|     10065 |    10062 | hSUSFy | 2022-08-08 22:31:55 |     123110 |
|     10048 |    10162 | HuaVAP | 2022-08-08 22:32:14 |     123110 |
|     10000 |    10065 | fHVJwo | 2022-08-08 22:32:23 |     123110 |
|     10059 |    10176 | xuFRts | 2022-08-08 22:32:40 |     123110 |
|     10064 |    10021 | NVqSpU | 2022-08-08 22:32:42 |     123110 |
+-----------+----------+--------+---------------------+------------+
5 rows in set (0.22 sec)
```

③添加索引

```
1  alter table student_info add index idx_sid(student_id);
```

④再查询。耗时0ms。性能提升杠杠的~

```
mysql> SELECT
    ->   course_id,
    ->   class_id,
    ->   NAME,
    ->   create_time,
    ->   student_id
    -> FROM
    ->   student_info
    -> WHERE student_id = 123110 ;
+-----------+----------+--------+---------------------+------------+
| course_id | class_id | NAME   | create_time         | student_id |
+-----------+----------+--------+---------------------+------------+
|     10065 |    10062 | hSUSFy | 2022-08-08 22:31:55 |     123110 |
|     10048 |    10162 | HuaVAP | 2022-08-08 22:32:14 |     123110 |
|     10000 |    10065 | fHVJwo | 2022-08-08 22:32:23 |     123110 |
|     10059 |    10176 | xuFRts | 2022-08-08 22:32:40 |     123110 |
|     10064 |    10021 | NVqSpU | 2022-08-08 22:32:42 |     123110 |
+-----------+----------+--------+---------------------+------------+
5 rows in set (0.00 sec)
```

3.经常GROUPBY和ORDERBY的列

索引就是让数据按照某种顺序进行存储或检索，因此当我们使用GROUP BY对数据进行分组查询，或者使用ORDER BY对数据进行排序的时候，就需要`对分组或者排序的字段进行索引`。如果待排序的列有多个，那么可以在这些列上建立`组合索引`。

比如，按照student_id对学生选修的课程进行分组，显示不同的student_id和课程数量，显示100个即可。如果我们不对student_id创建索引，执行下面的sQL语句：

```
SELECT student_id,count(*) as num FROM student_info group by student_id limit 100;
```

```
mysql> SELECT student_id,count(*) as num FROM student_info group by student_id limit 100;
+------------+-----+
| student_id | num |
+------------+-----+
|          1 |   1 |
|          2 |   2 |
|          3 |   4 |
|          4 |  10 |
......
|         98 |   6 |
|         99 |   5 |
|        100 |   5 |
|        101 |   4 |
+------------+-----+
100 rows in set (2.30 sec)
```

如果我们对student_id创建索引，再执行SQL语句。结果如下：

```
mysql> SELECT student_id,count(*) as num FROM student_info group by student_id limit 100;
+------------+-----+
| student_id | num |
+------------+-----+
|          1 |   1 |
|          2 |   2 |
|          3 |   4 |
|          4 |  10 |
......
|         98 |   6 |
|         99 |   5 |
|        100 |   5 |
|        101 |   4 |
+------------+-----+
100 rows in set (0.00 sec)
```

运行结果（100 条记录，运行时间`0.00s`），效率提升很明显。而且，得到的结果中 student_id字段的数值也是`按照顺序展示`的。

同样，如果是ORDER BY，也需要对字段创建索引。

如果同时有GROUP BY和ORDER BY的情况：比如我们按照student_id进行分组，同时按照创建时间降序的方式进行排序，这时我们就需要同时进行GROUP BY和ORDER BY，那么是不是需要单独创建student_id的索引和create_time的索引呢?

当我们对student_id和create_time`分别创建索引`，执行下面的 SQL查询：

```
SELECT student_id,COUNT(*) AS num FROM student_info
GROUP BY student_id
ORDER BY create_time DESC
LIMIT 100;
```

运行结果：

```
mysql> SELECT student_id,COUNT(*) AS num FROM student_info GROUP BY
student_id ORDER BY create_time DESC LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|      15181 |   1 |
|     103247 |   1 |
|     172915 |   1 |
|     146587 |   1 |
|     132944 |   1 |
......
|     162351 |   1 |
|      45294 |   1 |
|      73588 |   1 |
|      40499 |   1 |
|      33809 |   1 |
+------------+-----+
100 rows in set (3.90 sec)
```

添加单列索引

```
ALTER TABLE student_info ADD INDEX idx_sid(student_id);
ALTER TABLE student_info ADD INDEX idx_cre_time(create_time);
```

再查询

```
mysql> SELECT student_id,COUNT(*) AS num FROM student_info GROUP BY
student_id ORDER BY create_time DESC LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|      15181 |   1 |
|     103247 |   1 |
|     172915 |   1 |
|     146587 |   1 |
|     132944 |   1 |
......
|     162351 |   1 |
|      45294 |   1 |
|      73588 |   1 |
|      40499 |   1 |
|      33809 |   1 |
+------------+-----+
100 rows in set (2.70 sec)
```

```
mysql> EXPLAIN SELECT student_id,COUNT(*) AS num FROM student_info GROUP BY
student_id ORDER BY create_time DESC LIMIT 100\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: student_info
   partitions: NULL
         type: index
possible_keys: idx_sid
          key: idx_sid
      key_len: 4
          ref: NULL
         rows: 997194
     filtered: 100.00
        Extra: Using temporary; Using filesort
1 row in set, 1 warning (0.00 sec)
```

没使用create_time索引

添加联合索引

```
ALTER TABLE student_info
    ADD INDEX idx_sid_cre_time (student_id, create_time DESC);
```

```
mysql> SELECT student_id,COUNT(*) AS num FROM student_info GROUP BY
student_id ORDER BY create_time DESC LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|      15181 |   1 |
|     103247 |   1 |
|     172915 |   1 |
|     146587 |   1 |
|     132944 |   1 |
......
|     162351 |   1 |
|      45294 |   1 |
|      73588 |   1 |
|      40499 |   1 |
|      33809 |   1 |
+------------+-----+
100 rows in set (0.22 sec)
```

```
mysql>  EXPLAIN SELECT student_id,COUNT(*) AS num FROM student_info GROUP BY
student_id ORDER BY create_timee DESC LIMIT 100\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: student_info
   partitions: NULL
         type: index
possible_keys: idx_sid_cre_time
          key: idx_sid_cre_time
      key_len: 10
```

```
          ref: NULL
         rows: 997194
     filtered: 100.00
        Extra: Using index; Using temporary; Using filesort
1 row in set, 1 warning (0.00 sec)
```

再进一步

```
ALTER TABLE student_info
    ADD INDEX idx_cre_time_sid ( create_time desc,student_id);
drop index idx_sid_cre_time on student_info;
```

```
mysql>  EXPLAIN SELECT student_id,COUNT(*) AS num FROM student_info GROUP BY
student_id ORDER BY create_time DESC LIMIT 100\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: student_info
   partitions: NULL
         type: index
possible_keys: idx_cre_time_sid,idx_sid
          key: idx_sid
      key_len: 4
          ref: NULL
         rows: 997194
     filtered: 100.00
        Extra: Using temporary; Using filesort
1 row in set, 1 warning (0.00 sec)

```

```
mysql> SELECT student_id,COUNT(*) AS num FROM student_info GROUP BY
student_id ORDER BY create_time DESC LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|      15181 |   1 |
|     103247 |   1 |
|     172915 |   1 |
|     146587 |   1 |
|     132944 |   1 |
......
|     162351 |   1 |
|      45294 |   1 |
|      73588 |   1 |
|      40499 |   1 |
|      33809 |   1 |
+------------+-----+
100 rows in set (3.28 sec)
```

说明：多个单列索引在多条件查询时`只会生效一个索引`（MySQL会选择其中一个限制最严格的作为索引），所以在多条件联合查询的时候`最好创建联合索引`。接着，我们创建联合索引（student_id，create_time），查询时间为0.22s，效率提升了很多。

如果我们创建联合索引的顺序为（create_time，student_id）呢？运行时间为2。164s，因为在进行SELECT查询的时候，先进行GROUP BY，再对数据进行ORDERBY的操作，所以按照（student_id，create_time)这个联合索引的顺序效率是最高的。

索引其实就是让数据按照某种顺序进行存储或检没使用create_time索引索。当我们使用 GROUP BY 对数据进行分组查询，或者使用 ORDER BY 对数据进行排序的时候，如果`对分组或者排序的字段建立索引`，本身索引的数据就已经排好序了，进行分组查询和排序操作性能不是很nice吗？另外，如果待排序的列有多个，那么可以在这些列上建立`组合索引`。

①下面在有`student_id`索引的情况下，查询：

```
mysql> SELECT student_id,COUNT(*) AS num
    -> FROM student_info
    -> GROUP BY student_id
    -> LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|          1 |   5 |
.....此处省略n行......
|          3 |   4 |
|        101 |   7 |
+------------+-----+
100 rows in set (0.00 sec)
```

②删除索引

```
#删除idx_sid索引
DROP INDEX idx_sid ON student_info;
```

③再次查询，慢的像蜗牛~

```
mysql> SELECT student_id,COUNT(*) AS num
    -> FROM student_info
    -> GROUP BY student_id
    -> LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|      95666 |   9 |
.....此处省略n行......
|     173440 |  14 |
|      67234 |   9 |
+------------+-----+
100 rows in set (0.78 sec)
```

同样，如果是ORDER BY，也需要对字段创建索引

④如果同时使用`GROUP BY`和`ORDER BY`，先看看不加索引的情况

```
mysql> SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;
ERROR 1055 (42000): Expression #1 of ORDER BY clause is not in GROUP BY clause and contains nonaggregated column 'atguigudb1.student_info.create_time' which is not functionally dependent on columns in GROUP BY clause; this is incompatible with sql_mode=only_full_group_by
```

⑤出现了一个异常信息，这是因为我们使用的`sql_mode`是`only_full_group_by`。修改下再来查询，时间代价是6.61s

```
mysql> SELECT @@sql_mode;
+--------------------------------------------------------------------------------------------+
| @@sql_mode                                                                                 |
+--------------------------------------------------------------------------------------------+
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION |
+--------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SET @@sql_mode =
'STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO
,NO_ENGINE_SUBSTITUTION';
Query OK, 0 rows affected (0.00 sec); # 去掉ONLY_FULL_GROUP_BY

mysql> SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|      21497 |   1 |
|      17311 |   1 |
.....此处省略n行.....
|     183509 |   1 |
+------------+-----+
100 rows in set (6.61 sec)
```

⑥再看看两个字段分别建立单列索引的情况，耗时5.26 s，快了一点点

```
mysql> ALTER TABLE student_info ADD INDEX idx_sid(student_id);
Query OK, 0 rows affected (1.77 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE student_info ADD INDEX idx_cre_time(create_time);
Query OK, 0 rows affected (1.49 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|      64044 |   1 |
.....此处省略n行.....
|     101052 |   1 |
|     152620 |   1 |
+------------+-----+
100 rows in set (5.26 sec)
```

注意：建立多个单列索引，并不会都走，像刚才这个例子，只会走idx_sid索引

⑦分析下它的查询过程,原来我们只用了一个索引，由于我们是先`GROUP BY student_id`,后`ORDER BY create_time`，我们实际上只使用了索引`idx_sid`

```
mysql> EXPLAIN SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;
+----+-------------+--------------+------------+-------+---------------+---
-----+---------+------+--------+----------+---------------------------------
+
| id | select_type | table        | partitions | type  | possible_keys | key
    | key_len | ref  | rows   | filtered | Extra                           |
+----+-------------+--------------+------------+-------+---------------+---
-----+---------+------+--------+----------+---------------------------------
+
|  1 | SIMPLE      | student_info | NULL       | index | idx_sid       |
idx_sid | 4       | NULL | 997130 |   100.00 | Using temporary; Using
filesort |
+----+-------------+--------------+------------+-------+---------------+---
-----+---------+------+--------+----------+---------------------------------
+
1 row in set, 1 warning (0.00 sec)
```

⑧建立联合索引的情况,芜湖起飞，直接0.25s。此时我们用`EXPLAIN`查看命中的也是 联合索引

```
mysql>  ALTER TABLE student_info ADD INDEX
idx_sid_cre_time(student_id,create_time DESC);
Query OK, 0 rows affected (2.09 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
```

```
|       1226 |   8 |
.....此处省略n行.....
|       1400 |   2 |
+------------+-----+
100 rows in set (0.25 sec)

mysql> EXPLAIN SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;
+----+-------------+--------------+------------+-------+-------------------
------+-----------------+---------+------+--------+----------+------------
---------------------------------+
| id | select_type | table        | partitions | type  | possible_keys
     | key             | key_len | ref  | rows   | filtered | Extra
                               |
+----+-------------+--------------+------------+-------+-------------------
------+-----------------+---------+------+--------+----------+------------
---------------------------------+
|  1 | SIMPLE      | student_info | NULL       | index |
idx_sid,idx_sid_cre_time | idx_sid_cre_time | 10      | NULL | 997130 |
100.00 | Using index; Using temporary; Using filesort |
+----+-------------+--------------+------------+-------+-------------------
------+-----------------+---------+------+--------+----------+------------
---------------------------------+
1 row in set, 1 warning (0.00 sec)
```

⑨再来测试,交换字段顺序建立联合索引`idx_cre_time_sid`，耗时5.24s。下面查询真正使用的索引`key`是`idx_sid`

```
mysql> ALTER TABLE student_info ADD INDEX idx_cre_time_sid(create_time
DESC,student_id);
Query OK, 0 rows affected (2.10 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> DROP INDEX idx_sid_cre_time ON student_info; #删除联合索引
idx_sid_cre_time
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> show INDEX from student_info; # 查看student_info中的索引
+--------------+------------+------------------+--------------+-------------
+-----------+-------------+----------+--------+------+------------+---------
+---------------+---------+------------+
| Table        | Non_unique | Key_name         | Seq_in_index | Column_name
| Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment
| Index_comment | Visible | Expression |
+--------------+------------+------------------+--------------+-------------
+-----------+-------------+----------+--------+------+------------+---------
+---------------+---------+------------+
| student_info |          0 | PRIMARY          |            1 | id
 | A         |      993366 |     NULL |   NULL |      | BTREE      |
|               | YES     | NULL       |
```

```
| student_info |          1 | idx_sid          |            1 | student_id
 | A         |      199180 |     NULL |   NULL |       | BTREE      |
|               | YES     | NULL       |
| student_info |          1 | idx_cre_time     |            1 | create_time
| A         |          82 |     NULL |   NULL | YES  | BTREE      |
|               | YES     | NULL       |
| student_info |          1 | idx_cre_time_sid |            1 | create_time
| D         |          77 |     NULL |   NULL | YES  | BTREE      |
|               | YES     | NULL       |
| student_info |          1 | idx_cre_time_sid |            2 | student_id
 | A         |      967825 |     NULL |   NULL |       | BTREE      |
|               | YES     | NULL       |
+--------------+------------+------------------+--------------+------------
+-----------+-------------+----------+--------+------+-----------+---------
+---------------+---------+------------+
5 rows in set (0.00 sec)

mysql> SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;
+------------+-----+
| student_id | num |
+------------+-----+
|      64044 |   1 |
.....此处省略n行......
|     101052 |   1 |
|     152620 |   1 |
+------------+-----+
100 rows in set (5.24 sec)

mysql> EXPLAIN SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;#起作用的是idx_sid
+----+-------------+--------------+------------+-------+-------------------
------+---------+---------+------+--------+----------+---------------------
-----------+
| id | select_type | table        | partitions | type  | possible_keys
     | key     | key_len | ref  | rows   | filtered | Extra
          |
+----+-------------+--------------+------------+-------+-------------------
------+---------+---------+------+--------+----------+---------------------
-----------+
|  1 | SIMPLE      | student_info | NULL       | index |
idx_sid,idx_cre_time_sid | idx_sid | 4       | NULL | 997130 |   100.00 |
Using temporary; Using filesort |
+----+-------------+--------------+------------+-------+-------------------
------+---------+---------+------+--------+----------+---------------------
-----------+
1 row in set, 1 warning (0.00 sec)
```

总结：如果我们仅仅使用GROUP BY 或者 ORDER BY，且后面只有一个字段，则单独建立索引；如果后面跟多个字段，则建立联合索引。如果既有GROUP BY 又有 ORDER BY，那就建立联合索引，且GROUP BY的字段写在前面，ORDER BY的字段写在后面。8.0后的版本也可以考虑使用降序索引

4.UPDATE、DELETE的WHERE条件列

当我们对某条数据进行UPDATE或者DELETE操作的时候，是否也需要对WHERE的条件列创建索引呢?

我们先看一下对数据进行UPDATE的情况：我们想要把name为462eed7ac6e791292a79对应的student_id修改为10002，当我们没有对name进行索引的时候，执行SQL语句：

```
UPDATE student_info SET student_id = 10002 WHERE name ='462eed7ac6e791292a79'
```

运行结果为Affected rows:1，运行时间为`0.578s`。

你能看到效率不高，但如果我们对name字段创建了索引，然后执行类似的SQL语句：

```
UPDATE student_info SET student_id = 10001 WHERE name ='462eed7ac6e791292a79'
```

运行结果为Affected rows:1，运行时间仅为`0.001s`。效率有了大幅的提升。

如果我们对某条数据进行DELETE，效率如何呢?

比如我们想删除name为462eed7ac6e791292a79的数据。当我们没有对name字段进行索引的时候，执行SQL语句：

```
DELETE FROM student_info WHERE name ='462eed7ac6e791292a79'
```

运行结果为Affected rows:1，运行时间为`0.627s`，效率不高。

如果我们对name创建了索引，再来执行这条SQL语句，运行时间为`0.03s`，效率有了大幅的提升。

对数据按照某个条件进行查询后再进行UPDATE或DELETE的操作，如果对WHERE字段创建了索引，就能大幅提升效率。原理是因为我们需要先根据WHERE条件列检索出来这条记录，然后再对它进行更新或删除。**如果进行更新的时候，更新的字段是非索引字段，提升的效率会更明显，这是因为非索引字段更新不需要对索引进行维护。**

```
mysql> UPDATE student_info SET student_id = 10002
    ->  WHERE NAME = '462eed7ac6e791292a79';# 550ms
Query OK, 0 rows affected (0.55 sec)
Rows matched: 0  Changed: 0  Warnings: 0

mysql> ALTER TABLE student_info
    -> ADD INDEX idx_name(NAME);
Query OK, 0 rows affected (2.26 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> UPDATE student_info SET student_id = 10002
    -> WHERE NAME = '462eed7ac6e791292a79';# 1ms
Query OK, 0 rows affected (0.001 sec)
Rows matched: 0  Changed: 0  Warnings: 0
```

5.DISTINCT字段需要创建索引

有时候我们需要对某个字段进行去重，使用`DISTINCT`，那么对这个字段创建索引，也会提升查询效率。

比如，我们想要查询课程表中不同的student_id都有哪些，如果我们没有对student_id创建索引，执行SQL语句：

```sql
SELECT DISTINCT(student_id) FROM `student_info`;
```

运行结果（600637条记录，运行时间`0.683s`）：

如果我们对student_id创建索引，再执行SQL语句：

```sql
SELECT DISTINCT(student_id) FROM `student_info`;
```

运行结果（600637条记录，运行时间`0.010s`）：

你能看到SQL查询效率有了提升，同时显示出来的student_id还是按照`递增的顺序`进行展示的。这是因为索引会对数据按照某种顺序进行排序，所以在去重的时候也会快很多。

6.多表JOIN连接操作时，创建索引注意事项

首先，`连接表的数量尽量不要超过3张`，因为每增加一张表就相当于增加了一次嵌套的循环，数量级增长会非常快(n ,n^2 , n^3...)，严重影响查询的效率。

其次，`对WHERE条件创建索引`，因为WHERE才是对数据条件的过滤。如果在数据量非常大的情况下，没有WHERE条件过滤是非常可怕的。

最后，`对用于连接的字段创建索引`，并且该字段在多张表中的`类型必须一致`。比如course_id在student_info表和course表中都为int(11)类型，而不能一个为int另一个为varchar类型。

举个例子，如果我们只对student_id创建索引，执行SQL语句：

```sql
SELECT course_id, name, student_info.student_id, course_name
FROM student_info JOIN course
ON student_info.course_id = course.course_id
WHERE name = '462eed7ac6e791292a79';
```

运行结果（1条数据，运行时间`0.189s`）：

这里我们对name创建索引，再执行上面的SQL语句，运行时间为`0.002s`。

> 注意：对于用连接的字段创建索引，这些字段在多张表中的 类型必须一致 。比如 course_id 在 student_info 表和 course 表中都为 int(11) 类型，而不能一个为 int 另一个为 varchar 类型。否则在查询时，虽然也会帮我们进行隐式的类型转换，转换时会使用函数，但会导致索引失效。索引失效情况在后续文章中还会给大家详细介绍，敬请期待。

举个例子，如果我们只对 student_id 创建索引，执行 SQL 语句，耗时0.21s

```
mysql> SELECT c.course_id, NAME, s.student_id, course_name
    -> FROM student_info s JOIN course c
    -> ON s.course_id = c.course_id
    -> WHERE NAME = 'WloNYD';
+-----------+--------+------------+-------------+
| course_id | NAME   | student_id | course_name |
+-----------+--------+------------+-------------+
|     10077 | WloNYD |      95666 | JfydVs      |
|     10077 | WloNYD |      95666 | nZkayq      |
|     10077 | WloNYD |      95666 | mTHDYg      |
|     10085 | wLonyD |      98444 | pZdpsR      |
+-----------+--------+------------+-------------+
4 rows in set (0.21 sec)
```

这时，我们对 name 创建索引，再执行上面的 SQL 语句，耗时 0.00s

```
mysql> ALTER TABLE student_info
    -> ADD INDEX idx_name(NAME);# 为name创建索引
Query OK, 0 rows affected (2.52 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT c.course_id, name, s.student_id, course_name
    -> FROM student_info s JOIN course c
    -> ON s.course_id = c.course_id
    -> WHERE name = 'WloNYD';
+-----------+--------+------------+-------------+
| course_id | name   | student_id | course_name |
+-----------+--------+------------+-------------+
|     10077 | WloNYD |      95666 | mTHDYg      |
|     10077 | WloNYD |      95666 | nZkayq      |
|     10085 | wLonyD |      98444 | pZdpsR      |
|     10077 | WloNYD |      95666 | JfydVs      |
+-----------+--------+------------+-------------+
4 rows in set (0.00 sec)
```

7.使用列的类型小的创建索引

我们这里所说的`类型大小`指的就是该类型表示的数据范围的大小。

我们在定义表结构的时候要显式的指定列的类型，以整数类型为例，有`TINYINT`、`MEDIUMINT`、`INT`、`BIGINT`等，它们占用的存储空间依次递增，能表示的整数范围当然也是依次递增。如果我们想要对某个整数列建立索引的话，在表示的整数范围允许的情况下，尽量让索引列使用较小的类型，比如我们能使用`INT`就不要使用`BIGINT`，能使用`MEDIUMINT`就不要使用`INT`。这是因为：

- 数据类型越小，在查询时进行的比较操作越快
- 数据类型越小，索引占用的存储空间就越少，在一个数据页内就可以`放下更多的记录`，从而减少磁盘`I/O`带来的性能损耗，也就意味着可以把更多的数据页缓存在内存中，从而加快读写效率。

这个建议对于表的`主键来说更加适用`，因为不仅是聚簇索引中会存储主键值，其他所有的二级索引的节点处都会存储一份记录的主键值，如果主键使用更小的数据类型，也就意味着节省更多的存储空间和更高效的I/O。

8.使用字符串前缀创建索引

假设我们的字符串很长，那存储一个字符串就需要占用很大的存储空间。在我们需要为这个字符串列建立索引时，那就意味着在对应的B+树中有这么两个问题：

- B+树索引中的记录需要把该列的完整字符串存储起来，更费时。而且字符串越长，`在索引中占用的存储空间越大`。
- 如果B+树索引引中索引列存储的字符串很长，那在做字符串`比较时会占用更多的时间`。

我们可以通过截取字段的前面一部分内容建立索引，这个就叫`前缀索引`。这样在查找记录时虽然不能精确的定位到记录的位置，但是能定位到相应前缀所在的位置，然后根据前缀相同的记录的主键值回表查询完整的字符串值。既`节约空间`，又`减少了字符串的比较时间`，还大体能解决排序的问题。

例如，TEXT和BLOG类型的字段，进行全文检索会很浪费时间，如果只检索字段前面的若干字符，这样可以提高检索速度。

创建一张商户表，因为地址字段比较长，在地址字段上建立前缀索引

```
create table shop(address varchar(120) not null);
alter table shop add index(address(12));
```

问题是，截取多少呢？截取得多了，达不到节省索引存储空间的目的；截取得少了，重复内容太多，字段的散列度(选择性)会降低。**怎么计算不同的长度的选择性呢?**

先看一下字段在全部数据中的选择度：

```
select count(distinct address) / count(*) from shop;
```

通过不同长度去计算，与全表的选择性对比：

公式：

```
count(distinct left(列名, 索引长度))/count(*)
```

例如：

```
select count(distinct left(address,10)) / count(*) as sub10, -- 截取前10个字符的选择度
count(distinct left(address,15)) / count(*) as sub11, -- 截取前15个字符的选择度
count(distinct left(address,20)) / count(*) as sub12, -- 截取前20个字符的选择度
count(distinct left(address,25)) / count(*) as sub13 -- 截取前25个字符的选择度
from shop;
```

> 🎯 拓展：Alibaba《Java开发手册》
> 【 强制 】在 varchar 字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据实际文本区分度决定索引长度。
> 说明：索引的长度与区分度是一对矛盾体，一般对字符串类型数据，长度为 20 的索引，区分度会高达90% 以上 ，可以使用 count(distinct left(列名, 索引长度))/count(*)的区分度来确定。

引申另一个问题：索引列前缀对排序的影响

如果使用了索引列前缀，比方说前边只把address列的`前12个字符`放到了二级索引中，下边这个查询可能就有点儿尴尬了：

```
SELECT * FROM shop
ORDER BY address
LIMIT 12;
```

因为二级索引中不包含完整的address列信息，所以无法对前12个字符相同，后边的字符不同的记录进行排序，也就是使用索引列前缀的方式`无法支持使用索引排序`，只能使用文件排序。

拓展：Alibaba《Java开发手册》

【`强制`】在varchar字段上建立索引时，必须指定索引长度，没必要对全字段建立索引，根据实际文本区分度决定索引长度。

说明：索引的长度与区分度是一对矛盾体，一般对字符串类型数据，长度为20的索引，区分度会`高达90%以上`，可以使用count(distinctleft(列名,索引长度))/count(*)的区分度来确定。

9.区分度高(散列性高)的列适合作为索引

`列的基数`指的是某一列中不重复数据的个数，比方说某个列包含值`2，5，8，2，5，8，2，5，8`，虽然有`9`条记录，但该列的基数却是`3`。也就是说，**在记录行数一定的情况下，列的基数越大，该列中的值越分散；列的基数越小，该列中的值越集中。**这个列的基数指标非常重要，直接影响我们是否能有效的利用索引。最好为列的基数大的列建立索引，为基数太小列的建立索引效果可能不好。

可以使用公式`select count（distinct a）/count（*）from t1`计算区分度，越接近1越好，一般超过33%就算是比较高效的索引了。

拓展：联合索引把区分度高（散列性高）的列放在前面。

10.使用最频繁的列放到联合索引的左侧

这样也可以较少的建立一些索引。同时，由于"最左前缀原则"，可以增加联合索引的使用率。

11.在多个字段都要创建索引的情况下，联合索引优于单值索引

- 索引建立的多，维护的成本也高。
- 多个字段进行联合查询时，其实只使用到一个索引。如下，只用到了idx_sid索引

```
mysql> EXPLAIN SELECT student_id,COUNT(*) AS num FROM student_info
    -> GROUP BY student_id
    -> ORDER BY create_time DESC
    -> LIMIT 100;
```

- 在建立联合索引的相关字段做查询时，联合索引都能生效，使用频率比较高。足够优化sql执行的速度了

3.3限制索引的数目

在实际工作中，我们也需要注意平衡，索引的数目不是越多越好。我们需要限制每张表上的索引数量，建议单张表索引数量`不超过6个`。原因：

①每个索索引都需要占用`磁盘空间`，索引越多，需要的磁盘空间就越大。

②索引会影响`INSERT、DELETE、 UPDATE等语句的性能`，因为表中的数据更改的同时，索引也会进行调整和更新，会造成负担。

③优化器在选择如何优化查询时，会根据统一信息，对每一个可以用到的`索引来进行评估`，以生成出一个最好的执行计划，如果同时有很多个索引都可以用于查询，会增加MySQL优化器生成执行计划时间，降低查询性能。

解释：表中创建的索引过多，优化器在possible_keys中选择合适的key 时需要的成本也会更多。比如下面查询中possible_keys有两个，实际使用的key只有一个，这其实优化器判断的哟。

```
mysql> EXPLAIN SELECT student_id,COUNT(*) AS num FROM student_info
   -> GROUP BY student_id
   -> ORDER BY create_time DESC
   -> LIMIT 100;
+----+-------------+--------------+------------+-------+-------------------
------+---------+---------+------+--------+----------+---------------------
-----------+
| id | select_type | table        | partitions | type  | possible_keys
     | key     | key_len | ref  | rows   | filtered | Extra
          |
+----+-------------+--------------+------------+-------+-------------------
------+---------+---------+------+--------+----------+---------------------
-----------+
|  1 | SIMPLE      | student_info | NULL       | index |
idx_sid,idx_cre_time_sid | idx_sid | 4       | NULL | 997449 |   100.00 |
Using temporary; Using filesort |
+----+-------------+--------------+------------+-------+-------------------
------+---------+---------+------+--------+----------+---------------------
-----------+
1 row in set, 1 warning (0.00 sec)
```

3.4哪些情况不适合创建索引

1.在where中使用不到的字段，不要设置索引

WHERE条件（包括GROUP BY、ORDER BY）里用不到的字段不需要创建索引，索引的价值是快速定位，如果起不到定位的字段通常是不需要创建索引的。举个例子：

```
SELECT course_id, student_id, create_time
FROM student_info
WHERE student_id = 41251;
```

因为我们是按照student_id来进行检索的，所以不需要对其他字段创建索引，即使这些字段出现在SELECT字段中。

2.数据量小的表最好不要使用索引

如果表记录太少，比如少于1000个，那么是不需要创建索引的。表记录太少，是否创建索引`对查询效率的影响并不大`。甚至说，查询花费的时间可能比遍历索引的时间还要短，索引可能不会产生优化效果。

举例：创建表1：

```
CREATE TABLE t_without_index(
    a INT PRIMARY KEY AUTO_INCREMENT,
    b INT
);
```

提供存储过程1：

```
#创建存储过程
DELIMITER //
CREATE PROCEDURE t_wout_insert()
BEGIN
```

```
    DECLARE i INT DEFAULT 1;
    WHILE i <= 900
    DO
        INSERT INTO t_without_index(b) SELECT RAND()*10000;
        SET i = i + 1;
    END WHILE;
    COMMIT;
END //
DELIMITER ;

#调用
CALL t_wout_insert();
```

创建表2:

```
CREATE TABLE t_with_index(
    a INT PRIMARY KEY AUTO_INCREMENT,
    b INT,
    INDEX idx_b(b)
);
```

创建存储过程2:

```
#创建存储过程
DELIMITER //
CREATE PROCEDURE t_with_insert()
BEGIN
    DECLARE i INT DEFAULT 1;
    WHILE i <= 900
    DO
        INSERT INTO t_with_index(b) SELECT RAND()*10000;
        SET i = i + 1;
    END WHILE;
    COMMIT;
END //
DELIMITER ;

#调用
CALL t_with_insert();
```

查询对比:

```
mysql> select * from t_without_index where b = 9879;
+------+------+
| a    | b    |
+------+------+
| 1242 | 9879 |
+------+------+
1 row in set (0.00 sec)

mysql> select * from t_with_index where b = 9879;
+-----+------+
| a   | b    |
+-----+------+
```

```
| 112 | 9879 |
+-----+------+
1 row in set (0.00 sec)
```

你能看到运行结果相同，但是在数据量不大的情况下，索引就发挥不出作用了。

> 结论：在数据表中的数据行数比较少的情况下，比如不到 1000 行，是不需要创建索引的。

3.有大量重复数据的列上不要建立索引

在条件表达式中经常用到的不同值较多的列上建立索引，但字段中如果有大量重复数据，也不用创建索引。比如在学生表的“`性别`”字段上只有“男”与"女"两个不同值，因此无须建立索引。如果建立索引，不但不会提高查询效率，反而会`严重降低数据更新速度`。

举例1：要在100万行数据中查找其中的50万行（比如性别为男的数据），一旦创建了索引，你需要先访问50万次索引，然后再访问50万次数据表，这样加起来的开销比不使用索引可能还要大。

举例2：假设有一个学生表，学生总数为100万人，男性只有10个人，也就是占总人口的10万分之1。

学生表student_gender结构如下。其中数据表中的student_gender字段取值为0或1，0代表女性，1代表男性。

```
CREATE TABLE student_gender(
    student_id INT(11) NOT NULL,
    student_name VARCHAR(50) NOT NULL,
    student_gender TINYINT(1) NOT NULL,
    PRIMARY KEY(student_id)
)ENGINE = INNODB;
```

如果我们要筛选出这个学生表中的男性，可以使用：

```
SELECT * FROM student_gender WHERE student_gender = 1
```

运行结果（10条数据，运行时间`0.696s`）：

student_id	student_name	student_gender
110000	student_100000	1
210000	student_200000	1
.....	.....	.....
1010000	student_1000000	1

你能看到在未创建索引的情况下，运行的效率并不高。如果针对student_gender字段创建索引呢？

```
SELECT * FROM student_gender WHERE student_gender = 1
```

同样是10条数据，运行结果相同，时间却缩短到了`0.052s`，大幅提升了查询的效率。

其实通过这两个实验你也能看出来，索引的价值是帮你快速定位。如果想要定位的数据有很多，那么索引就失去了它的使用价值，比如通常情况下的性别字段。

> 结论：当数据重复度大，比如`高于 10%`的时候，也不需要对这个字段使用索引。

4.避免对经常更新的表创建过多的索引

第一层含义：频繁更新的字段不一定要创建索引。因为更新数据的时候，也需要更新索引，如果索引太多，在更新索引的时候也会造成负担，从而影响效率。

第二层含义：避免对经常更新的表创建过多的索引，并且索引中的列尽可能少。此时，虽然提高了查询速度，同时却会降低更新表的速度。

你能看到在未创建索引的情况下，运行的效率并不高。如果针对student_gender字段创建索引呢?

```
SELECT * FROM student_gender WHERE student_gender = 1
```

同样是10条数据，运行结果相同，时间却缩短到了`0.052s`，大幅提升了查询的效率。

其实通过这两个实验你也能看出来，索引的价值是帮你快速定位。如果想要定位的数据有很多，那么索引就失去了它的使用价值，比如通常情况下的性别字段。

> 结论：当数据重复度大，比如高于10%的时候，也不需要对这个字段使用索引。

5.不建议用无序的值作为索引

例如身份证、UUID(在索引比较时需要转为ASCII，并且插入时可能造成页分裂)、MD5、HASH、无序长字符串等。

6.删除不再使用或者很少使用的索引

表中的数据被大量更新，或者数据的使用方式被改变后，原有的一些索引可能不再需要。数据库管理员应当定期找出这些索引，将它们删除，从而减少索引对更新操作的影响。

7.不要定义冗余或重复的索引

①冗余索引

有时候有意或者无意的就对同一个列创建了多个索引，比如：index（a,b,c）相当于index（a）、index（a,b)、index(a,b,c)。

举例：建表语句如下

```
CREATE TABLE person_info(
    id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    birthday DATE NOT NULL,
    phone_number CHAR(11) NOT NULL,
    country varchar(100) NOT NULL,
    PRIMARY KEY (id),
    KEY idx_name_birthday_phone_number (name(10), birthday, phone_number),
    KEY idx_name (name(10))
);
```

我们知道，通过`idx_name_birthday_phone_number`索引就可以对`name`列进行快速搜索，再创建一个专门针对`name`列的索引就算是一个`冗余索引`，维护这个索引只会增加维护的成本，并不会对搜索有什么好处。

②重复索引

另一种情况，我们可能会对某个列重复建立索引，比方说这样：

```
CREATE TABLE repeat_index_demo (
    col1 INT PRIMARY KEY,
    col2 INT,
    UNIQUE uk_idx_c1 (col1),
    INDEX idx_c1 (col1)
);
```

我们看到，col1既是主键、又给它定义为一个唯一索引，还给它定义了一个普通索引，可是主键本身就会生成聚簇索引，所以定义的唯一索引和普通索引是重复的，这种情况要避免。

3.5 小结

索引是一把`双刃剑`，可提高查询效率，但也会降低插入和更新的速度并占用磁盘空间。

选择索引的最终目的是为了使查询的速度变快，上面给出的原则是最基本的准则，但不能拘泥于上面的准则，大家要在以后的学习和工作中进行不断的实践，根据应用的实际情况进行分析和判断，选择最合适的索引方式。