分页查询优化

```
示例表：
CREATE TABLE `employees` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `name` varchar(24) NOT NULL DEFAULT '' COMMENT '姓名',
 `age` int(11) NOT NULL DEFAULT '0' COMMENT '年龄',
 `position` varchar(20) NOT NULL DEFAULT '' COMMENT '职位',
 `hire_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '入职时间',
 PRIMARY KEY (`id`),
 KEY `idx_name_age_position` (`name`,`age`,`position`) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8 COMMENT='员工记录表';

```

很多时候我们业务系统实现分页功能可能会用如下sql实现

```
mysql> select * from employees limit 10000,10;
```

表示从表 employees 中取出从 10001 行开始的 10 行记录。看似只查询了 10 条记录，实际这条 SQL 是先读取 10010 条记录，然后抛弃前 10000 条记录，然后读到后面 10 条想要的数据。因此要查询一张大表比较靠后的数据，执行效率是非常低的。

>>常见的分页场景优化技巧：

1、根据自增且连续的主键排序的分页查询

首先来看一个根据自增且连续主键排序的分页查询的例子：

```
mysql> select * from employees limit 90000,5;
```

id	name	age	position	hire_time
90001	zhuge90001	90001	dev	2019-08-20 12:08:16
90002	zhuge90002	90002	dev	2019-08-20 12:08:16
90003	zhuge90003	90003	dev	2019-08-20 12:08:16
90004	zhuge90004	90004	dev	2019-08-20 12:08:16
90005	zhuge90005	90005	dev	2019-08-20 12:08:16

该 SQL 表示查询从第 90001开始的五行数据，没添加单独 order by，表示通过**主键排序。**我们再看表 employees，因为主键是自增并且连续的，所以可以改写成按照主键去查询从第 90001开始的五行数据，如下：

```
mysql> select * from employees where id > 90000 limit 5;
```

id	name	age	position	hire_time
90001	zhuge90001	90001	dev	2019-08-20 12:08:16
90002	zhuge90002	90002	dev	2019-08-20 12:08:16
90003	zhuge90003	90003	dev	2019-08-20 12:08:16
90004	zhuge90004	90004	dev	2019-08-20 12:08:16
90005	zhuge90005	90005	dev	2019-08-20 12:08:16

查询的结果是一致的。我们再对比一下执行计划：

```
mysql> EXPLAIN select * from employees limit 90000,5;
```

id	select_type	table	partitio	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	100183	100	(Null)

```
mysql> EXPLAIN select * from employees where id > 90000 limit 5;
```

id	select_type	table	partitio	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	(Null)	range	PRIMARY	PRIMARY	4	(Null)	19250	100	Using where

显然改写后的 SQL 走了索引，而且扫描的行数大大减少，执行效率更高。

但是，这条改写的SQL 在很多场景并不实用，因为表中可能某些记录被删后，主键空缺，导致结果不一致，如下图试验所示（先删除一条前面的记录，然后再测试原 SQL 和优化后的 SQL）：

查询创建工具 查询编辑器

```
1  select * from employees limit 90000,5;
2
```

信息 结果1 概况 状态

id	name	age	position	hire_time
90002	zhuge90002	90002	dev	2019-08-20 12:08:16
90003	zhuge90003	90003	dev	2019-08-20 12:08:16
90004	zhuge90004	90004	dev	2019-08-20 12:08:16
90005	zhuge90005	90005	dev	2019-08-20 12:08:16
90006	zhuge90006	90006	dev	2019-08-20 12:08:16

查询创建工具 查询编辑器

```
1  select * from employees where id > 90000 limit 5;
2
```

信息 结果1 概况 状态

id	name	age	position	hire_time
90001	zhuge90001	90001	dev	2019-08-20 12:08:16
90002	zhuge90002	90002	dev	2019-08-20 12:08:16
90003	zhuge90003	90003	dev	2019-08-20 12:08:16
90004	zhuge90004	90004	dev	2019-08-20 12:08:16
90005	zhuge90005	90005	dev	2019-08-20 12:08:16

两条 SQL 的结果并不一样，因此，如果主键不连续，不能使用上面描述的优化方法。

另外如果原 SQL 是 order by 非主键的字段，按照上面说的方法改写会导致两条 SQL 的结果不一致。所以这种改写得满足以下两个条件：

- 主键自增且连续
- 结果是按照主键排序的

2、根据非主键字段排序的分页查询

再看一个根据非主键字段排序的分页查询，SQL 如下：

mysql> select * from employees ORDER BY name limit 90000,5;

查询创建工具 查询编辑器

```
1  select * from employees ORDER BY name limit 90000,5;
2
```

信息 结果1 概况 状态

id	name	age	position	hire_time
90999	zhuge90999	90999	dev	2019-08-20 12:08:17
91	zhuge91	91	dev	2019-08-20 12:05:36
910	zhuge910	910	dev	2019-08-20 12:05:37
9100	zhuge9100	9100	dev	2019-08-20 12:05:52
91000	zhuge91000	91000	dev	2019-08-20 12:08:17

mysql> EXPLAIN select * from employees ORDER BY name limit 90000,5;

查询创建工具 查询编辑器

```
1  EXPLAIN select * from employees ORDER BY name limit 90000,5;
2
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	100183	100	Using filesort

发现并没有使用 name 字段的索引（key 字段对应的值为 null），具体原因上节课讲过：**扫描整个索引并查找到没索引的行(可能要遍历多个索引树)的成本比扫描全表的成本更高，所以优化器放弃使用索引。**

知道不走索引的原因，那么怎么优化呢?

其实关键是**让排序时返回的字段尽可能少**，所以可以让排序和分页操作先查出主键，然后根据主键查到对应的记录，SQL改写如下

```
mysql> select * from employees e inner join (select id from employees order by name limit 90000,5) ed
on e.id = ed.id;
```

查询创建工具 查询编辑器

```
select * from employees e inner join (select id from employees order by name limit 90000,5) ed on e.id = ed.id;
```

信息 结果1 概况 状态

id	name	age	position	hire_time	id1
90999	zhuge90999	90999	dev	2019-08-20 12:08:17	90999
91	zhuge91	91	dev	2019-08-20 12:05:36	91
910	zhuge910	910	dev	2019-08-20 12:05:37	910
9100	zhuge9100	9100	dev	2019-08-20 12:05:52	9100
91000	zhuge91000	91000	dev	2019-08-20 12:08:17	91000

需要的结果与原 SQL 一致，执行时间减少了一半以上，我们再对比优化前后sql的执行计划：

查询创建工具 查询编辑器

```
EXPLAIN select * from employees e inner join (select id from employees order by name limit 90000,5) ed on e.id = ed.id;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	90005	100	(Null)
1	PRIMARY	e	(Null)	eq_ref	PRIMARY	PRIMARY	4	ed.id	1	100	(Null)
2	DERIVED	employees	(Null)	index	(Null)	idx_name_age_position	140	(Null)	90005	100	Using index

原 SQL 使用的是 filesort 排序，而优化后的 SQL 使用的是索引排序。

Join关联查询优化

```
-- 示例表:
CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_a` (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

create table t2 like t1;

-- 插入一些示例数据
-- 往t1表插入1万行记录
drop procedure if exists insert_t1;
delimiter ;;
create procedure insert_t1()
begin
  declare i int;
  set i=1;
  while(i<=10000)do
    insert into t1(a,b) values(i,i);
    set i=i+1;
  end while;
end;;
delimiter ;
```

```
call insert_t1();

-- 往t2表插入100行记录
drop procedure if exists insert_t2;
delimiter ;;
create procedure insert_t2()
begin
  declare i int;
  set i=1;
  while(i<=100)do
    insert into t2(a,b) values(i,i);
    set i=i+1;
  end while;
end;;
delimiter ;
call insert_t2();
```

mysql的表关联常见有两种算法

- Nested-Loop Join 算法
- Block Nested-Loop Join 算法

1、 嵌套循环连接 Nested-Loop Join(NLJ) 算法

一次一行循环地从第一张表（称为**驱动表**）中读取行，在这行数据中取到关联字段，根据关联字段在另一张表（**被驱动表**）里取出满足条件的行，然后取出两张表的结果合集。

```
mysql> EXPLAIN select * from t1 inner join t2 on t1.a= t2.a;
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	partitior	type	possible_key	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t2	(Null)	ALL	idx_a	(Null)	(Null)	(Null)	100	100	Using where
1	SIMPLE	t1	(Null)	ref	idx_a	idx_a	5	test.t2.a	1	100	(Null)

从执行计划中可以看到这些信息：

- 驱动表是 t2，被驱动表是 t1。先执行的就是驱动表(执行计划结果的id如果一样则按从上到下顺序执行sql)；优化器一般会优先选择**小表做驱动表。所以使用 inner join 时，排在前面的表并不一定就是驱动表。**
- 当使用left join时，左表是驱动表，右表是被驱动表，当使用right join时，右表时驱动表，左表是被驱动表，当使用join时，mysql会选择数据量比较小的表作为驱动表，大表作为被驱动表。
- 使用了 NLJ算法。一般 join 语句中，如果执行计划 Extra 中未出现 **Using join buffer** 则表示使用的 join 算法是 NLJ。

上面sql的大致流程如下：

1. 从表 t2 中读取一行数据（如果t2表有查询过滤条件的，会从过滤结果里取出一行数据）；
2. 从第 1 步的数据中，取出关联字段 a，到表 t1 中查找；
3. 取出表 t1 中满足条件的行，跟 t2 中获取到的结果合并，作为结果返回给客户端；
4. 重复上面 3 步。

整个过程会读取 t2 表的所有数据(**扫描100行**)，然后遍历这每行数据中字段 a 的值，根据 t2 表中 a 的值索引扫描 t1 表中的对应行(**扫描100次 t1 表的索引，1次扫描可以认为最终只扫描 t1 表一行完整数据，也就是总共 t1 表也扫描了100行**)。因此整个过程扫描了 **200 行。**

如果被驱动表的关联字段没索引，**使用NLJ算法性能会比较低(下面有详细解释)**，mysql会选择Block Nested-Loop Join 算法。

2、 基于块的嵌套循环连接 Block Nested-Loop Join(BNL)算法

把**驱动表**的数据读入到 join_buffer 中，然后扫描**被驱动表**，把**被驱动表**每一行取出来跟 join_buffer 中的数据做对比。

```
mysql>EXPLAIN select * from t1 inner join t2 on t1.b= t2.b;
```

信息 结果1 概况 状态

id	select_type	table	partitior	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t2	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	100	100	(Null)
1	SIMPLE	t1	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	10337	10	Using where; Using join buffer (Block Nested Loo

Extra 中 的Using join buffer (Block Nested Loop)说明该关联查询使用的是 BNL 算法。

上面sql的大致流程如下：

1. 把 t2 的所有数据放入到 **join_buffer** 中
2. 把表 t1 中每一行取出来，跟 join_buffer 中的数据做对比
3. 返回满足 join 条件的数据

整个过程对表 t1 和 t2 都做了一次全表扫描，因此扫描的总行数为10000(表 t1 的数据总量) + 100(表 t2 的数据总量) = **10100。**并且 join_buffer 里的数据是无序的，因此对表 t1 中的每一行，都要做 100 次判断，所以内存中的判断次数是 100 * 10000= **100 万次。**

这个例子里表 t2 才 100 行，要是表 t2 是一个大表，join_buffer 放不下怎么办呢?

join_buffer 的大小是由参数 join_buffer_size 设定的，默认值是 256k。如果放不下表 t2 的所有数据话，策略很简单，就是**分段放。**

比如 t2 表有1000行记录， join_buffer 一次只能放800行数据，那么执行过程就是先往 join_buffer 里放800行记录，然后从 t1 表里取数据跟 join_buffer 中数据对比得到部分结果，然后清空 join_buffer ，再放入 t2 表剩余200行记录，再次从 t1 表里取数据跟 join_buffer 中数据对比。所以就多扫了一次 t1 表。

被驱动表的关联字段没索引为什么要选择使用 BNL 算法而不使用 Nested-Loop Join 呢?

如果上面第二条sql使用 Nested-Loop Join，那么扫描行数为 100 * 10000 = 100万次，这个是**磁盘扫描。**

很显然，用BNL磁盘扫描次数少很多，相比于磁盘扫描，BNL的内存计算会快得多。

因此MySQL对于被驱动表的关联字段没索引的关联查询，一般都会使用 BNL 算法。如果有索引一般选择 NLJ 算法，有索引的情况下 NLJ 算法比 BNL算法性能更高

对于关联sql的优化

- **关联字段加索引**，让mysql做join操作时尽量选择NLJ算法
- **小表驱动大表**，写多表连接sql时如果**明确知道**哪张表是小表可以用straight_join写法固定连接驱动方式，省去mysql优化器自己判断的时间

straight_join解释：straight_join功能同join类似，但能让左边的表来驱动右边的表，能改表优化器对于联表查询的执行顺序。

比如：select * from t2 straight_join t1 on t2.a = t1.a; 代表指定mysql选着 t2 表作为驱动表。

- **straight_join**只适用于inner join，并不适用于left join，right join。（因为left join，right join已经代表指定了表的执行顺序）
- 尽可能让优化器去判断，因为大部分情况下mysql优化器是比人要聪明的。使用**straight_join**一定要慎重，因为部分情况下人为指定的执行顺序并不一定会比优化引擎要靠谱。

对于小表定义的明确

在决定哪个表做驱动表的时候，应该是两个表按照各自的条件过滤，**过滤完成之后**，计算参与 join 的各个字段的总数据量，**数据量小的那个表，就是“小表”**，应该作为驱动表。

in和exsits优化

原则：**小表驱动大表**，即小的数据集驱动大的数据集

in：当B表的数据集小于A表的数据集时，in优于exists

```
select * from A where id in (select id from B)
```

```
#等价于:
    for(select id from B){
 select * from A where A.id = B.id
 }
```

exists：当A表的数据集小于B表的数据集时，exists优于in

将主查询A的数据，放到子查询B中做条件验证，根据验证结果（true或false）来决定主查询的数据是否保留

```
select * from A where exists (select 1 from B where B.id = A.id)
#等价于:
 for(select * from A){
 select * from B where B.id = A.id
 }

#A表与B表的ID字段应建立索引
```

1、EXISTS (subquery)只返回TRUE或FALSE,因此子查询中的SELECT * 也可以用SELECT 1替换,官方说法是实际执行时会忽略SELECT清单,因此没有区别

2、EXISTS子查询的实际执行过程可能经过了优化而不是我们理解上的逐条对比

3、EXISTS子查询往往也可以用JOIN来代替，何种最优需要具体问题具体分析

count(*)查询优化

```
-- 临时关闭mysql查询缓存，为了查看sql多次执行的真实时间
mysql> set global query_cache_size=0;
mysql> set global query_cache_type=0;

mysql> EXPLAIN select count(1) from employees;
mysql> EXPLAIN select count(id) from employees;
mysql> EXPLAIN select count(name) from employees;
mysql> EXPLAIN select count(*) from employees;
```

注意：以上4条sql只有根据某个字段count不会统计字段为null值的数据行

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	(Null)	index	(Null)	idx_name_age_position	140	(Null)	100140	100	Using index

四个sql的执行计划一样，说明这四个sql执行效率应该差不多

字段有索引：count(*)≈count(1)>count(字段)>count(主键 id)　//字段有索引，count(字段)统计走二级索引，二级索引存储数据比主键索引少，所以count(字段)>count(主键 id)

字段无索引：count(*)≈count(1)>count(主键 id)>count(字段)　//字段没有索引count(字段)统计走不了索引，count(主键 id)还可以走主键索引，所以count(主键 id)>count(字段)

count(1)跟count(字段)执行过程类似，不过count(1)不需要取出字段统计，就用常量1做统计，count(字段)还需要取出字段，所以理论上count(1)比count(字段)会快一点。

count(*) 是例外，mysql并不会把全部字段取出来，而是专门做了优化，不取值，按行累加，效率很高，所以不需要用count(列名)或count(常量)来替代 count(*)。

为什么对于count(id)，mysql最终选择辅助索引而不是主键聚集索引？因为二级索引相对主键索引存储数据更少，检索性能应该更高，mysql内部做了点优化(应该是在5.7版本才优化)。

常见优化方法

1、查询mysql自己维护的总行数

对于**myisam存储引擎**的表做不带where条件的count查询性能是很高的，因为myisam存储引擎的表的总行数会被mysql存储在磁盘上，查询不需要计算

```
9  EXPLAIN select count(*) from test_myisam;
10
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Select tables optimized away

对于**innodb存储引擎**的表mysql不会存储表的总记录行数(因为有MVCC机制，后面会讲)，查询count需要实时计算

2、show table status

如果只需要知道表总行数的估计值可以用如下sql查询，性能很高

```
11  show table status like 'employees';
12
```

信息 | 结果1 | 概况 | 状态

Name	Engine	Version	Row_format	Rows	Avg_row_length	D
employees	InnoDB	10	Dynamic	99806	57	5

3、将总数维护到Redis里

插入或删除表数据行的时候同时维护redis里的表总行数key的计数值(用incr或decr命令)，但是这种方式可能不准，很难保证表操作和redis操作的事务一致性

4、增加数据库计数表

插入或删除表数据行的时候同时维护计数表，让他们在同一个事务里操作

阿里巴巴Mysql规范解读

补充：MySQL数据类型选择

在MySQL中，选择正确的数据类型，对于性能至关重要。一般应该遵循下面两步：

（1）确定合适的大类型：数字、字符串、时间、二进制；

（2）确定具体的类型：有无符号、取值范围、变长定长等。

在MySQL数据类型设置方面，尽量用更小的数据类型，因为它们通常有更好的性能，花费更少的硬件资源。并且，尽量把字段定义为NOT NULL，避免使用NULL。

1、数值类型

类型	大小	范围（有符号）	范围（无符号）	用途
TINYINT	1 字节	(-128, 127)	(0, 255)	小整数值
SMALLINT	2 字节	(-32 768, 32 767)	(0, 65 535)	大整数值
MEDIUMINT	3 字节	(-8 388 608, 8 388 607)	(0, 16 777 215)	大整数值
INT或INTEGER	4 字节	(-2 147 483 648, 2 147 483 647)	(0, 4 294 967 295)	大整数值
BIGINT	8 字节	(-9 233 372 036 854 775 808, 9 223 372 036 854 775 807)	(0, 18 446 744 073 709 551 615)	极大整数值
FLOAT	4 字节	(-3.402 823 466 E+38, 1.175 494 351 E-38)，0，(1.175 494 351 E-38，3.402 823 466 351 E+38)	0, (1.175 494 351 E-38, 3.402 823 466 E+38)	单精度浮点数值
DOUBLE	8 字节	(1.797 693 134 862 315 7 E+308, 2.225 073 858 507 201 4 E-308), 0, (2.225 073 858 507 201 4	0, (2.225 073 858 507 201 4 E-308, 1.797 693 134 862 315 7 E+308)	双精度浮点数值

		E-308, 1.797 693 134 862 315 7 E+308)		
DECIMAL	对DECIMAL(M,D)，如果M>D，为M+2否则为D+2	依赖于M和D的值	依赖于M和D的值	小数值

优化建议

1. 如果整形数据没有负数，如ID号，建议指定为UNSIGNED无符号类型，容量可以扩大一倍。
2. 建议使用TINYINT代替ENUM、BITENUM、SET。
3. 避免使用整数的显示宽度(参看文档最后)，也就是说，不要用INT(10)类似的方法指定字段显示宽度，直接用INT。
4. DECIMAL最适合保存准确度要求高，而且用于计算的数据，比如价格。但是在使用DECIMAL类型的时候，注意长度设置。
5. 建议使用整形类型来运算和存储实数，方法是，实数乘以相应的倍数后再操作。
6. 整数通常是最佳的数据类型，因为它速度快，并且能使用AUTO_INCREMENT。

2、日期和时间

类型	大小(字节)	范围	格式	用途
DATE	3	1000-01-01 到 9999-12-31	YYYY-MM-DD	日期值
TIME	3	'-838:59:59' 到 '838:59:59'	HH:MM:SS	时间值或持续时间
YEAR	1	1901 到 2155	YYYY	年份值
DATETIME	8	1000-01-01 00:00:00 到 9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	4	1970-01-01 00:00:00 到 2038-01-19 03:14:07	YYYYMMDDhhmmss	混合日期和时间值，时间戳

优化建议

1. MySQL能存储的最小时间粒度为秒。
2. 建议用DATE数据类型来保存日期。MySQL中默认的日期格式是yyyy-mm-dd。
3. 用MySQL的内建类型DATE、TIME、DATETIME来存储时间，而不是使用字符串。
4. 当数据格式为TIMESTAMP和DATETIME时，可以用CURRENT_TIMESTAMP作为默认（MySQL5.6以后），MySQL会自动返回记录插入的确切时间。
5. TIMESTAMP是UTC时间戳，与时区相关。
6. DATETIME的存储格式是一个YYYYMMDD HH:MM:SS的整数，与时区无关，你存了什么，读出来就是什么。
7. 除非有特殊需求，一般的公司建议使用TIMESTAMP，它比DATETIME更节约空间，但是像阿里这样的公司一般会用DATETIME，因为不用考虑TIMESTAMP将来的时间上限问题。
8. 有时人们把Unix的时间戳保存为整数值，但是这通常没有任何好处，这种格式处理起来不太方便，我们并不推荐它。

3、字符串

类型	大小	用途
CHAR	0-255字节	定长字符串，char(n)当插入的字符串实际长度不足n时，插入空格进行补充保存。在进行检索时，尾部的空格会被去掉。
VARCHAR	0-65535 字节	变长字符串，varchar(n)中的n代表最大列长度，插入的字符串实际长度不足n时不会补充空格
TINYBLOB	0-255字节	不超过 255 个字符的二进制字符串
TINYTEXT	0-255字节	短文本字符串
BLOB	0-65 535字节	二进制形式的长文本数据
TEXT	0-65 535字节	长文本数据

MEDIUMBLOB	0-16 777 215字节	二进制形式的中等长度文本数据
MEDIUMTEXT	0-16 777 215字节	中等长度文本数据
LONGBLOB	0-4 294 967 295字节	二进制形式的极大文本数据
LONGTEXT	0-4 294 967 295字节	极大文本数据

优化建议

1. 字符串的长度相差较大用VARCHAR；字符串短，且所有值都接近一个长度用CHAR。
2. CHAR和VARCHAR适用于包括人名、邮政编码、电话号码和不超过255个字符长度的任意字母数字组合。那些要用来计算的数字不要用VARCHAR类型保存，因为可能会导致一些与计算相关的问题。换句话说，可能影响到计算的准确性和完整性。
3. 尽量少用BLOB和TEXT，如果实在要用可以考虑将BLOB和TEXT字段单独存一张表，用id关联。
4. BLOB系列存储二进制字符串，与字符集无关。TEXT系列存储非二进制字符串，与字符集相关。
5. BLOB和TEXT都不能有默认值。

PS：INT显示宽度

我们经常会使用命令来创建数据表，而且同时会指定一个长度，如下。但是，这里的长度并非是TINYINT类型存储的最大长度，而是显示的最大长度。

```
CREATE TABLE `user`(
 `id` TINYINT(2) UNSIGNED
);
```

这里表示user表的id字段的类型是TINYINT，可以存储的最大数值是255。所以，在存储数据时，如果存入值小于等于255，如200，虽然超过2位，但是没有超出TINYINT类型长度，所以可以正常保存；如果存入值大于255，如500，那么MySQL会自动保存为TINYINT类型的最大值255。

在查询数据时，不管查询结果为何值，都按实际输出。这里TINYINT(2)中2的作用就是，当需要在查询结果前填充0时，命令中加上ZEROFILL就可以实现，如：

```
`id` TINYINT(2) UNSIGNED ZEROFILL
```

这样，查询结果如果是5，那输出就是05。如果指定TINYINT(5)，那输出就是00005，其实实际存储的值还是5，而且存储的数据不会超过255，只是MySQL输出数据时在前面填充了0。

换句话说，在MySQL命令中，字段的类型长度TINYINT(2)、INT(11)不会影响数据的插入，只会在使用ZEROFILL时有用，让查询结果前填充0。

文档：05-VIP-Mysql索引优化实战二.note

```
http://note.youdao.com/noteshare?id=df15aba3aa76c225090d04d0dc776dd9&sub=B0D6000AA824407489097F4C87B3F36D
```