## MYSQL 研发规范

表设计规范：

1. 所有数据库表，函数，视图等对象都[定义为小写,字符之间用下划线连接](#库名、表名、字段名必须使用小写字母，并采用下划线分割)
2. 库名、表名、字段名禁止使用MySQL[保留字](http://www.360doc.com/content/12/0203/11/3688062_183844013.shtml)，且名称长度不能超过32字节
3. 默认使用[INNODB存储引擎](#使用INNODB存储引擎)，使用其他存储引擎需要DBA评审
4. [表必须有主键](#表必须有主键),推荐使用 自增id列作为主键
5. 字符集默认使用UTF-8，特殊定制会导致数据迁移，jdbc配置等出现不兼容问题。
6. 执行DDL、SHARDING、单表恢复等运维成本，在小库中不建议采用分区表。
7. 禁止使用[临时表](#临时表)。CREATE TEMPORARY TABLE

列定义规范：

1. 为了节省数据库空间，请参照[字段类型](#数据类型范围)详细说明表选择最少空间占用类型
2. [UNSIGNED](http://www.cnblogs.com/blankqdb/archive/2012/11/03/blank_qdb.html)存储非负数值，建议在没有负数的业务场景内使用此种类型进行自增列主键定义
3. 自增列在数据库重启动和truncate时会根据表内的MAX(ID)[重新计数](#AUTO_INCREMENT存在一个bug).
4. 同财务有关的浮点数必须使用精确存储的DECIMAL数值类型
5. 原则上不建议数据库字段定义为TXT 类型，仅仅当字符数量可能超过 20000 个的时候，可以使用 TEXT/MEDIUMTEXT 类型来存放字符类数据。推荐所有使用 TEXT/MEDIUMTEXT类型的字段和原表进行分拆，与原表主键单独组成另外一个表进行存放；
6. 使用UTF8时，因其是变长的字符集，故固定和动态长度的字符串都应用VARCHAR
7. 只需要精确到**天**的字段全部使用 DATE 类型
8. 不允许使用BLOB等大对象类型。
9. 将大字段、访问频率低的字段拆分到单独的表中存储，分离冷热数据。　　有利于有效利用缓存，防止读入无用的冷数据，较少磁盘IO,同时保证热数据常驻内存提高缓存命中率。
10. 所有字段在业务上最好都能保证为非空，[建议用用0、特殊值或空串代替NULL值](#所有索引字段建议定义为NOTNULL)
11. MYSQL的新增和修改列的操作相当于重建表，表设计要一步到位，尽量避免大表的DDL操作。 （TIPS:可以预定义一些列留作将来业务扩展，如：当前只需要10个字段，考虑到未来发展，可以预留10个字段，表上总共创建20个字段）

索引规范：

1. 表数据超过1000行就需要创建索引。
2. 常用查询条件都需要创建索引,选择度高的列建议作为索引的第一个字段
3. 为了降低索引维护成本，禁止冗余索引，增大IO压力。（a,b,c）、（a,b），后者为冗余索引。可以利用前缀索引来达到加速目的，减轻维护负担。
4. 禁止创建重复索引。 如primary key a;uniq index a; 在mysql内会分别创建两个索引.
5. 索引数量最好控制在 5个以内

SQL语句书写规范：

1. 与ORACLE数据库的支持能力不同，MYSQL数据库内仅仅存放数据，业务逻辑需要上移到应用层处理，报表分析型需求使用大数据平台，可以缓存数据使用redis等缓存处理。
2. 从主备数据异步复制的数据安全性角度出发，禁止使用SYSDATE()函数获取当前时间,可以使用now()
3. 隐式转换会导致用不到相应索引,在查询条件书写时避免产生对查询列的隐式转换 .详见[隐式转换规则](#隐式转换规则)部分
4. 禁止使用[LAST\_INSERT\_ID()](#LAST_INSERT_ID)
5. 为了防止SQL注入，禁止使用[SLEEP()、BENCHMARK()](#SLEEPBENCHMARK)类型函数
6. 由于UUID分布的随机性和长度较长，非常不适合作为InnoDB存储引擎表的主键或唯一性的标识，非特殊原因或特殊场景不应使用UUID() 函数
7. WHERE子句中的数据扫描不超过表总数据量的30%
8. WHERE子句禁止出现NULL值计算COLUMN\_A = NULL 可以写成 COLUMN\_A IS NULL
9. Where 子句中 in 或者or 的值不能一次性输入超过500个
10. Where条件中 列的查询条件输入值类型要与列定义匹配。
11. 同一列的查询使用[IN和OR](#WHERE子句中同一字段的值OR可用IN替换) 有时执行计划会不同，推荐用in， 如果in没有选择合适索引改成OR条件试试看。
12. 使用LIKE时，%不要放在首字符位置，会导致无法使用相应索引

1. [WHERE子句的表字段上禁止使用表达式或是函数](#WHERE子句的表字段上禁止使用表达式或是函数)，会导致无法使用相应索引
2. 按需使用UNION ALL和UNION ,没有去重的需求情况下，[建议优先使用UNION ALL子句。](#按需使用UNIONALL和UNION)
3. UPDATE和DELETE操作尽量依据主键或唯一索引进行操纵，且事务之间处理的 数据没有交集。
4. 查询语句要减少或避免[二次排序](#查询语句要减少或避免二次排序)
5. DML语句中不使用不确定性函数 (如 sysdate() )和随机函数 （ 如 rand()）
6. [INSERT语句最好选用批提交模式](#INSERT语句批提交模式)
7. 尽量不使用存储过程、触发器、函数、UDF

存储过程（去除db内计算）、触发器（影响效率）、函数 （create function去除db内计算）、UDF (用户自定义函数 自己写一段代码(如c) 嵌入MYSQL)

1. MYSQL 里面没有hash joion链接方式，只有NESTLOOP, SQL 语句需要[规避不符合嵌套循环算法JOIN操作](#规避不符合嵌套循环算法JOIN操作)，最好使用主键进行表之间的关联。
2. SQL语句中IN包含的值不超过500
3. Where 条件中不能对条件列进行计算，如果有计算需求需要改写成在常量上计算 比如： where a+1>10 改成 where a>10-1
4. 禁止使用临时表 **create temporary table**
5. 禁止使用[order by rand（），](#禁止使用orderbyrand) 相当于对所有结果集进行排序，资源消耗大
6. 每个语句运行时 使用EXPLAIN + sql statement 确认是否有使用索引（ 坏的执行计划除了 key 为不正确索引外，还有 执行计划中出现了 sort ，tmp file等字样 ）
7. 拆分复杂SQL为多个小SQL
8. MYSQL 连接书写规范语法

DML操作规范：

1. Mysql数据库中事物不允许出现一次超过1000行以上未提交的操作
2. 单次数据操作修改量超过1000万 ，请服务中心报备给DBA监控
3. 慎用 **insert into table\_a select \* from table\_b where** 操作。在此例子中会锁定 table\_b.
4. 为了减少数据库的交互建议采用批量操作语句进行插入 ， INSERT INTO VALUES（），但是禁止使用 这两种形式的[批量insert 语句](#INSERT语句批提交模式) 。replace into t1(id,data1) values (1,1); （merge）insert ignore into t1(id,data1) values (1,1); 忽略冲突数据
5. 禁止使用如下两种 删除和修改语句：**Delete from table\_a where column\_a in (select column\_b from table\_b where …..);**

**Update table\_a set ….. where column\_a in (select column\_b from table\_b where …..);**

**补充 成 iner joion 样例**

DDL操作规范

**MYSQL 5.6 以后版本**MYSQL部分的DDL 操作会复制旧表，在磁盘上分配与当前表大小相同的一套空间进行表的DDL 操作，与此同时，数据库内不能有在该表上的任何操作才能保证操作成功。

1. 最大限度减少表上的DDL操作频度
2. DDL操作会需要锁表，可能导致应用查询阻塞，**所以需要在业务停机窗口进行**
3. DDL操作之前需要确认数据库内表上已经没有相应事务。
4. 对于超过500万以上的表进行DDL操作，需提前请DBA评估后台空间是否满足要求
5. [对同一个表的多次alter操作必须合并为一次操作](#对同一个表的多次alter操作必须合并为一次操作)

**注： MYSQL 5.6之前的版本，所有的ddl操作都会阻塞查询和复制表，所以必须在数据库内对应表上没有任何相关操作时进行。**

***附： MYSQL数据库开发指引 (内有规范内相关知识点的讲解)***

1.库名、表名、字段名必须使用小写字母，并采用下划线分割。

　　a）MySQL有配置参数lower\_case\_table\_names,不可动态更改，[linux](http://linux.chinaitlab.com/)系统默认为 0,即库表名以实际情况[存储](http://www.storworld.com/)，大小写敏感。如果是1,以小写[存储](http://www.storworld.com/)，大小写不敏感。如果是2,以实际情况存储，但以小写比较。

　　b）如果大小写混合使用，可能存在abc,Abc,ABC等多个表共存，容易导致混乱。

　　c）字段名显式区分大小写，但实际使用不区分，即不可以建立两个名字一样但大小写不一样的字段。

　　d）为了统一规范， 库名、表名、字段名使用小写字母。

　　2.库名、表名、字段名禁止超过32个字符。

　　库名、表名、字段名支持最多64个字符，但为了统一规范、易于辨识以及减少传输量，禁止超过32个字符。

　　3.使用INNODB存储引擎。

　　INNODB引擎是MySQL5.5版本以后的默认引擘，支持事务、行级锁，有更好的数据恢复能力、更好的并发性能，同时对多核、大内存、SSD等硬件支持更好，支持数据热备份等，因此INNODB相比MyISAM有明显优势。

　　4.库名、表名、字段名禁止使用MySQL保留字。

当库名、表名、字段名等属性含有保留字时，SQL语句必须用反引号引用属性名称，这将使得SQL语句书写、SHELL脚本中变量的转义等变得非常复杂。

http://www.360doc.com/content/12/0203/11/3688062\_183844013.shtml

　　5.慎用使用分区表。

　　分区表对分区键有严格要求；分区表在表变大后，执行DDL、SHARDING、单表恢复等都变得更加困难。因此禁止使用分区表，并建议业务端手动SHARDING.

　　6.建议使用UNSIGNED存储非负数值。

　　同样的字节数，非负存储的数值范围更大。如TINYINT有符号为 -128-127,无符号为0-255.

　　7.建议使用INT UNSIGNED存储IPV4.

　　UNSINGED INT存储IP地址占用4字节，CHAR（15）则占用15字节。另外，计算机处理整数类型比字符串类型快。使用INT UNSIGNED而不是CHAR（15）来存储IPV4地址，通过MySQL函数inet\_ntoa和inet\_aton来进行转化。IPv6地址目前没有转化函数，需要使用DECIMAL或两个BIGINT来存储。

　　例如：

　　SELECT INET\_ATON（'209.207.224.40'）； 3520061480

　　SELECT INET\_NTOA（3520061480）； 209.207.224.40

　　8.强烈建议使用TINYINT来代替ENUM类型。

　　ENUM类型在需要修改或增加枚举值时，需要在线DDL,成本较大；ENUM列值如果含有数字类型，可能会引起默认值混淆。

　　9.使用VARBINARY存储大小写敏感的变长字符串或二进制内容。

　　VARBINARY默认区分大小写，没有字符集概念，速度快。

　　10.INT类型固定占用4字节存储，例如INT（4）仅代表显示字符宽度为4位，不代表存储长度。

　　数值类型括号后面的数字只是表示宽度而跟存储范围没有关系，比如INT（3）默认显示3位，空格补齐，超出时正常显示，python、java客户端等不具备这个功能。

　　11.区分使用DATETIME和TIMESTAMP.存储年使用YEAR类型。存储日期使用DATE类型。 存储时间（精确到秒）建议使用TIMESTAMP类型。

　　DATETIME和TIMESTAMP都是精确到秒，优先选择TIMESTAMP,因为TIMESTAMP只有4个字节，而DATETIME8个字节。同时TIMESTAMP具有自动赋值以及自动更新的特性。注意：在5.5和之前的版本中，如果一个表中有多个timestamp列，那么最多只能有一列能具有自动更新功能。

　　如何使用TIMESTAMP的自动赋值属性？

　　a）自动初始化，并自动更新： column1 TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

　　b）只是自动初始化： column1 TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

　　c）自动更新，初始化的值为0: column1 TIMESTAMP DEFAULT 0 ON UPDATE CURRENT\_TIMESTAMP

　　d）初始化的值为0: column1 TIMESTAMP DEFAULT 0

　　12.所有索引字段建议定义为NOT NULL.

　　a）对表的每一行，每个为NULL的列都需要额外的空间来标识。

　　b）如果索引字段可以为NULL,索引统计信息可能不准确，影响执行计划。

c）建议用0、特殊值或空串代替NULL值。

**TIPS:mysql 里面 IS null 可能会用到索引，is not null 不会用到索引**

　　13.将大字段、访问频率低的字段拆分到单独的表中存储，分离冷热数据。

　　有利于有效利用缓存，防止读入无用的冷数据，较少磁盘IO,同时保证热数据常驻内存提高缓存命中率。

　　14.禁止在数据库中存储明文密码。

　　采用加密字符串存储密码，并保证密码不可解密，同时采用随机字符串加盐保证密码[安全](http://security.chinaitlab.com/)。防止数据库数据被公司内部人员或黑客获取后，采用字典攻击等方式暴力破解用户密码。

　　15.表必须有主键，推荐使用UNSIGNED自增列作为主键。

表没有主键，INNODB会默认设置隐藏的主键列；没有主键的表在定位数据行的时候非常困难，也会降低基于行复制的效率。

网络资料：

<http://17173ops.com/2014/09/14/mysql-faq-why-innodb-table-using-autoinc-int-as-pk.shtml>

　　16.禁止冗余索引。

　　索引是双刃剑，会增加维护负担，增大IO压力。（a,b,c）、（a,b），后者为冗余索引。可以利用前缀索引来达到加速目的，减轻维护负担。

　　17.禁止重复索引。

　　primary key a;uniq index a;重复索引增加维护负担、占用磁盘空间，同时没有任何益处。

　　18.不在低基数列上建立索引，例如"性别".

　　大部分场景下，低基数列上建立索引的精确查找，相对于不建立索引的全表扫描没有任何优势，而且增大了IO负担。

　　19.合理使用覆盖索引减少IO,避免排序。

　　覆盖索引能从索引中获取需要的所有字段，从而避免回表进行二次查找，节省IO.INNODB存储引擎中， secondary index（非主键索引，又称为辅助索引、二级索引）没有直接存储行地址，而是存储主键值。如果用户需要查询secondary index中所不包含的数据列，则需要先通过secondary index查找到主键值，然后再通过主键查询到其他数据列，因此需要查询两次。覆盖索引则可以在一个索引中获取所有需要的数据，因此效率较高。主键查询是天然的覆盖索引。例如SELECT email,uid FROM user\_email WHERE uid=xx,如果uid 不是主键，适当时候可以将索引添加为index（uid,email），以获得性能提升。

　　20.可以用IN代替OR.SQL语句中IN包含的值不应过多，应少于500个。

　　IN是范围查找，MySQL内部会对IN的列表值进行排序后查找，比OR效率更高。

　　21.表字符集使用UTF8,必要时可申请使用UTF8MB4字符集。

　　a）UTF8字符集存储汉字占用3个字节，存储英文字符占用一个字节。

　　b）UTF8统一而且通用，不会出现转码出现乱码风险。

　　c）如果遇到EMOJ等表情符号的存储需求，可申请使用UTF8MB4字符集。

　　22.用UNION ALL代替UNION.

　　UNION ALL不需要对结果集再进行排序。

　　23.禁止使用order by rand（）。

　　order by rand（）会为表增加一个伪列，然后用rand（）函数为每一行数据计算出rand（）值，然后基于该行排序， 这通常都会生成磁盘上的临时表，因此效率非常低。建议先使用rand（）函数获得随机的主键值，然后通过主键获取数据。

　　24.建议使用合理的分页方式以提高分页效率。

　　第一种分页写法：

　　select \*

　　from t

　　where thread\_id = 771025

　　and deleted = 0

　　order by gmt\_create asc limit 0, 15;

　　select \* from t

　　where thread\_id = 771025

　　and deleted = 0

　　order by gmt\_create asc limit 0, 15;

　　原理：一次性根据过滤条件取出所有字段进行排序返回。

数据访问开销=索引IO+索引全部记录结果对应的表数据IO

　　缺点：该种写法越翻到后面执行效率越差，时间越长，尤其表数据量很大的时候。

　　适用场景：当中间结果集很小（10000行以下）或者查询条件复杂（指涉及多个不同查询字段或者多表连接）时适用。

　　第二种分页写法：

　　select t.\* from （

　　select id from t

　　where thread\_id = 771025 and deleted = 0 order by gmt\_create asc limit 0, 15） a, t

　　where a.id = t.id;

　　前提：假设t表主键是id列，且有覆盖索引secondary key:（thread\_id, deleted, gmt\_create）

　　原理：先根据过滤条件利用覆盖索引取出主键id进行排序，再进行join操作取出其他字段。

　　数据访问开销=索引IO+索引分页后结果（例子中是15行）对应的表数据IO.

　　优点：每次翻页消耗的资源和时间都基本相同，就像翻第一页一样。

　　适用场景：当查询和排序字段（即where子句和order by子句涉及的字段）有对应覆盖索引时，且中间结果集很大的情况时适用。

　　25.SELECT只获取必要的字段，禁止使用SELECT \*.

　　减少网络带宽消耗；

　　能有效利用覆盖索引；

　　表结构变更对程序基本无影响。

　　26.SQL中避免出现now（）、rand（）、sysdate（）、current\_user（）等不确定结果的函数。

　　语句级复制场景下，引起主从数据不一致；不确定值的函数，产生的SQL语句无法利用QUERY CACHE.

　　27.采用合适的分库分表策略。例如千库十表、十库百表等。

　　采用合适的分库分表策略，有利于业务发展后期快速对数据库进行水平拆分，同时分库可以有效利用MySQL的多线程复制特性。

　　28.减少与数据库交互次数，尽量采用批量SQL语句。

　　使用下面的语句来减少和db的交互次数：

　　INSERT INTO VALUES（）,(),()…();

　　29.拆分复杂SQL为多个小SQL。

　　30.对同一个表的多次alter操作必须合并为一次操作。

　　mysql对表的修改绝大部分操作都需要锁表并重建表，而锁表则会对线上业务造成影响。为减少这种影响，必须把对表的多次alter操作合并为一次操作。例如，要给表t增加一个字段b,同时给已有的字段aa建立索引， 通常的做法分为两步：

　　alter table t add column b varchar（10）；

　　然后增加索引：

　　alter table t add index idx\_aa（aa）；

　　正确的做法是：

　　alter table t add column b varchar（10），add index idx\_aa（aa）；

　　31.避免使用存储过程、触发器、视图、自定义函数等。

　　这些高级特性有性能问题，以及未知BUG较多。业务逻辑放到数据库会造成数据库的DDL、SCALE OUT、 SHARDING等变得更加困难。

　　32.禁止有super权限的应用程序账号存在。

[安全](http://security.chinaitlab.com/)第一。super权限会导致read only失效，导致较多诡异问题而且很难追踪。

　　　34.不要在MySQL数据库中存放业务逻辑。

数据库是有状态的服务，变更复杂而且速度慢，如果把业务逻辑放到数据库中，将会限制业务的快速发展。建议把业务逻辑提前，放到前端或中间逻辑层，而把数据库作为存储层，实现逻辑与存储的分离。

## 数据类型

* **数据类型分类**

1. 数值类型 -- TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT、 DECIMAL
2. 时间类型 -- DATE、DATETIME、TIMESTAMP
3. 字符串类型 -- CHAR、VARCHAR、TEXT
4. 特有类型 -- ENUM、SET

**特别说明：**

* VARCHAR(N) 或 CHAR(N)中N的含义

MySQL中两类字符串VARCHAR(N) 或CHAR(N)定义时候填写的长度N，不是字节数的意思 ，而是**字符数的含义**。

* 浮点数

同财务有关的浮点数必须使用精确存储的DECIMAL数值类型。为减少浮点类型FLOAT和DOUBLE的精度不够而可能造成数值计算存在偏差，故非财务相关的浮点数字段也采用DECIMAL数值类型存储，例如：weight DECIMAL(10,4) NOT NULL DEFAULT 0。

* **数据类型范围**

|  |  |  |  |
| --- | --- | --- | --- |
| **数据类型** | **类型分类** | **表达的范围** | **存储需求** |
| TINYINT[(M)] [UNSIGNED] [ZEROFILL] | 数值整型 | -128到127 或 0到255 | 1个字节 |
| SMALLINT[(M)] [UNSIGNED] [ZEROFILL] | -32768到32767 或 0到65535 | 2个字节 |
| MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL] | -8388608到8388607 或 0到16777215 | 3个字节 |
| INT[(M)] [UNSIGNED] [ZEROFILL] | -2147483648到2147483647 或 0到4294967295 | 4个字节 |
| BIGINT[(M)] [UNSIGNED] [ZEROFILL] | -9223372036854775808到9223372036854775807 或 0到18446744073709551615 | 8个字节 |
| DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL] | 数值混合型 | 整数最大位数(M)为65，小数位数最大(D)为30 | 变长 |
| DATE | 日期类型 | YYYY-MM-DD 精确到天 | 3个字节 |
| DATETIME | YYYY-MM-DD HH:MM:SS（1001年到9999年的范围） | 8个字节 |
| TIMESTAMP | YYYY-MM-DD HH:MM:SS（1970年到2037年的范围） | 4个字节 |
| ENUM | 枚举类型 | 1到65535个元数据 | 1个字节或2个字节 |
| SET | 集合类型 | 1到64个元数据 | 1,2,3,4或8个字节 |
| CHAR(M) | 字符串类型 | 0<M<=255（建议CHAR(1)外，超过此长度的统一用VARCHAR(M)替代） | M\*N个字节，其中N大小由字符集编码，以及是否为中文还是字母数字等有关系 |
| VARCHAR(M) | 0<M<65532/N 个字符 |
| TEXT | 64K个字节 | 能存储的字符个数，与字符集编码、存储的字符串值有关系 |

* **详细说明**

1. 类似于状态，且有限类别的字段，推荐使用可比较明显表示出实际意义的字符串，而不应该使用INT之类的数字来代替，故推荐使用枚举类型ENUM和集合类型SET；
2. 使用UTF8时，因其是变长的字符集，故固定和动态长度的字符串都应用VARCHAR；
3. 仅仅当字符数量可能超过 20000 个的时候，可以使用 TEXT/MEDIUMTEXT 类型来存放字符类数据。推荐所有使用 TEXT/MEDIUMTEXT类型的字段和原表进行分拆，与原表主键单独组成另外一个表进行存放；
4. 需要精确时间（年月日时分秒）的字段可以使用DATETIME 或TIMESTAMP，但请注意各自能表达的范围，以及是否需要用到TIMESTAMP的特性，尽量使用TIMESTAMP类型替代DATETIME以减少数据存储空间的占用；需要毫秒、微妙时，使用TIMESTAMP (3)或TIMESTAMP (6)，以及DATETIME(3)和DATETIME(6)（注：社区版5.6开始DATATIME也支持CURRENT\_TIMESTAMP.，且支持任意数量与组合）；
5. 所有只需要精确到天的字段全部使用 DATE 类型，而不应该使用 TIMESTAMP或者DATETIME 类型；
6. 自增序列特性的字段只能使用 INT 或者 BIGINT，强烈推荐明确标识为无符号类型 (关键词：UNSIGNED)，除非确实会出现负数，仅当该字段的数值会超过42亿，才使用 BIGINT 类型，并且自增字段必须作为主键或主键的一部分；
7. INT(N)中N的含义

MySQL中各类整形也可包含长度，该长度的含义和Oracle的NUMBER类型的长度含

义不同，N值的大小完全不影响数值的存储范围，只影响有ZEROFILL时显示的长度。

1. 空字符串的含义

MySQL中字符串存在三种容易混淆的值：空字符串''、空值NULL和NULL字符串

'NULL'，需要注意判断。和Oracle的最大区别为Oracle不区分空字符串''和空值NULL。

## 字符集

字符集UTF-8编码基本上表达我们日常所使用到的汉字、字母或字符，也能够兼容更多语种；其次，顺丰速运的研发项目所用编程语言为JAVA，而JAVA开发工具的语言也是默认为UTF-8；另外，顺丰速运多数项目的业务数据不需要区分数据的大小写，为此我们最终的建议：字符集统一为UTF-8，字符集校对规则为utf8\_general\_ci。

若研发项目中存在区分存储数据的字母大小的需求，则字符集统一为UTF-8，字符集校对规则为：utf8\_bin。

采用UTF8编码时，对于中文汉字及符号，实际存储时占**三个字节**，而数值、字母和符号（注：英文输入法下），则只占**一个字节**。例如：

* CREATE TABEL gl\_user(username VARCHAR(40));则username最多能存储40个字符；
* username存储’U-Cloud’ 则占用存储空间为：7个字节（注释：变长会额外增加1个字节或2个字节，此处实际占用存储空间为：8个字节）；

**特别说明：**

随着移动互联网业务发展，越来越多的移动终端支持输入一些特殊符号，字符编码空间超过3个字节，则可使用4个字节编码的UTF8mb4。不区分字母大小的示例，如下：

* CREATE TABEL gl\_user(username VARCHAR(40) CHARACTER SET utf8mb4 COLLATE utf8mb4\_general\_ci);则username最多能存储40个字符，字符自身占用最大存储空间160个字节；

## 函数

* + - * **字符串连接函数**

MySQL数据库中字符串连接方法，需使用CONCAT() 或CONCAT\_ WS()函数，语法如下：

CONCAT(string1,string2,...)

CONCAT\_ WS(separator,string1,string2,..)

MySQL中的CONCAT和Oracle中的CONCAT不同，可以接受任意多个参数，可以较为简单的替代'||'符号的作用。MySQL也可以通过sql\_mode配置双管道符作为字符串连接运算符，但不推荐这样做。

* + - * **字符串长度统计函数**

LENGTH(string) #返回string字符串所占的字节数

CHAR\_LENGTH(string) #返回string字符串中的字符个数

统计字符个数，就不区分是汉字还是字母或数字，也跟字符集没有关系，若统计的是字节数，则由字符是汉字、字母或数字类型，以及字符集共同决定。

**特别说明：**

我们所有的MySQL数据库都将会采用统一的UTF8编码，所以一个汉字占3个字节，中文输入法（或称全角输入模式）下的字母或数字占3个字节；英文输入法（或半角输入法模式）下一个字母或数字占1个字节。

* **字符串判断函数**
* IF(exp1,exp2,exp3)：若是exp1 为真，返回exp2；若是exp1为假，返回exp3；
* IFNULL(exp1,exp2)：若是exp1 IS NOT NULL，返回exp1，否则返回exp2；
* NULLIF(exp1,exp2)：若是exp1=exp2，返回NULL，否则返回exp1；
* **字符串替换函数**
* LTRIM(exp1)：去掉exp1中字符串开头的空格；
* RTRIM(exp1)：去掉exp1中字符串结尾的空格；
* TRIM(exp1)：去掉exp1中的开头和结尾的空格；
* TRIM(exp2,exp1)：去除掉exp1中存在的字符串exp2；
* **字符串查找函数**
* SUBSTRING\_INDEX(exp1,delim,count)

exp1为字符串，delim为分割符号，count表示第几个分割符号，例如：SUBSTRING\_INDEX(‘hotpu.cn’,‘.’,1)，返回：hotpu。

* SUBSTRING(exp1,pos,len)

exp1为字符串，pos为位置，len为长度，例如：SUBSTRING(‘hotpu.cn’,1,5)，返回：hotpu。

* LOCAL(substr,str)

查找substr在str中的第一个位置，例如：LCOAL(hotpu.cn’,‘.’)，返回：6。

* **字母大小写转换函数**
* UPPER(exp1)：把字符串exp1转换成大写；
* LOWER(exp1)：把字符串exp1转换成小写；
* **数学函数**
* ABS(value)：返回value的绝对值，例：ABS(-101)，返回：101
* FLOOR(value)：去掉value的小数，例：FLOOR(2013.8)，返回：2013
* MOD(N,M)：返回N除以M的余数，例：MOD(2013,10)，返回：3
* ROUND(value)：返回value的四舍五入值，例：FLOOR(2013.8)，返回：2014
* ROUND(value,num)：保留num位小数value的四舍五入值，例：FLOOR(2013.867,1)，返回：2013.9
* RAND()：返回随机数值
* **日期操作函数**
* 获取当前时间函数：NOW()、CURDATE()、CURTIME()

NOW()函数精确到秒， 格式：YYYY-MM-DD HH:MM:SS

CURDATE()函数精确到天， 格式：YYYY-MM-DD

CURTIME()函数提供小时、分钟、秒， 格式：HH:MM:SS

**特别说明：**

从主备数据异步复制的数据安全性角度出发，禁止使用SYSDATE()函数获取当前时间。

* 日期数值的加减函数

DATE\_ADD(*date*,INTERVAL *expr* *type*)

DATE\_ SUB(*date*,INTERVAL *expr* *type*)

常用的几种type类型：YEAR、MONTH、DAY、HOUR、MINUTE，其中expr可以为正数或负数，我们在开过程中，一般使用DATE\_ADD()函数，若要做日期减去一个数字的方式，就使用负数。

MySQL中不能像Oracle那样直接对时间类型进行加减运算，直接使用加减运算符不会得到符合预期的结果，需要使用函数进行运算。

DATEDIFF(expr1,expr2)，是返回 开始日期expr1与 结束日期expr2之间，相差的天数 ，返回值为正数或负数。

* 返回日期某部分信息的函数

YEAR(expr1) 返回日期expr1部分的年份；

MONTH(expr1) 返回日期expr1部分的月份；

DAY(expr1)返回expr1部分的天数；

WEEKDAY(expr1)返回expr1对应的星期数字，0表示星期一，1表示星期二，其他依次类推；

* **类型转换函数**
* 日期类型格式转换

字符串转换成日期方式，DATE\_FORMAT()或STR\_TO\_DATE()，两个函数的格式如下：

DATE\_FORMAT(expr1,format)

STR\_TO\_DATE(expr1, format)

**特别说明：**

* STR\_TO\_DATE()是为兼容Oracle数据库对应函数。

常用的日期格式YYYY-MM-DD HH:MM:SS 对应的FORMAT为：%Y-%m-%d %H:%i:%S。

* 通用类型转换函数

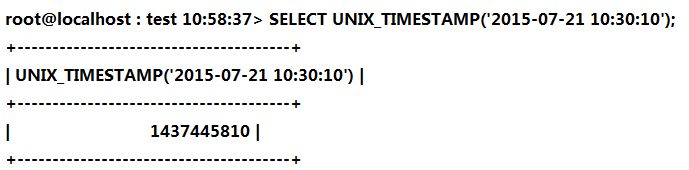
CAST(*expr* AS *type*) 则是把expr数字或字符串 转换为type类型；

CONVERT(*expr*,*type*) 则是把expr数字或字符串 转换为type类型；

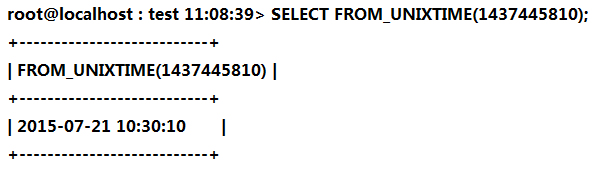
CONVERT(expr USING transcoding\_name) 则是转换字符串或字段的字符集编码；

* 日期整型转换函数

UNIX\_TIMESTAMP(date\_string) 则是将字符串格式表达的日期转换成INT无符号类型的整型数值，例如：



FROM\_UNIXTIME(int\_value) 则是将用整型数值表达的日期转换成字符串格式的日期，例如：



* IP地址转换

可以使用INET\_ATON()、INET\_NTOA()、INET6\_ATON()、INET6\_NTOA()实现IP地址和整型值之间的转换。

* 隐式转换规则

在不同类型的列之间发生比较或运算时，MySQL遵循如下规则：

* NULL和NULL比较不转换；
* 字符串和不同的字符串比较，则使用字符串比较；
* 整型和不同的整型比较，则使用整型比较；
* 十六进制值和整型以外的类型比较时十六进制值视为二进制字符串；
* 时间类型和字符串常量比较时，字符串转换为时间类型进行比较；
* DECIMAL类型的比较方式取决于另一个值的类型，如果是整型，则使用DECIMAL比较；如果是浮点型，则使用浮点类型比较。其它情况，使用浮点比较；
* 特殊函数
* SYSDATE()

原意为获取调用函数时刻的时间，通常在标准化的MySQL配置中，会将其修改为NOW()的同义词，为SQL语句开始执行的时间。从而规避SYSDATE带来的复制问题与sysdate-is-now参数带来的行为改变导致的潜在的配置不一致风险两个角度考虑，故建议禁止使用函数SYSDATE()。

* LAST\_INSERT\_ID()

获取最近成功插入带自增长的表的数据行的自增长ID值，对于需要查询刚刚INSERT的记录的ID的场景，例如外键约束的需要等等，可以考虑使用此函数。多行INSERT时，返回的是第一行获取到的自增长ID值，这种情况不推荐使用此方法获取ID。由于使用连接池时，不同的事务可能获取到不同的数据库连接，对该函数的查询应当保证在同一事务内进行。

* SLEEP()、BENCHMARK()

应用程序中要禁止使用此类函数，及防SQL注入方式对待。

mysql存在注入，并且注入的sleep语句如果传入一个足够大的参数，比如：sleep(9999999999).

如果数据库用的是myisam引擎，且注入点是某个会锁表的语句（insert,replace,update,delete），那么整个数据表的访问都会被阻塞。

使用该表的所有应用的读库请求都会被阻塞。

如果数据库使用的是主从分离的架构， 那么Master和Slave的同步会被sleep语句阻塞，导致从库无法从主库正常同步数据。一些依赖于主从同步的应用也会无法正常工作。

就算仅仅是读操作，经过有限次的请求，也会很快的达到数据库的max\_connections限制，而导致数据库拒绝服务。

* UUID()

生产全局唯一ID的函数，MySQL使用通用的算法，算法版本为1，该函数不能确保绝对不发生冲突，但是冲突的概率极低。由于UUID分布的随机性，非常不适合作为InnoDB存储引擎表的主键或唯一性的标识，非特殊原因或特殊场景不应使用。

* GET\_LOCK()、RELEASE\_LOCK()、IS\_FREE\_LOCK()、IS\_USED\_LOCK()

用户锁函数，对于定时调度存储过程，该系列函数可以用于确保只有一个存储过程正在运行类型的特殊功能，或者控制应用程序单线程运行类型的特殊功能。

## 查询语句编写指导

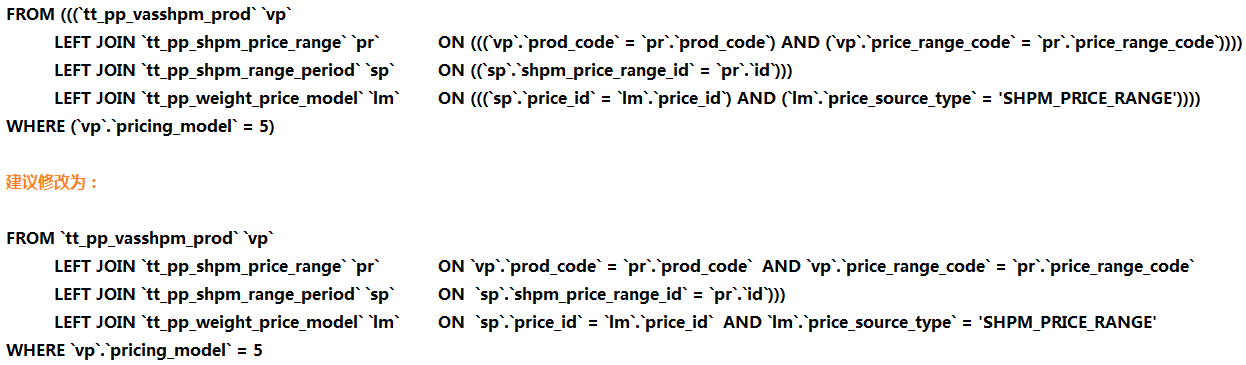
* 去掉不必要的括号

((a AND b) AND c OR (((a AND b) AND (c AND d))))

修改成

(a AND b AND c) OR (a AND b AND c AND d)

示例：



* 去掉重叠常量

示例：

(a<b AND b=c) AND a=100

修改成 b>100 AND b=c AND a=100

* 因常量重叠而去除部分常量条件

示例：

(B>=100 AND B=100) OR (B=200 AND 100=100) OR (B=300 AND 100=200)

修改成 B=100 OR B=200

* 去掉无意义的筛选条件

示例：

1=1，2>1，1<2等

修改成 这些无意义的筛选条件可以直接从WHERE子句中去掉。

* 开发过程中不使用拼字符串的方式来完成WHERE子句
* 多使用等值操作，少使用非等值操作

WHERE条件中的非等值条件（IN、BETWEEN、<、<=、>、>=）会导致后面的条件使用不了索引，因为不能同时用到两个范围条件。

* 常数表优先，字典表或小表其次，大表最后

常数表指：空表或只有1行的表。与在一个PRIMARY KEY或UNIQUE索引的WHERE子句一起使用的表。例如：

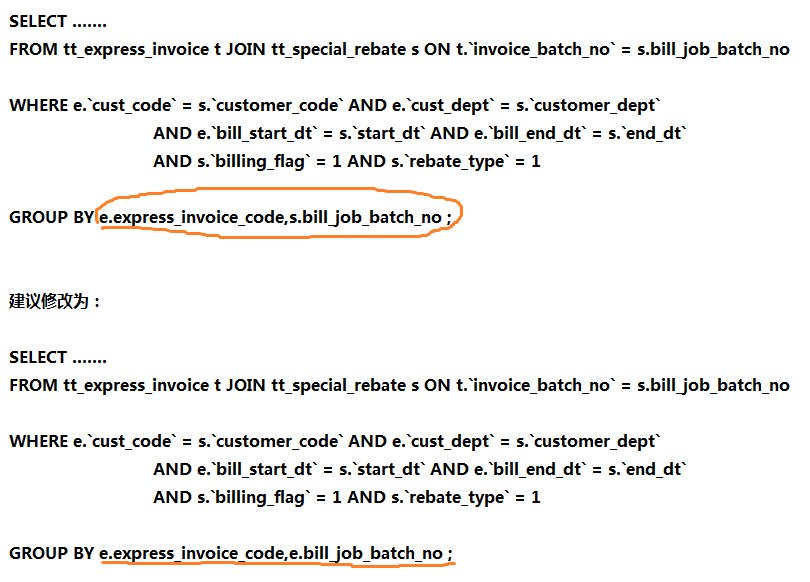
SELECT \* FROM t WHERE primary\_key=1;

SELECT \* FROM t1,t2 WHERE t1.primary\_key=1 AND t2.primary\_key=t1.id;

字典表指：小数据量的行，一般都是某一些业务或应用的配置数据。例如：自定义的自增字段表，而不使用MySQL的AUTO\_INCREMENT；或快递行业的产品种类配置表，往往此类表的数据量不会超过30条。

* 减少或避免临时表

如果有一个ORDER BY子句和不同的GROUP BY子句，或如果ORDER BY或GROUP BY包含联接队列中的第一个表之外的其它表的列，则创建一个临时表。故ORDER子句或GROUP子句涉及多张表的字段，统一含义的字段尽可能选择同一张表的。例如：



* WHERE子句中的数据扫描不超过表总数据量的30%

比如：WHERE primary\_key <> 1或者primary\_key not in(…)，这样扫描表的数据往往会超过30%。

WHERE status=1，其中1值非常少，主要是0值，比如一个表的记录删除用了一个状态位，而删除的记录又比较少。

* WHERE子句中同一个表的不同的字段组合建议小于等于5组，否则建议业务逻辑拆分或分表。
* WHERE子句禁止出现NULL值计算

WHERE子句中不允许出现IS NULL或IS NOT NULL的计算方式，更不允许出现错误的计算方式：Column1=NULL 和 (column1>NULL 或Column1<NULL)。故应该在字段定义时设置约束为NOT NULL。

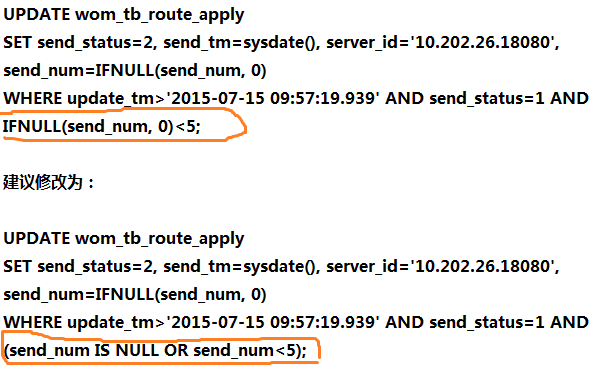
* 使用LIKE时，%不要放在首字符位置

WHERE子句中出现%放在首字符位置，会导致字段无法使用上索引，故建议禁止出现。

如果%必须放在首字符位置，执行频率极低的话，则建议直接使用MySQL存储引擎自身的全文检索功能；若执行频率较高或高的话，则建议使用索引存储引擎Sphinx或Lucene实现。

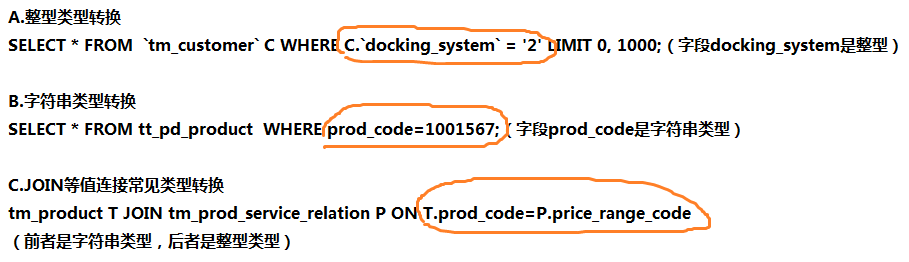
* WHERE子句的表字段上禁止使用表达式或是函数

当WHERE子句的表字段上使用函数后，将会导致该字段无法使用上索引，一般都是建议在字段的值域上加函数、计算或转换的表达式。例如：



* WHERE子句字段类型发生隐式转换
* WHERE子句的表字段类型为整型，向其传值一个字符串格式的数值在高并发情况下会偶尔诱发类型隐式转换；
* WHERE子句的表字段类型为字符串，向其传值一个整型的数值，会诱发类型隐式转换；
* 表与表关联操作的等值连接条件，常容易出现2张表的关联字段类型不同或者数值类型不同值域范围，而诱发类型隐式转换。

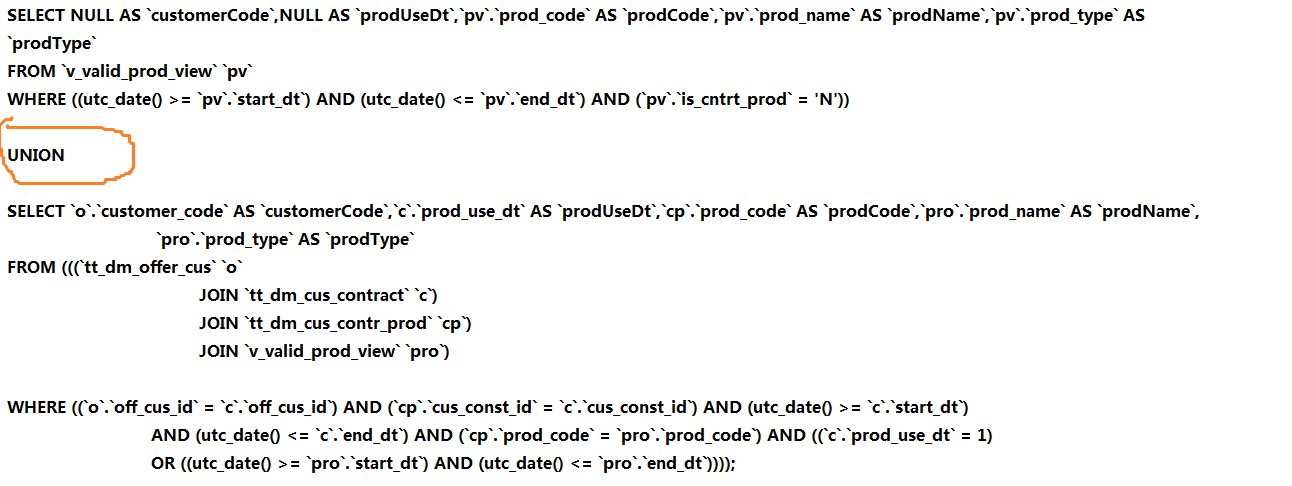
示例：



* 按需使用UNION ALL和UNION

UNION ALL不会连接的2个SQL语句结果集进行去重，从而减少了排序操作过程，故执行效率远高于UNION，消耗资源也更少，故没有去重的需求情况下，建议优先使用UNION ALL子句。

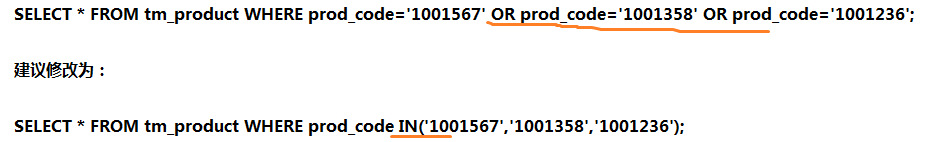
示例：



* WHERE子句中同一字段的值OR可用IN替换

MySQL数据库中WHERE子句的表字段IN条件，最终实现依然是转换成OR操作，为方便大家程序阅读和放送SQL语句包大小综合因素，推荐大家可用IN取代同一字段的OR连接。

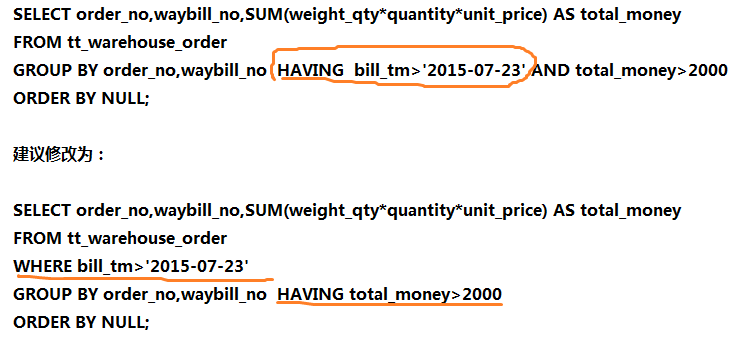
示例：



* 用WHERE子句替换HAVING子句

两种子句的区别：先执行WHERE子句，后执行HAVING子句；HAVING子句一般同GROUP子句配合使用；WHERE子句是对元数据的过滤，HAVING子句是对数据结果集的过滤；HAVING子句往往依据某一列、多列或表达式计算出来的列值进行过滤，或新列值别名进行筛选。

示例：



* UPDATE和DELETE操作尽量依据主键或唯一索引进行操纵

设计DML类型的UPDATE和DELETE操作时，尽可能地让应用程序获得主键，然后依据用户的操作转换成走主键条件去更新数据库表的数据，性能效率更高，锁记录或锁范围的时间更短，占用资源也更少。

* SELECT子句部分不要出现前端或后续处理不需要的字段，尤其不要出现TEXT/BLOB等大字段。

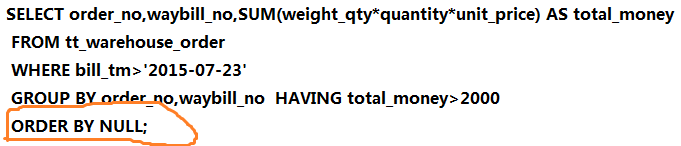
* 查询语句要减少或避免二次排序

1. 使用上索引完成排序

查询语句的排序字段上有相关的索引可用，查询语句涉及多张表的情况下，尽可能用同一张表的字段排序。

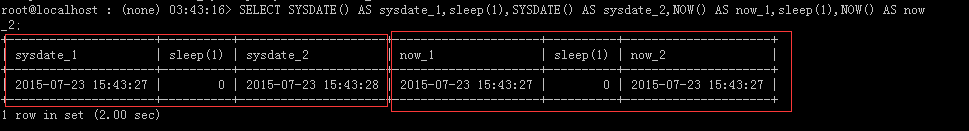
2）GROUP子句涉及的排序操作

GROUP BY子句操作隐含地带有一个排序的操作，若只需要分组计算功能，不需要排序，则可以建议增加ORDER BY NULL的显示模式，规避掉排序操作。例如：



* DML语句中不使用不确定性函数和随机函数

不确定性函数：通一条执行语句中，传入同一个数值，调用2次及以上，则执行结果集中获得的值不同。例如：



DML语句中不要使用常见的不确定性函数SYSDATE()等，不要使用随机函数RAND()等。

* INSERT语句批提交模式

合理地使用INSERT批提交数据写入模式，有利于减少每条SQL语句都需要申请初始化表和关闭表的操作，从而极大提高数据写入的效率。但一味地批量提交过多的记录，可能引发自增序列全局锁和间隙锁等。

INSERT INTO tablename(col1,clo2,.....) VALUES(),(),()„„，其中VALUES子句部分的括号个数不超过500，及注意数据包最大值是否小于max\_allowed\_packet。

* 尽量不使用存储过程、触发器、函数、UDF

1. UDF

要使用MySQL数据库自定义函数功能，会造成数据库服务及数据库服务器安全隐患，以及需要编写对应的C/C++代码程序，故建议禁止使用，生产环境此权限也是设置为关闭状态。

1. 触发器

使用触发器取代后台应用程序启用事务的方式DML数据，首先是容易造成数据逻辑维护关系不清晰，尤其是人员流失和时间长久之后；其次，在高并发系统中，使用触发器维护数据，容易造成锁资源争用更严重和死锁出现。

1. 存储过程

集中式技术架构往往会考虑使用存储过程实现部分或大量业务逻辑、数据计算的工作。对于当下业务和数据处理的爆发性增长，建议不要把业务逻辑和数据计算的工作实现存储过程实现，这样不利于扩展，及加速数据库资源成为瓶颈。业务逻辑和合理的数据计算放到应用程序端，可以做到自由伸缩扩展，而数据库的扩展性远低于此。

1. 函数

为了实现某些特殊业务处理和减少SQL复杂度，确实可以使用自己编写的函数处理，但建议不要把函数当存储过程使用。

**特别说明：**

对于存储过程、函数的创建及权限等信息，可以咨询顺丰DBA团队及向其获取相关资料。

* 规避不符合嵌套循环算法JOIN操作

MySQL数据库的Server和InnoDB存储引擎未提供经典复杂的Hash JOIN算法，只有Nested Loop JOIN算法，故设计数据库表结构的时候，就要考虑到此问题和SQL语句编写相结合。对于实现无法规避的情况，尽可能保证其有索引可走。

* 减少使用视图

项目中尽量不使用视图。视图已无太多实际意义，尤其是高并发系统中，反而会增加SQL语句的复杂度，降低执行效率。

* SQL语句中IN包含的值不超过500。
* UPDATE和DELETE语句不使用LIMIT

DML类型的UPDATE和DELETE修改语句禁止使用LIMIT子句。若修改类型的语句使用LIMIT子句，容易导致出现死锁和锁资源争用，还有导致主备数据不一致的风险，做二进制日志增量恢复的时候更复杂。此外，在ROW或MIXED模式下，会明细那增加二进制日志量，消耗更多数据库服务器的物理IO。

确实需要分批删除数据的方式，推荐使用SELECT查询语句拿到被删除数据的主键值，然后使用UPDATE/DELETE .... WHERE ...IN(pk1,pk2...)的语句方式实现。

* JDBC使用PREPARE STATEMNET

应用程序使用JDBC提供的预处理相关函数，可以做到加快数据写入速度，减少资源消耗和预防SQL注入，故鼓励研发工程师使用。

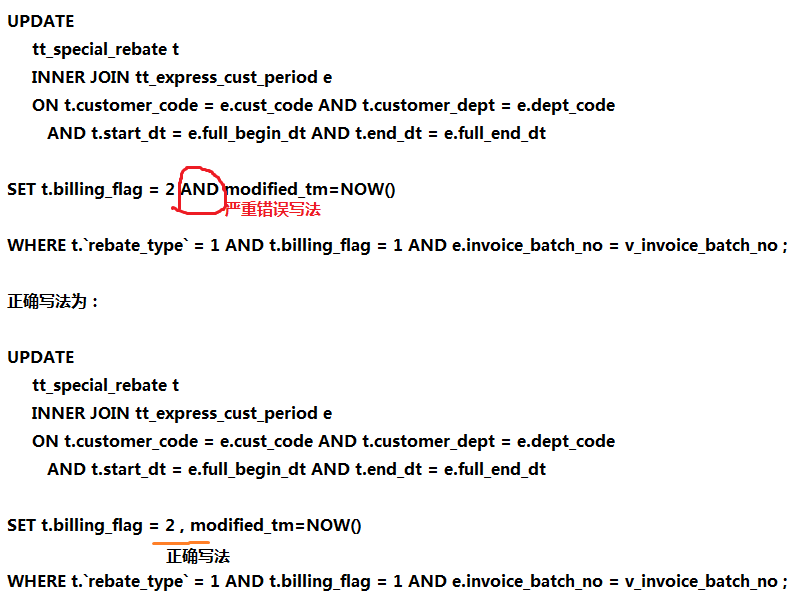
在存储过程、函数、事件中，除为了实现动态SQL执行的功能，其他情况下建议不要使用，无实际意义。

* 数据库表计数COUNT(\*)操作

正常用户自行统计推荐使用COUNT(\*)实现，一般不建议使用计数表的方式增加程序复杂度和研发工作量。若涉及到实时性要求极高的计数器业务场景，推荐使用行业内更加高效代价低的Redis解决方案，定时同步到数据库表中持久化。

* 禁止在UPDATE操作的SET子句中出现AND连接符号

若把SET子句的连接符号 “,” 错误写成 “AND”，将会导致寓意完全错误，故禁止在SET子句中出现 AND连接符号字样。举例说明：



正确示例：SET t.billing\_flag =2 , modified\_tm=NOW();

错误示例：SET t.billing\_flag = 2 AND modified\_tm=NOW();

此时“ut.billing\_flag = 2 AND modified\_tm=NOW()”将这个与运算的结果集作为值赋给ut.billing\_flag，且无任何警告信息提示。

* SELECT \* 的使用规范
* 基本原则

在不必要查询中使用“\*”列出所有字段，且存在GROUP BY或ORDER BY的时候，禁止使用SELECT \* 一次取出所有的字段。对于表连接的JOIN 语句，禁止使用SELECT \* 来进行查询。含有text 字段的表，当不需要取出TEXT 字段的时候，也禁止使用SELECT \* 进行查询。

* 详细说明

1> 进行GROUP BY或ORDER BY的时候不允许使用SELECT \* 是为了确保MySQL能够使用最新的优化排序算法；

2> JOIN 语句不允许使用SELECT \* 是为了防止仅仅只需要索引即可完成的查询需要回表取数；

3> 存有TEXT 字段表,在不需要取出TEXT字段的时候，不允许使用SELECT \* ,因为TEXT 字段是存放在和普通记录不一样的物理位置，会造成大量的IO操作；

4> SELECT \* 会增加CPU、IO开销和占用更多的网络带宽，也影响性能；

5> 避免因增删字段而没有修改相关SQL及相关程序代码导致程序BUG，而禁用SELECT \*；

* **子查询、EXISTS子句、IN子句、关联子查询**

严格意义的子查询指在一个语句中内含的SELECT查询语句。例如select \* from (select id,data from t1) t2;中select id,data from t1是严格意义的子查询。但是通常，我们将包含子查询的查询语句称为子查询。大部分子查询语句应当转化为其它类型语句执行，效率会更高，可读性也更好。

* **非关联子查询**
* select data1,(select data1 from t1 where ....) from t2 where ....;

此类型的子查询语句要求子查询最多只返回一个数据。为执行效率考虑，应当先执行子查询，获取子查询返回值，然后根据返回值生成SQL语句：“select data1,'子查询返回值' from t2 where ....;”，再执行生成的SQL语句。

对于select data1,function((select data1 from t1 where ....)) from t2 where ....;也可以类似处理。

* select data1 from (select data1 from t1 where ...) t2 where ...;

此类型子查询语句应当转化为普通查询语句“select data1 from t1 where ... and ...”。此类型的子查询语句是子查询语句改造的重点。

示例：

表中数据如下：

root@localhost : test 06:19:08> select \* from t1;

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

| id | data1 |

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

| 1 | 0.6298021847659743 |

| 2 | 0.09627149169917357 |

| 3 | 0.5919509349315901 |

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

3 rows in set (0.00 sec)

查询语句及结果如下：

root@localhost : test 06:20:35> select data1 from (select data1 from t1 where data1>0.5) t2 where data1>0.6;

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

| data1 |

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

| 0.6298021847659743 |

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

1 row in set (0.00 sec)

原始执行计划：

root@localhost : test 06:20:38> explain extended select data1 from (select data1 from t1 where data1>0.5) t2 where data1>0.6;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | filtered | Extra |

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

| 1 | PRIMARY | <derived2> | ALL | NULL | NULL | NULL | NULL | 3 | 100.00 | Using where |

| 2 | DERIVED | t1 | ALL | NULL | NULL | NULL | NULL | 3 | 100.00 | Using where |

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

2 rows in set, 1 warning (0.00 sec)

Note (Code 1003): /\* select#1 \*/ select `t2`.`data1` AS `data1` from (/\* select#2 \*/ select `test`.`t1`.`data1` AS `data1` from `test`.`t1` where (`test`.`t1`.`data1` > 0.5)) `t2` where (`t2`.`data1` > 0.6)

修改后的语句及执行结果：

root@localhost : test 06:21:21> select data1 from t1 where data1>0.5 and data1>0.6;

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

| data1 |

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

| 0.6298021847659743 |

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

1 row in set (0.00 sec)

修改后执行计划：

root@localhost : test 06:22:46> explain extended select data1 from t1 where data1>0.5 and data1>0.6;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | filtered | Extra |

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

| 1 | SIMPLE | t1 | ALL | NULL | NULL | NULL | NULL | 3 | 100.00 | Using where |

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

1 row in set, 1 warning (0.00 sec)

Note (Code 1003): /\* select#1 \*/ select `test`.`t1`.`data1` AS `data1` from `test`.`t1` where ((`test`.`t1`.`data1` > 0.5) and (`test`.`t1`.`data1` > 0.6))

root@localhost : test 06:23:00>

修改后的执行结果完全相同，但执行计划优于修改前的执行计划。

* select data1 from (select data1 from t1 where ... group by ...) t2 where ...;

此类型子查询语句应当转化为普通having查询语句“select data1 from t1 where ... group by ... having ...”。

* select data1 from t2 where data1=(select data1 from t1 where ... ) ...;

此类型的子查询语句要求子查询最多只返回一个数据。为执行效率考虑，应当先执行子查询，获取子查询返回值，然后根据返回值生成SQL语句：“select data1 from t2 where data1='子查询返回值' ...;”，再执行生成的SQL语句。

* select data1 from t1 where exist (select \* from t2 ....) ...

此类型的子查询语句应当先执行加上limit 1的子查询，获取子查询返回值，然后根据是否有返回数据判断是否要执行“select data1 from t1 ...”。

* select data1 from t1 where id in (select id from t2 where t2...) and t1...

此类型的子查询语句应当改写成连接语句：“select t1.data1 from t1 join t2 on (t1.id=t2.id and t2...) where t1...”。

* select data1 from t1 where id in (select id from t1 where ... limit N,N);

此类型的子查询语句在子查询部分可以走覆盖索引时可以取得速度上的优势，无需改写。如果不能覆盖索引，则应修改成“select data1 from t1 where ... limit N,N;”。

* **关联子查询**
* select data1,(select data1 from t1 where t1.id=t2.id ....) from t2 where t2.data...;

此类型的子查询语句要求子查询匹配外查询的数据，且最多返回一个数据，如果不匹配返回空值。此类查询应当转化为左外连接：“select t2.data1,t1.data1 from t2 left join t1 on (t2.id=t1.id ...) where t2.data...;”。如果从业务的角度能够确认子查询匹配外查询的数据有且只有一个，则可以使用普通join。

* select data1 from t2 where data1=(select data1 from t1 where t1.id=t2.id... ) ...;

此类型的子查询语句要求子查询匹配外查询的数据，且最多返回一个数据，如果不匹配返回空值。此类型的子查询语句应当改写成连接语句“select t2.data1 from t2 join t1 on (t2.id=t1.id ...and t2.data1=t1.data1) ...;”。

* select data1 from t1 where exist (select \* from t2 where t2.id=t1.id ....) ...

此类型的子查询语句不需要改写。

* select data1 from t1 where id2 in (select id2 from t2 where t2.id=t1.id and t2...) and t1...

此类型的子查询语句应当改写成连接语句：“select t1.data1 from t1 join t2 on (t1.id=t2.id and t1.id2=t2.id2 and t2...) where t1...”。

### 表连接写法建议

* **左连接**

左连接又称左外连接，表示在输出结果时，即使左表没有匹配到其它表的记录，依旧输出左表的记录，对应其它表的字段为NULL。

语法示例：

select \* from t1 left join t2 on (t1.id>t2.id ...) where ....

select \* from t1 left outer join t2 on (t1.id>t2.id ...) where ....

示例：

t1、t2表数据如下：

root@localhost : test 09:37:30> select \* from t1;

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

| id | data1 |

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

| 1 | 0.6298021847659743 |

| 2 | 0.09627149169917357 |

| 3 | 0.5919509349315901 |

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

3 rows in set (0.00 sec)

root@localhost : test 09:37:33> select \* from t2;

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

| id | data1 |

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

| 1 | 0.7138233685063415 |

| 2 | 0.44927900978296903 |

| 3 | 0.10492497412946539 |

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

3 rows in set (0.00 sec)

左连接SQL语句及结果如下：

root@localhost : test 09:37:34> select \* from t1 left join t2 on (t1.id=t2.id and t1.data1>t2.data1);

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

| id | data1 | id | data1 |

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

| 3 | 0.5919509349315901 | 3 | 0.10492497412946539 |

| 1 | 0.6298021847659743 | NULL | NULL |

| 2 | 0.09627149169917357 | NULL | NULL |

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

3 rows in set (0.00 sec)

对比后续内连接的输出结果可以发现，无法在t2表中匹配到记录的t1表数据（id为1、2的两条数据），也输出在结果集中。

* 开发规范：

1> select \* from t1 left join t2 on (t1.id=t2.id ...) where t1... ;

此类型的左连接语句过滤条件只在t1上，如果业务上确实需要t1中无法匹配的数据，则无需改写；否则应改写为普通连接语句；

2> select \* from t1 left join t2 on (t1.id=t2.id ...) where t2.data=X and ...;

此类型的左连接语句过滤条件包括对其它表字段的过滤，而且是一个具体的值，过滤后结果集不再可能出现其它表字段值为NULL的记录，因此该语句为不正确的左连接语句。语句应当改写为普通连接语句“select \* from t1 join t2 on (t1.id=t2.id ...) where t2.data=X and ...;”；

3> select \* from t1 left join t2 on (t1.id=t2.id ...) where t2.id is null and ...;

此类型的左连接语句要求获取其它表连接条件字段为NULL的记录，无需改写。

* 右连接

右连接又称右外连接，表示在输出结果时，即使右表没有匹配到其它表的记录，依旧输出右表的记录，对应其它表的字段为NULL。

语法示例：

select \* from t1 right join t2 on (t1.id>t2.id ...) where ....

select \* from t1 right outer join t2 on (t1.id>t2.id ...) where ....

示例：

使用的数据和左连接演示中使用的数据相同。

右连接SQL语句及结果如下：

root@localhost : test 10:38:46> select \* from t1 right join t2 on (t1.id=t2.id and t1.data1>t2.data1);

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

| id | data1 | id | data1 |

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

| 3 | 0.5919509349315901 | 3 | 0.10492497412946539 |

| NULL | NULL | 1 | 0.7138233685063415 |

| NULL | NULL | 2 | 0.44927900978296903 |

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

3 rows in set (0.00 sec)

对比后续内连接的输出结果可以发现，无法在t1表中匹配到记录的t2表数据（id为1、2的两条数据），也输出在结果集中。

* 开发规范：

通常不建议使用右连接，建议右连接全部改写为使用左连接。

1> select \* from t1 right join t2 on (t1.id=t2.id ...) where t2... ;

此类型的右连接语句过滤条件只在t2上，如果业务上确实需要t2中无法匹配的数据，则无需改写；否则应改写为普通连接语句；

2> select \* from t1 right join t2 on (t1.id=t2.id ...) where t1.data=X and ...;

此类型的右连接语句过滤条件包括对其它表字段的过滤，而且是一个具体的值，过滤后结果集不再可能出现其它表字段值为NULL的记录，因此该语句为不正确的右连接语句。语句应当改写为普通连接语句“select \* from t1 join t2 on (t1.id=t2.id ...) where t1.data=X and ...;”；

3> select \* from t1 right join t2 on (t1.id=t2.id ...) where t1.id is null and ...;

此类型的右连接语句要求获取其它表连接条件字段为NULL的记录，无需改写。

* 内连接

内连接是最普通的连接方式，会输出所有满足条件的记录。

语法示例：

select \* from t1 join t2 on (t1.id>t2.id ...) where ...

select \* from t1 inner join t2 on (t1.id>t2.id ...) where ...

select \* from t1 , t2 where t1.id>t2.id and ...

示例：

使用的数据和左连接演示中使用的数据相同。

内连接SQL语句及结果如下：

root@localhost : test 10:48:50> select \* from t1 join t2 on (t1.id=t2.id and t1.data1>t2.data1);

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

| id | data1 | id | data1 |

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

| 3 | 0.5919509349315901 | 3 | 0.10492497412946539 |

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

1 row in set (0.00 sec)

* 开发规范：

1> select \* from t1 join t2 on (t1.id=t2.id ...) where ...

此类型的内连接语句无需改写；

2> select \* from t1 , t2 where t1.id=t2.id and ...

此类型的内连接语句无需改写。

* **自然连接**

自然连接的含义为使用两个表中列名相同的列进行等值连接，且合并列名相同的列。

语法示例：

select \* from t1 natural join t2 where ...

select \* from t1 natural left outer join t2 where ...

示例：

使用的数据和左连接演示中使用的数据相同。

自然连接SQL语句及结果如下：

root@localhost : test 11:07:53> select \* from t1 natural join t2;

Empty set (0.00 sec)

root@localhost : test 11:08:00> alter table t2 change `data1` `data2` varchar(30) DEFAULT NULL;

Query OK, 0 rows affected (0.00 sec)

Records: 0 Duplicates: 0 Warnings: 0

root@localhost : test 11:08:27> select \* from t1 natural join t2;

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

| id | data1 | data2 |

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

| 1 | 0.6298021847659743 | 0.7138233685063415 |

| 2 | 0.09627149169917357 | 0.44927900978296903 |

| 3 | 0.5919509349315901 | 0.10492497412946539 |

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

3 rows in set (0.00 sec)

root@localhost : test 11:08:29> alter table t2 change `data2` `data1` varchar(30) DEFAULT NULL;

Query OK, 0 rows affected (0.00 sec)

Records: 0 Duplicates: 0 Warnings: 0

* **全外连接**

MySQL不支持全外连接。必须要实现此功能时，需要使用此方式：“select \* from t1 left join t2 on (t1.id=t2.id ...) where ... union select \* from t1 right join t2 on (t1.id=t2.id ...) where ...”，但是非常不推荐使用这样的方法，且这个方法最终结果等同于隐含地带上了DISTINCT，可能不符合业务需要。

* **强制顺序连接**

MySQL支持STRAIGHT\_JOIN关键字，指定MySQL使用编写的顺序连接表。

语法示例：

select \* from t2 straight\_join t1 on (t1.id>t2.id ...) where ...

select \* from t2 straight\_join t1 where ...

示例：

使用的数据和左连接演示中使用的数据相同。

强制顺序连接SQL语句及结果如下：

root@localhost : test 02:03:38> select \* from t2 straight\_join t1 where t1.id>1;

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

| id | data1 | id | data1 |

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

| 1 | 0.7138233685063415 | 2 | 0.09627149169917357 |

| 2 | 0.44927900978296903 | 2 | 0.09627149169917357 |

| 3 | 0.10492497412946539 | 2 | 0.09627149169917357 |

| 1 | 0.7138233685063415 | 3 | 0.5919509349315901 |

| 2 | 0.44927900978296903 | 3 | 0.5919509349315901 |

| 3 | 0.10492497412946539 | 3 | 0.5919509349315901 |

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

6 rows in set (0.00 sec)

root@localhost : test 02:03:48> explain extended select \* from t2 straight\_join t1 where t1.id>1;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | filtered | Extra |

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

| 1 | SIMPLE | t2 | ALL | NULL | NULL | NULL | NULL | 3 | 100.00 | NULL |

| 1 | SIMPLE | t1 | range | id | id | 8 | NULL | 2 | 100.00 | Using where; Using join buffer (Block Nested Loop) |

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

2 rows in set, 1 warning (0.00 sec)

Note (Code 1003): /\* select#1 \*/ select `test`.`t2`.`id` AS `id`,`test`.`t2`.`data1` AS `data1`,`test`.`t1`.`id` AS `id`,`test`.`t1`.`data1` AS `data1` from `test`.`t2` straight\_join `test`.`t1` where (`test`.`t1`.`id` > 1)

root@localhost : test 02:04:02> explain extended select \* from t2 join t1 where t1.id>1;

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

| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | filtered | Extra |

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

| 1 | SIMPLE | t1 | range | id | id | 8 | NULL | 2 | 100.00 | Using where |

| 1 | SIMPLE | t2 | ALL | NULL | NULL | NULL | NULL | 3 | 100.00 | Using join buffer (Block Nested Loop) |

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

2 rows in set, 1 warning (0.00 sec)

Note (Code 1003): /\* select#1 \*/ select `test`.`t2`.`id` AS `id`,`test`.`t2`.`data1` AS `data1`,`test`.`t1`.`id` AS `id`,`test`.`t1`.`data1` AS `data1` from `test`.`t2` join `test`.`t1` where (`test`.`t1`.`id` > 1)

### WHERE子句与函数

WHERE子句中经常存在需要使用函数的情形，由于在数据列上使用函数将导致大量计算，以及将导致无法使用索引，应当尽量避免在列上使用函数，转为在常量上运算，或其它方式实现。

* 开发规范：

1> select \* from t1 where data1+10>100;

该查询语句对列进行数学运算，可以化简运算到常量列进行。可以修改为查询语句：“select \* from t1 where data1>100-10;”

2> select \* from t1 where date\_add(data1,interval 1 day)>now();

该语句属于对列进行时间类型的函数运算。可以修改为查询语句：“select \* from t1 where data1>date\_sub(now(),interval 1 day);”

3> select \* from t1 where from\_unixtime(data1)='2015-07-23 14:45:23';

该语句属于对列进行时间类型的函数运算。可以修改为查询语句：“select \* from t1 where data1=unix\_timestamp('2015-07-23 14:45:23');”

4> select \* from t1 where year(data1)=2015;

该语句属于对列进行时间类型的函数运算。可以修改为查询语句：“select \* from t1 where data1 between '2015-01-01 00:00:00' and '2015-12-31 23:59:59';”

* **NULL值计算**
* select \* from t1 where ifnull(data1,0)=0;

如果字段定义是not null的，可以修改为查询语句“select \* from t1 where data1=0;”。如果字段定义允许空，并且应用混合使用null与0作业务角度的空值，建议推进将0作为业务角度的空值，并修改字段定义为not null，修改为查询语句“select \* from t1 where data1=0;”。如果无法推进修改，建议修改为查询语句“select \* from t1 where (data1 is null or data1=0);”。

* select \* from t1 where data1=null;

该SQL语句是错误的语句，原因为MySQL中null值与null值是不相等的，需要使用is null或者null安全等于操作符“<=>”。需要修改语句为“select \* from t1 where data1 is null;”或“select \* from t1 where data1<=>null;”。

* **特殊INSERT**
* INSERT ... ON DUPLICATE KEY UPDATE ...

语法示例：

insert into t1 set id=1,data1=1 on duplicate key update data1=data1+1 ;

insert into t1(id,data1) values (1,1) on duplicate key update data1=-1 ;

* 开发规范：

不推荐在大并发或高TPS的场景使用此语句，该语句执行、加锁过程比较复杂，容易导致死锁，带来错误处理的复杂度。另外，由于在存在逻辑问题的场景下，使用类似语句可能掩盖错误，因此要慎重使用该语句。

考虑到操作的原子性以及方便的角度，在压力很小，重要性不高的环境，可以合理使用这一语句降低开发复杂度。

* REPLACE

replace into t1(id,data1) values (1,1);

由于在存在逻辑问题的场景下，使用类似语句可能掩盖错误，因此禁止使用该语句。

* INSERT IGNORE

insert ignore into t1(id,data1) values (1,1);

由于在存在逻辑问题的场景下，使用类似语句可能掩盖错误，因此禁止使用该语句。

* INSERT DELAYED

INSERT DELAYED只适用于MyISAM、MEMORY、ARCHIVE、BLACKHOLE引擎；且在MySQL5.6中，已经过时，使用时会给出警告，未来将不支持该语法。因此应当禁止使用此语法。

### 临时表

* 临时表存储

CREATE TEMPORARY TABLE

创建和删除TEMPORARY表的过程本质上都是DDL操作，与普通DML操作相比，存在相对比较大的风险，需要慎用。

使用数据库连接池的场景，由于同一个数据库连接会被不同的线程复用，为避免出现问题，建议禁止使用TEMPORARY表。

* 使用临时表存储中间结果

这里的临时表是指业务意义上的临时表，而非MySQL角度的临时表，通常在一些批量计算、任务调度的场景使用。建议禁止使用这一方式处理中间结果的存储问题，原因为：

1> 通常带来大量的数据移动，以及操作大量数据的大事务。

2> INSERT SELECT语句将锁住select的表，潜在带来锁竞争的问题。

3> AUTO\_INCREMENT存在一个bug，MySQL服务器重启后会根据表中当前最大值重新计算；因此，如果删除最新的数据，加上重启MySQL数据库，将无法保证AUTO\_INCREMENT正向增长。使用临时表存储中间数据的场景涉及临时数据的删除，且可能涉及最新数据的删除，如果存在对临时表产生的自增长ID值的依赖，会比较容易被此bug影响。