



TiDB 开发指南（企业版）

产品版本：v5.4 | 文档版本：20220331



法律声明

PingCAP 提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过 PingCAP 网站或 PingCAP 提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为 PingCAP 的保密信息，您应当严格遵守保密义务；未经 PingCAP 事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经 PingCAP 事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。PingCAP 保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在 PingCAP 授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过 PingCAP 网站或 PingCAP 授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用 PingCAP 产品及服务的参考性指引，PingCAP 以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。PingCAP 在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但 PingCAP 在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证，本文档所引用的性能数据和程序示例仅用于说明目的，实际的性能结果可能因特定配置和操作条件而异。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，PingCAP 不承担任何法律责任。在任何情况下，PingCAP 均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使 PingCAP 已被告知该等损失的可能性）。
5. 本文档中及 PingCAP 网站上所有内容，包括但不限于作品、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计等，均由 PingCAP 和/或其关联公司依法拥有其知识产权，包括但不限于商标专利权、著作权、商业秘密等。非经 PingCAP 和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表 PingCAP 网站、产品程序或包括本文档在内的内容。此外，未经 PingCAP 事先书面同意，任何人不得以任何目的使用 PingCAP 商标（包括但不限于单独为或以组合形式包含“PingCAP”等 PingCAP 和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别 PingCAP 和/或其关联公司）。

6. PingCAP 可能拥有涵盖本文档中描述的主题的专利或者专利申请，未经 PingCAP 事先书面许可，本文档并不授予您关于这些专利或专利申请的任何许可。

7. 本文档中如有关于 PingCAP 未来方向或意图的声明，仅表示目标或者目的，如有更改或撤销，恕不另行通知。

8. 如若发现本文档存在任何错误，请与 PingCAP 取得直接联系。PingCAP 可能会以它认为合适的任何方式使用或分发您提供的任何信息，而无需对您承担任何义务。

目 录

前言	10
目的	10
适用范围	10
高亮示意	10
注意事项	10
对象命名规范	11
原则	11
数据库命名规范	11
表命名规范字段	11
命名规范索引命	12
名规范	12
数据库对象设计	12
表的设计字	12
段的设计字	13
段默认值索	15
引设计权限	16
设计	17
数据模型设计	18
完整性	18
性能扩	19
展性	19
SQL 开发规范	19
建表删表规范	20

select * 使用规范	20
大事务处理	20
Region 热点	21
字段上使用函数规范	22
数据删除规范	23
分页查询规范	23
使用 order by 语法保障结果集输出顺序	24
使用 full group by 语法保障结果集稳定	25
其他规范	25
事务限制	25
隔离级别	25
SI 可以克服幻读SI	26
不能克服写偏斜不	26
支持 savepoint大事	27
务限制	28
隐式类型转	28
换结果集不	31
稳定group by	31
order by	34
由于 group_concat() 中没有使用 order by 导致结果集不稳定	37
索引注意事项	39
TiDB 中的索引复合	39
索引的设计自增列	40
注意事项	41

原理	41
最佳实践	41
人为赋值的后果及挽救措施	42
TiDB 中的各种超时	42
GC 超时	42
事务超时	43
SQL 超时	43
连接器和 API	43
使用 MySQL 连接器连接 TiDB	45
使用 MySQL C API 连接 TiDB	45
使用 MySQL 第三方 API 连接 TiDB	45
TiDB 支持的连接器版本	45
JDBC 最佳实践	46
MySQL Connector/J 推荐版本	46
JDBC 参数设置	46
缓解热点问题	47
写入热点	47
热点的观测和定	47
位写入热点的缓解	48
配置 SHARD_ROW_ID_BITS 参数打散写入热	48
点避免连续自增的主键设计	48
分区表	50
读取热点	50
分页的最佳实践	50

分页查询	50
单字段主键表的分页批处理	52
复合主键表的分页批处理	53
唯一序列号生成方案	56
自增列	56
序列 (Sequence)	56
类 Snowflake 方案	57
号段分配方案	57
流程规范	58

前言

目的

本文档旨在为使用 TiDB 数据库的应用和系统提供统一规范参考，标准化 TiDB 数据库的开发使用及SQL优化流程，提高业务开发系统的规范性和代码的可读性，减轻维护工作量，提高工作效率。

适用范围

- TiDB 数据库开发设计人员
- 数据库管理人员
- 数据库运营人员
- 数据分析人员
- 数据库架构师

高亮示意

- 黄色代表需要注意
- 红色代表必须遵守

注意事项

TiDB 暂时不支持的特性：

- 存储过程
- 触发器
- 自定义函数

- 外键约束
- 全文索引
- 空间索引
- 非 UTF8 字符集

对象命名规范

用于规范数据库对象的名命，如数据库（DATABASE）、表（TABLE）、索引（INDEX）、用户（USER）等的命名约定。

原则

- 命名建议使用具有意义的英文词汇，词汇中间以下划线分隔；
- 命名只能使用英文字母、数字、下划线；
- 避免用 TiDB 的保留字如：group，order 等作为单个字段名；
- 建议所有数据库对象使用小写字母。

数据库命名规范

建议按照业务、产品线或者其它指标进行区分，一般不要超过 20 个字符。如：临时库（tmp_crm）、测试库（test_crm）。

表命名规范

- 同一业务或者模块的表尽可能使用相同的前缀，表名称尽可能表达含义；
- 多个单词以下划线分隔，不推荐超过32个字符；
- 建议对表的用途进行注释说明，以便于统一认识。
 - 如：临时表（tmp_t_crm_relation_0425）

- 备份表 (bak_t_crm_relation_20170425)
- 业务运营临时统计表 (tmp_st_[业务代码]_[创建人缩写]_[日期])
- 账期归档表 (t_crm_ec_record_YYYY[MM][DD])
- 不同业务模块的表单独建立 DATABASE，并增加相应注释。
- 目前 TiDB 只支持将 lower-case-table-names 值设为 2，即按照大小写来保存表名，按照小写来比较（不区分大小写）。

字段命名规范

- 字段命名需要表示其实际含义的英文单词或简写；
- 建议各表之间相同意义的字段应同名；
- 字段也尽量添加注释，枚举型需指明主要值的含义，如” 0 - 离线，1 - 在线”；
- 布尔值列命名为 [is_描述]。如 member 表上表示为 enabled 的会员的列命名为 is_enabled；
- 字段名不建议超过 30 个字符，字段个数不建议大于 60。
- 尽量避免使用保留字，如 order、from、desc 等，请参考官方保留字。

索引命名规范

- 主键索引：pk_[表名称简写]_[字段名简写]
- 唯一索引：uk_[表名称简写]_[字段名简写]
- 普通索引：idx_[表名称简写]_[字段名简写]
- 多单词组成的 column_name，取尽可能代表意义的缩写。

数据库对象设计

表的设计

TiDB 中的一张表的 RowID 是按照主键的字节序排序的（整数类型的主键我们会使用特定的编码使其字节序和按大小排序一致），即使在 CREATE TABLE 语句中不显式的创建主键，TiDB 也会为该表分配一个隐式主键；

在进行 TiDB 表设计过程中，有以下几点需要注意：

- 为了保障主从集群复制以及增量备份的幂等性，**表需要有主键或者唯一索引，需要唯一索引所有字段非空**（避免出现多条空值的重复记录）。
- 出于为性能考虑，尽量避免存储超宽表，表字段数不建议超过 60 个，建议单行的总数据大小不要超过 64K，数据长度过大字段最好拆到另外的表；
- 不推荐使用复杂的数据类型；
- 需要 join 的字段，数据类型保障绝对一致，避免隐式转换。

TiDB 字符集默认就是 UTF8，而且目前只支持 UTF8

- 使用 utf8mb4 编码，它是 utf8 的超集，除了将编码改为 utf8mb4 外不需要做其他转换
- TiDB 中，utf8mb4 的默认排序规则为 utf8mb4_bin（区分大小写）
- 4.0 支持 utf8mb4_general_ci（不区分大小写）

字段的设计

整数类型

- TiDB 支持 MySQL 所有的整数类型，包括 INTEGER/INT、TINYINT、SMALLINT、MEDIUMINT 以及 BIGINT。
- INT：所有整数类型的字段推荐只使用 INT 或者 BIGINT；
- BIGINT：定义中不推荐添加长度；
- 推荐使用 INT(10) UNSIGNED 存储 IPv4 格式 IP 地址。

- TINYINT(1)、TINYINT(4) 都是存储一个字节，并不会因为括号里的数字改变。例如 TINYINT(4) 存储 22 则会显示 0022，因为最大宽度为4，达不到的情况下用0来补充。

浮点类型

- TiDB 支持 MySQL 所有的浮点类型，包括 FLOAT、DOUBLE、DECIMAL、NUMERIC等。
- DECIMAL(M,D)：推荐使用 DECIMAL 类型。float 和 double 在存储的时候，存在精度损失的问题，很可能在值的比较时，得到不正确的结果。DECIMAL 在与 VARCHAR、CHAR 类型比较时会转换为 DOUBLE 类型进行比较，也存在精度损失问题，建议在进行比较时使用显示类型转换，如 CAST 避免精度损失。

日期时间类型

- TiDB 支持 MySQL 所有的日期时间类型，包括 DATE、DATETIME、TIMESTAMP、TIME 以及 YEAR。
- DATE：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；
- DATETIME：所有需要精确到时间(时分秒)的字段均使用 DATETIME，不要使用 TIMESTAMP 类型。
- 时间字段使用时间日期类型，不要使用字符串类型存储。否则无法利用日期函数对日期字段进行操作，而需要使用字符串函数进行复杂的操作

处理日期和时间类型时，请记住下面这些：

- 尽管 TiDB 尝试解释不同的格式，日期部分必须是按 年-月-日 的顺序（比如，'98-09-04'），而不是 月-日-年 或者 日-月-年 的顺序。
- 日期值中包含两位数字的年份是有歧义的，TiDB 按下面规则解释：

范围在 70-99 之间的被转换成 1970-1999

范围在 00-69 之间的被转换成 2000-2069

- 如果上下文里面需要的是一个数值，TiDB 自动将日期或时间值转换成数值类型，反之亦然。
- TiDB 允许存储零值 ‘0000-00-00’，有时候这会比 NULL 值更方便一些。（即使是 STRICT_TRANS_TABLES, NO_ZERO_IN_DATE, NO_ZERO_DATE）
- 设置不同的 SQL mode 可以改变 TiDB 的行为。

字符串类型

- TiDB 支持 MySQL 所有的字符串类型，包括 CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM 以及 SET。
- VARCHAR(N)：所有动态长度字符串全部使用 VARCHAR 类型，N 表示的是字符数不是字节数，比如 VARCHAR(256)，需要根据实际的宽度来选择 N，N 尽可能小；
- CHAR：仅仅只有单个字符的字段使用 CHAR(1) 类型，例如性别字段；
- TEXT：仅仅当字符数量可能超过 20000 个的时候，才建议使用 TEXT 类型来存放字符串数据，因为所有 TiDB 数据库都会使用 UTF8 字符集。所有使用 TEXT 类型的字段建议和原表进行分拆，与原表主键单独组成另外一个表进行存放；
- 不建议使用 ENUM、SET 类型，尽量使用 TINYINT 来代替；

字段默认值

在一个数据类型描述中的 DEFAULT value 段描述了一个列的默认值。**这个默认值必须是常量，不可以是一个函数或者是表达式**。但是对于时间类型，可以例外的使用 NOW、CURRENT_TIMESTAMP、LOCALTIME、LOCALTIMESTAMP 等函数作为 DATETIME 或者 TIMESTAMP 的默认值。

BLOB、TEXT 以及 JSON 不可以设置默认值。

如果一个列的定义中没有 DEFAULT 的设置。TiDB 按照如下的规则决定：

- 如果该类型可以使用 NULL 作为值，那么这个列会在定义时添加隐式的默认值设置 DEFAULT NULL。
- 如果该类型无法使用 NULL 作为值，那么这个列在定义时不会添加隐式的默认值设置。

对于一个设置了 NOT NULL 但是没有显式设置 DEFAULT 的列，当 INSERT、REPLACE 没有涉及到该列的值时，TiDB 根据当时的 SQL_MODE 进行不同的行为：

- 如果此时是 strict sql mode，在事务中的语句会导致事务失败并回滚，非事务中的语句会直接报错。
- 如果此时不是 strict sql mode，TiDB 会为这列赋值为列数据类型的隐式默认值。

此时隐式默认值的设置按照如下规则：

- 对于数值类型，它们的默认值是 0。当有 AUTO_INCREMENT 参数时，默认值会按照增量情况赋予正确的值。
- 对于除了时间戳外的日期时间类型，默认值会是该类型的“零值”。时间戳类型的默认值会是当前的时间。
- TIMESTAMP 和 DATETIME 列可以被自动初始化或者更新为当前时间。
- 对于表里面任意的 TIMESTAMP 或者 DATETIME 列，可以将默认值或者自动更新值指定为 current timestamp，通过在列定义时指定 DEFAULT CURRENT_TIMESTAMP 和 ON UPDATE CURRENT_TIMESTAMP 可以设置这些属性。DEFAULT 也可以指定成某个特定的值，比如 DEFAULT 0 或者 DEFAULT '2000-01-01 00:00:00'。

```
CREATE TABLE t1 (
```

```
ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
```

```
dt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP);
```

- 对于除枚举以外的字符串类型，默认值会是空字符串。对于枚举类型，默认值是枚举中的第一个值。

索引设计

- 选择区分度大的列建立索引，不在低基数列上建立索引，例如：“性别”，“否是X”
- 单张表的索引数量控制在 5 个以内，避免冗余索引；
- 索引中的字段数建议不超过 5 个；
- 唯一索引建议由 3 个或更少的字段组成；
- 尽量不要在频繁更新的列上创建索引；
- 对于确定需要组成组合索引的多个字段，建议将选择性高的字段靠前放；
- 最左前缀原则，使用联合索引时，从左向右匹配，比如索引 idx_c1_c2_c3 (c1,c2,c3)，相当于创建了 (c1)、(c1,c2)、(c1,c2,c3) 三个索引，where 条件包含上面三种情况的字段比较则可以用到索引，但像 where c1=a and c3=c 只能用到c1列的索引，像 c2=b and c3=c 等情况就完全用不到这个索引；
- 很长的 VARCHAR 字段建立索引时，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即可。idx_table_name (name(10))
- 定期删除一些长时间未使用过的索引；
- ORDER BY，GROUP BY，DISTINCT 的字段需要添加在索引的后面，形成覆盖索引；
- 新的 select,update,delete 上线，都要先 explain，确保索引的正确性
- 不建议在 where 条件索引列上使用函数，会导致索引失效，如 lower(email)。

- 使用 like 模糊匹配，% 不要放首位，会导致索引失效。

权限设计

TiDB 在数据库初始化时会生成一个 'root'@'%' 的默认账户，生产环境建议调整 root 用户为强密码，且不对外开放。线上所需用户建议按照用户或者业务场景划分，根据实际情况对每个用户授予相应权限，例如：

序号	用户名	涵义	用途
1	root	超级用户	全局管理，禁止对外开放
2	dba	数据库管理员	数据库 DBA
3	app	应用开发	应用开发
4	tempuser	临时统计	线上业务临时统计，只读用户
5	other	其他用户	第三方人员访问

数据模型设计

数据库模型设计是指对于一个给定的应用环境，构造合理的数据库模式，建立数据库及其应用系统，有效存储数据，满足用户信息要求和处理要求。数据库设计在开发过程中处于一个非常重要的地位。一个高效的数据库模型是非常重要和必要的。

完整性

数据库完整性是指数据库中数据的正确性和相容性，数据库完整性是由完整性约束来保证的，数据库完整性对于数据库应用系统非常关键，其作用主要体现在以下几个方面：

- 利用完整性控制机制来实现业务规则，易于定义，容易理解，而且可以降低应用程序的复杂性，提高应用程序的运行效率。
- 合理的数据库完整性设计，能够同时兼顾数据库的完整性和系统的效能。在应用软件的功能测试中，完善的数据库完整性有助于尽早发现应用软件的错误。
- 为了在数据库和应用程序代码之间提供另一层抽象，可以为应用程序建立专门的视图而不必非要应用程序直接访问数据表。这样做还等于在处理数据库 变更时给你提供了更多的自由。

性能

性能是衡量一个系统的关键因素，在设计阶段就在性能方面就应该多关注，尽量减少后期的烦恼。在数据库设计阶段，性能上的考虑时需要注意：不能以范式作为唯一标准或者指导，在设计过程中，需要从实际需求出发，以性能提升为根本目标来展开设计工作，一些时候为了提升性能，甚至会做反范式设计。

另外还有一些设计上的方法和技巧：

- 设置合理的字段类型和长度。字段类型在满足需求后应尽量短，比如，能用int就尽量不要用bigint。另外不同数据库在varchar和text类型在长度和性能上也是不同的，选择时要谨慎。
- 选择高效的主键和索引。由于对表记录的读取都是直接或者间接地通过主键或索引来获取，因此应该根据具体应用特性来设计合理的主键或索引。同时索引长度的也应该关注，尽量减少索引长度。
- 适度冗余。适度的冗余可以避免关联查询，减少join查询。

扩展性

在大规模系统中，除了性能，可扩展性也是设计的关键点，而数据库表扩展性主要包含表逻辑结构、功能字段的增加、分表等。在扩展性上要把握的原则如下：

- 一表一实体。如果不同实体之间有关联时，可增加一个单独的表，不会影响以前的功能。
- 使用分区表。当前支持的类型包括 Range 分区和 Hash 分区。Range 分区可以用于解决业务中大量删除带来的性能问题，支持快速删除分区。Hash 分区则可以用于大量写入场景下的数据打散。

SQL 开发规范

建表删表规范

基本原则：

表的建立在遵循表命名规范前提下，建议业务应用内部封装建表删表语句增加判断逻辑，防止业务流程异常中断。

详细说明：

create table if not exists table_name 或者 drop table if exists table_name 语句建议增加 if 判断，避免应用侧由于表的改动造成的异常中断。

select * 使用规范

基本原则：

禁止使用 select * 进行查询。

详细说明：

按需求选择合适的字段列，杜绝直接 SELECT * 读取全部字段，减少网络带宽消耗，有效利用覆盖索引。

大事务处理

基本原则：

按需求选择合适的字段列，杜绝直接 SELECT * 读取全部字段，减少网络带宽消耗，有效利用覆盖索引；

避免大事务，TiDB 对单个事务的大小有限制，这层限制是在 KV 层面，反映在 SQL 层面的话，简单来说一行数据会映射为一个 KV entry，每多一个索引，也会增加一个 KV entry，所以这个限制反映在 SQL 层面是：

- 最大单行记录容量为 120MB（v5.0 及更高的版本可通过 tidb-server 配置项 performance.txn-entry-size-limit 调整，低于 v5.0 的版本支持的单行容量为 6MB）

- 支持的最大单个事务容量为 10GB (v4.0 及更高版本可通过 tidb-server 配置项 performance.txn-total-size-limit 调整, 低于 v4.0 的版本支持的最大单个事务容量为 100MB)

另外注意, 无论是大小限制还是行数限制, 还要考虑 TiDB 做编码以及事务额外 Key 开销, 在使用的时候, 为了使性能达到最优, 建议每 100 ~ 500 行写入一个事务。

Region 热点

Region 热点可能的原因:

- 这条 SQL 涉及到的 region 的 leader 全部在这个 TiKV 上
- 这条 SQL 只涉及到一个 region, 并且有大量的请求使用同样或者类似的 SQL

基本原则:

如果表的数据量比较小, 数据存储大概率只涉及到一个 region, 大量请求对该表进行写入或者读取都会造成该 region 热点, 可以通过手工拆分 region 方式进行调整, 也可以在建表时预先进行 split region。

详细说明:

如何有效避免热点?

- Handle ID 设计。避免连续自增主键的设计, 建议采用置换算法, 比如时间 12010120180601
- TiDB Partition。通过设置表分区方式来避免热点, 支持按照 Hash 以及按照 Range 分区
- SHARD_ROW_ID_BITS。这个 TABLE OPTION 是用来设置隐式 _tidb_rowid 的分片数量的 bit 位数。

对于 PK 非整数或没有 PK 的表，TiDB 会使用一个隐式的自增 rowid，大量 INSERT 时会把数据集中写入单个 region，造成写入热点。通过设置 SHARD_ROW_ID_BITS 可以把 rowid 打散写入多个不同的 region，缓解写入热点问题。但是设置的过大会造成 RPC 请求数放大，增加 CPU 和网络开销。

SHARD_ROW_ID_BITS = 4 代表 16 个分片，SHARD_ROW_ID_BITS = 6 表示 64 个分片，SHARD_ROW_ID_BITS = 0 就是默认值 1 个分片。

CREATE TABLE 语句示例: CREATE TABLE t (c int) SHARD_ROW_ID_BITS = 4;

ALTER TABLE 语句示例: ALTER TABLE t SHARD_ROW_ID_BITS = 4;

字段上使用函数规范

基本原则：

在取出字段上可以使用相关函数,但是在 Where 条件中的过滤条件字段上严禁使用任何函数,包括数据类型转换函数。

详细说明：

错误的写法：

```
select gmt_create  
  
from ...  
  
where date_format(gmt_create, '%Y%m%d %H:%i:%s') = '20090101 00:00:0'
```

正确的写法：

```
select date_format(gmt_create, '%Y%m%d %H:%i:%s')  
  
from ...
```

```
where gmt_create = str_to_date('20090101 00:00:00', '%Y%m%d %H:%i:s');
```

数据删除规范

基本原则：

删除表中全部的数据时，使用 TRUNCATE 或者 DROP 后重建方式，不要使用 DELETE

。

详细说明：

DELETE, TRUNCATE 和 DROP 都不会立即释放空间，对于 TRUNCATE 和 DROP 操作，在达到 TiDB 的 GC (garbage collection) 时间后（默认 10 分钟），TiDB 的 GC 机制会删除数据并释放空间。对于 DELETE 操作 TiDB 的 GC 机制会删除数据，但不会释放空间，而是当后续数据写入 RocksDB 且进行 compact 时对空间重新利用。

分页查询规范

基本原则：

分页查询语句全部都需要带有排序条件,除非业务方明确要求不要使用任何排序来随机展示数据。

详细说明：

- 常规分页语句写法(start:起始记录数,page_offset:每页记录数):

```
select * from table_a t order by gmt_modified desc limit start, page_offset;
```

- 多表 Join 的分页语句,如果过滤条件在单个表上，内查询语句必须走覆盖索引，先分页,再 Join：

错误的写法：

```
select a.column_a, a.column_b ... b.column_a, b.column_b ...  
  
from table_t a, table_b b  
  
where a.xxx...  
  
and a.column_c = b.column_d  
  
order by a.yyy limit  
  
start, page_offset
```

正确的写法：

```
select from a.column_a, a.column_b ... b.column_a, b.column_b ...  
  
where  
  
(select t.column_a, t.column_b ...  
  
from table_t t  
  
where t.xxx...  
  
order by t.yyy limit start, page_offset) a,  
  
table_b b  
  
a.column_c = b.column_d;  
  
select * from t limit 10000,10;  
  
select * from t order by c limit 10000,10;
```

使用 **order by** 语法保障结果集输出顺序

数据库是基于集合论的，而集合中的元素是没有顺序的。想要按顺序输出的结果集，需明确的把要排序的字段添加到 order by 子句中，这符合 SQL 的语义。

使用 full group by 语法保障结果集稳定

出与便捷的考量，MySQL “扩展” 了 group by 语法，使 select 子句可以引用未在 group by 子句中声明的非聚集字段，也就是 non-full group by 语法，在其他数据库中，这被认为是一种语法错误，因为这会导致结果集不稳定。

```
+-----+-----+-----+
| class   | stuname       | score |
+-----+-----+-----+
| 2018_CS_01 | MonkeyDLuffy | 95.0 |
|          |                |      |
| 2018_CS_03 | PatrickStar   | 99.0 |
|          |                |      |
| 2018_CS_03 | SpongeBob     | 99.0 |
+-----+-----+-----+
```

```
select class,stuname,max(score) as max_score from score group by class;
```

其他规范

- WHERE条件中不在索引列上进行数学运算或函数运算；
- 用 in() /union 替换 or，并注意 in 的个数小于 300；
- 禁止使用%前缀进行模糊前缀查询；

- 如应用使用 Multi Statements 执行 SQL，即将多个 SQL 使用分号连接一次性发给客户端执行，TiDB 只会返回第一个 SQL 的执行结果。

事务限制

隔离级别

TiDB 支持的隔离级别是 Snapshot Isolation (SI)，和 Repeatable Read (RR) 隔离级别基本等价。

SI 可以克服幻读

TiDB 的 SI 隔离级别可以克服幻读异常 (Phantom Reads)，但 ANSI/ISO SQL 标准中的 RR 不能。

所谓幻读是指：事务 A 首先根据条件查询得到 n 条记录，然后事务 B 改变了这 n 条记录之外的 m 条记录或者增添了 m 条符合事务 A 查询条件的记录，导致事务 A 再次发起请求时发现有 $n+m$ 条符合条件记录，就产生了幻读。

例如：系统管理员 A 将数据库中所有学生的成绩从具体分数改为 ABCDE 等级，但是系统管理员 B 就在这个时候插入了一条具体分数的记录，当系统管理员 A 改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

SI 不能克服写偏斜

TiDB 的 SI 隔离级别不能克服写偏斜异常 (Write Skew)，需要使用 Select for update 语法来克服写偏斜异常。

写偏斜异常是指两个并发的事务读取了不同但相关的记录，接着这两个事务各自更新了自己读到的数据，并最终都提交了事务，如果这些相关的记录之间存在着不能被多个事务并发修改的约束，那么最终结果将是违反约束的。

例如：值班表有两列，姓名以及值班状态，0 代表不值班，1 代表值班

姓名	值班状态
张三	0
李四	0
王五	0

有这样一个事务，它的逻辑是判断当前无人值班，则分配一个值班人。当该程序顺序执行时，只会分配一个值班人。但当它并行执行时，就可能出现多人同时为值班状态的错误，造成这个错误的原因就是写偏斜。

不支持 savepoint

TiDB 不支持 savepoint 机制，因此也不支持 **PROPAGATION_NESTED 传播行为**，基于 Java Spring 框架的应用如果使用了 PROPAGATION_NESTED 传播行为，需要在应用端做出调整。

不支持如下操作：

```
mysql> BEGIN;

mysql> INSERT INTO T2 VALUES(100);

mysql> SAVEPOINT svp1;
```

```
mysql> INSERT INTO T2 VALUES(200);

mysql> ROLLBACK TO SAVEPOINT svp1;

mysql> RELEASE SAVEPOINT svp1;

mysql> COMMIT;

mysql> SELECT * FROM T2;
```

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

```
| ID |
```

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

```
| 100 |
```

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

大事务限制

基于日志的数据库在面对大事务时，需要手动调大可用日志的容量，以避免日志被单一事务占满。

TiDB 处理大事务的性能相较于处理并发的小事务的性能要差，因此 TiDB 中对于事务量设定了一些限制：

- 最大单行记录容量为 120MB（v5.0 及更高的版本可通过 tidb-server 配置项 `performance.txn-entry-size-limit` 调整，低于 v5.0 的版本支持的单行容量为 6MB）

- 支持的最大单个事务容量为 10GB (v4.0 及更高版本可通过 tidb-server 配置项 performance.txn-total-size-limit 调整, 低于 v4.0 的版本支持的最大单个事务容量为 100MB)

隐式类型转换

当 SQL 中谓词两侧的数据类型不一致时, TiDB 将隐式的将一侧或两侧的数据类型进行转换, 将其变为兼容的数据类型, 以进行谓词运算。

TiDB 中隐式类型转换规则如下:

- 如果一个或两个参数都是 NULL, 比较的结果是 NULL, 除了 NULL 安全的 `<=>` 相等比较运算符。对于 `NULL <=> NULL`, 结果为 true。不需要转换
- 如果比较操作中的两个参数都是字符串, 则将它们作为字符串进行比较。
- 如果两个参数都是整数, 则将它们作为整数进行比较。
- 如果不与数字进行比较, 则将十六进制值视为二进制字符串
- 如果其中一个参数是十进制值, 则比较取决于另一个参数。如果另一个参数是十进制或整数值, 则将参数与十进制值进行比较, 如果另一个参数是浮点值, 则将参数与浮点值进行比较
- 如果其中一个参数是 TIMESTAMP 或 DATETIME 列, 另一个参数是常量, 则在执行比较之前将常量转换为时间戳。
- 在所有其他情况下, 参数都是作为浮点数 (double 类型) 比较的。

隐式类型转换增强了人机交互的易用性, 但在应用代码中, 我们应尽量避免隐式类型转换出现, 这是由于隐式类型转换会导致:

- 索引失效

如下案例，account_id 为主键，其数据类型为 varchar。通过执行计划可见，该SQL 发生了隐式类型转换，无法使用索引。

```

...

desc select * from account where account_id=6010000000009801;

+-----+-----+-----+-----+-----+
| id          | estRows  | task  | access object | operator info |
+-----+-----+-----+-----+-----+

| TableReader_7      | 8000628000.00 | root  |               | data:Selection_6 |
|
| Selection_6         | 8000628000.00 | cop[tikv] |               | eq(cast(findpt.account.account_id), 6.0100000000009801e+15) |
| TableFullScan_5    | 10000785000.00 | cop[tikv] | table:account | keep order:false |
|
+-----+-----+-----+-----+-----+

3 rows in set (0.00 sec)

...

```

● 精度丢失

如下案例，字段 a 的数据类型为 decimal(32,0)，从执行计划可以得知，出现了隐式类型转换，decimal 字段和字符串常值都被转换为 double 类型，而 double 类型的精度会丢失。

度没有 decimal 高，出现了精度丢失，在这个 case 中，造成了筛选出范围之外的结果集的错误。

...

```
desc select * from t1 where a between '12123123' and '1111222211111111200000';
```

```
+-----+-----+-----+-----+
+-----+
| id          | estRows | task      | access object | operator info |
+-----+
| TableReader_7 | 0.80    | root      |               | data:Selection_6 |
|
| Selection_6   | 0.80    | cop[tikv] |               | ge(cast(findpt.t1.a), 1.2123123e+07) |
| le(cast(findpt.t1.a), 1.1112222111111112e+21) |
|
| Selection_6   | 0.80    | cop[tikv] |               | ge(cast(findpt.t1.a), 1.2123123e+07) |
| le(cast(findpt.t1.a), 1.1112222111111112e+21) |
|
| TableFullScan_5 | 1.00    | cop[tikv] | table:t1      | keep order:false, stats:pseudo |
|
+-----+-----+-----+-----+
+-----+
```

3 rows in set (0.00 sec)

```
MySQL [findpt]> select * from t1 where a between '12123123' and '1111222211111111200000';
```

```
+-----+
| a          |
+-----+
```

© 2022 PingCAP 公司保留所有权利。除非版权法允许，否则在未得到本公司事先给出的书面许可情况下，严禁复制、改编或翻译本文。

```
+-----+
| 1111222211111111222211 |
+-----+

1 row in set (0.01 sec)

...
```

结果集不稳定

group by

出于便捷的考量，MySQL “扩展” 了 group by 语法，使 select 子句可以引用未在 group by 子句中声明的非聚集字段，也就是 non-full group by 语法，在其他数据库中，这被认为是一种语法错误，因为这会导致结果集不稳定。

在下例的 3 条 SQL 语句中，第一条 SQL 使用了 full group by 语法，所有在 select 子句中引用的字段，都在 group by 子句中有所声明，它的结果集是稳定的，可以看到 class 与 stuname 的全部组合共有三种；第二条与第三条是同一个 SQL，但它在两次执行时得到了不同的结果，这条 SQL 的 group by 子句中仅声明了一个 class 字段，因此结果集只会针对 class 进行聚集，class 的唯一值有两个，也就是说结果集中只会包含两行数据，而 class 与 stuname 的全部组合共有三种，班级 2018_CS_03 有两位同学，每次执行时返回哪位同学是没有语义上的限制的，都是符合语义的结果。

...

```
mysql> select a.class, a.stuname, max(b.coursescore) from stu_info a join stu_score b on
a.stuno=b.stuno group by a.class, a.stuname order by a.class, a.stuname;
```

```
+-----+-----+-----+ | class
```

```
| stuname | max(b.courscore) |+-----
```

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

```
| 2018_CS_01 | MonkeyDLuffy |      95.5 |
```

```
| 2018_CS_03 | PatrickStar |      99.0 |
```

```
| 2018_CS_03 | SpongeBob |      95.0 |
```

```
+-----+-----+-----+ 3 rows
```

in set (0.00 sec)

```
mysql> select a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on
a.stuno=b.stuno group by a.class order by a.class, a.stuname;
```

```
+-----+-----+-----+ | class
```

```
| stuname | max(b.courscore) |+-----
```

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

```
| 2018_CS_01 | MonkeyDLuffy |      95.5 |
```

```
| 2018_CS_03 | SpongeBob |      99.0 |
```

```
+-----+-----+-----+ 2 rows
```

in set (0.01 sec)

```
mysql> select a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on  
a.stuno=b.stuno group by a.class order by a.class, a.stuname;
```

```
+-----+-----+-----+ | class
```

```
| stuname | max(b.courscore) |+-----
```

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

```
| 2018_CS_01 | MonkeyDLuffy |      95.5 |
```

```
| 2018_CS_03 | PatrickStar |      99.0 |
```

```
+-----+-----+-----+ 2 rows
```

```
in set (0.01 sec)
```

```
...
```

因此，想保障 group by 语句结果集的稳定，请使用 full group by 语法

MySQL 提供了一个 SQL_MODE 开关 only_full_group_by 来控制是否进行 full group by 语法的检查，TiDB 也兼容了这个 SQL_MODE 开关：

```
...
```

```
mysql> select a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on  
a.stuno=b.stuno group by a.class order by a.class, a.stuname;
```

```
+-----+-----+-----+ | class
```

```
| stuname | max(b.courscore) |+-----
```

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

```
| 2018_CS_01 | MonkeyDLuffy |      95.5 |
```

| 2018_CS_03 | PatrickStar | 99.0 |

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

2 rows in set (0.01 sec)

mysql> set

@@sql_mode='STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION,ONLY_FULL_GROUP_BY';

Query OK, 0 rows affected (0.01 sec)

mysql> select a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on
a.stuno=b.stuno group by a.class order by a.class, a.stuname;

ERROR 1055 (42000): Expression #2 of ORDER BY is not in GROUP BY clause and contains
nonaggregated column " which is not functionally dependent on columns in GROUP BY clause; this is
incompatible with sql_mode=only_full_group_by

...

order by

在 SQL 的语义中，只有使用了 order by 语法才会保障结果集的顺序输出。而单机数据库由于数据都存储在一台服务器上，在不进行数据重组时，多次执行的结果往往是稳定的，有些数据库(尤其是 MySQL InnoDB 存储引擎)还会按照主键或索引的顺序进行结果集的输出。TiDB 是分布式数据库，数据被存储在多台服务器上，另外 TiDB 层不缓存数据页，因此不含 order by 的 SQL 语句的结果集展现顺序容易被感知到不稳定。想要按顺序输出的结果集，需明确的把要排序的字段添加到 order by 子句中，这符合 SQL 的语义。

在下面的案例中，用户只在 order by 子句中添加了一个字段，TiDB 只会按照这一个字段进行排序。

...

```
mysql> select a.class, a.stuname, b.course, b.courscore from stu_info a join stu_score b on
a.stuno=b.stuno order by a.class;
```

stuname	course	courscore
2018_CS_01 MonkeyDLuffy		60.5
PrinciplesofDatabase 2018_CS_01 MonkeyDLuffy		43.0
2018_CS_01 MonkeyDLuffy OpSwimming		67.0
2018_CS_01 MonkeyDLuffy OpFencing		76.0
2018_CS_01 MonkeyDLuffy FundamentalsofCompiling		88.0
2018_CS_01 MonkeyDLuffy OperatingSystem		90.5
2018_CS_01 MonkeyDLuffy PrincipleofStatistics		69.0
2018_CS_01 MonkeyDLuffy ProbabilityTheory		76.0
2018_CS_01 MonkeyDLuffy Physics		63.5
2018_CS_01 MonkeyDLuffy AdvancedMathematics		95.5
2018_CS_01 MonkeyDLuffy LinearAlgebra		92.5
2018_CS_01 MonkeyDLuffy DiscreteMathematics		89.0

	2018_CS_03		SpongeBob		PrinciplesofDatabase		88.0	
	2018_CS_03		SpongeBob		English		79.0	
	2018_CS_03		SpongeBob		OpBasketball		92.0	
	2018_CS_03		SpongeBob		OpTennis		94.0	
	2018_CS_03		PatrickStar		LinearAlgebra		6.5	
	2018_CS_03		PatrickStar				5.0	
	2018_CS_03		SpongeBob		DiscreteMathematics		72.0	
	2018_CS_03		PatrickStar		ProbabilityTheory		12.0	
	2018_CS_03		PatrickStar		PrincipleofStatistics		20.0	
	2018_CS_03		PatrickStar		OperatingSystem		36.0	
	2018_CS_03		PatrickStar		FundamentalsofCompiling		2.0	
	2018_CS_03		PatrickStar				14.0	
	2018_CS_03		PatrickStar		DiscreteMathematics		9.0	
	2018_CS_03		PatrickStar		English		60.0	
	2018_CS_03		PatrickStar				12.0	
	2018_CS_03		PatrickStar		OpTableTennis		99.0	
	2018_CS_03		SpongeBob		FundamentalsofCompiling		43.0	
	2018_CS_03		SpongeBob		OperatingSystem		95.0	
	2018_CS_03		SpongeBob				90.0	
	2018_CS_03		SpongeBob		PrincipleofStatistics		87.0	

```
| 2018_CS_03 | SpongeBob    | Physics          | 65.0 |
| 2018_CS_03 | SpongeBob    | AdvancedMathematics | 55.0 |
| 2018_CS_03 | SpongeBob    | LinearAlgebra     | 60.5 |
| 2018_CS_03 | PatrickStar  | Physics          | 6.0  |
```

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

36 rows in set (0.01 sec)

...

由于 `group_concat()` 中没有使用 `order by` 导致结果集不稳定

结果集不稳定是因为 TiDB 是并行地从存储层读取数据，所以 `group_concat()` 不加 `order by` 的情况下得到的结果集展现顺序容易被感知到不稳定。

`group_concat()` 要获取到按顺序输出的结果集，需要把用于排序的字段添加到 `order by` 子句中，这样才符合 SQL 的语义。在下面的案例中，使用 `group_concat()` 不加 `order by` 的情况下拼接 `customer_id` 结果集不稳定

1) 不加 `order by`

第一次查询：

...

```
mysql> select GROUP_CONCAT( customer_id SEPARATOR ',' ) FROM customer where customer_id like
'200002%';
```

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

```
| GROUP_CONCAT(customer_id SEPARATOR ',') |
+-----+
| 20000200992,20000200993,20000200994,20000200995,20000200996,20000200... |
+-----+
...
```

第二次查询：

```
mysql> select GROUP_CONCAT( customer_id SEPARATOR ',') FROM customer where customer_id like
'200002%';
+-----+
| GROUP_CONCAT(customer_id SEPARATOR ',') |
+-----+
| 20000203040,20000203041,20000203042,20000203043,20000203044,20000203... |
+-----+
...
```

2) 加 order by

第一次查询：

```
mysql> select GROUP_CONCAT( customer_id order by customer_id SEPARATOR ',') FROM customer
where customer_id like '200002%';
```

```
+-----+
| GROUP_CONCAT(customer_id SEPARATOR ',') |
+-----+
| 20000200000,20000200001,20000200002,20000200003,20000200004,20000200... |
+-----+
...
```

第二次查询：

```
mysql> select GROUP_CONCAT( customer_id order by customer_id SEPARATOR ',' ) FROM customer
where customer_id like '200002%';

+-----+
| GROUP_CONCAT(customer_id SEPARATOR ',') |
+-----+
| 20000200000,20000200001,20000200002,20000200003,20000200004,20000200... |
+-----+
...
```

索引|注意事项

TiDB 中的索引

索引也是数据，也要占用存储空间。和表中的数据一样，TiDB 中表的索引在存储引擎中也被作为 kv 来存储，一行索引是一个 kv 对。例如一张有 10 个索引的表，每插入一行数据的时候，会写入 11 个 kv 对。

TiDB 支持主键索引，唯一索引，也支持二级索引，构成以上索引的可以是单一列，也可以是多个列（复合索引）。

TiDB 目前（v5.0）还不支持反向/双向索引，全文索引，分区表的全局索引。

TiDB 中在查询的谓词是 =, >, <, >=, <=, like ‘...%’, not like ‘...%’, in, not in, <>, !=, is null, is not null, between...and ... 时能够使用索引，使用与否由优化器来决策。

TiDB 中在查询的谓词是 like ‘%...’, like ‘%...%’, not like ‘%...’, not like ‘%...%’, <=>

时，都无法使用索引。*注：目前版本（v5.0）还没有对 <=> 做处理，无法像 is null 一样使用索引。

复合索引的设计

由于目前的版本（v5.0）还没有 Index Merge 的功能，因此符合索引的设计尤为重要。

TiDB 中的复合索引形如 key tablekeyname (a,b,c)，与其他数据库一样，设计复合索引的一般原则是尽可能的把使用频率比较高的字段放在前面。在当前版本（v5.0 及以下的全部版本）使用中需要特别注意，复合索引中前一列的范围查询会中止后续索引列的使用，可以通过下面的案例来理解这个特性。在如下的查询中 `select a,b,c from tablename where a<predicate>’ <value1>’ and b<predicate>’ <value2>’ and c<predicate>’ <value3>’ ;`

如果 a 条件的谓词（语句中的 predicate）是 = 或 in，那么在 b 的查询条件上就可以利用到组合索引 (a,b,c)。例：`select a,b,c from tablename where a=1 and b<5 and c=’ abc’ ;`

同样的，如果 a 条件和 b 条件的谓词都是 = 或 in，那么在 c 上的查询就可以利用到组合索引 (a,b,c)。例：`select a,b,c from tablename where a in (1,2,3) and b=5 and c=’ abc’ ;`

如果 a 条件的谓词不是 = 也不是 in, 那么 b 上的查询就无法利用到组合索引(a,b,c)。此时 b 条件将在 a 条件筛选后的数据中进行无索引的数据扫描。例 `select a,b,c from tablename where a>1 and b<5 and c=' abc' ;`

这是由于在 TiDB 中, 复合索引中排在前面的列如果被用于范围查询, 那么后续列的查询就会在前列筛选后的数据范围中进行非索引的扫描。综上, 在 TiDB 中进行复合索引设计时, 需要尽可能的将使用频率高的, 经常被点查使用的列排在前面, 将经常进行范围查询的列排在后面。

另外形如 `select c, count(*) from tablename where a=1 and b=2 group by c order by c;` 的查询可以利用到索引 (a,b,c), 同样遵循上面的原则。

自增列注意事项

原理

TiDB 的自增 ID (auto_increment) 只保证自增且唯一, 并不保证连续分配。TiDB 目前采用批量分配的方式, 所以如果在多台 TiDB 上同时插入数据, 分配的自增 ID 会不连续。当多个线程并发往不同的 tidb-server 插入数据的时候, 有可能会发生后插入的数据自增 ID 小的情况。此外, TiDB 允许给数值类型的列指定 auto_increment, 且一个表只允许一个属性为 auto_increment 的列。

最佳实践

设置自增 ID 的目的一般是将它作为表内数据的唯一性约束, 因此被设计为主键或唯一索引, 此类列属性应带有 not null。

自增 ID 列的类型必须为整型，在几种整型类型中，我们建议使用 `bigint`，这是由于即使在单机数据库中也屡见 `int` 类型的自增 ID 被耗光的情况，而 TiDB 被用于处理比单机数据大得多的数据量，此外 TiDB 采用多线程的方式分配自增 ID，因此 `int` 类型无法满足需求。另外自增 ID 一般不需要存储负值，为列增加 `unsigned` 属性可以扩充一倍的 id 存储容量。`int` 无符号的范围是 0 到 4294967295，`bigint` 无符号的范围是 0 到 18446744073709551615

综上，自增 ID 设计的最佳实践如下：

```
`auto_inc_id` bigint unsigned not null unique key auto_increment comment '自增 ID'
```

人为赋值的后果及挽救措施

在集群中有多个 `tidb-server` 时，人为向自增列写入值之后，可能会导致 TiDB 分配自增 ID 冲突而报 “Duplicate entry” 错误：

假设有这样一个带有自增 ID 的表：`create table t(id int unique key auto_increment, c int);`

TiDB 实现自增 ID 的原理是每个 `tidb-server` 实例缓存一段 ID 值用于分配（目前会缓存 30000 个 ID），用完这段值再去取下一段。

假设集群中有两个 `tidb-server` 实例 A 和 B（A 缓存 [1,30000] 的自增 ID，B 缓存 [30001,60000] 的自增 ID），依次执行如下操作：

客户端向 B 插入一条将 id 设置为 1 的语句 `insert into t values (1, 1)`，并执行成功。

客户端向 A 发送 Insert 语句 `insert into t (c) (1)`，这条语句中没有指定 id 的值，所以会由 A 分配，当前 A 缓存了 [1, 30000] 这段 ID，所以会分配 1 为自增 ID 的值，并把本地计数器加 1。而此时数据库中已经存在 id 为 1 的数据，最终返回 `Duplicated Entry` 错误。

处理该问题只需要调大表上的 `AUTO_INCREMENT` 属性值即可让所有 `tidb-server` 重新获取一段自增 ID:

1. 确认表上自增值的最大值: `show create table t;`
2. 修改表上的自增值最大值到一个更大的值: `alter table t AUTO_INCREMENT=120000;`

TiDB 中的各种超时

GC 超时

TiDB 的事务的实现采用了 MVCC (多版本并发控制) 机制, 当新写入的数据覆盖旧的数据时, 旧的数据不会被替换掉, 而是与新写入的数据同时保留, 并以时间戳来区分版本。TiDB 通过定期 GC 的机制来清理不再需要的旧数据。

默认配置下 TiDB 可以保障每个 MVCC 版本 (一致性快照) 保存 10 分钟, 读取时间超过 10 分钟的事务, 会收到报错 `GC life time is shorter than transaction duration`

当用户确信自己需要更长的读取时间时, 比如在使用了 Mydumper 做全量备份的场景中 (Mydumper 备份的是一致性的快照), 可以通过调整 TiDB 中 `mysql.tidb` 表中的 `tikv_gc_life_time` 的值来调大 MVCC 版本保留时间, 需要注意的是 `tikv_gc_life_time` 的配置是立刻影响全局的, 调大它会为当前所有存在的快照增加生命时长, 调小它会立即缩短所有快照的生命时长。过多的 MVCC 版本会拖慢 TiKV 的处理效率, 在使用 Mydumper 做完全量备份后需要及时把 `tikv_gc_life_time` 调整回之前的设置。

更多关于 GC 的信息, 请参考官网文档:

<https://pingcap.com/docs-cn/stable/reference/garbage-collection/overview/>

事务超时

含 DML 语句的事务，除了受 `tikv_gc_life_time` 限制之外，还受到另外一个参数 `max-txn-time-use` 的影响，这个参数位于 `tidb-server` 的配置文件 `tidb.toml` 中，用于控制单个事务允许的最大执行时间。该参数的默认值为 590（秒），需要注意必须控制该参数的值小于 `tikv_gc_life_time` 的值。

形如 `insert into t10 select * from t1` 的 SQL 语句，即使执行时间没有达到 `tikv_gc_life_time` 限制，但超过了 `max-txn-time-use` 的限制，会由于超时而回滚。

SQL 超时

TiDB 还提供了一个系统变量来限制单条 SQL 语句的执行时间：`max_execution_time`，它的默认值为 0，表示无限制。`max_execution_time` 目前对所有类型的 statement 生效，并非只对 SELECT 语句生效。其单位为 ms，但实际精度在 100ms 级别，而非更准确的毫秒级别。

连接器和 API

数据库连接器为客户端提供了连接数据库服务端的方式，APIs 提供了使用 MySQL 协议和资源的底层接口。无论是连接器还是 API 都可以用来在不同的语言和环境内连接服务器并执行 sql 语句，包括 `odbc`、`java(jdbc)`、`Perl`、`Python`、`PHP`、`Ruby` 和 `C`。

TiDB 兼容 MySQL(5.6、5.7) 的所有连接器和 API，包括：

MySQL Connector/C

MySQL Connector/C++

MySQL Connector/J

MySQL Connector/Net

MySQL Connector/ODBC

MySQL Connector/Python

MySQL C API

MySQL PHP API

MySQL Perl API

MySQL Python API

MySQL Ruby APIs

MySQL Tcl API

MySQL Eiffel Wrapper

Mysql Go API

使用 MySQL 连接器连接 TiDB

Oracle 官方提供了以下 API , TiDB 可以兼容所有这些 API。

MySQL Connector/C: C 语言的客户端库, 是 libmysqlclient 的替代品

MySQL Connector/C++: C++ 语言的客户端库

MySQL Connector/J: Java 语言的客户端库, 基于标准 JDBC 接口

MySQL Connector/Net: .Net 语言的客户端库, MySQL for Visual Studio使用这个库, 支持 Microsoft Visual Studio 2012, 2013, 2015 和 2017 版本

MySQL Connector/ODBC: 标准的 ODBC 接口, 支持 Windows, Unix 和 OS X

MySQL Connector/Python: Python 语言的客户端包, 和 Python DB API version 2.0

一致

使用 MySQL C API 连接 TiDB

如果使用 C 语言程序直接连接 TiDB, 可以直接链接 libmysqlclient 库, 使用 MySQL 的 C API, 这是最主要的一种 C 语言连接方式, 被各种客户端和 API 广泛使用, 包括 Connector/C。

使用 MySQL 第三方 API 连接 TiDB

第三方 API 非 Oracle 官方提供, [链接](#)列出了常用的第三方 API:

TiDB 支持的连接器版本

Connector Connector version

Connector/C 6.1.0 GA

Connector/C++ 1.0.5 GA

Connector/J 5.1.8

Connector/Net 6.9.9 GA

Connector/Net 6.8.8 GA

Connector/ODBC 5.1

Connector/ODBC 3.51 (Unicode not supported)

Connector/Python 2.0

Connector/Python 1.2

JDBC 最佳实践

MySQL Connector/J 推荐版本

TiDB 服务端兼容 MySQL 5.7，客户端推荐使用 5.1.36 或更高版本的 5.1.x jdbc 驱动[下载链接](#)。

JDBC 参数设置

Java 应用常用的数据库连接池包括 weblogic、c3p0、Druid等。使用连接池配置数据源时，需要配置一系列参数，其中比较重要的包括 jdbc 的 url 配置，超时探活机制等。充分认识并理解各项参数有助于让 TiDB 发挥出更高的性能。Mysql 5.1版本 jdbc configuration properties 参见 [链接](#)：

一个建议的 url 配置如下：

...

```
spring.datasource.url=jdbc:mysql://{TiDBIP}:{TiDBPort}/{DBName}?characterEncoding=utf8&useSSL=false&useServerPrepStmts=true&prepStmtCacheSqlLimit=10000000000&useConfigs=maxPerformance&rewriteBatchedStatements=true&defaultFetchSize=-214783648
```

...

缓解热点问题

写入热点

© 2022 PingCAP 公司保留所有权利。除非版权法允许，否则在未得到本公司事先给出的书面许可情况下，严禁复制、改编或翻译本文。

大部分单机 RDBMS 采用 B+ tree 数据结构，主键往往是用于组织数据的关键索引（此时表被称作索引组织表），同一数据页内的记录按主键顺序存放。因此单机 RDBMS 产品一般推荐写入连续的序列号，这样每次写入新的记录，都会顺序添加到当前 B+ tree 索引节点的后续位置，当前的数据页写满时，会自动开始新一页的写入。相反，过于随机的主键值，会导致新记录被写入到数据页的某个中间位置，造成数据的移动而带来了额外的开销。

虽然 TiDB 具有不同于单机 RDBMS 的数据结构，但顺序的主键值写入，在 TiDB 上也会产生类似的效果：TiKV 上一个的 region 被写满，进而分裂出一个新的 region，后续的写入转由新的 region 来承载。但甲之蜜糖，乙之砒霜，单机 RDBMS 的最佳实践放到 TiDB 上，会使写入压力总是集中在一个 region 上，这样就构成了持续的写入热点，无法发挥出 TiDB 这种分布式数据库的并行写入能力，降低了业务写入瓶颈，造成了系统资源的浪费。

TiDB 发生写入热点的原因主要有以下几种：

- 小表，整个表只有一个 region，高并发的请求会造成读写热点。
- 大量写入时 Key 值的离散程度不足以跳过一个 region 大小（96MB）而引起写入热点。

热点的观测和定位

TiDB 从 v4.0 版本开始提供便于迅速识别集群负载的 Dashboard 流量可视化页面（Key Visualizer）。Key Visualizer 的具体使用方法请参考[官方文档](#)。

写入热点的缓解

配置 SHARD_ROW_ID_BITS 参数打散写入热点

对于非索引组织表（默认行为），TiDB 会使用一个隐式的自增 rowid，大量 INSERT 时会把数据集中写入单个 region，造成写入热点。通过设置可以把 SHARD_ROW_ID_BITS rowid 打散写入多个不同的 region，缓解写入热点问题。但是设置的过大会造成 RPC 请求数放大，增加 CPU 和网络开销。

用 SHARD_ROW_ID_BITS 来设置隐藏列 _tidb_rowid 分片数量的 bit 位数，默认值为 0，即 $2^0 = 1$ 个分片。SHARD_ROW_ID_BITS = 4 代表 16 个分片，SHARD_ROW_ID_BITS = 6 表示 64 个分片，SHARD_ROW_ID_BITS = 0 就是默认值 1 个分片。

CREATE TABLE 语句示例：`CREATE TABLE t (c int) SHARD_ROW_ID_BITS = 4 PRE_SPLIT_REGIONS=4;`

ALTER TABLE 语句示例：`ALTER TABLE t SHARD_ROW_ID_BITS = 4 PRE_SPLIT_REGIONS=4;`

避免连续自增的主键设计

对索引组织表来说，它无法利用到 SHARD_ROW_ID_BITS 的优化，可以通过修改序列号的方式来构成多个写入分片来分散写入热点。

如下表所示，左侧为原本要写入表中的序列号值，右侧为转换后的序列号。经过转换后的序列号不再连续且离散程度足以跳过一个 region 大小，以此方式来分散写入热点：

原始序列号	转换后的序列号
561632371724517376	566163237172451737

561632371728711680	506163237172871168
561632371728711681	516163237172871168
561632371728711682	526163237172871168
561632371732905984	546163237173290598
561632371732905985	556163237173290598
561632371732905986	566163237173290598
561632371732905987	576163237173290598
561632371732905988	586163237173290598
561632371737100288	586163237173710028

表 1. 将连续的写入转换为 10 个分片写入的案例

分区表

[分区表](#) ([partitioned table](#)) 可以将一张表的数据分散到多张物理表中，而多张物理表的数据是分散在多个 region 中的，因此通过合理的涉及分区规则，可以进一步避免写入热点问题。

读取热点

即使使用了上述的 SHARD_ROW_ID_BITS 方式打散数据在 region 上的分布，有些小表的热点问题也可能依然存在，典型场景是被频繁读取的参数配置表，高并发的读取集中在一个或几个 region 上，有可能成为整笔交易的性能瓶颈。

推荐做法是在应用侧做缓存处理，使用如 redis 这样的产品进行缓存。

分页的最佳实践

分页查询

基本原则：

分页查询语句全部都需要带有排序条件,除非业务方明确要求不要使用任何排序来随机展示数据。

详细说明：

- 常规分页语句写法(start:起始记录数,page_offset:每页记录数)：

...

```
select * from table_a t order by gmt_modified desc limit start, page_offset;
```

...

- 多表 Join 的分页语句,如果过滤条件在单个表上, 内查询语句必须走覆盖索引, 先分页,再 Join:

错误的写法:

...

```
select a.column_a, a.column_b ... b.column_a, b.column_b ...
```

```
from table_t a, table_b b
```

```
where a.xxx...
```

```
and a.column_c = b.column_d
```

```
order by a.yyy
```

```
limit start, page_offset
```

...

正确的写法:

...

```
select a.column_a, a.column_b ... b.column_a, b.column_b ...
```

```
from
```

```
(select t.column_a, t.column_b ...
```

```
from table_t t
```

```
where t.xxx...
```

```
order by t.yyy

limit start, page_offerset) a,

table_b b

where a.column_c = b.column_d;

select * from t limit 10000,10;

select * from t order by c limit 10000,10;

...
```

单字段主键表的分页批处理

常规的分页更新 SQL 一般使用主键或者唯一索引进行排序，再配合 MySQL limit 语法中非常好用的 offset 功能按固定行数拆分页面，然后把页面包装进独立的事务中，从而实现灵活的分页更新。但是，劣势也很明显：由于需要对主键或者唯一索引进行排序，越靠后的页面参与排序的行数就会越多，尤其当批量处理涉及的数据体量较大时，可能会占用过多计算资源。

首先将数据按照主键排序，然后调用窗口函数row_number() 为每一行数据生成行号，接着调用聚合函数按照设置好的页面大小对行号进行分组，最终计算出每页的最小值和最大值。

...

```
MySQL [demo]> select min(t.serialno) as start_key, max(t.serialno) as end_key, count(*)
as page_size from ( select *, row_number () over (order by serialno) as
row_num from tmp_loan ) t group by floor((t.row_num - 1) / 50000) order by
start_key;
```

```

+-----+-----+-----+      |
start_key | end_key  | page_size |

+-----+-----+-----+
| 200000000 | 200050001      50000 |
|          | 200050002      | 50000 |
200100007 | | 200100008      50000 |
| 200150008      |      | 50000 |
200150009 | 200200013 |      50000 |
| 200200014..| 200250017
| 201900019 | 201950018 |      50000 |
| 201950019 | 201999003 |      48985 |

+-----+-----+-----+ 40

rows in set (1.51 sec)

...

```

接下来，只需要使用 `serialno between start_key and end_key` 查询每个分片的数据即可。

如果修改数据时，也可以借助上面计算好的分片信息，实现高效数据更新。

改进方案由于规避了频繁的数据排序操作造成的性能损耗，显著改善了批量处理的效率。

复合主键表的分页批处理

对于非索引组织表，可以使用隐藏字段 `_tidb_rowid` 做分页使用。

对于索引组织表，可以用如下方式进行分页计算：

制作元信息表

...

```
mysql> SELECT floor(( t1.row_num - 1 ) / 600000 )+1 rn,
min(mvalue),max(mvalue),count(*) FROM (SELECT concat( '(', customer_id, ',',
customer_idno, ')') AS mvalue, row_number() over ( ORDER BY customer_id,
customer_idno ) AS row_num FROM findpt.customer) t1 GROUP BY floor((
t1.row_num - 1 ) / 600000 ) ORDER BY rn;
```

rn	min(mvalue)	max(mvalue)	count(*)
1	('100000000001','351421198512031871')		600000
2	('100006000001','410727197307043818')		600000

```
| 3 | ('10001200001','521527198406224414') |  
('10001800000','320209197609305969') | 600000 |  
  
| 4 | ('10001800001','220304197912193073')  
('10002400000','230504197308067651') | 600000 |  
|  
| 5 | ('10002400001','121711197208214015')  
('10003000000','430112199003258074') | 600000 |  
|  
| 6 | ('10003000001','330609198706142725')  
('10003600000','520519197407128506') | 600000 |  
|  
| 7 | ('10003600001','621108199508175476')  
('10004200000','631319197203254252') | 600000 |  
|  
| 8 | ('10004200001','350406198608214809')  
('10004800000','500827199406068657') | 600000 |  
|  
| 9 | ('10004800001','450311198612295355')  
('10005400000','430713199601229738') | 600000 |  
|  
| 10 | ('10005400001','640608199311094703')  
('10006000000','131222199007068025') | 600000 |  
|  
| 11 | ('10006000001','110724197808158121')  
('10006600000','410909199902088607') | 600000 |  
|  
| 12 | ('10006600001','371802199909286692')  
('10007200000','331616199104157617') | 600000 |  
|
```

```
| 13 | ('10007200001','631618198707015770') |  
(('10007800000','311424198409271703')) | 600000 |  
  
| 14 | ('10007800001','450212199805062337')  
(('10008400000','141520197703176129')) | 600000 |  
|  
| 15 | ('10008400001','130920197811106553')  
(('10009000000','640206197509055077')) | 600000 |  
|  
| 16 | ('10009000001','151822197801136758')  
(('10009600000','810620197505228665')) | 600000 |  
|  
| 17 | ('10009600001','230109198906203721')  
(('10010000000','340408198312036321')) | 400000 |  
|  
+----+-----+-----+-----+-----+
```

17 rows in set (26.42 sec)

...

批量处理 SQL 案例

...

```
delete from customer where (customer_id, customer_idno) >=  
(('10000000001','351421198512031871')) and (customer_id, customer_idno) <=  
(('10000600000','541420198607276566')) order by customer_id,customer_idno;
```

...

唯一序列号生成方案

自增列

自增 (auto_increment) 是大多数兼容 MySQL 协议的 RDBMS 上列的一种属性，通过配置该属性来使数据库为该列的值自动赋值，用户不需要为该列赋值，该列的值随着表内记录增加会自动增长，并确保唯一性。在大多数场景中，自增列被作为无业务涵义的代理主键使用。自增列的局限性在于：自增列只能采用整型字段，所赋的值也只能为整型。假设业务所需要的序列号由字母、数字及其他字符拼接而成，用户是难以通过自增列来获取序列号中所需的数字自增值的。

序列 (Sequence)

序列是一种数据库对象，应用程序通过调用某个序列可以产生递增的序列值，应用程序可以灵活的使用这个序列值为一张表或多张表赋值，也可以使用序列值进行更复杂的加工，来实现文本和数字的组合，来赋予代理键以一定的跟踪和分类的意义。TiDB 从v4.0 版本开始提供序列功能，详情请参考[官方文档](#)。

类 Snowflake 方案

Snowflake 是 Twitter 提出的分布式 ID 生成方案。目前有多种实现，较流行的是百度的 uid-generator 和美团的 leaf。下面以 uid-generator 为例展开说明。

uid-generator 生成的 64 位 ID 结构如下：

- sign：长度固定为 1 位。固定为 0，表示生成的 ID 始终为正数。
- delta seconds：默认 28 位。当前时间，表示为相对于某个预设时间基点 (默认 "2016-05-20") 的增量值，单位为秒。28 位最多可支持约 8.7 年。

- worker node id: 默认 22 位。表示机器 id, 通常在应用程序进程启动时从一个集中式的 ID 生成器取得。常见的集中式 ID 生成器是数据库自增列或者 Zookeeper。默认分配策略为用后即弃, 进程重启时会重新获取一个新的 worker node id, 22 位最多可支持约 420 万次启动。
- sequence: 默认 13 位。表示每秒的并发序列, 13 位可支持每秒 8192 个并发。

使用类 Snowflake 方案时需要注意几个问题:

- delta seconds 完全本地生成, 强依赖机器时钟。如果发生时钟回拨, 会导致发号重复或者服务会处于不可用状态。
- 可根据数据预期寿命调整 delta seconds 位数, 一般在 28 位至 44 位之间。
- delta seconds 时间基点不要使用默认值, 应该尽量贴近当前时间。
- worker node id 位数有限, 对应数值不超过 500 万。如果使用 TiDB 的自增列实现 worker node id, 每次 TiDB 实例的重启都会让自增列返回值增加至少 3 万, 这样最多 $500 / 3 = 166$ 次实例重启后, 自增列返回值就比 worker node id 可接受的最大值要大。这时就不能直接使用这个过大的值, 需要清空自增列所在的表, 把自增列值重置为零, 也可以在 Snowflake 实现层解决这个问题。

号段分配方案

号段分配方案可以理解为从数据库批量获取自增 ID。本方案需要一张序列号生成表, 每行记录表示一个序列对象。表定义示例如下:

字段名	字段类型	字段说明
-----	------	------

SEQ_NAME	varchar(128)	序列名称，用来区分不同业务
MAX_ID	bigint(20)	当前序列已被分配出去的最大值
STEP	int(11)	步长，表示每次分配的号段长度

应用程序每次按配置好的步长获取一段序列号，并同时更新数据库以持久化保存当前序列已被分配出去的最大值，然后在应用程序内存中即可完成序列号加工及分配动作。待一段号码耗尽之后，应用程序才会去获取新的号段，这样就有效降低了数据库写入压力。实际使用过程中，还可以适度调节步长以控制数据库记录的更新频度。

最后，需要注意的是，上述两种方案生成的 ID 都不够随机，不适合直接作为 TiDB 表的主键。实际使用过程中可以对生成的 ID 进行位反转（bit-reverse）后得到一个较为随机的新 ID。

流程规范

流程规范主要针对线上客户生产集群的调试，临时统计，上线，变更等操作流程规范定义，避免测试或者其它误操作情况造成集群性系列影响。

- 所有的建表操作需要提前告知 DBA 该表涉及的查询 SQL；
- 所有的建表需要确定建立哪些索引后才可以建表上线；

- 所有的改表结构、加索引操作都需要将涉及到所改表的查询 SQL 发出来告知 DBA 等相关人员；
- 在建新表加字段之前，建议开发人员提前发出给 dba 们评估、优化和审核；
- 批量导入、导出数据必须提前通知 DBA 协助观察；
- 大批量统计更新，如临时统计，避开高峰期，并通知 DBA；
- 推广活动或上线新功能必须提前通知 DBA 进行流量评估；
- 及时处理已下线业务的 SQL。