MySQL 开发及运维规范

知数堂发起

日期	版本	作者	说明
2016.7.26	1.0	知数堂	初始化

1 文档简介

文档是经验总结沉淀，方便传播及阅读。 随着时间的推移，文档里的一些知识也许就变成不合时宜，甚至是错的。因此，我们推崇一个理念：尽信书不如无书，要勇于挑战传统，坚持自己实践之后得出的认知。

本文档由知数堂发起，希望能借此促进行业的 MySQL 开发、使用规范，少走弯路，让大家都能在 MySQL 上做出好的应用。

1.1 联系我们

如果您觉得这份规范还不错，有进一步的想法， 或是有兴趣参与这个规范整理和补充，那就联系我们吧。

文档负责人：**叶金荣**（QQ/Weixin： 4700963），**吴炳锡**（QQ/Weixin： 82565387），也欢迎扫描下方二维码加入 QQ 群进交流。

1.2 适用范围

该文档可以用于各类项目中的 MySQL 使用场景，主要是表结构 DDL 设计建议，SQL 编写优化建议，及进行 SQL Review，日常 MySQL 运维管理等场景。

- 类型溢出

  以 MySQL5 版本，int 类型为例：

  #建表

  root@localhost(test2)14:46>create table test2 (a int(10) UNSIGNED);

  Query OK, 0 rows affected (0.12 sec)

  #插入数据

  root@localhost(test2)14:56>insert test2 values (10);

  Query OK, 1 row affected (0.00 sec)

  #模拟更新溢出

  root@localhost(test2)14:56>update test2 set a=a-11;

  Query OK, 1 row affected, 1 warning (0.00 sec)

  Rows matched: 1 Changed: 1 Warnings: 1

  #查看 warnings

  root@localhost(test2)14:57>show warnings;

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

  | Level | Code | Message |

```
+---------+------+-----------------------------------------+
| Warning | 1264 | Out of range value for column 'a' at row 1 |
+---------+------+-----------------------------------------+
1 row in set (0.00 sec)
#确定实际得到的值已经溢出
root@localhost(test2)14:57>select * from test2;
+-------------+
| a           |
+-------------+
| 4294967295 |
+-------------+
1 row in set (0.00 sec)
#清理数据
root@localhost(test2)14:59>delete from test2;
Query OK, 1 row affected (0.00 sec)
#模拟插入溢出
```

```
root@localhost(test2)14:59>insert test2 values (-1);
Query OK, 1 row affected, 1 warning (0.00 sec)
#查看 warnings
root@localhost(test2)14:59>show warnings;
+---------+------+-----------------------------------------+
| Level   | Code | Message                                 |
+---------+------+-----------------------------------------+
| Warning | 1264 | Out of range value for column 'a' at row 1 |
+---------+------+-----------------------------------------+
1 row in set (0.00 sec)
#确定实际得到的值已经溢出
root@localhost(test2)14:59>select * from test2;
+------+
| a    |
+------+
|    0 |
```

+------+

1 row in set (0.00 sec)

1、原因

int 占用 4 个字节，而 int 又分为无符号型和有符号性。对于无符号型的范围是 0 到 4294967295；有符号型的范围是-2147483648 到 2147483647。

举一反三，其他类型都可能有类似问题，均需要考量。

2、规避方法

可以把 sql_mode 的 STRICT_TRANS_TABLES 模式加上，也可以在应用程序端控制，比如：对表单项的值进行校验。

- 类型隐式转换

1、举例

类型溢出和隐式转换都可能导致 SQL 效率低下，应用响应变慢。

典型案例：

测试表：sid，有个字段是 varchar 类型，并且创建了索引，看看表 DDL：

```
CREATE TABLE `sid` (
  `id` mediumint(5) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(20) NOT NULL DEFAULT '',
```

```
  `vid` varchar(10) NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `i_vid` (`vid`)
) ENGINE=InnoDB AUTO_INCREMENT=262135 DEFAULT CHARSET=utf8;
```

来看看下面 2 个 SQL 的执行计划：

#SQL 1，条件值**带单引号**

```
mysql> desc select * from sid where vid = '12345'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sid
   partitions: NULL
         type: ref
possible_keys: i_vid
          key: i_vid
      key_len: 32
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

这个执行计划看起来没问题，能用到 i_vid 索引。

#SQL 2，条件值不带单引号

```
mysql> desc select * from sid where vid = 12345\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sid
   partitions: NULL
         type: ALL
possible_keys: i_vid
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 261862
     filtered: 10.00
        Extra: Using where
1 row in set, 3 warnings (0.00 sec)

mysql> show warnings;
+---------+------+-------------------------------------------------------------------------------------------------------------------------------------+
| Level   | Code | Message                                                                                                                             |
+---------+------+-------------------------------------------------------------------------------------------------------------------------------------+
| Warning | 1739 | Cannot use ref access on index 'i_vid' due to type or collation conversion on field 'vid'                                           |
| Warning | 1739 | Cannot use range access on index 'i_vid' due to type or collation conversion on field 'vid'                                         |
+---------+------+-------------------------------------------------------------------------------------------------------------------------------------+
```

可以看到执行计划非常差，完全用不到 i_vid 索引。需要进行全表扫描，预计需要扫描 26 万条记录，相比之下，SQL 1 预估只需要扫描 1 条记录，SQL 2 多了 26 万倍，真正执行起来的耗时相差也是很大的。

注：在 5.7 版本中，甚至还在执行计划中增加了告警提示，能看到这个 SQL 发生了类型转换，所以没办法使用 i_vid 索引进行等值及范围扫描。

| Warning | 1739 | Cannot use ref access on index 'i_vid' due to type or collation conversion on field 'vid'

| Warning | 1739 | Cannot use range access on index 'i_vid' due to type or collation conversion on field 'vid'

SQL 1 的执行耗时

```
mysql> select * from sid where vid = '12345';
+-------+---------+-------+
| id    | name    | vid   |
+-------+---------+-------+
| 12345 | aaaaaaa | 12345 |
+-------+---------+-------+
1 row in set (0.00 sec)
```

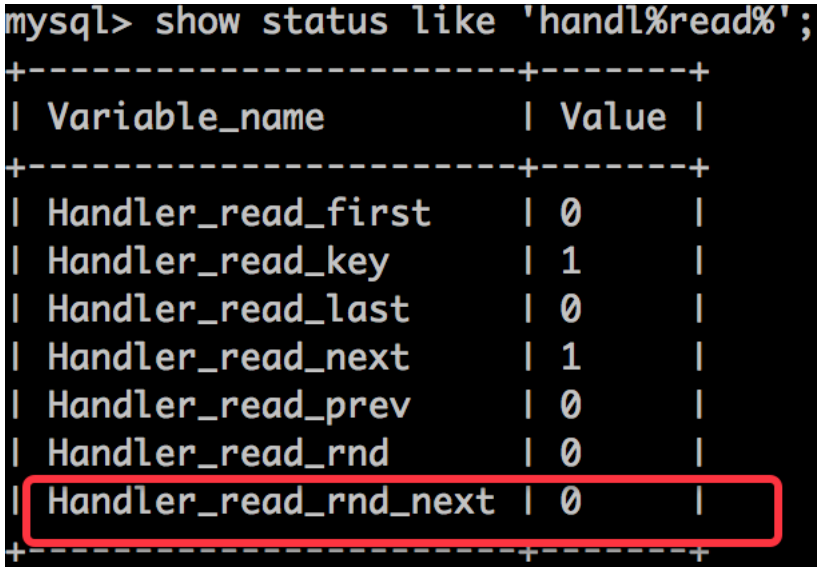
SQL 2 的执行耗时

```
mysql> select * from sid where vid = 12345;
+-------+---------+-------+
| id    | name    | vid   |
+-------+---------+-------+
| 12345 | aaaaaaa | 12345 |
+-------+---------+-------+
1 row in set (0.19 sec)
```

这还是在 PCIe SSD 卡设备环境中运行的结果，如果是在性能更差的服务器上，这个耗时相差会更严重。

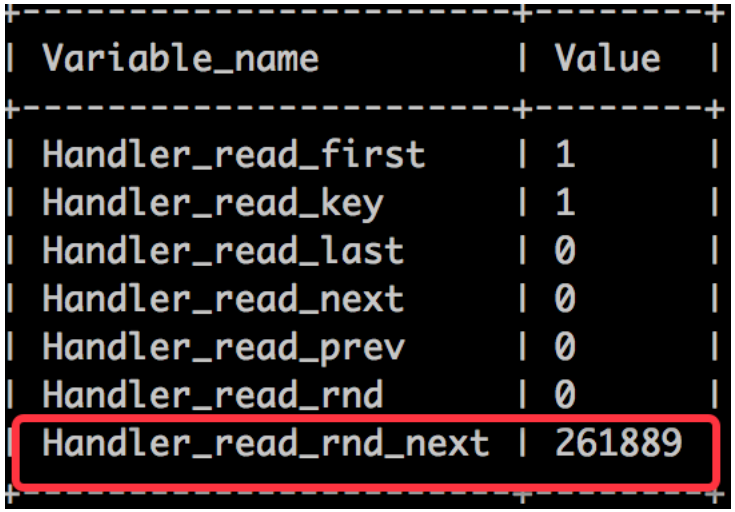
再来对比下 2 个 SQL 的代价：

SQL 1 的代价：



```
mysql> show status like 'handl%read%';
+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| Handler_read_first    | 0     |
| Handler_read_key      | 1     |
| Handler_read_last     | 0     |
| Handler_read_next     | 1     |
| Handler_read_prev     | 0     |
| Handler_read_rnd      | 0     |
| Handler_read_rnd_next | 0     |
+-----------------------+-------+
```

SQL 2 的代价：



Variable_name	Value
Handler_read_first	1
Handler_read_key	1
Handler_read_last	0
Handler_read_next	0
Handler_read_prev	0
Handler_read_rnd	0
Handler_read_rnd_next	261889

真的是相差了 26 万倍。

不过，如果 vid 列是整型，where 条件中带不带单引号都 不受影响，可以自行测试验证。

2、原因

vid 列是 varchar 类型，而条件值是整型，整型数值是可以转换成相应的 ASCII 码的。

```
mysql> desc select * from sid where vid = 12345\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: sid
   partitions: NULL
         type: ALL
possible_keys: i_vid
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 261862
     filtered: 10.00
        Extra: Using where
1 row in set, 3 warnings (0.00 sec)

mysql> show warnings;
+---------+------+-------------------------------------------------------------------------------------------------
-----------------------------------+
| Level   | Code | Message
                                   |
+---------+------+-------------------------------------------------------------------------------------------------
-----------------------------------+
| Warning | 1739 | Cannot use ref access on index 'i_vid' due to type or collation conversion on field 'vid'
                                   |
| Warning | 1739 | Cannot use range access on index 'i_vid' due to type or collation conversion on field 'vid'
```

反过来，如果是 int_col = ‘12345’ 这样的，后面的 ‘12345’ 虽然也会被快速强制转成 int，但这个效率就非常高了，不受影响。

3、规避方法

所有的 where 条件值，都带上引号。

3.2 索引规范

- 非唯一索引按照“i_表名_字段名”进行命名，例如：i_test_user
- 唯一索引按照“u_表名_字段名”进行命名，例如：u_test_uid
- 索引名称使用小写
- 索引中的字段数不超过 5 个
- 唯一索引不和主键重复
- 索引字段的顺序需要考虑字段值去重之后的个数（唯一性越大，基数越大），基数大的放在前面
- ORDER BY，GROUP BY，DISTINCT 的字段需要添加在索引的后面
- 单张表的索引数量控制在 5 个以内
- 若有多个字段都要用于单独查询而建索引时，需要经过 DBA 评估确认，但更建议从产品设计上进行重构
- 重要业务 SQL 上线前请使用 EXPLAIN 检查执行计划是否合理，避免 extra 列出现：Using Filesort，Using Temporary
- 对 VARCHAR 字段建立索引时，建议只创建不超过 15 个字符长度的前缀索引。举例：

  下面的表增加一列 url_crc32，然后对 url_crc32 建立索引，减少索引字段的长度，提高效率。

  ```
  CREATE TABLE all_url(

  ID INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,

  url VARCHAR(255) NOT NULL DEFAULT 0,

  url_crc32 BIGINT UNSIGNED NOT NULL DEFAULT 0,

  index i_url_crc32(url_crc32)
  ```

)engine = InnoDB;

- 合理创建联合索引（避免冗余），(a,b,c) 索引已经可以覆盖 (a)、(a,b)，就没必要创建后两个索引了
- 合理利用覆盖索引，也就是尽可能在 SELECT 中包含索引中的字段，减少回表读取产生的 I/O 读

3.3 分库分表规范

分库分表，通常指定表中的一个字段为分区 KEY，按 Range 或是 Hash 的方式进行拆分。

进行分库分表是为了让业务有更好、更快的扩展能力，以适应业务发展需要。同时也方便增删改字段、数据备份恢复等。如果业务稳定，也可以不用进行太多、太复杂的分表方案。服务器性能足够，架构、业务设计也合理的话，MySQL 处理上亿数据量的单表也是绰绰有余。

- 单表数据容量限制规则

无 CHAR/VARCHAR/TEXT/BLOB 的可以考虑千万到亿级左右，否则不建议超过千万级别。

- 单表空间物理大小限制规则

传统机械磁盘 10GB 左右， PCIe SSD 卡类不超过 100GB。

- 分表参考案例

例如有个业务表计划分成 100 个子表，那么可以分成 10 个实例各 10 个子表； 或者 100 个实例各 1 个子表。

备注：在做分库分表设计时，要考虑后期扩容怎么处理。是通过移动库的形式做扩容，或是通过移动表的形式扩容。是倍数扩容，或是基数扩容，都需要提前规划好。从而设计不同的拆分规则，才能更适用自己的业务。