mysql day04 课堂笔记

1、索引（index）

1.1、什么是索引？

索引是在数据库表的字段上添加的，是为了提高查询效率存在的一种机制。

一张表的一个字段可以添加一个索引，当然，多个字段联合起来也可以添加索引。

索引相当于一本书的目录，是为了缩小扫描范围而存在的一种机制。

对于一本字典来说，查找某个汉字有两种方式：

第一种方式：一页一页挨着找，直到找到为止，这种查找方式属于全字典扫描。效率比较低。

第二种方式：先通过目录（索引）去定位一个大概的位置，然后直接定位到这个位置，做局域性扫描，缩小扫描的范围，快速的查找。这种查找方式属于通过索引检索，效率较高。

t_user

id(idIndex)　name(nameIndex)　email(emailIndex)　address (emailAddressIndex)

1　zhangsan...

2　lisi

3　wangwu

4　zhaoliu

5　hanmeimei

6　jack

```
select * from t_user where name = 'jack';
```

以上的这条 SQL 语句会去 name 字段上扫描，为什么？

因为查询条件是：name='jack'

如果 name 字段上没有添加索引（目录），或者说没有给 name 字段创建索引，MySQL 会进行全扫描，会将 name 字段上的每一个值都比对一遍。效率比较低。

MySQL 在查询方面主要就是两种方式：

第一种方式：全表扫描

第二种方式：根据索引检索。

注意：

在实际中，汉语字典前面的目录是排序的，按照 a b c d e f....排序，为什么排序呢？因为只有排序了才会有区间查找这一说！（缩小扫描范围其实就是扫描某个区间罢了！）

在 mysql 数据库当中索引也是需要排序的，并且这个所以的排序和 TreeSet 数据结构相同。TreeSet（TreeMap）底层是一个自平衡的二叉树！在 mysql

当中索引是一个 B-Tree 数据结构。

遵循左小又大原则存放。采用中序遍历方式遍历取数据。

1.2、索引的实现原理？

假设有一张用户表：t_user

id(PK)	name	每一行记录在硬盘上都有物理存储编号
100	zhangsan	0x1111
120	lisi	0x2222
99	wangwu	0x8888
88	zhaoliu	0x9999
101	jack	0x6666
55	lucy	0x5555
130	tom	0x7777

提醒 1：在任何数据库当中主键上都会自动添加索引对象，id 字段上自动有索引，因为 id 是 PK。另外在 mysql 当中，一个字段上如果有 unique 约束的话，也会自动创建索引对象。

提醒 2：在任何数据库当中，任何一张表的任何一条记录在硬盘存储上都有一个硬盘的物理存储编号。

提醒 3：在 mysql 当中，索引是一个单独的对象，不同的存储引擎以不同的形式存在，在 MyISAM 存储引擎中，索引存储在一个.MYI 文件中。在 InnoDB 存储引擎中索引存储在一个逻辑名称叫做 tablespace 的当中。在 MEMORY 存储引擎当中索引被存储在内存当中。不管索引存储在哪里，索引在 mysql 当中都是一个树的形式存在。（自平衡二叉树：B-Tree）

1.3、在 mysql 当中，主键上，以及 unique 字段上都会自动添加索引的!!!!
什么条件下，我们会考虑给字段添加索引呢？
条件 1：数据量庞大（到底有多么庞大算庞大，这个需要测试，因为每一个硬件环境不同）
条件 2：该字段经常出现在 where 的后面，以条件的形式存在，也就是说这个字段总是被扫描。
条件 3：该字段很少的 DML(insert delete update)操作。（因为 DML 之后，索引需要重新排序。）

建议不要随意添加索引，因为索引也是需要维护的，太多的话反而会降低系统的性能。
建议通过主键查询，建议通过 unique 约束的字段进行查询，效率是比较高的。

1.4、索引怎么创建？怎么删除？语法是什么？

创建索引：

mysql> create index emp_ename_index on emp(ename);

给 emp 表的 ename 字段添加索引，起名：emp_ename_index

删除索引：

mysql> drop index emp_ename_index on emp;

将 emp 表上的 emp_ename_index 索引对象删除。

1.5、在 mysql 当中，怎么查看一个 SQL 语句是否使用了索引进行检索？

```
mysql> explain select * from emp where ename = 'KING';
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   14 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
```

扫描 14 条记录：说明没有使用索引。type=ALL

```
mysql> create index emp_ename_index on emp(ename);

mysql> explain select * from emp where ename = 'KING';
+----+-------------+-------+------+----------------+----------------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys  | key            | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+----------------+----------------+---------+-------+------+-------------+
|  1 | SIMPLE      | emp   | ref  | emp_ename_index | emp_ename_index | 33      | const |    1 | Using where |
+----+-------------+-------+------+----------------+----------------+---------+-------+------+-------------+
```

1.6、索引有失效的时候，什么时候索引失效呢？

失效的第 1 种情况：

```
select * from emp where ename like '%T';
```

ename 上即使添加了索引，也不会走索引，为什么？

原因是因为模糊匹配当中以“%”开头了！

尽量避免模糊查询的时候以“%”开始。

这是一种优化的手段/策略。

```
mysql> explain select * from emp where ename like '%T';

+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL |   14 | Using where |
+----+-------------+-------+------+---------------+------+---------+------+------+-------------+
```

失效的第 2 种情况：

使用 or 的时候会失效，如果使用 or 那么要求 or 两边的条件字段都要有索引，才会走索引，如果其中一边有一个字段没有索引，那么另一个字段上的索引也会实现。所以这就是为什么不建议使用 or 的原因。

```
mysql> explain select * from emp where ename = 'KING' or job = 'MANAGER';

+----+-------------+-------+------+----------------+------+---------+------+------+-------------+
| id | select_type | table | type | possible_keys  | key  | key_len | ref  | rows | Extra       |
+----+-------------+-------+------+----------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | emp   | ALL  | emp_ename_index | NULL | NULL    | NULL |   14 | Using where |
+----+-------------+-------+------+----------------+------+---------+------+------+-------------+
```

失效的第 3 种情况：

使用复合索引的时候，没有使用左侧的列查找，索引失效

什么是复合索引？

两个字段，或者更多的字段联合起来添加一个索引，叫做复合索引。

```
create index emp_job_sal_index on emp(job,sal);

mysql> explain select * from emp where job = 'MANAGER';

+----+-------------+-------+------+-----------------+-----------------+---------+-------+------+-------------+
| id | select_type | table | type | possible_keys   | key             |
```

```
key_len | ref   | rows | Extra       |

    +----+-------------+-------+------+-----------------+-----------------+--
------+-------+------+-------------+
         |  1 | SIMPLE      | emp   | ref  | emp_job_sal_index | emp_job_sal_index
| 30      | const |    3 | Using where |

    +----+-------------+-------+------+-----------------+-----------------+--
------+-------+------+-------------+

         mysql> explain select * from emp where sal = 800;

    +----+-------------+-------+------+---------------+------+---------+------+--
----+-------------+
         | id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra       |

    +----+-------------+-------+------+---------------+------+---------+------+--
----+-------------+
         |  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL
|   14 | Using where |

    +----+-------------+-------+------+---------------+------+---------+------+--
----+-------------+
```

失效的第 4 种情况：

在 where 当中索引列参加了运算，索引失效。

```
mysql> create index emp_sal_index on emp(sal);

explain select * from emp where sal = 800;

    +----+-------------+-------+------+---------------+---------------+---------+
-------+------+-------------+
         | id | select_type | table | type | possible_keys | key           | key_len
| ref   | rows | Extra       |

    +----+-------------+-------+------+---------------+---------------+---------+
-------+------+-------------+
         |  1 | SIMPLE      | emp   | ref  | emp_sal_index | emp_sal_index | 9       |
const |    1 | Using where |

    +----+-------------+-------+------+---------------+---------------+---------+
-------+------+-------------+

         mysql> explain select * from emp where sal+1 = 800;

    +----+-------------+-------+------+---------------+------+---------+------+--
```

```
----+-------------+
          | id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra       |

     +----+-------------+-------+------+---------------+------+---------+------+--
----+-------------+
          |  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL
|   14 | Using where |

     +----+-------------+-------+------+---------------+------+---------+------+--
----+-------------+
```

失效的第 5 种情况：

在 where 当中索引列使用了函数

```
explain select * from emp where lower(ename) = 'smith';

     +----+-------------+-------+------+---------------+------+---------+------+--
----+-------------+
          | id | select_type | table | type | possible_keys | key  | key_len | ref  |
rows | Extra       |

     +----+-------------+-------+------+---------------+------+---------+------+--
----+-------------+
          |  1 | SIMPLE      | emp   | ALL  | NULL          | NULL | NULL    | NULL
|   14 | Using where |

     +----+-------------+-------+------+---------------+------+---------+------+--
----+-------------+
```

失效的第 6...

失效的第 7...

1.7、索引是各种数据库进行优化的重要手段。优化的时候优先考虑的因素就是索引。

索引在数据库当中分了很多类？

单一索引：一个字段上添加索引。

复合索引：两个字段或者更多的字段上添加索引。

主键索引：主键上添加索引。

唯一性索引：具有 unique 约束的字段上添加索引。

.....

注意：唯一性比较弱的字段上添加索引用处不大。

2、视图(view)

2.1、什么是视图？

view:站在不同的角度去看待同一份数据。

2.2、怎么创建视图对象？怎么删除视图对象？

表复制：

```
mysql> create table dept2 as select * from dept;
```

dept2 表中的数据：

```
mysql> select * from dept2;
+--------+------------+----------+
| DEPTNO | DNAME      | LOC      |
+--------+------------+----------+
|     10 | ACCOUNTING | NEW YORK |
|     20 | RESEARCH   | DALLAS   |
|     30 | SALES      | CHICAGO  |
|     40 | OPERATIONS | BOSTON   |
+--------+------------+----------+
```

创建视图对象：

```
create view dept2_view as select * from dept2;
```

删除视图对象：

```
drop view dept2_view;
```

注意：只有 DQL 语句才能以 view 的形式创建。

```
create view view_name as 这里的语句必须是 DQL 语句;
```

2.3、用视图做什么？

我们可以面向视图对象进行增删改查，对视图对象的增删改查，会导致原表被操作！（视图的特点：通过对视图的操作，会影响到原表数据。）

```
//面向视图查询
select * from dept2_view;

// 面向视图插入
insert into dept2_view(deptno,dname,loc) values(60,'SALES', 'BEIJING');

// 查询原表数据
mysql> select * from dept2;
+--------+------------+----------+
| DEPTNO | DNAME      | LOC      |
+--------+------------+----------+
|     10 | ACCOUNTING | NEW YORK |
|     20 | RESEARCH   | DALLAS   |
|     30 | SALES      | CHICAGO  |
|     40 | OPERATIONS | BOSTON   |
```

```
|     60 | SALES      | BEIJING  |
+--------+------------+----------+

// 面向视图删除
mysql> delete from dept2_view;

// 查询原表数据
mysql> select * from dept2;
Empty set (0.00 sec)


// 创建视图对象
create view
     emp_dept_view
as
     select
          e.ename,e.sal,d.dname
     from
          emp e
     join
          dept d
     on
          e.deptno = d.deptno;

// 查询视图对象
mysql> select * from emp_dept_view;
+--------+---------+------------+
| ename  | sal     | dname      |
+--------+---------+------------+
| CLARK  | 2450.00 | ACCOUNTING |
| KING   | 5000.00 | ACCOUNTING |
| MILLER | 1300.00 | ACCOUNTING |
| SMITH  |  800.00 | RESEARCH   |
| JONES  | 2975.00 | RESEARCH   |
| SCOTT  | 3000.00 | RESEARCH   |
| ADAMS  | 1100.00 | RESEARCH   |
| FORD   | 3000.00 | RESEARCH   |
| ALLEN  | 1600.00 | SALES      |
| WARD   | 1250.00 | SALES      |
| MARTIN | 1250.00 | SALES      |
| BLAKE  | 2850.00 | SALES      |
| TURNER | 1500.00 | SALES      |
| JAMES  |  950.00 | SALES      |
+--------+---------+------------+

// 面向视图更新
update emp_dept_view set sal = 1000 where dname = 'ACCOUNTING';
```

```
// 原表数据被更新
mysql> select * from emp;
+-------+--------+-----------+------+------------+---------+---------+-------
-+
| EMPNO | ENAME  | JOB       | MGR  | HIREDATE   | SAL     | COMM    | DEPTNO |
+-------+--------+-----------+------+------------+---------+---------+-------
-+
|  7369 | SMITH  | CLERK     | 7902 | 1980-12-17 |  800.00 |    NULL |     20 |
|  7499 | ALLEN  | SALESMAN  | 7698 | 1981-02-20 | 1600.00 |  300.00 |     30 |
|  7521 | WARD   | SALESMAN  | 7698 | 1981-02-22 | 1250.00 |  500.00 |     30 |
|  7566 | JONES  | MANAGER   | 7839 | 1981-04-02 | 2975.00 |    NULL |     20 |
|  7654 | MARTIN | SALESMAN  | 7698 | 1981-09-28 | 1250.00 | 1400.00 |     30 |
|  7698 | BLAKE  | MANAGER   | 7839 | 1981-05-01 | 2850.00 |    NULL |     30 |
|  7782 | CLARK  | MANAGER   | 7839 | 1981-06-09 | 1000.00 |    NULL |     10 |
|  7788 | SCOTT  | ANALYST   | 7566 | 1987-04-19 | 3000.00 |    NULL |     20 |
|  7839 | KING   | PRESIDENT | NULL | 1981-11-17 | 1000.00 |    NULL |     10 |
|  7844 | TURNER | SALESMAN  | 7698 | 1981-09-08 | 1500.00 |    0.00 |     30 |
|  7876 | ADAMS  | CLERK     | 7788 | 1987-05-23 | 1100.00 |    NULL |     20 |
|  7900 | JAMES  | CLERK     | 7698 | 1981-12-03 |  950.00 |    NULL |     30 |
|  7902 | FORD   | ANALYST   | 7566 | 1981-12-03 | 3000.00 |    NULL |     20 |
|  7934 | MILLER | CLERK     | 7782 | 1982-01-23 | 1000.00 |    NULL |     10 |
+-------+--------+-----------+------+------------+---------+---------+-------
-+
```

2.4、视图对象在实际开发中到底有什么用？《方便，简化开发，利于维护》

```
create view
    emp_dept_view
as
    select
        e.ename,e.sal,d.dname
    from
        emp e
    join
        dept d
    on
        e.deptno = d.deptno;
```

假设有一条非常复杂的 SQL 语句，而这条 SQL 语句需要在不同的位置上反复使用。每一次使用这个 sql 语句的时候都需要重新编写，很长，很麻烦，怎么办？

可以把这条复杂的 SQL 语句以视图对象的形式新建。

在需要编写这条 SQL 语句的位置直接使用视图对象，可以大大简化开发。

并且利于后期的维护，因为修改的时候也只需要修改一个位置就行，只需要修改视图对象所映射的 SQL 语句。

我们以后面向视图开发的时候，使用视图的时候可以像使用 table 一样。
可以对视图进行增删改查等操作。视图不是在内存当中，视图对象也是
存储在硬盘上的，不会消失。

再提醒一下：
视图对应的语句只能是 DQL 语句。
但是视图对象创建完成之后，可以对视图进行增删改查等操作。

小插曲：
增删改查，又叫做：CRUD。
CRUD 是在公司中程序员之间沟通的术语。一般我们很少说增删改查。
一般都说 CRUD。

C:Create（增）
R:Retrive（查：检索）
U:Update（改）
D:Delete（删）

3、DBA 常用命令？

重点掌握：
数据的导入和导出（数据的备份）
其它命令了解一下即可。（这个培训日志文档留着，以后忘了，可以打开文档复制粘贴。）

数据导出？
注意：在 windows 的 dos 命令窗口中：
mysqldump bjpowernode>D:\bjpowernode.sql -uroot -p123456

可以导出指定的表吗？
mysqldump bjpowernode emp>D:\bjpowernode.sql -uroot -p123456

数据导入？
注意：需要先登录到 mysql 数据库服务器上。
然后创建数据库：create database bjpowernode;
使用数据库：use bjpowernode
然后初始化数据库：source D:\bjpowernode.sql

4、数据库设计三范式

4.1、什么是数据库设计范式？
数据库表的设计依据。教你怎么进行数据库表的设计。

4.2、数据库设计范式共有？
3 个。
第一范式：要求任何一张表必须有主键，每一个字段原子性不可再分。

第二范式：建立在第一范式的基础之上，要求所有非主键字段完全依赖主键，
不要产生部分依赖。

第三范式：建立在第二范式的基础之上，要求所有非主键字段直接依赖主键，
不要产生传递依赖。

声明：三范式是面试官经常问的，所以一定要熟记在心！

设计数据库表的时候，按照以上的范式进行，可以避免表中数据的冗余，空间的浪费。

4.3、第一范式

最核心，最重要的范式，所有表的设计都需要满足。
必须有主键，并且每一个字段都是原子性不可再分。

学生编号 学生姓名 联系方式

1001	张三	zs@gmail.com,1359999999
1002	李四	ls@gmail.com,13699999999
1001	王五	ww@163.net,13488888888

以上是学生表，满足第一范式吗？

不满足，第一：没有主键。第二：联系方式可以分为邮箱地址和电话

学生编号(pk)	学生姓名	邮箱地址	联系电话
1001	张三	zs@gmail.com	1359999999
1002	李四	ls@gmail.com	13699999999
1003	王五	ww@163.net	13488888888

4.4、第二范式：

建立在第一范式的基础之上，
要求所有非主键字段必须完全依赖主键，不要产生部分依赖。

学生编号 学生姓名 教师编号 教师姓名

1001	张三	001	王老师
1002	李四	002	赵老师
1003	王五	001	王老师
1001	张三	002	赵老师

这张表描述了学生和老师的关系：（1 个学生可能有多个老师，1 个老师有多个学生）
这是非常典型的：多对多关系！

分析以上的表是否满足第一范式？

不满足第一范式。

怎么满足第一范式呢？修改

学生编号+教师编号(pk)		学生姓名	教师姓名
1001	001	张三	王老师
1002	002	李四	赵老师
1003	001	王五	王老师
1001	002	张三	赵老师

学生编号 教师编号，两个字段联合做主键，复合主键（PK: 学生编号+教师编号）
经过修改之后，以上的表满足了第一范式。但是满足第二范式吗？

不满足，“张三”依赖 1001，“王老师”依赖 001，显然产生了部分依赖。

产生部分依赖有什么缺点？

数据冗余了。空间浪费了。“张三”重复了，“王老师”重复了。

为了让以上的表满足第二范式，你需要这样设计：

使用三张表来表示多对多的关系!!!!

学生表

学生编号(pk)	学生名字
1001	张三
1002	李四
1003	王五

教师表

教师编号(pk)	教师姓名
001	王老师
002	赵老师

学生教师关系表

id(pk)	学生编号(fk)	教师编号(fk)
1	1001	001
2	1002	002
3	1003	001
4	1001	002

背口诀：

多对多怎么设计？

多对多，三张表，关系表两个外键!!!!!!!!!!!!!!!

4.5、第三范式

第三范式建立在第二范式的基础之上

要求所有非主键字典必须直接依赖主键，不要产生传递依赖。

学生编号（PK） 学生姓名 班级编号 班级名称

1001	张三	01	一年一班
1002	李四	02	一年二班
1003	王五	03	一年三班
1004	赵六	03	一年三班

以上表的设计是描述：班级和学生的关系。很显然是 1 对多关系！
一个教室中有多个学生。

分析以上表是否满足第一范式？
满足第一范式，有主键。

分析以上表是否满足第二范式？
满足第二范式，因为主键不是复合主键，没有产生部分依赖。主键是单一主键。

分析以上表是否满足第三范式？
第三范式要求：不要产生传递依赖！
一年一班依赖 01，01 依赖 1001，产生了传递依赖。
不符合第三范式的要求。产生了数据的冗余。

那么应该怎么设计一对多呢？

班级表：一

班级编号(pk)	班级名称
01	一年一班
02	一年二班
03	一年三班

学生表：多

学生编号（PK）	学生姓名	班级编号(fk)
1001	张三	01
1002	李四	02
1003	王五	03
1004	赵六	03

背口诀：
一对多，两张表，多的表加外键!!!!!!!!!!!!

4.6、总结表的设计？

一对多：
一对多，两张表，多的表加外键!!!!!!!!!!!!

多对多：

多对多，三张表，关系表两个外键!!!!!!!!!!!!!!!

一对一：

一对一放到一张表中不就行了吗？为啥还要拆分表？

在实际的开发中，可能存在一张表字段太多，太庞大。这个时候要拆分表。

一对一怎么设计？

没有拆分表之前：一张表

t_user

id　　login_name　　login_pwd　　real_name　　email　　address.....

1　　zhangsan　　123　　张三　　zhangsan@xxx

2　　lisi　　123　　李四　　lisi@xxx

...

这种庞大的表建议拆分为两张：

t_login 登录信息表

id(pk)　　login_name　　login_pwd

1　　zhangsan　　123

2　　lisi　　123

t_user 用户详细信息表

id(pk)　　real_name　　email　　address.....　　login_id(fk+unique)

100　　张三　　zhangsan@xxx　　1

200　　李四　　lisi@xxx　　2

口诀：一对一，外键唯一!!!!!!!!!!

4.7、嘱咐一句话：

数据库设计三范式是理论上的。

实践和理论有的时候有偏差。

最终的目的都是为了满足客户的需求，有的时候会拿冗余换执行速度。

因为在 sql 当中，表和表之间连接次数越多，效率越低。（笛卡尔积）

有的时候可能会存在冗余，但是为了减少表的连接次数，这样做也是合理的，并且对于开发人员来说，sql 语句的编写难度也会降低。

面试的时候把这句话说上：他就不会认为你是初级程序员了！