平凯数据库开发指南

v7.1.8

平凯星辰

20250214

目录

1 法律声明

平凯星辰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. 您应当通过平凯星辰网站或本公司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。本文档的内容视为平凯星辰的保密信息，您应当严格遵守保密义务；未经平凯星辰事先书面同意，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. 未经平凯星辰事先书面许可，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。平凯星辰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，并在平凯星辰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平凯星辰网站或平凯星辰授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用平凯星辰产品及服务的参考性指引，平凯星辰以产品及服务的“现状”、“有缺陷”和“当前功能” 的状态提供本文档。平凯星辰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，但平凯星辰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，平凯星辰不承担任何法律责任。在任何情况下，平凯星辰均不对任何间接性、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（即使平凯星辰已被告知 该等损失的可能性）。

5. 本文档中及平凯星辰网站上所有内容，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，均由平凯星辰和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经平凯星辰和/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、发行或公开发表平凯星辰网站、产品程序或包括本文档在内的内容。此外，未经平凯星辰事先书面同意，任何人不得以任何目的使用平凯星辰商标（包括但不限于单独为或以组合形式包含“平凯星辰” 等平凯星辰和/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平凯星辰和/或其关联公司）。

6. 平凯星辰可能拥有涵盖本文档中描述的主题的专利或者专利申请，未经平凯星辰事先书面许可，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. 本文档中如有关于平凯星辰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，请与平凯星辰取得直接联系。平凯星辰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。

2 开发者手册概览

本文是为应用程序开发者所编写的，如果你对平凯数据库的内部原理感兴趣，或希望参与到平凯数据库的开发中来，那么可前往阅读 TiDB Kernel Development Guide 来获取更多平凯数据库的相关信息。

本手册将展示如何使用平凯数据库来快速构建一个应用，并且阐述使用平凯数据库期间可能出现的场景以及可能会遇到的问题。因此，在阅读此页面之前，建议你先行阅读在本地快速部署平凯数据库测试集群。

此外，你还可以通过视频的形式学习免费的平凯数据库 SQL 开发在线课程。

2.1 平凯数据库基础

在你开始使用平凯数据库之前，你需要了解一些关于平凯数据库数据库的一些重要工作机制：

- 阅读平凯数据库事务概览来了解平凯数据库的事务运作方式，或查看为应用开发程序员准备的事务说明查看应用开发程序员需要了解的事务部分。
- 学习免费在线课程平凯数据库架构与特点，了解构建平凯数据库分布式数据库集群的核心组件及其概念。
- 了解应用程序与平凯数据库交互的方式。

2.2 平凯数据库事务机制

平凯数据库支持分布式事务，而且提供乐观事务与悲观事务两种事务模式。平凯数据库当前版本中默认采用 **悲观事务** 模式，这让你在平凯数据库事务时可以像使用传统的单体数据库 (如: MySQL) 事务一样。

你可以使用 BEGIN 开启一个事务，或者使用 BEGIN PESSIMISTIC 显式的指定开启一个**悲观事务**，使用 BEGIN OPTIMISTIC 显式的指定开启一个**乐观事务**。随后，使用 COMMIT 提交事务，或使用 ROLLBACK 回滚事务。

TiDB 会为你保证 BEGIN 开始到 COMMIT 或 ROLLBACK 结束间的所有语句的原子性，即在这期间的所有语句全部成功，或者全部失败。用以保证你在应用开发时所需的数据一致性。

若你不清楚**乐观事务**是什么，请暂时不要使用它。因为使用**乐观事务**的前提是需要应用程序可以正确的处理 COMMIT 语句所返回的所有错误。如果不确定应用程序如何处理，请直接使用**悲观事务**。

2.3 应用程序与平凯数据库交互的方式

平凯数据库高度兼容 MySQL 协议，平凯数据库支持大多数 MySQL 的语法及特性，因此大部分的 MySQL 的连接库都与平凯数据库兼容。如果你的应用程序框架或语言无 PingCAP 的官方适配，那么建议你使用 MySQL 的客户端库。同时，也有越来越多的三方数据库主动支持平凯数据库的差异特性。

因为平凯数据库兼容 MySQL 协议，且兼容 MySQL 语法，因此大多数支持 MySQL 的 ORM 也兼容平凯数据库。

2.4 扩展阅读

- 快速开始
- 选择驱动或 ORM 框架
- 连接到平凯数据库

- 数据库模式设计
- 数据写入
- 数据读取
- 事务
- 优化 SQL 性能
- 示例程序

3 快速开始

3.1 在本地快速部署平凯数据库测试集群

本指南介绍如何快速上手体验平凯数据库。对于非生产环境，你可以选择以下任意一种方式部署平凯数据库：

- 部署本地测试集群（支持 macOS 和 Linux）
- 在单机上模拟部署生产环境集群（支持 Linux）

注意：

本指南中的平凯数据库部署方式仅适用于快速上手体验，不适用于生产环境。

3.1.1 部署本地测试集群

- 适用场景：利用本地 macOS 或者单机 Linux 环境快速部署平凯数据库测试集群，体验平凯数据库集群的基本架构，以及 TiDB、TiKV、PD、监控等基础组件的运行。

平凯数据库是一个分布式系统。最基础的平凯数据库测试集群通常由 2 个 TiDB 实例、3 个 TiKV 实例、3 个 PD 实例和可选的 TiFlash 实例构成。通过 TiUP Playground，可以快速搭建出上述的一套基础测试集群，步骤如下：

1. 下载并安装 TiUP。

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

安装完成后会提示如下信息：

```
Successfully set mirror to https://tiup-mirrors.pingcap.com
Detected shell: zsh
Shell profile:  /Users/user/.zshrc
/Users/user/.zshrc has been modified to add tiup to PATH
open a new terminal or source /Users/user/.zshrc to use it
Installed path: /Users/user/.tiup/bin/tiup
===============================================
Have a try:     tiup playground
===============================================
```

2. 声明全局环境变量。

   注意：

   TiUP 安装完成后会提示 Shell profile 文件的绝对路径。在执行以下 source 命令前，需要将 ${your_shell_profile} 修改为 Shell profile 文件的实际位置。

   ```
   source ${your_shell_profile}
   ```

3. 在当前 session 执行以下命令启动集群。

   注意：

   - 如果按以下方式执行 playground，在结束部署测试后，TiUP 会自动清理掉原集群数据，重新执行命令会得到一个全新的集群。
   - 如果希望持久化数据，需要在启动集群时添加 TiUP 的 --tag 参数，详见启动集群时指定 tag 以保留数据。

     ```
     tiup playground --tag ${tag_name}
     ```

   - 直接执行 tiup playground 命令会运行最新版本的 TiDB 集群，其中 TiDB、TiKV、PD 和 TiFlash 实例各 1 个：

     ```
     tiup playground
     ```

- 也可以指定 TiDB 版本以及各组件实例个数，命令类似于：

  tiup playground v7.1.8-5.1 --db 2 --pd 3 --kv 3

  上述命令会在本地下载并启动某个版本的集群（例如 v7.1.8-5.1）。最新版本可以通过执行 tiup list tidb 来查看。运行结果将显示集群的访问方式：

  ```
  CLUSTER START SUCCESSFULLY, Enjoy it ^-^
  To connect TiDB: mysql --comments --host 127.0.0.1 --port 4001 -u root -p (no password)
  To connect TiDB: mysql --comments --host 127.0.0.1 --port 4000 -u root -p (no password)
  To view the dashboard: http://127.0.0.1:2379/dashboard
  PD client endpoints: [127.0.0.1:2379 127.0.0.1:2382 127.0.0.1:2384]
  To view the Prometheus: http://127.0.0.1:9090
  To view the Grafana: http://127.0.0.1:3000
  ```

  注意：

  v5.2.0 及以上版本的 TiDB 支持在 Apple M1 芯片的机器上运行 tiup playground。

4. 新开启一个 session 以访问 TiDB 数据库。

   - 使用 TiUP client 连接 TiDB：

     tiup client

   - 也可使用 MySQL 客户端连接 TiDB：

     mysql --host 127.0.0.1 --port 4000 -u root

5. 通过 http://127.0.0.1:9090 访问 TiDB 的 Prometheus 管理界面。

6. 通过 http://127.0.0.1:2379/dashboard 访问 TiDB Dashboard 页面，默认用户名为 root，密码为空。

7. 通过 http://127.0.0.1:3000 访问 TiDB 的 Grafana 界面，默认用户名和密码都为 admin。

8. （可选）将数据加载到 TiFlash 进行分析。

9. 测试完成之后，可以通过执行以下步骤来清理集群：

    1. 按下 Control+C 键停掉上述启用的 TiDB 服务。

    2. 等待服务退出操作完成后，执行以下命令：

        tiup clean --all

注意：

TiUP Playground 默认监听 127.0.0.1，服务仅本地可访问；若需要使服务可被外部访问，可使用 --host 参数指定监听网卡绑定外部可访问的 IP。

TiDB 是一个分布式系统。最基础的 TiDB 测试集群通常由 2 个 TiDB 实例、3 个 TiKV 实例、3 个 PD 实例和可选的 TiFlash 实例构成。通过 TiUP Playground，可以快速搭建出上述的一套基础测试集群，步骤如下：

1. 下载并安装 TiUP。

    curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh

    安装完成后会提示如下信息：

    ```
    Successfully set mirror to https://tiup-mirrors.pingcap.com
    Detected shell: bash
    Shell profile:  /home/user/.bashrc
    /home/user/.bashrc has been modified to add tiup to PATH
    open a new terminal or source /home/user/.bashrc to use it
    Installed path: /home/user/.tiup/bin/tiup
    ===============================================
    Have a try:     tiup playground
    ===============================================
    ```

2. 声明全局环境变量。

    注意：

TiUP 安装完成后会提示 Shell profile 文件的绝对路径。在执行以下 source 命令前，需要将 ${your_shell_profile} 修改为 Shell profile 文件的实际位置。

```
source ${your_shell_profile}
```

3. 在当前 session 执行以下命令启动集群。

   注意：

   - 如果按以下方式执行 playground，在结束部署测试后，TiUP 会自动清理掉原集群数据，重新执行命令会得到一个全新的集群。
   - 如果希望持久化数据，需要在启动集群时添加 TiUP 的 --tag 参数，详见启动集群时指定 tag 以保留数据。

     ```
     tiup playground --tag ${tag_name}
     ```

   - 直接运行 tiup playground 命令会运行最新版本的 TiDB 集群，其中 TiDB、TiKV、PD 和 TiFlash 实例各 1 个：

     ```
     tiup playground
     ```

   - 也可以指定 TiDB 版本以及各组件实例个数，命令类似于：

     ```
     tiup playground v7.1.8-5.1 --db 2 --pd 3 --kv 3
     ```

     上述命令会在本地下载并启动某个版本的集群（例如 v7.1.8-5.1）。最新版本可以通过执行 tiup list tidb 来查看。运行结果将显示集群的访问方式：

     ```
     CLUSTER START SUCCESSFULLY, Enjoy it ^-^
     To connect TiDB: mysql --host 127.0.0.1 --port 4000 -u root -p (no passwo
     rd) --comments
     To view the dashboard: http://127.0.0.1:2379/dashboard
     PD client endpoints: [127.0.0.1:2379]
     To view the Prometheus: http://127.0.0.1:9090
     To view the Grafana: http://127.0.0.1:3000
     ```

4. 新开启一个 session 以访问平凯数据库数据库。

- 使用 TiUP client 连接平凯数据库：

  tiup client

- 也可使用 MySQL 客户端连接 TiDB：

  mysql --host 127.0.0.1 --port 4000 -u root

5. 通过 http://127.0.0.1:9090 访问 TiDB 的 Prometheus 管理界面。
6. 通过 http://127.0.0.1:2379/dashboard 访问 TiDB Dashboard 页面，默认用户名为 root，密码为空。
7. 通过 http://127.0.0.1:3000 访问 TiDB 的 Grafana 界面，默认用户名和密码都为 admin。
8. （可选）将数据加载到 TiFlash 进行分析。
9. 测试完成之后，可以通过执行以下步骤来清理集群：
   1. 按下 Control+C 键停掉上述启用的 TiDB 服务。
   2. 等待服务退出操作完成后，执行以下命令：

   tiup clean --all

注意：

TiUP Playground 默认监听 127.0.0.1，服务仅本地可访问。若需要使服务可被外部访问，可使用 --host 参数指定监听网卡绑定外部可访问的 IP。

3.1.2 在单机上模拟部署生产环境集群

- 适用场景：希望用单台 Linux 服务器，体验平凯数据库最小完整拓扑的集群，并模拟生产环境下的部署步骤。

本节介绍如何参照 TiUP 最小拓扑的一个 YAML 文件部署平凯数据库集群。

3.1.2.1 准备环境

开始部署平凯数据库集群前，准备一台部署主机，确保其软件满足需求：

- 推荐安装 CentOS 7.3 及以上版本
- 运行环境可以支持互联网访问，用于下载平凯数据库及相关软件安装包

最小规模的平凯数据库集群拓扑包含以下实例：

注意：

下表中拓扑实例的 IP 为示例 IP。在实际部署时，请替换为实际的 IP。

实例	个数	IP	配置
TiKV	3	10.0.1.1 10.0.1.1 10.0.1.1	避免端口和目录冲突
TiDB	1	10.0.1.1	默认端口 全局目录配置
PD	1	10.0.1.1	默认端口 全局目录配置
TiFlash	1	10.0.1.1	默认端口 全局目录配置
Monitor	1	10.0.1.1	默认端口 全局目录配置

部署主机软件和环境要求如下：

- 部署需要使用部署主机的 root 用户及密码
- 部署主机关闭防火墙或者开放 TiDB 集群的节点间所需端口
- 目前 TiUP Cluster 支持在 x86_64（AMD64）和 ARM 架构上部署平凯数据库集群
    - 在 AMD64 架构下，建议使用 CentOS 7.3 及以上版本 Linux 操作系统
    - 在 ARM 架构下，建议使用 CentOS 7.6 1810 版本 Linux 操作系统

3.1.2.2 实施部署

注意：

你可以使用 Linux 系统的任一普通用户或 root 用户登录主机，以下步骤以 root 用户为例。

1. 下载并安装 TiUP：

   ```
   curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
   ```

2. 声明全局环境变量：

   注意：

   TiUP 安装完成后会提示对应 Shell profile 文件的绝对路径。在执行以下 source 命令前，需要将 ${your_shell_profile} 修改为 Shell profile 文件的实际位置。

   ```
   source ${your_shell_profile}
   ```

3. 安装 TiUP 的 cluster 组件：

   ```
   tiup cluster
   ```

4. 如果机器已经安装 TiUP cluster，需要更新软件版本：

   ```
   tiup update --self && tiup update cluster
   ```

5. 由于模拟多机部署，需要通过 root 用户调大 sshd 服务的连接数限制：

   1. 修改 /etc/ssh/sshd_config 将 MaxSessions 调至 20。

   2. 重启 sshd 服务：

      ```
      service sshd restart
      ```

6. 创建并启动集群

   按下面的配置模板，编辑配置文件，命名为 topo.yaml，其中：

- user: "tidb"：表示通过 tidb 系统用户（部署会自动创建）来做集群的内部管理，默认使用 22 端口通过 ssh 登录目标机器
- replication.enable-placement-rules：设置这个 PD 参数来确保 TiFlash 正常运行
- host：设置为本部署主机的 IP

配置模板如下：

```
# # Global variables are applied to all deployments and used as the default value of
# # the deployments if a specific deployment value is missing.
global:
 user: "tidb"
 ssh_port: 22
 deploy_dir: "/tidb-deploy"
 data_dir: "/tidb-data"

# # Monitored variables are applied to all the machines.
monitored:
 node_exporter_port: 9100
 blackbox_exporter_port: 9115

server_configs:
 tidb:
   instance.tidb_slow_log_threshold: 300
 tikv:
   readpool.storage.use-unified-pool: false
   readpool.coprocessor.use-unified-pool: true
 pd:
   replication.enable-placement-rules: true
   replication.location-labels: ["host"]
 tiflash:
   logger.level: "info"

pd_servers:
 - host: 10.0.1.1

tidb_servers:
 - host: 10.0.1.1

tikv_servers:
```

```
  - host: 10.0.1.1
    port: 20160
    status_port: 20180
    config:
      server.labels: { host: "logic-host-1" }

  - host: 10.0.1.1
    port: 20161
    status_port: 20181
    config:
      server.labels: { host: "logic-host-2" }

  - host: 10.0.1.1
    port: 20162
    status_port: 20182
    config:
      server.labels: { host: "logic-host-3" }

tiflash_servers:
  - host: 10.0.1.1

monitoring_servers:
  - host: 10.0.1.1

grafana_servers:
  - host: 10.0.1.1
```

7. 执行集群部署命令：

   tiup cluster deploy <cluster-name> <version> ./topo.yaml --user root -p

   - 参数 <cluster-name> 表示设置集群名称
   - 参数 <version> 表示设置集群版本，例如 v7.1.8-5.1。可以通过 tiup list tidb 命令来查看当前支持部署的 TiDB 版本
   - 参数 -p 表示在连接目标机器时使用密码登录

     注意：

如果主机通过密钥进行 SSH 认证，请使用 -i 参数指定密钥文件路径，-i 与 -p 不可同时使用。

按照引导，输入”y”及 root 密码，来完成部署：

```
Do you want to continue? [y/N]:  y
Input SSH password:
```

8. 启动集群：

```
tiup cluster start <cluster-name>
```

9. 访问集群：

   - 安装 MySQL 客户端。如果已安装 MySQL 客户端则可跳过这一步骤。

     ```
     yum -y install mysql
     ```

   - 访问 TiDB 数据库，密码为空：

     ```
     mysql -h 10.0.1.1 -P 4000 -u root
     ```

   - 访问 TiDB 的 Grafana 监控：

     通过 http://{grafana-ip}:3000 访问集群 Grafana 监控页面，默认用户名和密码均为 admin。

   - 访问 TiDB 的 Dashboard：

     通过 http://{pd-ip}:2379/dashboard 访问集群 TiDB Dashboard 监控页面，默认用户名为 root，密码为空。

   - 执行以下命令确认当前已经部署的集群列表：

     ```
     tiup cluster list
     ```

   - 执行以下命令查看集群的拓扑结构和状态：

     ```
     tiup cluster display <cluster-name>
     ```

3.2 使用平凯数据库的增删改查 SQL

本章将简单介绍平凯数据库的增删改查 SQL 的使用方法。

3.2.1 在开始之前

请确保你已经连接到平凯数据库集群，若未连接，请参考在本地快速部署平凯数据库测试集群来创建一个平凯数据库集群。

3.2.2 基本 SQL 操作

> **注意：**
>
> 此处文档引用并简化自平凯数据库文档中的 SQL 基本操作，你可直接前往此文档获取更全面、深入的 SQL 基本操作信息。

成功部署平凯数据库集群之后，便可以在平凯数据库中执行 SQL 语句了。因为平凯数据库兼容 MySQL，你可以使用 MySQL 客户端连接 TiDB，并且大多数情况下可以直接执行 MySQL 语句。

SQL 是一门声明性语言，它是数据库用户与数据库交互的方式。它更像是一种自然语言，好像在用英语与数据库进行对话。本文档介绍基本的 SQL 操作。完整的 SQL 语句列表，参见 SQL 语句概览。

3.2.3 分类

SQL 语言通常按照功能划分成以下的 4 个部分：

- **DDL (Data Definition Language)**：数据定义语言，用来定义数据库对象，包括库、表、视图和索引等。
- **DML (Data Manipulation Language)**：数据操作语言，用来操作和业务相关的记录。
- **DQL (Data Query Language)**：数据查询语言，用来查询经过条件筛选的记录。

- **DCL (Data Control Language)**：数据控制语言，用来定义访问权限和安全级别。

此文档中，主要介绍 DML 和 DQL，即数据操作语言和数据查询语言。其余部分可查看 SQL 基本操作或 SQL 语句概览获得更多信息。

3.2.4 DML 数据操作语言

数据操作语言可完成数据的增删改。

使用 INSERT 语句向表内插入表记录。例如：

```
INSERT INTO person VALUES(1,'tom','20170912');
```

使用 INSERT 语句向表内插入包含部分字段数据的表记录。例如：

```
INSERT INTO person(id,name) VALUES('2','bob');
```

使用 UPDATE 语句向表内修改表记录的部分字段数据。例如：

```
UPDATE person SET birthday='20180808' WHERE id=2;
```

使用 DELETE 语句向表内删除部分表记录。例如：

```
DELETE FROM person WHERE id=2;
```

> **注意：**
>
> UPDATE 和 DELETE 操作如果不带 WHERE 过滤条件是对全表进行操作。

3.2.5 DQL 数据查询语言

数据查询语言是从一个表或多个表中检索出想要的数据行，通常是业务开发的核心内容。

使用 SELECT 语句检索单表内数据。例如：

```
SELECT * FROM person;
```

在 SELECT 后面加上要查询的列名。例如：

```
SELECT name FROM person;
```

运行结果为：

```
+------+
| name |
+------+
| tom  |
+------+
1 rows in set (0.00 sec)
```

使用 WHERE 子句，对所有记录进行是否符合条件的筛选后再返回。例如：

```
SELECT * FROM person WHERE id < 5;
```

4 示例程序

4.1 Java

4.1.1 使用 JDBC 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。JDBC 是 Java 的数据访问 API。MySQL Connector/J 是 MySQL 对 JDBC 的实现。

本文档将展示如何使用平凯数据库和 JDBC 来完成以下任务：

- 配置你的环境。
- 使用 JDBC 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.1.1.1 前置需求

- 推荐 **Java Development Kit** (JDK) **17** 及以上版本。你可以根据公司及个人需求，自行选择 OpenJDK 或 Oracle JDK。
- Maven **3.8** 及以上版本。
- Git。
- 平凯数据库集群。

如果你还没有平凯数据库集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署平凯数据库正式生产集群，创建一个本地集群

4.1.1.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到平凯数据库。

4.1.1.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-java-jdbc-quickstart.git
cd tidb-java-jdbc-quickstart
```

4.1.1.2.2 第 2 步：配置连接信息

1. 运行以下命令，将 env.sh.example 复制并重命名为 env.sh：

   ```
   cp env.sh.example env.sh
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 env.sh 中。需更改部分示例结果如下：

   ```
   export TIDB_HOST='{host}'
   export TIDB_PORT='4000'
   export TIDB_USER='root'
   export TIDB_PASSWORD='{password}'
   export TIDB_DB_NAME='test'
   export USE_SSL='false'
   ```

   注意替换 {} 中的占位符为你的 TiDB 对应的值，并设置 USE_SSL 为 false。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 env.sh 文件。

4.1.1.2.3 第 3 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

   ```
   make
   ```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.1.1.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-java-jdbc-quickstart。

4.1.1.3.1 连接到平凯数据库

```
public MysqlDataSource getMysqlDataSource() throws SQLException {
    MysqlDataSource mysqlDataSource = new MysqlDataSource();

    mysqlDataSource.setServerName(${tidb_host});
    mysqlDataSource.setPortNumber(${tidb_port});
    mysqlDataSource.setUser(${tidb_user});
    mysqlDataSource.setPassword(${tidb_password});
    mysqlDataSource.setDatabaseName(${tidb_db_name});
    if (${tidb_use_ssl}) {
        mysqlDataSource.setSslMode(PropertyDefinitions.SslMode.VERIFY_IDENTITY.name
());
        mysqlDataSource.setEnabledTLSProtocols("TLSv1.2,TLSv1.3");
    }

    return mysqlDataSource;
}
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name} 等替换为你的 TiDB 集群的实际值。

4.1.1.3.2 插入数据

```
public void createPlayer(PlayerBean player) throws SQLException {
    MysqlDataSource mysqlDataSource = getMysqlDataSource();
    try (Connection connection = mysqlDataSource.getConnection()) {
        PreparedStatement preparedStatement = connection.prepareStatement("INSERT IN
TO player (id, coins, goods) VALUES (?, ?, ?)");
        preparedStatement.setString(1, player.getId());
        preparedStatement.setInt(2, player.getCoins());
        preparedStatement.setInt(3, player.getGoods());

        preparedStatement.execute();
```

```
    }
}
```

更多信息参考插入数据。

4.1.1.3.3 查询数据

```
public void getPlayer(String id) throws SQLException {
    MysqlDataSource mysqlDataSource = getMysqlDataSourceByEnv();
    try (Connection connection = mysqlDataSource.getConnection()) {
        PreparedStatement preparedStatement = connection.prepareStatement("SELECT * F
ROM player WHERE id = ?");
        preparedStatement.setString(1, id);
        preparedStatement.execute();

        ResultSet res = preparedStatement.executeQuery();
        if(res.next()) {
            PlayerBean player = new PlayerBean(res.getString("id"), res.getInt("coins"), res.get
Int("goods"));
            System.out.println(player);
        }
    }
}
```

更多信息参考查询数据。

4.1.1.3.4 更新数据

```
public void updatePlayer(String id, int amount, int price) throws SQLException {
    MysqlDataSource mysqlDataSource = getMysqlDataSourceByEnv();
    try (Connection connection = mysqlDataSource.getConnection()) {
        PreparedStatement transfer = connection.prepareStatement("UPDATE player SET go
ods = goods + ?, coins = coins + ? WHERE id=?");
        transfer.setInt(1, -amount);
        transfer.setInt(2, price);
        transfer.setString(3, id);
        transfer.execute();
    }
}
```

更多信息参考更新数据。

4.1.1.3.5 删除数据

```
public void deletePlayer(String id) throws SQLException {
    MysqlDataSource mysqlDataSource = getMysqlDataSourceByEnv();
    try (Connection connection = mysqlDataSource.getConnection()) {
        PreparedStatement deleteStatement = connection.prepareStatement("DELETE FRO
M player WHERE id=?");
        deleteStatement.setString(1, id);
        deleteStatement.execute();
    }
}
```

更多信息参考删除数据。

4.1.1.4 注意事项

4.1.1.4.1 使用驱动程序还是 ORM 框架?

Java 驱动程序提供对数据库的底层访问，但要求开发者：

- 手动建立和释放数据库连接
- 手动管理数据库事务
- 手动将数据行映射为数据对象

建议仅在需要编写复杂的 SQL 语句时使用驱动程序。其他情况下，建议使用 ORM 框架进行开发，例如 Hibernate、MyBatis 或 Spring Data JPA。ORM 可以帮助你：

- 减少管理连接和事务的模板代码
- 使用数据对象代替大量 SQL 语句来操作数据

4.1.1.5 下一步

- 关于 MySQL Connector/J 的更多使用方法，可以参考 MySQL Connector/J 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。

- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。
- 我们还额外提供针对 Java 开发者的课程：使用 Connector/J - TiDB v6 及在 TiDB 上开发应用的最佳实践 - TiDB v6。

4.1.1.6 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.1.2 使用 MyBatis 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。MyBatis 是当前比较流行的开源 Java 应用持久层框架。

本文档将展示如何使用平凯数据库和 MyBatis 来完成以下任务：

- 配置你的环境。
- 使用 MyBatis 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.1.2.1 前置需求

- 推荐 **Java Development Kit** (JDK) **17** 及以上版本。你可以根据公司及个人需求，自行选择 OpenJDK 或 Oracle JDK。
- Maven **3.8** 及以上版本。
- Git。
- 平凯数据库集群。

如果你还没有平凯数据库集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署平凯数据库正式生产集群，创建一个本地集群

4.1.2.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到平凯数据库。

4.1.2.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-java-mybatis-quickstart.git
cd tidb-java-mybatis-quickstart
```

4.1.2.2.2 第 2 步：配置连接信息

1. 运行以下命令，将 env.sh.example 复制并重命名为 env.sh：

   ```
   cp env.sh.example env.sh
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 env.sh 中。需更改部分示例结果如下：

   ```
   export TIDB_HOST='{host}'
   export TIDB_PORT='4000'
   export TIDB_USER='root'  # e.g. xxxxxx.root
   export TIDB_PASSWORD='{password}'
   export TIDB_DB_NAME='test'
   export USE_SSL='false'
   ```

   注意替换 {} 中的占位符为你的 TiDB 对应的值，并设置 USE_SSL 为 false。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 env.sh 文件。

4.1.2.2.3 第 3 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

   ```
   make
   ```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.1.2.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-java-mybatis-quickstart。

4.1.2.3.1 连接到平凯数据库

编写配置文件 mybatis-config.xml：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
      PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
      "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <settings>
    <setting name="cacheEnabled" value="true"/>
    <setting name="lazyLoadingEnabled" value="false"/>
    <setting name="aggressiveLazyLoading" value="true"/>
    <setting name="logImpl" value="LOG4J"/>
  </settings>

  <environments default="development">
    <environment id="development">
      <!-- JDBC transaction manager -->
      <transactionManager type="JDBC"/>
      <!-- Database pool -->
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.cj.jdbc.Driver"/>
        <property name="url" value="${TIDB_JDBC_URL}"/>
        <property name="username" value="${TIDB_USER}"/>
        <property name="password" value="${TIDB_PASSWORD}"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="${MAPPER_LOCATION}.xml"/>
  </mappers>
</configuration>
```

请将 ${TIDB_JDBC_URL}、${TIDB_USER}、${TIDB_PASSWORD} 等替换为你的 TiDB 集群的实际值。并替换 ${MAPPER_LOCATION} 的值为你的 mapper XML 配置文件的

位置。如果你有多个 mapper XML 配置文件，需要添加多个 <mapper/> 标签。随后编写以下函数：

```
public SqlSessionFactory getSessionFactory() {
   InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");
   SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStrea
m);
}
```

4.1.2.3.2 插入数据

在 mapper XML 中添加节点，并在 XML 配置文件的 mapper.namespace 属性中配置的接口类中添加同名函数：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/
dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.pingcap.model.PlayerMapper">
   <insert id="insert" parameterType="com.pingcap.model.Player">
   INSERT INTO player (id, coins, goods)
   VALUES (#{id, jdbcType=VARCHAR}, #{coins, jdbcType=INTEGER}, #{goods, jdbcType=
INTEGER})
   </insert>
</mapper>
```

更多信息参考插入数据。

4.1.2.3.3 查询数据

在 mapper XML 中添加节点，并在 XML 配置文件的 mapper.namespace 属性中配置的接口类中添加同名函数。特别地，如果你在 MyBatis 的查询函数中使用 resultMap 作为返回类型，需要额外注意配置文件的 <resultMap/> 节点配置是否正确。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/
dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.pingcap.model.PlayerMapper">
   <resultMap id="BaseResultMap" type="com.pingcap.model.Player">
      <constructor>
         <idArg column="id" javaType="java.lang.String" jdbcType="VARCHAR" />
```

```
        <arg column="coins" javaType="java.lang.Integer" jdbcType="INTEGER" />
        <arg column="goods" javaType="java.lang.Integer" jdbcType="INTEGER" />
      </constructor>
    </resultMap>

    <select id="selectByPrimaryKey" parameterType="java.lang.String" resultMap="BaseR
esultMap">
    SELECT id, coins, goods
    FROM player
    WHERE id = #{id, jdbcType=VARCHAR}
    </select>
</mapper>
```

更多信息参考查询数据。

4.1.2.3.4 更新数据

在 mapper XML 中添加节点，并在 XML 配置文件的 mapper.namespace 属性中配置的接口类中添加同名函数：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/
dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.pingcap.model.PlayerMapper">
    <update id="updateByPrimaryKey" parameterType="com.pingcap.model.Player">
    UPDATE player
    SET coins = #{coins, jdbcType=INTEGER},
      goods = #{goods, jdbcType=INTEGER}
    WHERE id = #{id, jdbcType=VARCHAR}
    </update>
</mapper>
```

更多信息参考更新数据。

4.1.2.3.5 删除数据

在 mapper XML 中添加节点，并在 XML 配置文件的 mapper.namespace 属性中配置的接口类中添加同名函数：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/
dtd/mybatis-3-mapper.dtd">
```

```
<mapper namespace="com.pingcap.model.PlayerMapper">
  <delete id="deleteByPrimaryKey" parameterType="java.lang.String">
  DELETE FROM player
  WHERE id = #{id, jdbcType=VARCHAR}
  </delete>
</mapper>
```

更多信息参考删除数据。

4.1.2.4 下一步

- 关于 MyBatis 的更多使用方法，可以参考 MyBatis 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。
- 我们还额外提供针对 Java 开发者的课程：使用 Connector/J - TiDB v6 及在 TiDB 上开发应用的最佳实践 - TiDB v6。

4.1.2.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.1.3 使用 Hibernate 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。Hibernate 是当前比较流行的开源 Java 应用持久层框架，且 Hibernate 在版本 6.0.0.Beta2 及以上支持了平凯数据库方言，完美适配了平凯数据库的特性。

本文档将展示如何使用平凯数据库和 Hibernate 来完成以下任务：

- 配置你的环境。
- 使用 Hibernate 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.1.3.1 前置需求

- 推荐 **Java Development Kit** (JDK) **17** 及以上版本。你可以根据公司及个人需求，自行选择 OpenJDK 或 Oracle JDK。
- Maven **3.8** 及以上版本。
- Git。
- 平凯数据库集群。

如果你还没有平凯数据库集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署平凯数据库正式生产集群，创建一个本地集群

4.1.3.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到平凯数据库。

4.1.3.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-java-hibernate-quickstart.git
cd tidb-java-hibernate-quickstart
```

4.1.3.2.2 第 2 步：配置连接信息

1. 运行以下命令，将 env.sh.example 复制并重命名为 env.sh：

   ```
   cp env.sh.example env.sh
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 env.sh 中。需更改部分示例结果如下：

   ```
   export TIDB_HOST='{host}'
   export TIDB_PORT='4000'
   export TIDB_USER='root'
   export TIDB_PASSWORD='{password}'
   export TIDB_DB_NAME='test'
   export USE_SSL='false'
   ```

注意替换 {} 中的占位符为你的 TiDB 对应的值，并设置 USE_SSL 为 false。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 env.sh 文件。

4.1.3.2.3 第 3 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

   make

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.1.3.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-java-hibernate-quickstart。

4.1.3.3.1 连接到平凯数据库

编写配置文件 hibernate.cfg.xml：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>

      <!-- Database connection settings -->
      <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
      <property name="hibernate.dialect">org.hibernate.dialect.TiDBDialect</property>
      <property name="hibernate.connection.url">${tidb_jdbc_url}</property>
      <property name="hibernate.connection.username">${tidb_user}</property>
      <property name="hibernate.connection.password">${tidb_password}</property>
      <property name="hibernate.connection.autocommit">false</property>
```

```
        <!-- Required so a table can be created from the 'PlayerDAO' class -->
        <property name="hibernate.hbm2ddl.auto">create-drop</property>

        <!-- Optional: Show SQL output for debugging -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

请将 ${tidb_jdbc_url}、${tidb_user}、${tidb_password} 等替换为你的 TiDB 集群的实际值。随后编写以下函数：

```
public SessionFactory getSessionFactory() {
    return new Configuration()
            .configure("hibernate.cfg.xml")
            .addAnnotatedClass(${your_entity_class})
            .buildSessionFactory();
}
```

在使用该函数时，你需要替换 ${your_entity_class} 为自己的数据实体类。如果你有多个实体类，需要添加多个 .addAnnotatedClass(${your_entity_class}) 语句。此外，这仅是 Hibernate 的其中一种配置方式。在配置中遇到任何问题，或想了解更多关于 Hibernate 的信息，你可参考 Hibernate 官方文档。

4.1.3.3.2 插入或更新数据

```
try (Session session = sessionFactory.openSession()) {
    session.persist(new PlayerBean("id", 1, 1));
}
```

更多信息参考插入数据和更新数据。

4.1.3.3.3 查询数据

```
try (Session session = sessionFactory.openSession()) {
    PlayerBean player = session.get(PlayerBean.class, "id");
    System.out.println(player);
}
```

更多信息参考查询数据。

4.1.3.3.4 删除数据

```
try (Session session = sessionFactory.openSession()) {
    session.remove(new PlayerBean("id", 1, 1));
}
```

更多信息参考删除数据。

4.1.3.4 下一步

- 关于 Hibernate 的更多使用方法，可以参考 Hibernate 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。
- 我们还额外提供针对 Java 开发者的课程：使用 Connector/J - TiDB v6 及在 TiDB 上开发应用的最佳实践 - TiDB v6。

4.1.3.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.1.4 使用 Spring Boot 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。Spring 是当前比较流行的开源 Java 容器框架，本文选择 Spring Boot 作为使用 Spring 的方式。

本文档将展示如何使用平凯数据库和 Spring Data JPA 及 Hibernate 作为 JPA 提供者来完成以下任务：

- 配置你的环境。
- 使用 Spring Data JPA 与 Hibernate 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.1.4.1 前置需求

- 推荐 **Java Development Kit** (JDK) **17** 及以上版本。你可以根据公司及个人需求，自行选择 OpenJDK 或 Oracle JDK。
- Maven **3.8** 及以上版本。
- Git。
- 平凯数据库集群。

如果你还没有平凯数据库集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署平凯数据库正式生产集群，创建一个本地集群

4.1.4.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.1.4.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-java-springboot-jpa-quickstart.git
cd tidb-java-springboot-jpa-quickstart
```

4.1.4.2.2 第 2 步：配置连接信息

1. 运行以下命令，将 env.sh.example 复制并重命名为 env.sh：

   ```
   cp env.sh.example env.sh
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 env.sh 中。需更改部分示例结果如下：

   ```
   export TIDB_HOST='{host}'
   export TIDB_PORT='4000'
   export TIDB_USER='root'  # e.g. xxxxxx.root
   export TIDB_PASSWORD='{password}'
   export TIDB_DB_NAME='test'
   export USE_SSL='false'
   ```

注意替换 {} 中的占位符为你的 TiDB 对应的值，并设置 USE_SSL 为 false。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 env.sh 文件。

4.1.4.2.3 第 3 步：运行代码并查看结果

1. 运行下述命令，启动示例代码编写的服务：

   make

2. 打开另一个终端，开启请求脚本：

   make request

3. 查看 Expected-Output.txt，并与你的服务程序输出进行比较。结果近似即为连接成功。

4.1.4.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-java-springboot-jpa-quickstart。

4.1.4.3.1 连接到平凯数据库

编写配置文件 application.yml：

```
spring:
  datasource:
    url: ${TIDB_JDBC_URL:jdbc:mysql://localhost:4000/test}
    username: ${TIDB_USER:root}
    password: ${TIDB_PASSWORD:}
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    show-sql: true
    database-platform: org.hibernate.dialect.TiDBDialect
    hibernate:
      ddl-auto: create-drop
```

请在配置后将环境变量 TIDB_JDBC_URL、TIDB_USER 和 TIDB_PASSWORD 设置为你的 TiDB 集群的实际值。此配置文件带有环境变量默认配置，即在不配置环境变量时，变量的值为：

- TIDB_JDBC_URL: "jdbc:mysql://localhost:4000/test"
- TIDB_USER: "root"
- TIDB_PASSWORD: ""

4.1.4.3.2 数据管理

Spring Data JPA 通过 @Entity 注册数据实体，并绑定数据库的表。

```
@Entity
@Table(name = "player_jpa")
public class PlayerBean {
}
```

PlayerRepository 通过继承 JpaRepository 接口，由 JpaRepositoryFactoryBean 为其自动注册对应的 Repository Bean。同时，JpaRepository 接口的默认实现类 SimpleJpaRepository 提供了增删改查函数的具体实现。

```
@Repository
public interface PlayerRepository extends JpaRepository<PlayerBean, Long> {
}
```

随后，在需要使用 PlayerRepository 的类中，你可以通过 @Autowired 自动装配，这样就可以直接使用增删改查函数了。示例代码如下：

```
@Autowired
private PlayerRepository playerRepository;
```

4.1.4.3.3 插入或更新数据

```
playerRepository.save(player);
```

更多信息参考插入数据和更新数据。

4.1.4.3.4 查询数据

```
PlayerBean player = playerRepository.findById(id).orElse(null);
```

更多信息参考查询数据。

4.1.4.3.5 删除数据

```
playerRepository.deleteById(id);
```

更多信息参考删除数据。

4.1.4.4 下一步

- 关于本文使用到的第三方库及框架，可以参考各自官方文档：
    - Spring Framework 官方文档
    - Spring Boot 官方文档
    - Spring Data JPA 官方文档
    - Hibernate 官方文档
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。
- 我们还额外提供针对 Java 开发者的课程：使用 Connector/J - TiDB v6 及在 TiDB 上开发应用的最佳实践 - TiDB v6。

4.1.4.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.2 Go

4.2.1 使用 Go-MySQL-Driver 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。Go-MySQL-Driver 是 database/sql 接口的 MySQL 实现。

本文档将展示如何使用平凯数据库和 Go-MySQL-Driver 来完成以下任务：

- 配置你的环境。
- 使用 Go-MySQL-Driver 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.2.1.1 前置需求

- 推荐 Go **1.20** 及以上版本。
- Git。
- 平凯数据库集群。

如果你还没有平凯数据库集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署平凯数据库正式生产集群，创建一个本地集群

4.2.1.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到平凯数据库。

4.2.1.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-golang-sql-driver-quickstart.git
cd tidb-golang-sql-driver-quickstart
```

4.2.1.2.2 第 2 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 .env 中。示例结果如下：

   ```
   TIDB_HOST='{host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   TIDB_PASSWORD='{password}'
   ```

```
TIDB_DB_NAME='test'
USE_SSL='false'
```

注意替换 {} 中的占位符为你的 TiDB 对应的值，并设置 USE_SSL 为 false。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.2.1.2.3 第 3 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

```
make
```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.2.1.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-golang-sql-driver-quickstart。

4.2.1.3.1 连接到平凯数据库

```
func openDB(driverName string, runnable func(db *sql.DB)) {
    dsn := fmt.Sprintf("%s:%s@tcp(%s:%s)/%s?charset=utf8mb4&tls=%s",
        ${tidb_user}, ${tidb_password}, ${tidb_host}, ${tidb_port}, ${tidb_db_name}, ${use_ss
l})
    db, err := sql.Open(driverName, dsn)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    runnable(db)
}
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name} 等替换为你的 TiDB 集群的实际值。

4.2.1.3.2 插入数据

```
openDB("mysql", func(db *sql.DB) {
    insertSQL := "INSERT INTO player (id, coins, goods) VALUES (?, ?, ?)"
    _, err := db.Exec(insertSQL, "id", 1, 1)

    if err != nil {
        panic(err)
    }
})
```

更多信息参考插入数据。

4.2.1.3.3 查询数据

```
openDB("mysql", func(db *sql.DB) {
    selectSQL := "SELECT id, coins, goods FROM player WHERE id = ?"
    rows, err := db.Query(selectSQL, "id")
    if err != nil {
        panic(err)
    }

    // This line is extremely important!
    defer rows.Close()

    id, coins, goods := "", 0, 0
    if rows.Next() {
        err = rows.Scan(&id, &coins, &goods)
        if err == nil {
            fmt.Printf("player id: %s, coins: %d, goods: %d\n", id, coins, goods)
        }
    }
})
```

更多信息参考查询数据。

4.2.1.3.4 更新数据

```
openDB("mysql", func(db *sql.DB) {
    updateSQL := "UPDATE player set goods = goods + ?, coins = coins + ? WHERE id = ?
"
    _, err := db.Exec(updateSQL, 1, -1, "id")

    if err != nil {
        panic(err)
```

```
    }
})
```

更多信息参考更新数据。

4.2.1.3.5 删除数据

```
openDB("mysql", func(db *sql.DB) {
    deleteSQL := "DELETE FROM player WHERE id=?"
    _, err := db.Exec(deleteSQL, "id")

    if err != nil {
        panic(err)
    }
})
```

更多信息参考删除数据。

4.2.1.4 注意事项

4.2.1.4.1 使用驱动程序还是 ORM 框架?

Golang 驱动程序提供对数据库的底层访问，但要求开发者：

- 手动建立和释放数据库连接
- 手动管理数据库事务
- 手动将数据行映射为数据对象

建议仅在需要编写复杂的 SQL 语句时使用驱动程序。其他情况下，建议使用 ORM 框架进行开发，例如 GORM。ORM 可以帮助你：

- 减少管理连接和事务的模板代码
- 使用数据对象代替大量 SQL 语句来操作数据

4.2.1.5 下一步

- 关于 Go-MySQL-Driver 的更多使用方法，可以参考 Go-MySQL-Driver 官方文档。

- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.2.1.6 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.2.2 使用 GORM 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。GORM 是当前比较流行的开源 Golang ORM 框架并且适配了平凯数据库 的 AUTO_RANDOM 等特性。同时，平凯数据库为 GORM 的默认支持数据库。

本文档将展示如何使用平凯数据库和 GORM 来完成以下任务：

- 配置你的环境。
- 使用 GORM 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.2.2.1 前置需求

- 推荐 Go **1.20** 及以上版本。
- Git。
- 平凯数据库集群。

如果你还没有平凯数据库集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署平凯数据库正式生产集群，创建一个本地集群

4.2.2.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到平凯数据库。

4.2.2.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-golang-gorm-quickstart.git
cd tidb-golang-gorm-quickstart
```

4.2.2.2.2 第 2 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 .env 中。示例结果如下：

   ```
   TIDB_HOST='{host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   TIDB_PASSWORD='{password}'
   TIDB_DB_NAME='test'
   USE_SSL='false'
   ```

   注意替换 {} 中的占位符为你的 TiDB 对应的值，并设置 USE_SSL 为 false。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.2.2.2.3 第 3 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

   ```
   make
   ```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.2.2.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-golang-gorm-quickstart。

4.2.2.3.1 连接到平凯数据库

```
func createDB() *gorm.DB {
    dsn := fmt.Sprintf("%s:%s@tcp(%s:%s)/%s?charset=utf8mb4&tls=%s",
        ${tidb_user}, ${tidb_password}, ${tidb_host}, ${tidb_port}, ${tidb_db_name}, ${use_ss
l})

    db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{
        Logger: logger.Default.LogMode(logger.Info),
    })
    if err != nil {
        panic(err)
    }

    return db
}
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name} 等替换为你的 TiDB 集群的实际值。

4.2.2.3.2 插入数据

```
db.Create(&Player{ID: "id", Coins: 1, Goods: 1})
```

更多信息参考插入数据。

4.2.2.3.3 查询数据

```
var queryPlayer Player
db.Find(&queryPlayer, "id = ?", "id")
```

更多信息参考查询数据。

4.2.2.3.4 更新数据

```
db.Save(&Player{ID: "id", Coins: 100, Goods: 1})
```

更多信息参考更新数据。

4.2.2.3.5 删除数据

```
db.Delete(&Player{ID: "id"})
```

更多信息参考删除数据。

4.2.2.4 下一步

- 关于 GORM 的更多使用方法，可以参考 GORM 官方文档 及 GORM 官方文档中的 TiDB 章节。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.2.2.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.3 Python

4.3.1 使用 mysqlclient 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。mysqlclient 为当前流行的开源 Python Driver 之一。

本文档将展示如何使用平凯数据库和 mysqlclient 来完成以下任务：

- 配置你的环境。
- 使用 mysqlclient 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.3.1.1 前置需求

- 推荐 Python 3.8 及以上版本。
- Git。
- TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.3.1.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.3.1.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-python-mysqlclient-quickstart.git
cd tidb-python-mysqlclient-quickstart
```

4.3.1.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 mysqlclient）：

```
pip install -r requirements.txt
```

如果遇到安装问题，请参考 mysqlclient 官方文档。

4.3.1.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 .env 中。示例结果如下：

   ```
   TIDB_HOST='{host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   ```

```
TIDB_PASSWORD='{password}'
TIDB_DB_NAME='test'
```

注意替换 {} 中的占位符为你的 TiDB 对应的值，并删除 CA_PATH 这行。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.3.1.2.4 第 4 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

```
python mysqlclient_example.py
```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.3.1.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-python-mysqlclient-quickstart。

4.3.1.3.1 连接到平凯数据库

```
def get_mysqlclient_connection(autocommit:bool=True) -> MySQLdb.Connection:
    db_conf = {
        "host": '${tidb_host}',
        "port": '${tidb_port}',
        "user": '${tidb_user}',
        "password": '${tidb_password}',
        "database": '${tidb_db_name}',
        "autocommit": autocommit
    }

    if '${ca_path}':
        db_conf["ssl_mode"] = "VERIFY_IDENTITY"
        db_conf["ssl"] = {"ca": '${ca_path}'}

    return MySQLdb.connect(**db_conf)
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name}、${ca_path} 等替换为你的 TiDB 集群的实际值。

4.3.1.3.2 插入数据

```
with get_mysqlclient_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player = ("test", 1, 1)
        cursor.execute("INSERT INTO players (id, coins, goods) VALUES (%s, %s, %s)", playe
r)
```

更多信息参考插入数据。

4.3.1.3.3 查询数据

```
with get_mysqlclient_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        cur.execute("SELECT count(*) FROM players")
        print(cur.fetchone()[0])
```

更多信息参考查询数据。

4.3.1.3.4 更新数据

```
with get_mysqlclient_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player_id, amount, price="test", 10, 500
        cur.execute(
            "UPDATE players SET goods = goods + %s, coins = coins + %s WHERE id = %s",
            (-amount, price, player_id),
        )
```

更多信息参考更新数据。

4.3.1.3.5 删除数据

```
with get_mysqlclient_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player_id = "test"
        cur.execute("DELETE FROM players WHERE id = %s", (player_id,))
```

更多信息参考删除数据。

4.3.1.4 注意事项

4.3.1.4.1 使用驱动程序还是 ORM 框架?

Python 驱动程序提供对数据库的底层访问，但要求开发者：

- 手动建立和释放数据库连接
- 手动管理数据库事务
- 手动将数据行（在 mysqlclient 中表示为元组 (tuple)）映射为数据对象

建议仅在需要编写复杂的 SQL 语句时使用驱动程序。其他情况下，建议使用 ORM 框架进行开发，例如 SQLAlchemy、Peewee 和 Django。ORM 可以帮助你：

- 减少管理连接和事务的模板代码
- 使用数据对象代替大量 SQL 语句来操作数据

4.3.1.5 下一步

- 关于 mysqlclient 的更多使用方法，可以参考 mysqlclient 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.3.1.6 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.3.2 使用 MySQL Connector/Python 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。MySQL Connector/Python 是由 MySQL 开发的 Python Driver。

本文档将展示如何使用平凯数据库和 MySQL Connector/Python 来完成以下任务：

- 配置你的环境。
- 使用 MySQL Connector/Python 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.3.2.1 前置需求

- 推荐 Python 3.8 及以上版本。
- Git。
- TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.3.2.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.3.2.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-python-mysqlconnector-quickstart.git
cd tidb-python-mysqlconnector-quickstart
```

4.3.2.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 mysql-connector-python）：

```
pip install -r requirements.txt
```

4.3.2.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 .env 中。示例结果如下：

```
TIDB_HOST='{host}'
TIDB_PORT='4000'
TIDB_USER='root'
TIDB_PASSWORD='{password}'
TIDB_DB_NAME='test'
```

注意替换 {} 中的占位符为你的 TiDB 对应的值，并删除 CA_PATH 这行。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.3.2.2.4 第 4 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

```
python mysql_connector_example.py
```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.3.2.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-python-mysqlconnector-quickstart。

4.3.2.3.1 连接到平凯数据库

```
def get_connection(autocommit: bool = True) -> MySQLConnection:
    db_conf = {
        "host": '${tidb_host}',
        "port": '${tidb_port}',
        "user": '${tidb_user}',
        "password": '${tidb_password}',
        "database": '${tidb_db_name}',
        "autocommit": autocommit,
        "use_pure": True,
    }

    if '${ca_path}':
        db_conf["ssl_verify_cert"] = True
```

```
        db_conf["ssl_verify_identity"] = True
        db_conf["ssl_ca"] = '${ca_path}'
    return mysql.connector.connect(**db_conf)
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name} 以及 ${ca_path} 替换为你的 TiDB 集群的实际值。

4.3.2.3.2 插入数据

```
with get_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player = ("test", 1, 1)
        cursor.execute("INSERT INTO players (id, coins, goods) VALUES (%s, %s, %s)", playe
r)
```

更多信息参考插入数据。

4.3.2.3.3 查询数据

```
with get_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        cur.execute("SELECT count(*) FROM players")
        print(cur.fetchone()[0])
```

更多信息参考查询数据。

4.3.2.3.4 更新数据

```
with get_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player_id, amount, price="test", 10, 500
        cur.execute(
            "UPDATE players SET goods = goods + %s, coins = coins + %s WHERE id = %s",
            (-amount, price, player_id),
        )
```

更多信息参考更新数据。

4.3.2.3.5 删除数据

```
with get_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player_id = "test"
        cur.execute("DELETE FROM players WHERE id = %s", (player_id,))
```

更多信息参考删除数据。

4.3.2.4 注意事项

4.3.2.4.1 使用驱动程序还是 ORM 框架?

Python 驱动程序提供对数据库的底层访问，但要求开发者：

- 手动建立和释放数据库连接
- 手动管理数据库事务
- 手动将数据行（在 mysql-connector-python 中表示为元组 (Tuple) 或者字典 (Dictionary)）映射为数据对象

建议仅在需要编写复杂的 SQL 语句时使用驱动程序。其他情况下，建议使用 ORM 框架进行开发，例如 SQLAlchemy、Peewee 和 Django。ORM 可以帮助你：

- 减少管理连接和事务的模板代码
- 使用数据对象代替大量 SQL 语句来操作数据

4.3.2.5 下一步

- 关于 mysql-connector-python 的更多使用方法，可以参考 MySQL Connector/Python 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.3.2.6 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.3.3 使用 PyMySQL 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。PyMySQL 为当前流行的开源 Python Driver 之一。

本文档将展示如何使用平凯数据库和 PyMySQL 来完成以下任务：

- 配置你的环境。
- 使用 PyMySQL 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.3.3.1 前置需求

- 推荐 Python 3.8 及以上版本。
- Git。
- TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.3.3.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.3.3.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-python-pymysql-quickstart.git
cd tidb-python-pymysql-quickstart
```

4.3.3.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 pymysql）：

```
pip install -r requirements.txt
```

4.3.3.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 .env 中。示例结果如下：

   ```
   TIDB_HOST='{host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   TIDB_PASSWORD='{password}'
   TIDB_DB_NAME='test'
   ```

   注意替换 {} 中的占位符为你的 TiDB 对应的值，并删除 CA_PATH 这行。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.3.3.2.4 第 4 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

   ```
   python pymysql_example.py
   ```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.3.3.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-python-pymysql-quickstart。

4.3.3.3.1 连接到平凯数据库

```
from pymysql import Connection
from pymysql.cursors import DictCursor
```

```
def get_connection(autocommit: bool = True) -> Connection:
    config = Config()
    db_conf = {
        "host": ${tidb_host},
        "port": ${tidb_port},
        "user": ${tidb_user},
        "password": ${tidb_password},
        "database": ${tidb_db_name},
        "autocommit": autocommit,
        "cursorclass": DictCursor,
    }

    if ${ca_path}:
        db_conf["ssl_verify_cert"] = True
        db_conf["ssl_verify_identity"] = True
        db_conf["ssl_ca"] = ${ca_path}

    return pymysql.connect(**db_conf)
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name} 以及 ${ca_path} 替换为你的 TiDB 集群的实际值。

4.3.3.3.2 插入数据

```
with get_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player = ("1", 1, 1)
        cur.execute("INSERT INTO players (id, coins, goods) VALUES (%s, %s, %s)", player)
```

更多信息参考插入数据。

4.3.3.3.3 查询数据

```
with get_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        cur.execute("SELECT count(*) FROM players")
        print(cursor.fetchone()["count(*)"])
```

更多信息参考查询数据。

4.3.3.3.4 更新数据

```
with get_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player_id, amount, price = "1", 1, 50
        cur.execute(
            "UPDATE players SET goods = goods + %s, coins = coins + %s WHERE id = %s",
            (-amount, price, player_id),
        )
```

更多信息参考更新数据。

4.3.3.3.5 删除数据

```
with get_connection(autocommit=True) as conn:
    with conn.cursor() as cur:
        player_id = "1"
        cur.execute("DELETE FROM players WHERE id = %s", player_id)
```

更多信息参考删除数据。

4.3.3.4 注意事项

4.3.3.4.1 使用驱动程序还是 ORM 框架?

Python 驱动程序提供对数据库的底层访问，但要求开发者：

- 手动建立和释放数据库连接
- 手动管理数据库事务
- 手动将数据行（在 pymysql 中表示为元组 (tuple) 或者字典 (dict)）映射为数据对象

建议仅在需要编写复杂的 SQL 语句时使用驱动程序。其他情况下，建议使用 ORM 框架进行开发，例如 SQLAlchemy、Peewee 和 Django。ORM 可以帮助你：

- 减少管理连接和事务的模板代码
- 使用数据对象代替大量 SQL 语句来操作数据

4.3.3.5 下一步

- 关于 PyMySQL 的更多使用方法，可以参考 PyMySQL 官方文档。

- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.3.3.6 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.3.4 使用 SQLAlchemy 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。SQLAlchemy 为当前流行的开源 Python ORM (Object Relational Mapper) 之一。

本文档将展示如何使用平凯数据库和 SQLAlchemy 来完成以下任务：

- 配置你的环境。
- 使用 SQLAlchemy 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.3.4.1 前置需求

- 推荐 Python 3.8 及以上版本。
- Git。
- TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.3.4.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.3.4.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-python-sqlalchemy-quickstart.git
cd tidb-python-sqlalchemy-quickstart
```

4.3.4.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 SQLAlchemy 和 PyMySQL）：

```
pip install -r requirements.txt
```

为什么安装 PyMySQL?

SQLAlchemy 是一个支持多种数据库的 ORM 库。它是对数据库的高层抽象，可以帮助开发者以更面向对象的方式编写 SQL 语句。但 SQLAlchemy 并不提供数据库驱动，因此需要单独安装用于连接 TiDB 的驱动。本示例项目使用 PyMySQL 作为数据库驱动。PyMySQL 是一个与 TiDB 兼容的纯 Python 实现的 MySQL 客户端库，并可以在所有平台上安装。

你也可以使用其他数据库驱动，例如 mysqlclient 以及 mysql-connector-python。但是它们不是纯 Python 库，需要安装对应的 C/C++ 编译器和 MySQL 客户端库进行编译。更多信息，参考 SQLAlchemy 官方文档。

4.3.4.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 .env 中。示例结果如下：

   ```
   TIDB_HOST='{host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   TIDB_PASSWORD='{password}'
   TIDB_DB_NAME='test'
   ```

注意替换 {} 中的占位符为你的 TiDB 对应的值，并删除 CA_PATH 这行。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.3.4.2.4 第 4 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

   ```
   python sqlalchemy_example.py
   ```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.3.4.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-python-sqlalchemy-quickstart。

4.3.4.3.1 连接到平凯数据库

```
from sqlalchemy import create_engine, URL
from sqlalchemy.orm import sessionmaker

def get_db_engine():
    connect_args = {}
    if ${ca_path}:
        connect_args = {
            "ssl_verify_cert": True,
            "ssl_verify_identity": True,
            "ssl_ca": ${ca_path},
        }
    return create_engine(
        URL.create(
            drivername="mysql+pymysql",
            username=${tidb_user},
            password=${tidb_password},
            host=${tidb_host},
            port=${tidb_port},
            database=${tidb_db_name},
```

```
        ),
        connect_args=connect_args,
    )

engine = get_db_engine()
Session = sessionmaker(bind=engine)
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name} 以及 ${ca_path} 替换为你的 TiDB 集群的实际值。

4.3.4.3.2 声明数据对象

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Player(Base):
    id = Column(Integer, primary_key=True)
    name = Column(String(32), unique=True)
    coins = Column(Integer)
    goods = Column(Integer)

    __tablename__ = "players"
```

更多信息参考 SQLAlchemy 声明式映射表。

4.3.4.3.3 插入数据

```
with Session() as session:
    player = Player(name="test", coins=100, goods=100)
    session.add(player)
    session.commit()
```

更多信息参考插入数据以及 SQLAlchemy Query。

4.3.4.3.4 查询数据

```
with Session() as session:
    player = session.query(Player).filter_by(name == "test").one()
    print(player)
```

更多信息参考查询数据以及 SQLAlchemy Query。

4.3.4.3.5 更新数据

```
with Session() as session:
    player = session.query(Player).filter_by(name == "test").one()
    player.coins = 200
    session.commit()
```

更多信息参考更新数据以及 SQLAlchemy Query。

4.3.4.3.6 删除数据

```
with Session() as session:
    player = session.query(Player).filter_by(name == "test").one()
    session.delete(player)
    session.commit()
```

更多信息参考删除数据以及 SQLAlchemy Query。

4.3.4.4 下一步

- 关于 SQLAlchemy 的更多使用方法，可以参考 SQLAlchemy 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.3.4.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.3.5 使用 peewee 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。peewee 为当前流行的开源 Python ORM (Object Relational Mapper) 之一。

本文档将展示如何使用平凯数据库和 peewee 来完成以下任务：

- 配置你的环境。
- 使用 peewee 连接到平凯数据库集群。

- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.3.5.1 前置需求

- 推荐 Python 3.8 及以上版本。
- Git。
- TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.3.5.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.3.5.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-python-peewee-quickstart.git
cd tidb-python-peewee-quickstart
```

4.3.5.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 peewee 和 PyMySQL）：

```
pip install -r requirements.txt
```

为什么安装 PyMySQL?

peewee 是一个支持多种数据库的 ORM 库。它是对数据库的高层抽象，可以帮助开发者以更面向对象的方式编写 SQL 语句。但 peewee 并不提供数据库驱动，因此需要单独安装用于连接 TiDB 的驱动。本示例项目使用 PyMySQL 作为数据库驱动。PyMySQL 是一个与 TiDB 兼容的纯 Python 实现的 MySQL 客户端库，并可以在所有平台上安装。更多信息，参考 peewee 官方文档。

4.3.5.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 .env 中。示例结果如下：

   ```
   TIDB_HOST='{host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   TIDB_PASSWORD='{password}'
   TIDB_DB_NAME='test'
   ```

   注意替换 {} 中的占位符为你的 TiDB 对应的值，并删除 CA_PATH 这行。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.3.5.2.4 第 4 步：运行代码并查看结果

1. 运行下述命令，执行示例代码：

   ```
   python peewee_example.py
   ```

2. 查看 Expected-Output.txt，并与你的程序输出进行比较。结果近似即为连接成功。

4.3.5.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-python-peewee-quickstart。

4.3.5.3.1 连接到平凯数据库

```
from peewee import MySQLDatabase

def get_db_engine():
    connect_params = {}
    if '${ca_path}':
```

```
        connect_params = {
            "ssl_verify_cert": True,
            "ssl_verify_identity": True,
            "ssl_ca": '${ca_path}',
        }
    return MySQLDatabase(
        '${tidb_db_name}',
        host='${tidb_host}',
        port='${tidb_port}',
        user='${tidb_user}',
        password='${tidb_password}',
        **connect_params,
    )
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name} 以及 ${ca_path} 替换为你的 TiDB 集群的实际值。

4.3.5.3.2 声明数据对象

```
from peewee import Model, CharField, IntegerField

db = get_db_engine()

class BaseModel(Model):
    class Meta:
        database = db

class Player(BaseModel):
    name = CharField(max_length=32, unique=True)
    coins = IntegerField(default=0)
    goods = IntegerField(default=0)

    class Meta:
        table_name = "players"

    def __str__(self):
        return f"Player(name={self.name}, coins={self.coins}, goods={self.goods})"
```

更多信息参考 peewee 模型与字段。

4.3.5.3.3 插入数据

```
### 插入单个对象
Player.create(name="test", coins=100, goods=100)

### 插入多个对象
data = [
    {"name": "test1", "coins": 100, "goods": 100},
    {"name": "test2", "coins": 100, "goods": 100},
]
Player.insert_many(data).execute()
```

更多信息参考插入数据。

4.3.5.3.4 查询数据

```
### 查询所有对象
players = Player.select()

### 查询单个对象
player = Player.get(Player.name == "test")

### 查询多个对象
players = Player.select().where(Player.coins == 100)
```

更多信息参考查询数据。

4.3.5.3.5 更新数据

```
### 更新单个对象
player = Player.get(Player.name == "test")
player.coins = 200
player.save()

### 批量更新多个对象
Player.update(coins=200).where(Player.coins == 100).execute()
```

更多信息参考更新数据。

4.3.5.3.6 删除数据

```
### 删除单个对象
player = Player.get(Player.name == "test")
player.delete_instance()

### 批量删除多个对象
Player.delete().where(Player.coins == 100).execute()
```

更多信息参考删除数据。

4.3.5.4 下一步

- 关于 peewee 的更多使用方法，可以参考 peewee 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.3.5.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.3.6 使用 Django 连接到平凯数据库

TiDB 是一个兼容 MySQL 的数据库。Django 为当前流行的 Python Web 框架之一，它内部实现了一个强大的 ORM (Object Relational Mapper) 系统。

本文档将展示如何使用平凯数据库和 Django 来完成以下任务：

- 配置你的环境。
- 使用 Django 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.3.6.1 前置需求

- 推荐 Python 3.8 及以上版本。
- Git。
- TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.3.6.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.3.6.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-python-django-quickstart.git
cd tidb-python-django-quickstart
```

4.3.6.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 Django、django-tidb 和 mysqlclient）：

```
pip install -r requirements.txt
```

如果你在安装 mysqlclient 时遇到问题，请参考 mysqlclient 官方文档。

django-tidb 是一个为 Django 提供的 TiDB 适配器，它解决了 TiDB 与 Django 之间的兼容性问题。

安装 django-tidb 时，请选择与你的 Django 版本匹配的版本。例如，如果你使用的是 django==4.2.*，则应安装 django-tidb>=4.2.0,<4.3.0，其中 minor 版本号不需要完全相同。建议使用最新的 minor 版本。

更多信息，请参考 django-tidb 仓库。

4.3.6.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 复制并粘贴对应 TiDB 的连接字符串至 .env 中。示例结果如下：

   ```
   TIDB_HOST='{host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   TIDB_PASSWORD='{password}'
   TIDB_DB_NAME='test'
   ```

   注意替换 {} 中的占位符为你的 TiDB 对应的值，并删除 CA_PATH 这行。如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.3.6.2.4 第 4 步：初始化数据库

在示例项目根目录执行以下命令，初始化数据库：

```
python manage.py migrate
```

```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sample_project, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
```

```
Applying auth.0004_alter_user_username_opts... OK
Applying auth.0005_alter_user_last_login_null... OK
Applying auth.0006_require_contenttypes_0002... OK
Applying auth.0007_alter_validators_add_error_messages... OK
Applying auth.0008_alter_user_username_max_length... OK
Applying auth.0009_alter_user_last_name_max_length... OK
Applying auth.0010_alter_group_name_max_length... OK
Applying auth.0011_update_proxy_permissions... OK
Applying auth.0012_alter_user_first_name_max_length... OK
Applying sample_project.0001_initial... OK
Applying sessions.0001_initial... OK
```

4.3.6.2.5 第 5 步：运行示例应用程序

1. 在开发模式下运行示例应用程序：

   ```
   python manage.py runserver
   ```

   应用程序默认在 8000 端口上运行。如果你想要使用其他端口号，可以在命令后添加端口号，例如：

   ```
   python manage.py runserver 8080
   ```

2. 打开浏览器，在地址栏输入 http://localhost:8000/，访问示例应用程序，你可以进行以下操作：

   - 创建一个新的 Player
   - 批量创建 Player
   - 查看所有的 Player
   - 更新 Player
   - 删除 Player
   - 在两个 Player 之间交易物品

4.3.6.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-python-django-quickstart。

4.3.6.3.1 配置数据库连接

打开 sample_project/settings.py 文件，添加以下配置：

```
DATABASES = {
    "default": {
        "ENGINE": "django_tidb",
        "HOST": '${tidb_host}',
        "PORT": '${tidb_port}',
        "USER": '${tidb_user}',
        "PASSWORD": '${tidb_password}',
        "NAME": '${tidb_db_name}',
        "OPTIONS": {
            "charset": "utf8mb4",
        },
    }
}

TIDB_CA_PATH = '${ca_path}'
if TIDB_CA_PATH:
    DATABASES["default"]["OPTIONS"]["ssl_mode"] = "VERIFY_IDENTITY"
    DATABASES["default"]["OPTIONS"]["ssl"] = {
        "ca": TIDB_CA_PATH,
    }
```

在使用该函数时，你需要将 ${tidb_host}、${tidb_port}、${tidb_user}、${tidb_password}、${tidb_db_name} 以及 ${ca_path} 替换为你的 TiDB 集群的实际值。

4.3.6.3.2 声明数据对象

```
from django.db import models

class Player(models.Model):
    name = models.CharField(max_length=32, blank=False, null=False)
    coins = models.IntegerField(default=100)
    goods = models.IntegerField(default=1)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

更多信息参考 Django 模型。

4.3.6.3.3 插入数据

```
### 插入单个对象
player = Player.objects.create(name="player1", coins=100, goods=1)

### 批量插入多个对象
Player.objects.bulk_create([
    Player(name="player1", coins=100, goods=1),
    Player(name="player2", coins=200, goods=2),
    Player(name="player3", coins=300, goods=3),
])
```

更多信息参考插入数据。

4.3.6.3.4 查询数据

```
### 查询单个对象
player = Player.objects.get(name="player1")

### 查询多个对象
filtered_players = Player.objects.filter(name="player1")

### 查询所有对象
all_players = Player.objects.all()
```

更多信息参考查询数据。

4.3.6.3.5 更新数据

```
### 更新单个对象
player = Player.objects.get(name="player1")
player.coins = 200
player.save()

### 批量更新多个对象
Player.objects.filter(coins=100).update(coins=200)
```

更多信息参考更新数据。

4.3.6.3.6 删除数据

```
### 删除单个对象
player = Player.objects.get(name="player1")
player.delete()

### 批量删除多个对象
Player.objects.filter(coins=100).delete()
```

更多信息参考删除数据。

4.3.6.4 下一步

- 关于 Django 的更多使用方法，可以参考 Django 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.3.6.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.4 Node.js

4.4.1 使用 node-mysql2 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。node-mysql2 是一个与 mysqljs/mysql 兼容的面向 Node.js 的 MySQL 驱动。

本文档将展示如何使用平凯数据库和 node-mysql2 来完成以下任务：

- 配置你的环境。
- 使用 node-mysql2 驱动连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.4.1.1 前置需求

为了能够顺利完成本教程，你需要提前：

- 在你的机器上安装 Node.js 16.x 或以上版本。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.4.1.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.4.1.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-nodejs-mysql2-quickstart.git
cd tidb-nodejs-mysql2-quickstart
```

4.4.1.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖 (包括 mysql2 和 dotenv 依赖包)：

```
npm install
```

在你现有的项目当中，你可以通过以下命令安装 mysql2 和 dotenv 依赖包（dotenv 用于从 .env 文件中读取环境变量）：

```
npm install mysql2 dotenv --save
```

4.4.1.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

    ```
    cp .env.example .env
    ```

2. 编辑 .env 文件，按照如下格式设置连接信息，将占位符 {} 替换为你的 TiDB 集群的连接参数值：

```
TIDB_HOST='{host}'
TIDB_PORT='4000'
TIDB_USER='root'
TIDB_PASSWORD='{password}'
TIDB_DATABASE='test'
```

3. 保存 .env 文件。

4.4.1.2.4 第 4 步：运行代码并查看结果

运行下述命令，执行示例代码：

```
npm run start
```

预期输出结果：

如果连接成功，你的终端将会输出所连接集群的版本信息：

```
🔌 Connected to TiDB cluster! (TiDB version: 8.0.11-TiDB-v7.1.8-5.1)
⌛ Loading sample game data...
✅ Loaded sample game data.

🆕 Created a new player with ID 12.
ℹ️ Got Player 12: Player { id: 12, coins: 100, goods: 100 }
🔢 Added 50 coins and 50 goods to player 12, updated 1 row.
🚮 Deleted 1 player data.
```

4.4.1.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。完整代码及其运行方式，见代码仓库 tidb-samples/tidb-nodejs-mysql2-quickstart。

4.4.1.3.1 连接到平凯数据库

下面的代码使用环境变量中定义的连接选项来建立与 TiDB 集群的连接。

```
// 步骤 1. 导入 'mysql2' 和 'dotenv' 依赖包。
import { createConnection } from "mysql2/promise";
```

```
import dotenv from "dotenv";
import * as fs from "fs";

// 步骤 2. 将连接参数从 .env 文件中读取到 process.env 中。
dotenv.config();

async function main() {
  // 步骤 3. 创建与 TiDB 集群的连接。
  const options = {
    host: process.env.TIDB_HOST || '127.0.0.1',
    port: process.env.TIDB_PORT || 4000,
    user: process.env.TIDB_USER || 'root',
    password: process.env.TIDB_PASSWORD || '',
    database: process.env.TIDB_DATABASE || 'test',
    ssl: process.env.TIDB_ENABLE_SSL === 'true' ? {
      minVersion: 'TLSv1.2',
      ca: process.env.TIDB_CA_PATH ? fs.readFileSync(process.env.TIDB_CA_PATH) : un
defined
    } : null,
  }
  const conn = await createConnection(options);

  // 步骤 4. 执行 SQL 语句。

  // 步骤 5. 关闭连接。
  await conn.end();
}

void main();
```

4.4.1.3.2 插入数据

```
const [rsh] = await conn.query('INSERT INTO players (coins, goods) VALUES (?, ?);', [100,
100]);
console.log(rsh.insertId);
```

更多信息参考插入数据。

4.4.1.3.3 查询数据

```
const [rows] = await conn.query('SELECT id, coins, goods FROM players WHERE id = ?;',
[1]);
console.log(rows[0]);
```

更多信息参考查询数据。

4.4.1.3.4 更新数据

```
const [rsh] = await conn.query(
   'UPDATE players SET coins = coins + ?, goods = goods + ? WHERE id = ?;',
   [50, 50, 1]
);
console.log(rsh.affectedRows);
```

更多信息参考更新数据。

4.4.1.3.5 删除数据

```
const [rsh] = await conn.query('DELETE FROM players WHERE id = ?;', [1]);
console.log(rsh.affectedRows);
```

更多信息参考删除数据。

4.4.1.4 注意事项

- 推荐使用连接池来管理数据库连接，以减少频繁建立和销毁连接所带来的性能开销。
- 为了避免 SQL 注入的风险，推荐使用预处理语句执行 SQL。
- 在不涉及大量复杂 SQL 语句的场景下，推荐使用 ORM 框架 (例如：Sequelize、TypeORM 或 Prisma) 来提升你的开发效率。
- 当你在数据表中使用到 BIGINT 和 DECIMAL 类型列时，需要开启 Driver 的 supportBigNumbers: true 选项。
- 为了避免由于网络原因出现的 read ECONNRESET Socket 错误，可以在 Driver 上开启 enableKeepAlive: true 选项。（相关 Issue: sidorares/node-mysql2#683）

4.4.1.5 下一步

- 关于 node-mysql2 的更多使用方法，可以参考 node-mysql2 的 GitHub 仓库。

- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供了专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.4.2 使用 mysql.js 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。mysql.js 是一个纯 Node.js 代码编写的实现了 MySQL 协议的 JavaScript 客户端。

本文档将展示如何使用平凯数据库和 mysql.js 来构造一个简单的 CRUD 应用程序。

- 配置你的环境。
- 使用 mysql.js 驱动连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.4.2.1 前置需求

为了能够顺利完成本教程，你需要提前：

- 在你的机器上安装 Node.js 16.x 或以上版本。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.4.2.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.4.2.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-nodejs-mysqljs-quickstart.git
cd tidb-nodejs-mysqljs-quickstart
```

4.4.2.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖 (包括 mysql 和 dotenv 依赖包)：

```
npm install
```

在你现有的项目当中，你可以通过以下命令安装 mysql 和 dotenv 依赖包（dotenv 用于从 .env 文件中读取环境变量）：

```
npm install mysql dotenv --save
```

4.4.2.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 编辑 .env 文件，按照如下格式设置连接信息，将占位符 {} 替换为你的 TiDB 集群的连接参数值：

   ```
   TIDB_HOST='{host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   TIDB_PASSWORD='{password}'
   TIDB_DATABASE='test'
   ```

3. 保存 .env 文件。

4.4.2.2.4 第 4 步：运行代码并查看结果

运行下述命令，执行示例代码：

```
npm run start
```

预期输出结果：

如果连接成功，你的终端将会输出所连接集群的版本信息：

```
🔌 Connected to TiDB cluster! (TiDB version: 8.0.11-TiDB-v7.1.8-5.1)
⌛ Loading sample game data...
✅ Loaded sample game data.

🆕 Created a new player with ID 12.
ℹ️ Got Player 12: Player { id: 12, coins: 100, goods: 100 }
🔢 Added 50 coins and 50 goods to player 12, updated 1 row.
🚮 Deleted 1 player data.
```

4.4.2.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。完整代码及其运行方式，见代码仓库 tidb-samples/tidb-nodejs-mysqljs-quickstart。

4.4.2.3.1 连接到平凯数据库

下面的代码使用环境变量中定义的连接选项来建立与 TiDB 集群的连接。

```
// 步骤 1. 导入 'mysql' 和 'dotenv' 依赖包。
import { createConnection } from "mysql";
import dotenv from "dotenv";
import * as fs from "fs";

// 步骤 2. 将连接参数从 .env 文件中读取到 process.env 中。
dotenv.config();

// 步骤 3. 创建与 TiDB 集群的连接。
const options = {
  host: process.env.TIDB_HOST || '127.0.0.1',
  port: process.env.TIDB_PORT || 4000,
  user: process.env.TIDB_USER || 'root',
  password: process.env.TIDB_PASSWORD || '',
  database: process.env.TIDB_DATABASE || 'test',
  ssl: process.env.TIDB_ENABLE_SSL === 'true' ? {
    minVersion: 'TLSv1.2',
    ca: process.env.TIDB_CA_PATH ? fs.readFileSync(process.env.TIDB_CA_PATH) : undefi
ned
  } : null,
}
```

```
const conn = createConnection(options);

// 步骤 4. 执行 SQL 语句。

// 步骤 5. 关闭连接。
conn.end();
```

4.4.2.3.2 插入数据

下面的代码创建了一条 Player 记录，并返回了该记录的 ID。

```
conn.query('INSERT INTO players (coins, goods) VALUES (?, ?);', [coins, goods], (err, ok)
=> {
  if (err) {
    console.error(err);
  } else {
    console.log(ok.insertId);
  }
});
```

更多信息参考插入数据。

4.4.2.3.3 查询数据

下面的查询返回了 ID 为 1 的 Player 记录。

```
conn.query('SELECT id, coins, goods FROM players WHERE id = ?;', [1], (err, rows) => {
  if (err) {
    console.error(err);
  } else {
    console.log(rows[0]);
  }
});
```

更多信息参考查询数据。

4.4.2.3.4 更新数据

下面的查询为 ID 为 1 的 Player 记录增加了 50 个金币和 50 个物品。

```
conn.query(
  'UPDATE players SET coins = coins + ?, goods = goods + ? WHERE id = ?;',
```

```
  [50, 50, 1],
  (err, ok) => {
    if (err) {
      console.error(err);
    } else {
      console.log(ok.affectedRows);
    }
  }
);
```

更多信息参考更新数据。

4.4.2.3.5 删除数据

下面的查询删除了 ID 为 1 的 Player 记录。

```
conn.query('DELETE FROM players WHERE id = ?;', [1], (err, ok) => {
  if (err) {
    reject(err);
  } else {
    resolve(ok.affectedRows);
  }
});
```

更多信息参考删除数据。

4.4.2.4 注意事项

- 推荐使用连接池来管理数据库连接，以减少频繁建立和销毁连接所带来的性能开销。
- 为了避免 SQL 注入的风险，请在执行 SQL 语句前传递到 SQL 中的值进行转义。

  Note

  mysqljs/mysql 包目前还不支持预处理语句，它只在客户端对值进行转义 (相关 issue: mysqljs/mysql#274)。

  如果你希望使用预处理语句来避免 SQL 注入或提升批量插入/更新的效率，推荐使用 mysql2 包。

- 在不涉及大量复杂 SQL 语句的场景下, 推荐使用 ORM 框架 (例如：Sequelize, TypeORM, 或 Prisma) 来提升你的开发效率.
- 当你在数据表中使用到 BIGINT 和 DECIMAL 类型列时，需要开启 Driver 的 supportBigNumbers: true 选项.

4.4.2.5 下一步

- 关于 mysql.js 驱动的更多使用方法，可以参考 mysql.js 的 GitHub 仓库。
- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供了专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.4.3 使用 Prisma 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。Prisma 是当前流行的 Node.js ORM 框架之一。

本文档将展示如何使用平凯数据库和 Prisma 来构造一个简单的 CRUD 应用程序。

- 配置你的环境。
- 使用 Prisma 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.4.3.1 前置需求

为了能够顺利完成本教程，你需要提前：

- 在你的机器上安装 Node.js 16.x 或以上版本。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.4.3.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.4.3.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-nodejs-prisma-quickstart.git
cd tidb-nodejs-prisma-quickstart
```

4.4.3.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖 (包括 prisma 依赖包)：

```
npm install
```

在你现有的项目当中，你可以通过以下命令安装所需要的依赖包：

```
npm install prisma typescript ts-node @types/node --save-dev
```

4.4.3.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 编辑 .env 文件，按照如下格式设置连接信息，将占位符 {} 替换为你的 TiDB 集群的连接参数值：

   ```
   DATABASE_URL='mysql://{user}:{password}@{host}:4000/test'
   ```

   如果你在本地运行 TiDB 集群，默认的 Host 是 127.0.0.1, 默认用户名为 root, 密码为空。

3. 保存 .env 文件。

4. 在 prisma/schema.prisma 文件中，将 provider 修改为 mysql，并将 url 修改为 env("DATABASE_URL")：

```
datasource db {
  provider = "mysql"
  url      = env("DATABASE_URL")
}
```

4.4.3.2.4 第 4 步：初始化表结构

运行以下命令，使用 Prisma Migrate 根据 prisma/schema.prisma 文件中的数据模型定义来初始化数据库表结构：

```
npx prisma migrate dev
```

prisma.schema 文件中的模型定义：

```
// 定义一个 Player 模型，表示 `players` 表。
model Player {
  id        Int      @id @default(autoincrement())
  name      String   @unique(map: "uk_player_on_name") @db.VarChar(50)
  coins     Decimal  @default(0)
  goods     Int      @default(0)
  createdAt DateTime @default(now()) @map("created_at")
  profile   Profile?

  @@map("players")
}

// 定义一个 Profile 模型，表示 `profiles` 表。
model Profile {
  playerId  Int    @id @map("player_id")
  biography String @db.Text

  // 定义 `Player` 和 `Profile` 模型之间的 1:1 关系，并使用外键约束。
  player    Player @relation(fields: [playerId], references: [id], onDelete: Cascade, map: "fk_
profile_on_player_id")

  @@map("profiles")
}
```

你可以通过查阅 Prisma 的 Data model 文档来了解如何在 prisma.schema 文件里定义数据模型。

预期执行结果：

Your database is now in sync with your schema.

✓ Generated Prisma Client (5.1.1 | library) to ./node_modules/@prisma/client in 54ms

这个命令同时会根据 prisma/schema.prisma 文件中的模型定义，生成用于与数据库交互的 Prisma Client 的代码。

4.4.3.2.5 第 5 步：运行代码并查看结果

运行下述命令，执行示例代码：

npm start

示例代码中的主要逻辑：

```
// 步骤 1. 导入自动生成的 `@prisma/client` 依赖包。
import {Player, PrismaClient} from '@prisma/client';

async function main(): Promise<void> {
  // 步骤 2. 创建一个新的 `PrismaClient` 实例。
  const prisma = new PrismaClient();
  try {

    // 步骤 3. 使用 Prisma Client 执行一些 CRUD 操作。

  } finally {
    // 步骤 4. 断开 Prisma Client 的连接。
    await prisma.$disconnect();
  }
}

void main();
```

预期输出结果：

如果连接成功，在你的终端上会输出所连接集群的版本信息。

```
🔌 Connected to TiDB cluster! (TiDB version: 8.0.11-TiDB-v7.1.8-5.1)
🆕 Created a new player with ID 1.
ℹ️ Got Player 1: Player { id: 1, coins: 100, goods: 100 }
🔢 Added 50 coins and 50 goods to player 1, now player 1 has 150 coins and 150 goods.
🚮 Player 1 has been deleted.
```

4.4.3.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-nodejs-prisma-quickstart。

4.4.3.3.1 插入数据

下面的查询会创建一条新的 Player 记录，并返回一个包含自增 ID 的 Player 对象：

```
const player: Player = await prisma.player.create({
  data: {
    name: 'Alice',
    coins: 100,
    goods: 200,
    createdAt: new Date(),
  }
});
console.log(player.id);
```

更多信息参考插入数据。

4.4.3.3.2 查询数据

下面的查询会返回 ID 为 101 的 Player 记录，如果没有找到对应的记录，会返回 null：

```
const player: Player | null = prisma.player.findUnique({
    where: {
        id: 101,
    }
});
```

更多信息参考查询数据。

4.4.3.3.3 更新数据

下面的查询会将 ID 为 101 的 Player 记录的 coins 和 goods 字段的值分别增加 50：

```
await prisma.player.update({
  where: {
    id: 101,
  },
  data: {
    coins: {
      increment: 50,
    },
    goods: {
      increment: 50,
    },
  }
});
```

更多信息参考更新数据。

4.4.3.3.4 删除数据

下面的查询会删除 ID 为 101 的 Player 记录：

```
await prisma.player.delete({
  where: {
    id: 101,
  }
});
```

更多信息参考删除数据。

4.4.3.4 注意事项

4.4.3.4.1 外键约束与 Prisma Relation Mode

你可以使用外键约束或 Prisma Relation Mode 来检查参照完整性：

- 外键是从 v6.6.0 开始支持的，在 v7.1.8 成为正式功能。外键允许跨表交叉引用相关数据，外键约束则可以保证相关数据的一致性。

警告：

外键功能通常适用于为**中小规模**的数据提供完整性和一致性约束校验，但是在大数据量和分布式数据库系统下，使用外键可能会导致严重的性能问题，并对系统产生不可预知的影响。如果计划使用外键，请进行充分验证后谨慎使用。

- Prisma Relation Mode 是 Prisma Client 端对外键约束的模拟。该特性会对应用程序的性能产生一些影响，因为它需要额外的数据库查询来维护参照完整性。

4.4.3.5 下一步

- 关于 Prisma 的更多使用方法，可以参考 Prisma 的官方文档。
- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.4.4 使用 Sequelize 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。Sequelize 是当前流行的 Node.js ORM 框架之一。

本文档将展示如何使用平凯数据库和 Sequelize 来构造一个简单的 CRUD 应用程序。

- 配置你的环境。
- 使用 Sequelize 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.4.4.1 前置需求

为了能够顺利完成本教程，你需要提前：

- 在你的机器上安装 Node.js 18.x 或以上版本。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.4.4.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

> **Note**
>
> 完整代码及其运行方式，见代码仓库 tidb-samples/tidb-nodejs-sequelize-quickstart。

4.4.4.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone git@github.com:tidb-samples/tidb-nodejs-sequelize-quickstart.git
cd tidb-nodejs-sequelize-quickstart
```

4.4.4.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 sequelize）：

```
npm install
```

4.4.4.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 编辑 .env 文件，按照如下格式设置连接信息，将占位符 {} 替换为你的 TiDB 集群的连接参数值：

```
TIDB_HOST='{host}'
TIDB_PORT='4000'
TIDB_USER='root'
TIDB_PASSWORD='{password}'
TIDB_DB_NAME='test'
```

如果你在本地运行 TiDB 集群，默认的主机地址是 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.4.4.2.4 第 4 步：运行代码并查看结果

运行以下命令，执行示例代码：

```
npm start
```

预期输出结果（部分）：

```
INFO (app/10117): Getting sequelize instance...
Executing (default): SELECT 1+1 AS result
Executing (default): DROP TABLE IF EXISTS `players`;
Executing (default): CREATE TABLE IF NOT EXISTS `players` (`id` INTEGER NOT NULL auto_
increment  COMMENT 'The unique ID of the player.', `coins` INTEGER NOT NULL COMME
NT 'The number of coins that the player had.', `goods` INTEGER NOT NULL COMMENT 'T
he number of goods that the player had.', `createdAt` DATETIME NOT NULL, `updatedAt`
DATETIME NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `players`
Executing (default): INSERT INTO `players` (`id`,`coins`,`goods`,`createdAt`,`updatedAt`) VA
LUES (1,100,100,'2023-08-31 09:10:11','2023-08-31 09:10:11'),(2,200,200,'2023-08-31 09:1
0:11','2023-08-31 09:10:11'),(3,300,300,'2023-08-31 09:10:11','2023-08-31 09:10:11'),(4,40
0,400,'2023-08-31 09:10:11','2023-08-31 09:10:11'),(5,500,500,'2023-08-31 09:10:11','2023
-08-31 09:10:11');
Executing (default): SELECT `id`, `coins`, `goods`, `createdAt`, `updatedAt` FROM `players`
AS `players` WHERE `players`.`coins` > 300;
Executing (default): UPDATE `players` SET `coins`=?,`goods`=?,`updatedAt`=? WHERE `id`
= ?
Executing (default): DELETE FROM `players` WHERE `id` = 6
```

4.4.4.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-nodejs-sequelize-quickstart。

4.4.4.3.1 连接到平凯数据库

下面的代码使用环境变量中定义的连接选项来建立与 TiDB 集群的连接。

```
// src/lib/tidb.ts
import { Sequelize } from 'sequelize';

export function initSequelize() {
  return new Sequelize({
    dialect: 'mysql',
    host: process.env.TIDB_HOST || 'localhost',     // TiDB host
    port: Number(process.env.TIDB_PORT) || 4000,    // TiDB port, default: 4000
    username: process.env.TIDB_USER || 'root',      // TiDB user, for example: {prefix}.root
    password: process.env.TIDB_PASSWORD || 'root',  // TiDB password
    database: process.env.TIDB_DB_NAME || 'test',   // TiDB database name, default: test
    dialectOptions: {
      ssl:
        process.env?.TIDB_ENABLE_SSL === 'true'     // (Optional) Enable SSL
          ? {
              minVersion: 'TLSv1.2',
              rejectUnauthorized: true,
              ca: process.env.TIDB_CA_PATH          // (Optional) Path to the custom CA certific
ate
                ? readFileSync(process.env.TIDB_CA_PATH)
                : undefined,
            }
          : null,
    },
}

export async function getSequelize() {
  if (!sequelize) {
    sequelize = initSequelize();
    try {
      await sequelize.authenticate();
```

```
      logger.info('Connection has been established successfully.');
    } catch (error) {
      logger.error('Unable to connect to the database:');
      logger.error(error);
      throw error;
    }
  }
  return sequelize;
}
```

4.4.4.3.2 插入数据

下面的查询会创建一条单独的 Players 记录，并返回一个 Players 对象：

```
logger.info('Creating a new player...');
const newPlayer = await playersModel.create({
  id: 6,
  coins: 600,
  goods: 600,
});
logger.info('Created a new player.');
logger.info(newPlayer.toJSON());
```

更多信息参考插入数据。

4.4.4.3.3 查询数据

下面的查询会返回一条 Players 记录，其金币数量大于 300：

```
logger.info('Reading all players with coins > 300...');
const allPlayersWithCoinsGreaterThan300 = await playersModel.findAll({
  where: {
    coins: {
      [Op.gt]: 300,
    },
  },
});
logger.info('Read all players with coins > 300.');
logger.info(allPlayersWithCoinsGreaterThan300.map((p) => p.toJSON()));
```

更多信息参考查询数据。

4.4.4.3.4 更新数据

下面的查询会将 ID 为 6 的 Player 的金币数量和物品数量设置为 700，这个记录是在插入数据部分创建的：

```
logger.info('Updating the new player...');
await newPlayer.update({ coins: 700, goods: 700 });
logger.info('Updated the new player.');
logger.info(newPlayer.toJSON());
```

更多信息参考更新数据。

4.4.4.3.5 删除数据

下面的查询会删除在插入数据部分创建的 Player 记录，其 ID 为 6：

```
logger.info('Deleting the new player...');
await newPlayer.destroy();
const deletedNewPlayer = await playersModel.findByPk(6);
logger.info('Deleted the new player.');
logger.info(deletedNewPlayer?.toJSON());
```

更多信息参考删除数据。

4.4.4.4 下一步

- 关于 Sequelize 的更多使用方法，可以参考 Sequelize 的官方文档。
- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.4.4.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，或从 PingCAP 官方或 TiDB 社区获取支持。

4.4.5 使用 TypeORM 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。TypeORM 是当前流行的 Node.js ORM 框架之一。

本文档将展示如何使用平凯数据库和 TypeORM 来完成以下任务：

- 配置你的环境。
- 使用 TypeORM 连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.4.5.1 前置需求

为了能够顺利完成本教程，你需要提前：

- 在你的机器上安装 Node.js 16.x 或以上版本。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.4.5.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.4.5.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-nodejs-typeorm-quickstart.git
cd tidb-nodejs-typeorm-quickstart
```

4.4.5.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 typeorm 和 mysql2 依赖包）：

```
npm install
```

在现有的项目中安装依赖

在你现有的项目当中，你可以通过以下命令安装所需要的依赖包：

- typeorm：面向 Node.js 应用的 ORM 框架。
- mysql2：面向 Node.js 的 MySQL Driver 包。你也可以使用 mysql。
- dotenv：用于从 .env 文件中读取环境变量。
- typescript：TypeScript 编译器。
- ts-node：用于在不编译的情况下直接执行 TypeScript 代码。
- @types/node：用于提供 Node.js 的 TypeScript 类型定义。

```
npm install typeorm mysql2 dotenv --save
npm install @types/node ts-node typescript --save-dev
```

4.4.5.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 编辑 .env 文件，按照如下格式设置连接信息，将占位符 {} 替换为你的 TiDB 集群的连接参数值：

   ```
   TIDB_HOST={host}
   TIDB_PORT=4000
   TIDB_USER=root
   TIDB_PASSWORD={password}
   TIDB_DATABASE=test
   ```

   如果你在本机运行 TiDB，默认 Host 地址为 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.4.5.2.4 第 4 步：初始化表结构

运行以下命令，使用 TypeORM CLI 初始化数据库。TypeORM CLI 会根据 src/migrations 文件夹中的迁移文件生成 SQL 语句并执行。

```
npm run migration:run
```

预期的执行输出

下面的 SQL 语句创建了 players 表和 profiles 表，并通过外键关联了两个表。

```
query: SELECT VERSION() AS `version`
query: SELECT * FROM `INFORMATION_SCHEMA`.`COLUMNS` WHERE `TABLE_SCHEMA
` = 'test' AND `TABLE_NAME` = 'migrations'
query: CREATE TABLE `migrations` (`id` int NOT NULL AUTO_INCREMENT, `timestamp`
bigint NOT NULL, `name` varchar(255) NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoD
B
query: SELECT * FROM `test`.`migrations` `migrations` ORDER BY `id` DESC
0 migrations are already loaded in the database.
1 migrations were found in the source code.
1 migrations are new migrations must be executed.
query: START TRANSACTION
query: CREATE TABLE `profiles` (`player_id` int NOT NULL, `biography` text NOT NULL,
PRIMARY KEY (`player_id`)) ENGINE=InnoDB
query: CREATE TABLE `players` (`id` int NOT NULL AUTO_INCREMENT, `name` varchar(5
0) NOT NULL, `coins` decimal NOT NULL, `goods` int NOT NULL, `created_at` datetime
NOT NULL, `profilePlayerId` int NULL, UNIQUE INDEX `uk_players_on_name` (`name`), U
NIQUE INDEX `REL_b9666644b90ccc5065993425ef` (`profilePlayerId`), PRIMARY KEY (`i
d`)) ENGINE=InnoDB
query: ALTER TABLE `players` ADD CONSTRAINT `fk_profiles_on_player_id` FOREIGN K
EY (`profilePlayerId`) REFERENCES `profiles`(`player_id`) ON DELETE NO ACTION ON UP
DATE NO ACTION
query: INSERT INTO `test`.`migrations`(`timestamp`, `name`) VALUES (?, ?) -- PARAMETE
RS: [1693814724825,"Init1693814724825"]
Migration Init1693814724825 has been  executed successfully.
query: COMMIT
```

迁移文件是根据 src/entities 文件夹中定义的实体生成的。要了解如何在 TypeORM 中定义实体，请参考 TypeORM: Entities。

4.4.5.2.5 第 5 步：运行代码并查看结果

运行以下命令，执行示例代码：

```
npm start
```

预期输出结果：

如果连接成功，你的终端将会输出所连接集群的版本信息：

```
🔌 Connected to TiDB cluster! (TiDB version: 8.0.11-TiDB-v7.1.8-5.1)
🆕 Created a new player with ID 2.
ℹ️ Got Player 2: Player { id: 2, coins: 100, goods: 100 }
🔢 Added 50 coins and 50 goods to player 2, now player 2 has 100 coins and 150 goods.
🚮 Deleted 1 player data.
```

4.4.5.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-nodejs-typeorm-quickstart。

4.4.5.3.1 连接到平凯数据库

下面的代码使用环境变量中定义的连接选项来建立与 TiDB 集群的连接。

```
// src/dataSource.ts

// 加载 .env 文件中的环境变量到 process.env。
require('dotenv').config();

export const AppDataSource = new DataSource({
  type: "mysql",
  host: process.env.TIDB_HOST || '127.0.0.1',
  port: process.env.TIDB_PORT ? Number(process.env.TIDB_PORT) : 4000,
  username: process.env.TIDB_USER || 'root',
  password: process.env.TIDB_PASSWORD || '',
  database: process.env.TIDB_DATABASE || 'test',
  ssl: process.env.TIDB_ENABLE_SSL === 'true' ? {
    minVersion: 'TLSv1.2',
    ca: process.env.TIDB_CA_PATH ? fs.readFileSync(process.env.TIDB_CA_PATH) : undefine
```

```
d
  } : null,
  synchronize: process.env.NODE_ENV === 'development',
  logging: false,
  entities: [Player, Profile],
  migrations: [__dirname + "/migrations/**/*{.ts,.js}"],
});
```

4.4.5.3.2 插入数据

下面的代码创建了一条 Player 记录，并返回该记录的 id 字段，该字段由 TiDB 自动生成：

```
const player = new Player('Alice', 100, 100);
await this.dataSource.manager.save(player);
```

更多信息参考插入数据。

4.4.5.3.3 查询数据

下面的代码查询 ID 为 101 的 Player 记录，如果没有找到则返回 null：

```
const player: Player | null = await this.dataSource.manager.findOneBy(Player, {
  id: id
});
```

更多信息参考查询数据。

4.4.5.3.4 更新数据

下面的代码将 Player 记录的 goods 字段增加 50：

```
const player = await this.dataSource.manager.findOneBy(Player, {
  id: 101
});
player.goods += 50;
await this.dataSource.manager.save(player);
```

更多信息参考更新数据。

4.4.5.3.5 删除数据

下面的代码删除 ID 为 101 的 Player 记录：

```
await this.dataSource.manager.delete(Player, {
  id: 101
});
```

更多信息参考删除数据。

4.4.5.3.6 执行原生 SQL 查询

下面的代码执行原生 SQL 语句 (SELECT VERSION() AS tidb_version;) 并返回 TiDB 集群的版本信息：

```
const rows = await dataSource.query('SELECT VERSION() AS tidb_version;');
console.log(rows[0]['tidb_version']);
```

更多信息参考 TypeORM: DataSource API。

4.4.5.4 注意事项

4.4.5.4.1 外键约束

使用外键约束可以通过在数据库层面添加检查来确保数据的引用完整性。但是，在大数据量的场景下，这可能会导致严重的性能问题。

你可以通过使用 createForeignKeyConstraints 选项来控制在构建实体之间的关系时是否创建外键约束（默认值为 true）。

```
@Entity()
export class ActionLog {
    @PrimaryColumn()
    id: number

    @ManyToOne((type) => Person, {
        createForeignKeyConstraints: false,
    })
    person: Person
}
```

更多信息，请参考 TypeORM FAQ 和平凯数据库外键约束。

4.4.5.5 下一步

- 关于 TypeORM 的更多使用方法，可以参考 TypeORM 的官方文档。
- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.4.5.6 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

4.4.6 在 Next.js 中使用 mysql2 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库，mysql2 是当前流行的开源 Node.js Driver 之一。

本文档将展示如何在 Next.js 中使用平凯数据库和 mysql2 来完成以下任务：

- 配置你的环境。
- 使用 mysql2 驱动连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.4.6.1 前置需求

为了能够顺利完成本教程，你需要提前：

- 在你的机器上安装 Node.js 18.x 或以上版本。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.4.6.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

Note

完整代码及其运行方式，见代码仓库 tidb-nextjs-vercel-quickstart。

4.4.6.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone git@github.com:tidb-samples/tidb-nextjs-vercel-quickstart.git
cd tidb-nextjs-vercel-quickstart
```

4.4.6.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 mysql2 和 Next.js）：

```
npm install
```

4.4.6.2.3 第 3 步：配置连接信息

1. 运行以下命令，将 .env.example 复制并重命名为 .env：

   ```
   # Linux
   cp .env.example .env

   # Windows
   Copy-Item ".env.example" -Destination ".env"
   ```

2. 编辑 .env 文件，按照如下格式设置连接信息，将占位符 {} 替换为你的 TiDB 集群的连接参数值：

   ```
   TIDB_HOST='{tidb_server_host}'
   TIDB_PORT='4000'
   TIDB_USER='root'
   TIDB_PASSWORD='{password}'
   TIDB_DB_NAME='test'
   ```

如果你在本地运行 TiDB 集群，默认的主机地址是 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.4.6.2.4 第 4 步：运行代码并查看结果

1. 运行示例应用程序：

   ```
   npm run dev
   ```

2. 打开浏览器，在地址栏输入 http://localhost:3000，访问示例应用程序。请查看你的终端以获取实际的端口号，默认为 3000。

3. 点击 **RUN SQL** 执行示例代码。

4. 在终端中检查输出。如果输出类似于以下内容，则连接成功：

   ```
   {
     "results": [
       {
         "Hello World": "Hello World"
       }
     ]
   }
   ```

4.4.6.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-nextjs-vercel-quickstart。

4.4.6.3.1 连接到平凯数据库

下面的代码使用环境变量中定义的连接选项来建立与 TiDB 集群的连接。

```
// src/lib/tidb.js
import mysql from 'mysql2';

let pool = null;

export function connect() {
  return mysql.createPool({
```

```
    host: process.env.TIDB_HOST, // TiDB host
    port: process.env.TIDB_PORT || 4000, // TiDB port, default: 4000
    user: process.env.TIDB_USER, // TiDB user, for example: {prefix}.root
    password: process.env.TIDB_PASSWORD, // The password of TiDB user.
    database: process.env.TIDB_DATABASE || 'test', // TiDB database name, default: test
    ssl: {
      minVersion: 'TLSv1.2',
      rejectUnauthorized: true,
    },
    connectionLimit: 1, // Setting connectionLimit to "1" in a serverless function environme
nt optimizes resource usage, reduces costs, ensures connection stability, and enables sea
mless scalability.
    maxIdle: 1, // max idle connections, the default value is the same as `connectionLimit`
    enableKeepAlive: true,
  });
}

export function getPool() {
  if (!pool) {
    pool = createPool();
  }
  return pool;
}
```

4.4.6.3.2 插入数据

下面的查询会创建一条单独的 Player 记录，并返回一个 ResultSetHeader 对象：

```
const [rsh] = await pool.query('INSERT INTO players (coins, goods) VALUES (?, ?);', [100,
100]);
console.log(rsh.insertId);
```

更多信息参考插入数据。

4.4.6.3.3 查询数据

下面的查询会返回一条 Player 记录，其 ID 为 1：

```
const [rows] = await pool.query('SELECT id, coins, goods FROM players WHERE id = ?;',
[1]);
console.log(rows[0]);
```

更多信息参考查询数据。

4.4.6.3.4 更新数据

下面的查询会将 ID 为 1 的 Player 记录的 coins 和 goods 字段的值分别增加 50：

```
const [rsh] = await pool.query(
  'UPDATE players SET coins = coins + ?, goods = goods + ? WHERE id = ?;',
  [50, 50, 1]
);
console.log(rsh.affectedRows);
```

更多信息参考更新数据。

4.4.6.3.5 删除数据

下面的查询会删除一条 Player 记录，其 ID 为 1：

```
const [rsh] = await pool.query('DELETE FROM players WHERE id = ?;', [1]);
console.log(rsh.affectedRows);
```

更多信息参考删除数据。

4.4.6.4 注意事项

- 推荐使用连接池来管理数据库连接，以减少频繁建立和销毁连接所带来的性能开销。
- 为了避免 SQL 注入的风险，推荐使用预处理语句执行 SQL。
- 在不涉及大量复杂 SQL 语句的场景下，推荐使用 ORM 框架（例如：Sequelize、TypeORM 或 Prisma）来提升你的开发效率。

4.4.6.5 下一步

- 关于使用 ORM 框架和 Next.js 构建复杂应用程序的更多细节，可以参考 tidb-prisma-vercel-demo。
- 关于 mysql2 的更多使用方法，可以参考 mysql2 的官方文档。
- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。

- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.4.6.6 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，或从 PingCAP 官方或 TiDB 社区获取支持。

4.4.7 在 AWS Lambda 函数中使用 mysql2 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。AWS Lambda 函数是一个计算服务，mysql2 是当前流行的开源 Node.js Driver 之一。

本文档将展示如何在 AWS Lambda 函数中使用平凯数据库和 mysql2 来完成以下任务：

- 配置你的环境。
- 使用 mysql2 驱动连接到平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。
- 部署你的 AWS Lambda 函数。

4.4.7.1 前置需求

为了能够顺利完成本教程，你需要提前：

- 在你的机器上安装 Node.js 18.x 或以上版本。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。
- 准备一个具有管理员权限的 AWS IAM 用户。
- 在你的机器上安装 AWS CLI。
- 在你的机器上安装 AWS SAM CLI。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

如果你还没有 AWS 账户或用户，可以按照 Lambda 入门文档中的步骤来创建它们。

4.4.7.2 运行代码并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

> **Note**
>
> 完整代码及其运行方式，见代码仓库 tidb-samples/tidb-aws-lambda-quickstart。

4.4.7.2.1 第 1 步：克隆示例代码仓库到本地

运行以下命令，将示例代码仓库克隆到本地：

```
git clone git@github.com:tidb-samples/tidb-aws-lambda-quickstart.git
cd tidb-aws-lambda-quickstart
```

4.4.7.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 mysql2）：

```
npm install
```

4.4.7.2.3 第 3 步：配置连接信息

编辑 env.json 文件，按照如下格式设置连接信息，将占位符 {} 替换为你的 TiDB 集群的连接参数值：

```
{
  "Parameters": {
    "TIDB_HOST": "{tidb_server_host}",
    "TIDB_PORT": "4000",
    "TIDB_USER": "root",
    "TIDB_PASSWORD": "{password}"
  }
}
```

4.4.7.2.4 第 4 步：运行代码并查看结果

1. （前置需求）安装 AWS SAM CLI。
2. 构建应用程序包：

   ```
   npm run build
   ```

3. 调用示例 Lambda 函数：

   ```
   sam local invoke --env-vars env.json -e events/event.json "tidbHelloWorldFunction"
   ```

4. 检查终端中的输出。如果输出类似于以下内容，则表示连接成功：

   ```
   {"statusCode":200,"body":"{\"results\":[{\"Hello World\":\"Hello World\"}]}"}
   ```

确认连接成功后，你可以按照部署 AWS Lambda 函数中的步骤进行部署。

4.4.7.3 部署 AWS Lambda 函数

你可以通过 SAM CLI 或 AWS Lambda 控制台部署 AWS Lambda 函数。

4.4.7.3.1 通过 SAM CLI 部署（推荐）

1. （前置需求）安装 AWS SAM CLI。
2. 构建应用程序包：

   ```
   npm run build
   ```

3. 更新 template.yml 文件中的环境变量：

   ```
   Environment:
     Variables:
       TIDB_HOST: {tidb_server_host}
       TIDB_PORT: 4000
       TIDB_USER: {prefix}.root
       TIDB_PASSWORD: {password}
   ```

4. 参考使用短期凭证进行身份验证文档，设置 AWS 环境变量：

```
export AWS_ACCESS_KEY_ID={your_access_key_id}
export AWS_SECRET_ACCESS_KEY={your_secret_access_key}
export AWS_SESSION_TOKEN={your_session_token}
```

5. 部署 AWS Lambda 函数：

```
sam deploy --guided

# Example:

# Configuring SAM deploy
# ======================

#       Looking for config file [samconfig.toml] :  Not found

#       Setting default arguments for 'sam deploy'
#       =========================================
#       Stack Name [sam-app]: tidb-aws-lambda-quickstart
#       AWS Region [us-east-1]:
#       #Shows you resources changes to be deployed and require a 'Y' to initiate deploy
#       Confirm changes before deploy [y/N]:
#       #SAM needs permission to be able to create roles to connect to the resources in your template
#       Allow SAM CLI IAM role creation [Y/n]:
#       #Preserves the state of previously provisioned resources when an operation fails
#       Disable rollback [y/N]:
#       tidbHelloWorldFunction may not have authorization defined, Is this okay? [y/N]: y
#       tidbHelloWorldFunction may not have authorization defined, Is this okay? [y/N]: y
#       tidbHelloWorldFunction may not have authorization defined, Is this okay? [y/N]: y
#       tidbHelloWorldFunction may not have authorization defined, Is this okay? [y/N]: y
#       Save arguments to configuration file [Y/n]:
#       SAM configuration file [samconfig.toml]:
#       SAM configuration environment [default]:

#       Looking for resources needed for deployment:
#       Creating the required resources...
#       Successfully created!
```

4.4.7.3.2 通过网页控制台部署

1. 构建应用程序包：

   npm run build

   ```
   # Bundle for AWS Lambda
   # =====================
   # dist/index.zip
   ```

2. 访问 AWS Lambda 控制台。

3. 按照使用 Node.js 构建 Lambda 函数中的步骤创建一个 Node.js Lambda 函数。

4. 按照 Lambda 部署程序包中的步骤，上传 dist/index.zip 文件。

5. 在 Lambda 函数中复制并配置相应的连接字符串。

   1. 在 Lambda 控制台的**函数**页面中，选择**配置** > **环境变量**。

   2. 点击**编辑**。

   3. 按照以下步骤添加数据库访问凭证：

      - 选择**添加环境变量**，然后在**键**中输入 TIDB_HOST，在**值**中输入主机名。
      - 选择**添加环境变量**，然后在**键**中输入 TIDB_PORT，在**值**中输入端口号（默认 4000）。
      - 选择**添加环境变量**，然后在**键**中输入 TIDB_USER，在**值**中输入用户名。
      - 选择**添加环境变量**，然后在**键**中输入 TIDB_PASSWORD，在**值**中输入数据库的密码。
      - 点击**保存**。

4.4.7.4 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-aws-lambda-quickstart。

4.4.7.4.1 连接到平凯数据库

下面的代码使用环境变量中定义的连接选项来建立与 TiDB 集群的连接。

```
// lib/tidb.ts
import mysql from 'mysql2';

let pool: mysql.Pool | null = null;

function connect() {
  return mysql.createPool({
    host: process.env.TIDB_HOST, // TiDB host
    port: process.env.TIDB_PORT ? Number(process.env.TIDB_PORT) : 4000, // TiDB port, d
efault: 4000
    user: process.env.TIDB_USER, // TiDB user, for example: {prefix}.root
    password: process.env.TIDB_PASSWORD, // TiDB password
    database: process.env.TIDB_DATABASE || 'test', // TiDB database name, default: test
    ssl: {
      minVersion: 'TLSv1.2',
      rejectUnauthorized: true,
    },
    connectionLimit: 1, // Setting connectionLimit to "1" in a serverless function environme
nt optimizes resource usage, reduces costs, ensures connection stability, and enables sea
mless scalability.
    maxIdle: 1, // max idle connections, the default value is the same as `connectionLimit`
    enableKeepAlive: true,
  });
}

export function getPool(): mysql.Pool {
  if (!pool) {
    pool = connect();
  }
  return pool;
}
```

4.4.7.4.2 插入数据

下面的查询会创建一条单独的 Player 记录，并返回一个 ResultSetHeader 对象：

```
const [rsh] = await pool.query('INSERT INTO players (coins, goods) VALUES (?, ?);', [100,
100]);
console.log(rsh.insertId);
```

更多信息参考插入数据。

4.4.7.4.3 查询数据

下面的查询会返回一条 Player 记录，其 ID 为 1：

```
const [rows] = await pool.query('SELECT id, coins, goods FROM players WHERE id = ?;',
[1]);
console.log(rows[0]);
```

更多信息参考查询数据。

4.4.7.4.4 更新数据

下面的查询会将 ID 为 1 的 Player 记录的 coins 和 goods 字段的值分别增加 50：

```
const [rsh] = await pool.query(
  'UPDATE players SET coins = coins + ?, goods = goods + ? WHERE id = ?;',
  [50, 50, 1]
);
console.log(rsh.affectedRows);
```

更多信息参考更新数据。

4.4.7.4.5 删除数据

下面的查询会删除一条 Player 记录，其 ID 为 1：

```
const [rsh] = await pool.query('DELETE FROM players WHERE id = ?;', [1]);
console.log(rsh.affectedRows);
```

更多信息参考删除数据。

4.4.7.5 注意事项

- 推荐使用连接池来管理数据库连接，以减少频繁建立和销毁连接所带来的性能开销。
- 为了避免 SQL 注入的风险，推荐使用预处理语句执行 SQL。
- 在不涉及大量复杂 SQL 语句的场景下，推荐使用 ORM 框架（例如：Sequelize、TypeORM 或 Prisma）来提升你的开发效率。
- 如需为你的应用程序构建一个 RESTful API，建议将 AWS Lambda 与 Amazon API Gateway 结合使用。

4.4.7.6 下一步

- 关于在 AWS Lambda 函数中使用 TiDB 的更多细节，可以参考 TiDB-Lambda-integration/aws-lambda-bookstore 示例程序。你也可以使用 AWS API Gateway 为你的应用程序构建 RESTful API。
- 关于 mysql2 的更多使用方法，可以参考 mysql2 的官方文档。
- 关于 AWS Lambda 的更多使用方法，可以参考 AWS Lambda 开发者指南。
- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.4.7.7 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，或从 PingCAP 官方或 TiDB 社区获取支持。

4.5 Ruby

4.5.1 使用 mysql2 连接平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。mysql2 是 Ruby 中最受欢迎的 MySQL 驱动之一。

本文档将展示如何使用平凯数据库和 mysql2 来完成以下任务：

- 设置你的环境。
- 使用 mysql2 连接你的平凯数据库集群。
- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.5.1.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Ruby 3.0 或以上版本。
- 在你的机器上安装 Bundler。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.5.1.2 运行示例应用程序并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.5.1.2.1 第 1 步：克隆示例代码仓库到本地

在你的终端窗口中运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-ruby-mysql2-quickstart.git
cd tidb-ruby-mysql2-quickstart
```

4.5.1.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 mysql2 和 dotenv 依赖包）：

```
bundle install
```

为现有项目安装依赖

对于你的现有项目，运行以下命令安装这些包：

```
bundle add mysql2 dotenv
```

4.5.1.2.3 第 3 步：配置连接信息

1. 运行以下命令复制 .env.example 并将其重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 编辑 .env 文件，按照以下方式设置环境变量，并将占位符 {}替换为你的 TiDB 集群的连接参数：

   ```
   DATABASE_HOST={host}
   DATABASE_PORT=4000
   DATABASE_USER={user}
   DATABASE_PASSWORD={password}
   DATABASE_NAME=test
   ```

   如果你在本地运行 TiDB，那么默认的主机地址是 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.5.1.2.4 第 4 步：运行代码并查看结果

运行以下命令来执行示例代码：

```
ruby app.rb
```

如果连接成功，你的终端将会输出所连接集群的版本信息：

```
🔌 Connected to TiDB cluster! (TiDB version: 8.0.11-TiDB-v7.1.8-5.1)
⌛ Loading sample game data...
✅ Loaded sample game data.

🆕 Created a new player with ID 12.
ℹ️ Got Player 12: Player { id: 12, coins: 100, goods: 100 }
🔢 Added 50 coins and 50 goods to player 12, updated 1 row.
🚮 Deleted 1 player data.
```

4.5.1.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-ruby-mysql2-quickstart。

4.5.1.3.1 连接到平凯数据库

下面的代码使用环境变量中定义的连接选项来建立与 TiDB 集群的连接。

```ruby
require 'dotenv/load'
require 'mysql2'
Dotenv.load # 从 .env 文件中加载环境变量

options = {
  host: ENV['DATABASE_HOST'] || '127.0.0.1',
  port: ENV['DATABASE_PORT'] || 4000,
  username: ENV['DATABASE_USER'] || 'root',
  password: ENV['DATABASE_PASSWORD'] || '',
  database: ENV['DATABASE_NAME'] || 'test'
}
options.merge(ssl_mode: :verify_identity) unless ENV['DATABASE_ENABLE_SSL'] == 'fal
se'
options.merge(sslca: ENV['DATABASE_SSL_CA']) if ENV['DATABASE_SSL_CA']
client = Mysql2::Client.new(options)
```

4.5.1.3.2 插入数据

以下查询创建一个具有两个字段的 player，并返回 last_insert_id：

```ruby
def create_player(client, coins, goods)
  result = client.query(
    "INSERT INTO players (coins, goods) VALUES (#{coins}, #{goods});"
  )
  client.last_id
end
```

更多信息，请参考插入数据。

4.5.1.3.3 查询数据

以下查询通过 ID 返回特定 player 的记录：

```
def get_player_by_id(client, id)
  result = client.query(
    "SELECT id, coins, goods FROM players WHERE id = #{id};"
  )
  result.first
end
```

更多信息，请参考查询数据。

4.5.1.3.4 更新数据

以下查询通过 ID 更新特定 player 的记录：

```
def update_player(client, player_id, inc_coins, inc_goods)
  result = client.query(
    "UPDATE players SET coins = coins + #{inc_coins}, goods = goods + #{inc_goods} WH
ERE id = #{player_id};"
  )
  client.affected_rows
end
```

更多信息，请参考更新数据。

4.5.1.3.5 删除数据

以下查询删除特定 player 的记录：

```
def delete_player_by_id(client, id)
  result = client.query(
    "DELETE FROM players WHERE id = #{id};"
  )
  client.affected_rows
end
```

更多信息，请参考删除数据。

4.5.1.4 最佳实践

默认情况下，mysql2 gem 可以按照特定的顺序搜索现有的 CA 证书，直到找到相应的文件。

1. 对于 Debian、Ubuntu、Gentoo、Arch 或 Slackware，证书的默认存储路径为 /etc/ssl/certs/ca-certificates.crt。
2. 对于 RedHat、Fedora、CentOS、Mageia、Vercel 或 Netlify，证书的默认存储路径为 /etc/pki/tls/certs/ca-bundle.crt。
3. 对于 OpenSUSE，证书的默认存储路径为 /etc/ssl/ca-bundle.pem。
4. 对于 macOS 或 Alpine（docker 容器），证书的默认存储路径为 /etc/ssl/cert.pem。

尽管可以手动指定 CA 证书路径，但在多环境部署场景中这可能会引起不必要的麻烦，因为不同的机器和环境可能存储 CA 证书的位置不同。因此，建议将 sslca 设置为 nil，方便在不同环境中灵活且方便地部署。

4.5.1.5 下一步

- 从 mysql2 的文档中了解更多关于 mysql2 驱动的使用情况。
- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.5.1.6 需要帮助?

在 AskTUG 论坛上提问。

4.5.2 使用 Rails 框架和 ActiveRecord ORM 连接平凯数据库

平凯数据库是一个兼容 MySQL 的数据库，Rails 是用 Ruby 编写的流行的 Web 框架，而 ActiveRecord ORM 是 Rails 中的对象关系映工具。

本文档将展示如何使用平凯数据库和 Rails 来完成以下任务：

- 设置你的环境。
- 使用 Rails 连接你的平凯数据库集群。

- 构建并运行你的应用程序。你也可以参考示例代码片段，完成基本的 CRUD 操作。

4.5.2.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Ruby 3.0 或以上版本。
- 在你的机器上安装 Bundler。
- 在你的机器上安装 Git。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

4.5.2.2 运行示例应用程序并连接到平凯数据库

本小节演示如何运行示例应用程序的代码，并连接到 TiDB。

4.5.2.2.1 第 1 步：克隆示例代码仓库到本地

在你的终端窗口中运行以下命令，将示例代码仓库克隆到本地：

```
git clone https://github.com/tidb-samples/tidb-ruby-rails-quickstart.git
cd tidb-ruby-rails-quickstart
```

4.5.2.2.2 第 2 步：安装依赖

运行以下命令，安装示例代码所需要的依赖（包括 mysql2 和 dotenv 依赖包）：

```
bundle install
```

为现有项目安装依赖

对于你的现有项目，运行以下命令以安装这些包：

```
bundle add mysql2 dotenv
```

4.5.2.2.3 第 3 步：配置连接信息

1. 运行以下命令复制 .env.example 并将其重命名为 .env：

   ```
   cp .env.example .env
   ```

2. 编辑 .env 文件，按照以下方式设置 DATABASE_URL 环境变量，并将 {user}、{password}、{host}、{port} 和 {database} 替换为你自己的 TiDB 连接信息：

   ```
   DATABASE_URL=mysql2://{user}:{password}@{host}:{port}/{database}
   ```

   如果你在本地运行 TiDB，那么默认的主机地址是 127.0.0.1，密码为空。

3. 保存 .env 文件。

4.5.2.2.4 第 4 步：运行代码并查看结果

1. 创建数据库和表：

   ```
   bundle exec rails db:create
   bundle exec rails db:migrate
   ```

2. 填充示例数据：

   ```
   bundle exec rails db:seed
   ```

3. 运行以下命令执行示例代码：

   ```
   bundle exec rails runner ./quickstart.rb
   ```

如果连接成功，你的终端将会输出所连接集群的版本信息：

```
🔌 Connected to TiDB cluster! (TiDB version: 8.0.11-TiDB-v7.1.8-5.1)
⌛ Loading sample game data...
✅ Loaded sample game data.

🆕 Created a new player with ID 12.
ℹ️ Got Player 12: Player { id: 12, coins: 100, goods: 100 }
🔢 Added 50 coins and 50 goods to player 12, updated 1 row.
🚮 Deleted 1 player data.
```

4.5.2.3 示例代码片段

你可参考以下关键代码片段，完成自己的应用开发。

完整代码及其运行方式，见代码仓库 tidb-samples/tidb-ruby-rails-quickstart。

4.5.2.3.1 连接到平凯数据库

config/database.yml 中的以下代码使用 DATABASE_URL 系统变量的配置连接到 TiDB：

```
default: &default
  adapter: mysql2
  encoding: utf8mb4
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  url: <%= ENV["DATABASE_URL"] %>

development:
  <<: *default

test:
  <<: *default
  database: quickstart_test

production:
  <<: *default
```

4.5.2.3.2 插入数据

以下查询创建了一个具有两个字段的 Player，并返回创建的 Player 对象：

```
new_player = Player.create!(coins: 100, goods: 100)
```

更多信息，请参考插入数据。

4.5.2.3.3 查询数据

以下通过 ID 查询返回特定 Player 的记录：

```
player = Player.find_by(id: new_player.id)
```

更多信息，请参考查询数据。

4.5.2.3.4 更新数据

以下查询更新特定 Player 对象：

```
player.update(coins: 50, goods: 50)
```

更多信息，请参考更新数据。

4.5.2.3.5 删除数据

以下查询删除特定 Player 对象：

```
player.destroy
```

更多信息，请参考删除数据。

4.5.2.4 最佳实践

默认情况下，mysql2 gem 可以按照特定的顺序搜索现有的 CA 证书，直到找到相应的文件。

1. 对于 Debian、Ubuntu、Gentoo、Arch 或 Slackware，证书的默认存储路径为 /etc/ssl/certs/ca-certificates.crt。
2. 对于 RedHat、Fedora、CentOS、Mageia、Vercel 或 Netlify，证书的默认存储路径为 /etc/pki/tls/certs/ca-bundle.crt。
3. 对于 OpenSUSE，证书的默认存储路径为 /etc/ssl/ca-bundle.pem。
4. 对于 macOS 或 Alpine（docker 容器），证书的默认存储路径为 /etc/ssl/cert.pem。

尽管可以手动指定 CA 证书路径，但在多环境部署场景中这可能会引起不必要的麻烦，因为不同的机器和环境可能存储 CA 证书的位置不同。因此，建议将 sslca 设置为 nil，方便在不同环境中灵活且方便地部署。

4.5.2.5 下一步

- 从 ActiveRecord 文档中了解更多关于 ActiveRecord ORM 的用法。

- 你可以继续阅读开发者文档的其它章节来获取更多 TiDB 应用开发的最佳实践。例如：插入数据，更新数据，删除数据，单表读取，事务，SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程支持，并在考试后提供相应的资格认证。

4.5.2.6 需要帮助?

在 AskTUG 论坛上提问。

5 连接到平凯数据库

5.1 GUI 数据库工具

5.1.1 使用 MySQL Workbench 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。MySQL Workbench 是为 MySQL 数据库用户提供的 GUI 工具集。

警告

- 尽管由于 MySQL Workbench 兼容 MySQL，你可以使用 MySQL Workbench 连接到平凯数据库，但 MySQL Workbench 并不完全支持平凯数据库。由于 MySQL Workbench 将平凯数据库视为 MySQL，因此在使用过程中可能会遇到一些问题。
- 建议使用平凯数据库完全支持的其它 GUI 工具进行连接，例如 DataGrip，DBeaver 以及 VS Code SQLTools。平凯数据库完全支持的 GUI 工具的完整列表，参考平凯数据库支持的第三方工具。

在本文档中，你可以学习如何使用 MySQL Workbench 连接到平凯数据库集群。

5.1.1.1 前置需求

为了能够顺利完成本文中的操作，你需要：

- MySQL Workbench **8.0.31** 或以上版本。

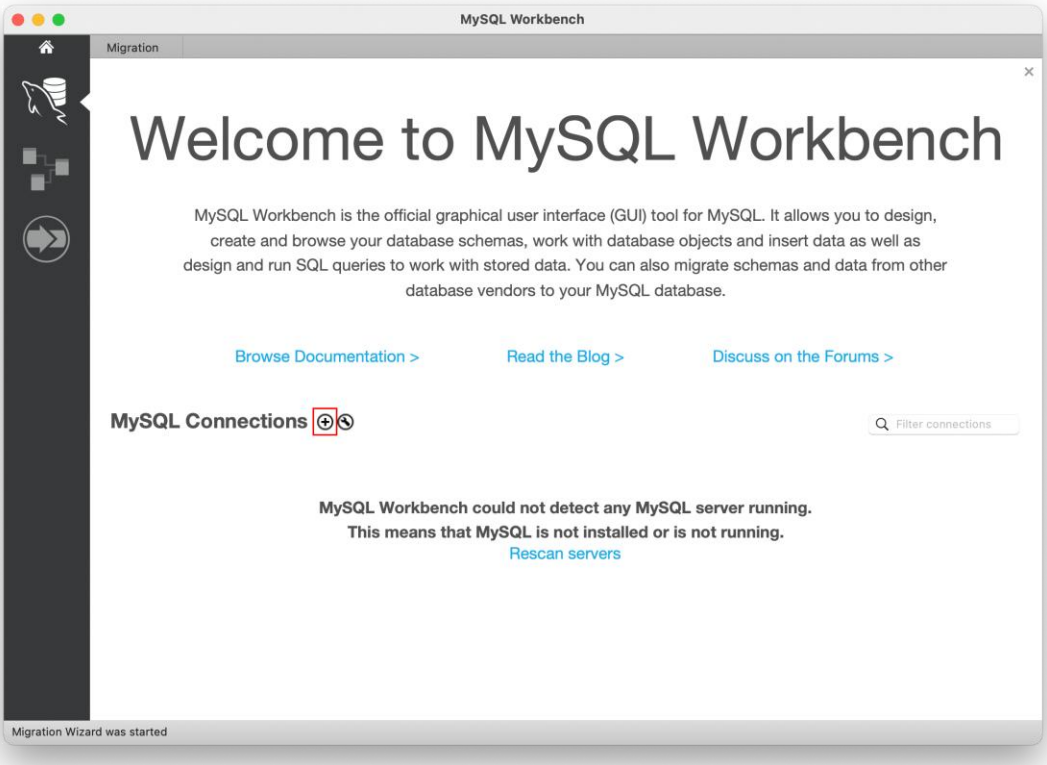
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

5.1.1.2 连接到平凯数据库

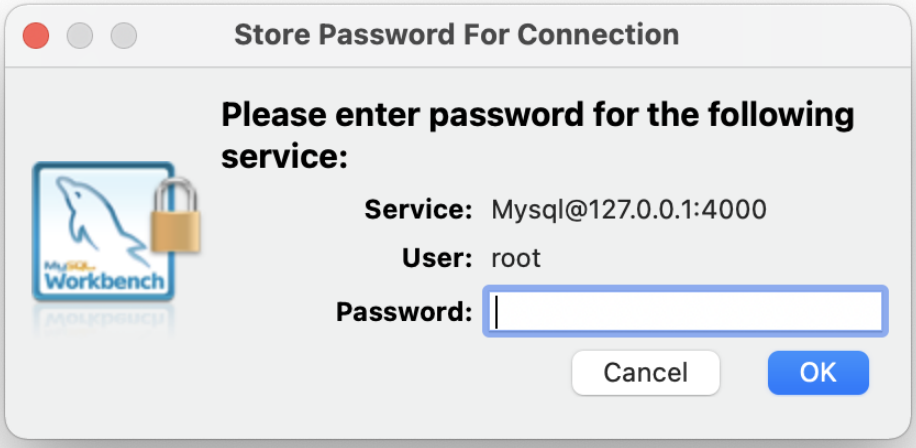
1. 启动 MySQL Workbench，并点击 **MySQL Connections** 标题旁边的 +。



MySQL Workbench: add new connection

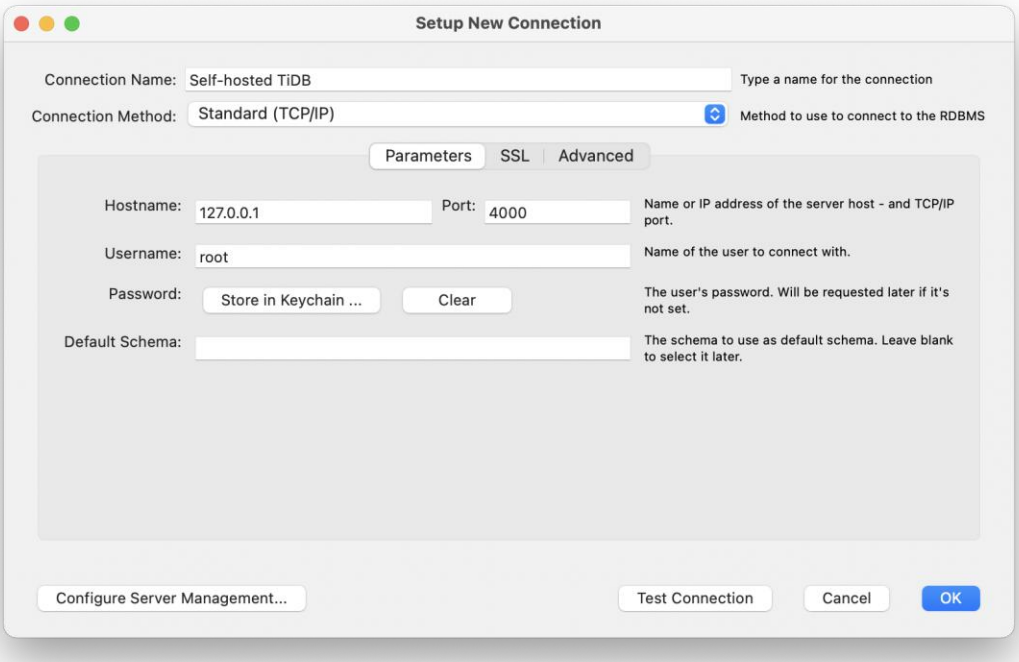
2. 在 **Setup New Connection** 对话框中，配置以下连接参数：
    - **Connection Name**：为该连接指定一个有意义的名称。
    - **Hostname**：输入本地部署 TiDB 集群的 IP 地址或域名。
    - **Port**：输入本地部署 TiDB 集群的端口号。

- **Username**：输入用于连接到 TiDB 的用户名。
- **Password**：点击 **Store in Keychain ...**，输入用于连接 TiDB 集群的密码，然后点击 **OK** 保存密码。



MySQL Workbench: store the password of TiDB Self-Managed in keychain

下图显示了连接参数的示例：



MySQL Workbench: configure connection settings for TiDB Self-Managed

3. 点击 **Test Connection** 以验证与本地部署 TiDB 集群的连接。
4. 如果连接测试成功，你可以看到 **Successfully made the MySQL connection** 信息。点击 **OK** 保存连接配置。

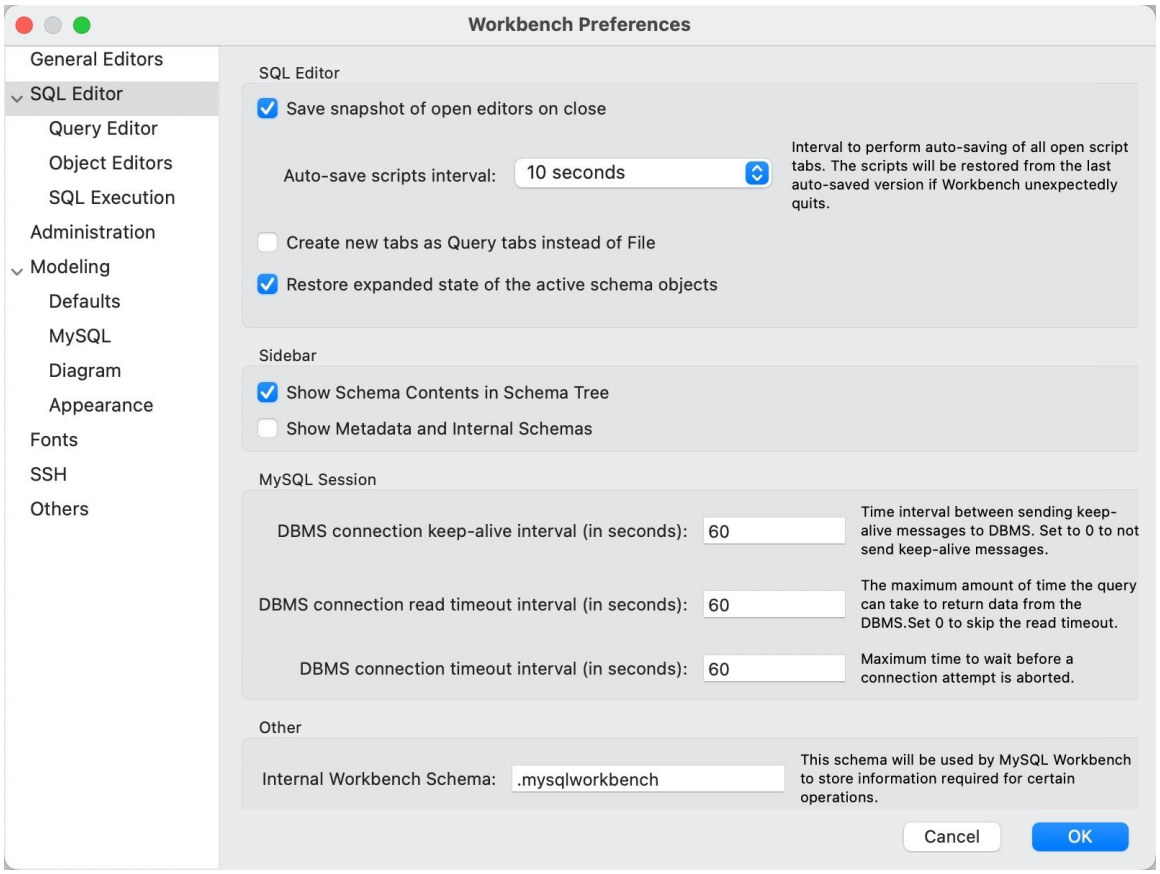
5.1.1.3 常见问题

5.1.1.3.1 如何处理连接超时错误 “Error Code: 2013. Lost connection to MySQL server during query”?

这个错误表示查询执行时间超过了超时限制。要解决这个问题，可以按照以下步骤调整超时设置：

1. 启动 MySQL Workbench 并打开 **Workbench Preferences** 页面。

2. 在 **SQL Editor** > **MySQL Session** 部分，调整 **DBMS connection read timeout interval (in seconds)** 的设置。该字段控制了 MySQL Workbench 在断开与服务器的连接之前查询可以执行的最长时间（以秒为单位）。



MySQL Workbench: adjust timeout option in SQL Editor settings

更多信息，可以参考 MySQL Workbench 常见问题。

5.1.1.4 下一步

- 关于 MySQL Workbench 的更多使用方法，可以参考 MySQL Workbench 官方文档。
- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。

- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程 支持，并在考试后提供相应的资格认证。

5.1.1.5 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，或从 PingCAP 官方或 TiDB 社区获取支持。

5.1.2 使用 Navicat 连接到平凯数据库

平凯数据库是一个兼容 MySQL 的数据库。Navicat 是为数据库用户提供的 GUI 工具集。本教程使用 Navicat Premium 工具连接平凯数据库。

在本文档中，你可以学习如何使用 Navicat 连接到平凯数据库集群。

5.1.2.1 前置需求

为了能够顺利完成本文中的操作，你需要：

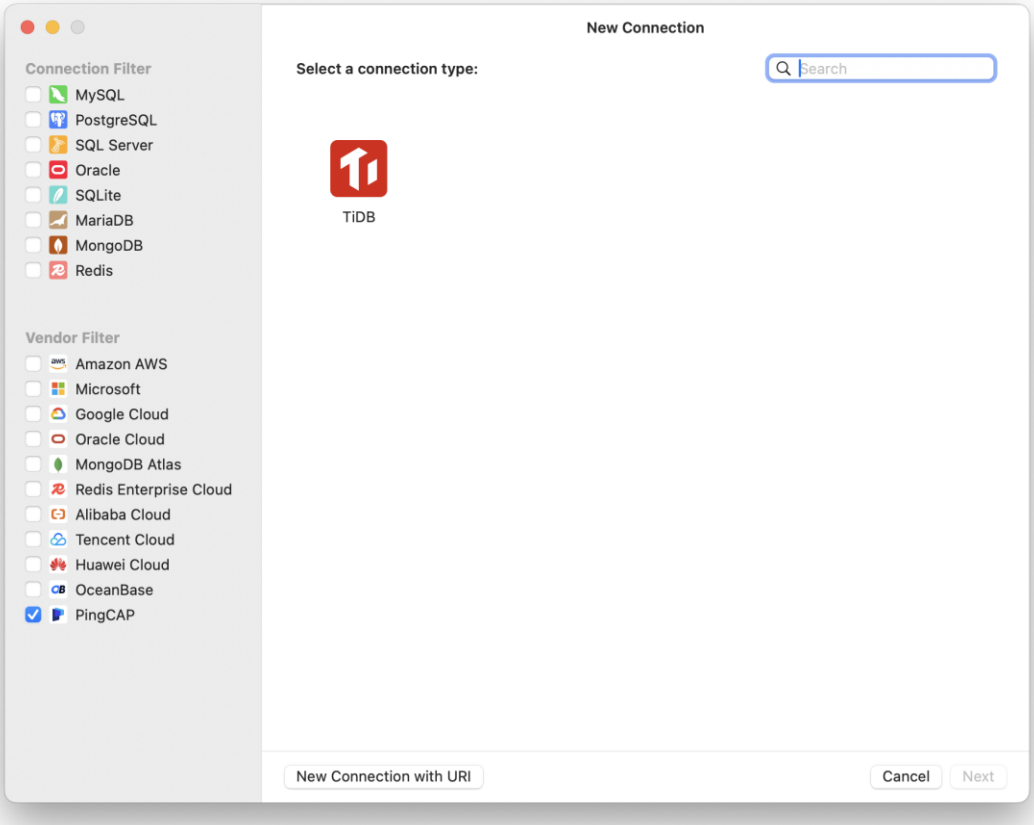
- Navicat Premium **17.1.6** 或以上版本。
- 一个 Navicat Premium 的付费账号。
- 准备一个 TiDB 集群。

如果你还没有 TiDB 集群，可以按如下方式创建一个：

- 在本地快速部署平凯数据库测试集群
- 部署 TiDB 正式生产集群，创建一个本地集群

5.1.2.2 连接到平凯数据库

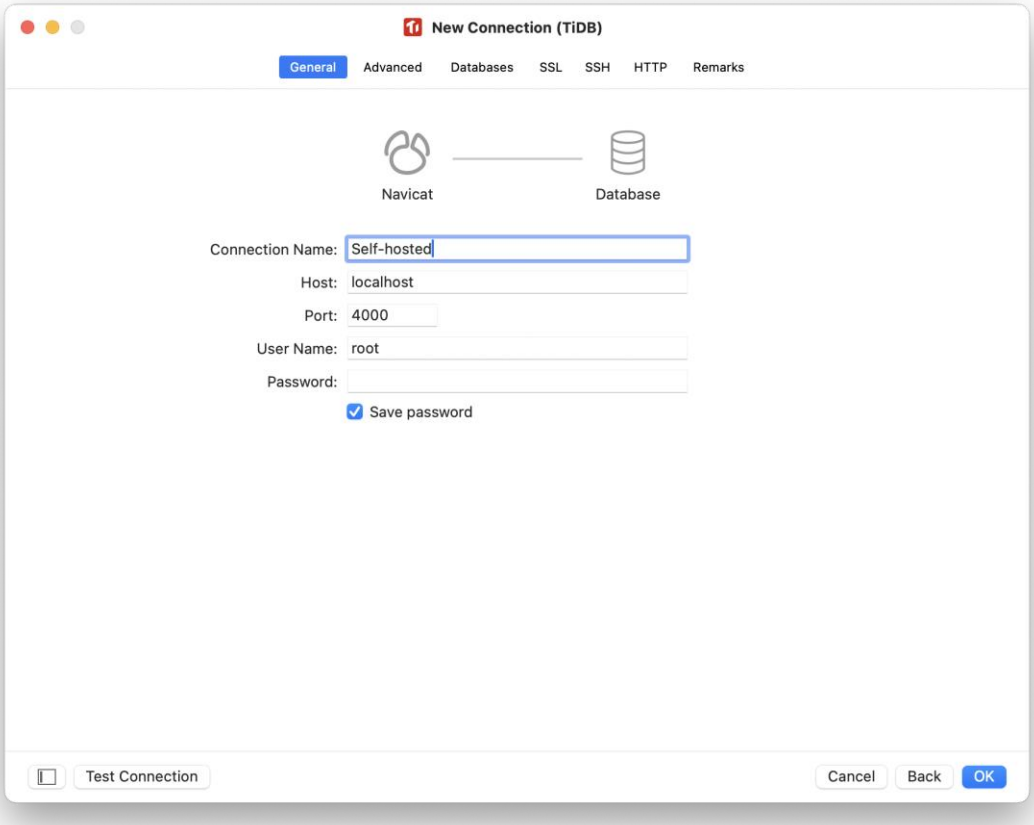
1. 启动 Navicat Premium，点击左上角的 **Connection**，在 **Vendor Filter** 中勾选 **PingCAP**，并双击右侧面板中的 **TiDB**。



Navicat: add new connection

2. 在 **New Connection (TiDB)** 对话框中，配置以下连接参数：

    - **Connection Name**：为该连接指定一个有意义的名称。
    - **Host**：输入本地部署 TiDB 集群的 IP 地址或域名。
    - **Port**：输入本地部署 TiDB 集群的端口号。
    - **User Name**：输入用于连接到 TiDB 的用户名。
    - **Password**：输入用于连接到 TiDB 的密码。



Navicat: configure connection general panel for self-hosted TiDB

3. 点击 **Test Connection** 以验证与本地部署 TiDB 集群的连接。
4. 如果连接测试成功，你可以看到 **Connection Successful** 信息。点击 **OK** 完成连接配置。

5.1.2.3 下一步

- 你可以继续阅读开发者文档，以获取更多关于 TiDB 应用开发的最佳实践。例如：插入数据、更新数据、删除数据、单表读取、事务、SQL 性能优化等。
- 如果你更倾向于参与课程进行学习，我们也提供专业的 TiDB 开发者课程 支持，并在考试后提供相应的资格认证。

5.1.2.4 需要帮助?

如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，或从 PingCAP 官方或 TiDB 社区获取支持。

5.2 选择驱动或 ORM 框架

注意：

平凯数据库支持等级说明：

- **Full**：表明平凯数据库已经兼容该工具的绝大多数功能，并且在该工具的新版本中对其保持兼容。PingCAP 将定期地对平凯数据库支持的第三方工具中的新版本进行兼容性测试。
- **Compatible**：表明由于该工具已适配 MySQL，而平凯数据库高度兼容 MySQL 协议，因此 TiDB 可以兼容该工具的大部分功能。但 PingCAP 并未对该工具作出完整的兼容性验证，有可能出现一些意外的行为。

关于平凯数据库支持的更多第三方工具，你可以查看平凯数据库支持的第三方工具。

平凯数据库兼容 MySQL 的协议，但存在部分与 MySQL 不兼容或有差异的特性。

5.2.1 Java

本节介绍 Java 语言的 Driver 及 ORM 的使用方式。

5.2.1.1 Java Drivers

支持等级：**Full**

按照 MySQL 文档中的说明下载并配置 Java JDBC 驱动程序即可使用。对于 v7.1.0 及以上版本，建议使用 MySQL Connector/J 最新 GA 版本。

有关一个完整的实例应用程序，可参阅平凯数据库和 JDBC 的简单 CRUD 应用程序。

支持等级：**Full**

TiDB-JDBC 是基于 MySQL 8.0.29 的定制版本。TiDB-JDBC 基于 MySQL 官方 8.0.29 版本编译，修复了原 JDBC 在 prepare 模式下多参数、多字段 EOF 的错误，并新增 TiCDC snapshot 自动维护和 SM3 认证插件等功能。

基于 SM3 的认证仅在 TiDB 的 TiDB-JDBC 中支持。

如果你使用的是 Maven，请将以下内容添加到你的 <dependencies></dependencies>：

```
<dependency>
  <groupId>io.github.lastincisor</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.29-tidb-1.0.0</version>
</dependency>
```

如果你需要使用 SM3 认证，请将以下内容添加到你的 <dependencies></dependencies>：

```
<dependency>
  <groupId>io.github.lastincisor</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.29-tidb-1.0.0</version>
</dependency>
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcprov-jdk15on</artifactId>
    <version>1.67</version>
</dependency>
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcpkix-jdk15on</artifactId>
    <version>1.67</version>
</dependency>
```

如果你使用的是 Gradle，请将以下内容添加到你的 dependencies：

```
implementation group: 'io.github.lastincisor', name: 'mysql-connector-java', version: '8.0.29-tidb-1.0.0'
```

```
implementation group: 'org.bouncycastle', name: 'bcprov-jdk15on', version: '1.67'
implementation group: 'org.bouncycastle', name: 'bcpkix-jdk15on', version: '1.67'
```

5.2.1.2 Java ORM 框架

注意：

- Hibernate 当前不支持嵌套事务。
- TiDB 支持 Savepoint，你可在 @Transactional 中使用 Propagation.NESTED 事务传播选项，即 @Transactional(propagation = Propagation.NESTED)。

支持等级：**Full**

你可以使用 Gradle 或 Maven 获取你的应用程序的所有依赖项，且会帮你下载依赖项的间接依赖，而无需你手动管理复杂的依赖关系。注意，只有 Hibernate 6.0.0.Beta2 及以上版本才支持 TiDB 方言。

如果你使用的是 Maven，请将以下内容添加到你的 <dependencies></dependencies>：

```
<dependency>
   <groupId>org.hibernate.orm</groupId>
   <artifactId>hibernate-core</artifactId>
   <version>6.2.3.Final</version>
</dependency>

<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>8.0.33</version>
</dependency>
```

如果你使用的是 Gradle，请将以下内容添加到你的 dependencies：

```
implementation 'org.hibernate:hibernate-core:6.2.3.Final'
implementation 'mysql:mysql-connector-java:8.0.33'
```

- 有关原生 Java 使用 Hibernate 进行 TiDB 应用程序构建的例子，可参阅平凯数据库和 Hibernate 的简单 CRUD 应用程序。
- 有关 Spring 使用 Spring Data JPA、Hibernate 进行 TiDB 应用程序构建的例子，可参阅使用 Spring Boot 构建平凯数据库应用程序。

额外的，你需要在 Hibernate 配置文件中指定 TiDB 方言 org.hibernate.dialect.TiDBDialect，此方言在 Hibernate 6.0.0.Beta2 以上才可支持。若你无法升级 Hibernate 版本，那么请你直接使用 MySQL 5.7 的方言 org.hibernate.dialect.MySQL57Dialect。但这可能造成不可预料的使用结果，及部分 TiDB 特有特性的缺失，如：序列等。

支持等级：**Full**

你可以使用 Gradle 或 Maven 获取应用程序的所有依赖项包括间接依赖，无需手动管理复杂的依赖关系。

如果你使用的是 Maven，请将以下内容添加到你的 <dependencies></dependencies>：

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.5.13</version>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.33</version>
</dependency>
```

如果你使用的是 Gradle，请将以下内容添加到你的 dependencies：

```
implementation 'org.mybatis:mybatis:3.5.13'
implementation 'mysql:mysql-connector-java:8.0.33'
```

使用 MyBatis 进行 TiDB 应用程序构建的例子，可参阅平凯数据库和 MyBatis 的简单 CRUD 应用程序。

5.2.1.3 Java 客户端负载均衡

tidb-loadbalance

支持等级：**Full**

tidb-loadbalance 是应用端的负载均衡组件。通过 tidb-loadbalance，你可以实现自动维护 TiDB server 的节点信息，根据节点信息使用 tidb-loadbalance 策略在客户端分发 JDBC 连接。客户端应用与 TiDB server 之间使用 JDBC 直连，性能高于使用负载均衡组件。

目前 tidb-loadbalance 已实现轮询、随机、权重等负载均衡策略。

> **注意：**
>
> tidb-loadbalance 需配合 mysql-connector-j 一起使用。

如果你使用的是 Maven，请将以下内容添加到你的 <dependencies></dependencies>：

```
<dependency>
  <groupId>io.github.lastincisor</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.29-tidb-1.0.0</version>
</dependency>
<dependency>
  <groupId>io.github.lastincisor</groupId>
  <artifactId>tidb-loadbalance</artifactId>
  <version>0.0.5</version>
</dependency>
```

如果你使用的是 Gradle，请将以下内容添加到你的 dependencies：

```
implementation group: 'io.github.lastincisor', name: 'mysql-connector-java', version: '8.0.29-tidb-1.0.0'
implementation group: 'io.github.lastincisor', name: 'tidb-loadbalance', version: '0.0.5'
```

5.2.2 Golang

本节介绍 Golang 语言的 Driver 及 ORM 的使用方式。

5.2.2.1 Golang Drivers

go-sql-driver/mysql

支持等级：**Full**

按照 go-sql-driver/mysql 文档中的说明获取并配置 Golang 驱动程序即可使用。

有关一个完整的实例应用程序，可参阅使用 Go-MySQL-Driver 连接到 TiDB。

5.2.2.2 Golang ORM 框架

GORM

支持等级：**Full**

GORM 是一个流行的 Golang 的 ORM 框架，你可以使用 go get 获取你的应用程序的所有依赖项。

```
go get -u gorm.io/gorm
go get -u gorm.io/driver/mysql
```

使用 GORM 进行 TiDB 应用程序构建的例子，可参阅使用 GORM 连接到 TiDB。

5.2.3 Python

本节介绍 Python 语言的 Driver 及 ORM 的使用方式。

5.2.3.1 Python Drivers

支持等级：**Compatible**

按照 PyMySQL 文档中的说明下载并配置驱动程序即可使用。建议使用 **1.0.2** 及以上版本。

使用 PyMySQL 构建 TiDB 应用程序的例子，可参阅使用 PyMySQL 连接到 TiDB。

支持等级：**Compatible**

按照 mysqlclient 文档中的说明下载并配置驱动程序即可使用。建议使用 **2.1.1** 及以上版本。

使用 mysqlclient 构建 TiDB 应用程序的例子，可参阅使用 mysqlclient 连接到 TiDB。

支持等级：**Compatible**

按照 MySQL Connector/Python 文档中的说明下载并配置驱动程序即可使用。建议使用 **8.0.31** 及以上版本。

使用 MySQL Connector/Python 构建 TiDB 应用程序的例子，可参阅使用 MySQL Connector/Python 连接到 TiDB。

5.2.3.2 Python ORM 框架

支持等级：**Full**

Django 是一个流行的 Python Web 开发框架。为解决 TiDB 与 Django 的兼容性问题，PingCAP 开发了一个专门的适配器 django-tidb。你可以参考 django-tidb 文档进行安装。

使用 Django 构建 TiDB 应用程序的例子，可参阅使用 Django 连接到 TiDB。

支持等级：**Full**

SQLAlchemy 是一个流行的 Python 的 ORM 框架，你可以使用 pip install SQLAlchemy==1.4.44 获取你的应用程序的所有依赖项。建议使用 **1.4.44** 及以上版本。

使用 SQLAlchemy 构建 TiDB 应用程序的例子，可参阅使用 SQLAlchemy 连接到 TiDB。

支持等级：**Compatible**

peewee 是一个流行的 Python 的 ORM 框架，你可以使用 pip install peewee==3.15.4 获取你的应用程序的所有依赖项。建议使用 **3.15.4** 及以上版本。

使用 peewee 构建 TiDB 应用程序的例子，可参阅使用 peewee 连接到 TiDB。

5.3 连接到平凯数据库

平凯数据库高度兼容 MySQL 协议，全量的客户端连接参数列表，请参阅 MySQL Client Options。

平凯数据库支持 MySQL 客户端/服务器协议。这使得大多数客户端驱动程序和 ORM 框架可以像连接到 MySQL 一样地连接到平凯数据库。

5.3.1 MySQL

你可以选择使用 MySQL Client 或 MySQL Shell 连接到 TiDB。

你可以使用 MySQL Client 作为平凯数据库的命令行工具连接到平凯数据库。下面以基于 YUM 的 Linux 发行版为例，介绍如何安装 MySQL Client。

```
sudo yum install mysql
```

安装完成后，你可以使用如下命令连接到平凯数据库：

```
mysql --host <tidb_server_host> --port 4000 -u root -p --comments
```

你可以使用 MySQL Shell 作为平凯数据库的命令行工具连接到平凯数据库。参考 MySQL Shell 文档进行安装。安装完成后，你可以使用如下命令连接到 TiDB：

```
mysqlsh --sql mysql://root@<tidb_server_host>:4000
```

5.3.2 JDBC

你可以使用 JDBC 驱动连接到 TiDB，这需要创建一个 MysqlDataSource 或 MysqlConnectionPoolDataSource 对象（它们都实现了 DataSource 接口），并使用 setURL 函数设置连接字符串。

例如：

```
MysqlDataSource mysqlDataSource = new MysqlDataSource();
mysqlDataSource.setURL("jdbc:mysql://{host}:{port}/{database}?user={username}&password={password}");
```

有关 JDBC 连接的更多信息，可参考 JDBC 官方文档。

连接参数

参数名	描述
{username}	需要连接到 TiDB 集群的 SQL 用户
{password}	需要连接到 TiDB 集群的 SQL 用户的密码
{host}	TiDB 节点运行的 Host
{port}	TiDB 节点正在监听的端口
{database}	(已经存在的)数据库的名称

5.3.3 Hibernate

你可以使用 Hibernate ORM 连接到 TiDB，请将 Hibernate 的配置中的 hibernate.connection.url 设置为合法的 TiDB 连接字符串。

例如，你的配置被写在 hibernate.cfg.xml 文件中，那么你的配置文件应该为：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
```

```
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.dialect">org.hibernate.dialect.TiDBDialect</property>
        <property name="hibernate.connection.url">jdbc:mysql://{host}:{port}/{database}?user={user}&amp;amp;password={password}</property>
    </session-factory>
</hibernate-configuration>
```

随后，使用代码读取配置文件，从而获得 SessionFactory 对象：

```
SessionFactory sessionFactory = new Configuration().configure("hibernate.cfg.xml").buildSessionFactory();
```

这里有几个需要注意的点：

1. 因为使用的配置文件 hibernate.cfg.xml 为 XML 格式，而 & 字符，在 XML 中属于特殊字符，因此，需将 & 更改为 &。即，连接字符串 hibernate.connection.url 由 jdbc:mysql://{host}:{port}/{database}?user={user}&password={password} 改为了 jdbc:mysql://{host}:{port}/{database}?user={user}&password={password}。
2. 在你使用 Hibernate 时，建议使用 TiDB 方言，即 hibernate.dialect 设置为 org.hibernate.dialect.TiDBDialect。
3. Hibernate 在版本 6.0.0.Beta2 及以上可支持 TiDB 方言，因此推荐使用 6.0.0.Beta2 及以上版本的 Hibernate。

更多有关 Hibernate 连接参数的信息，请参阅 Hibernate 官方文档。

连接参数

参数名	描述
{username}	需要连接到 TiDB 集群的 SQL 用户
{password}	需要连接到 TiDB 集群的 SQL 用户的密码
{host}	TiDB 节点运行的 Host
{port}	TiDB 节点正在监听的端口

参数名	描述
{database}	(已经存在的)数据库的名称

5.4 连接池与连接参数

5.4.1 连接池参数

TiDB (MySQL) 连接建立是比较昂贵的操作（至少对于 OLTP 来讲），除了建立 TCP 连接外还需要进行连接鉴权操作，所以客户端通常会把 TiDB (MySQL) 连接保存到连接池中进行复用。

Java 的连接池实现很多 (HikariCP, tomcat-jdbc, druid, c3p0, dbcp)，TiDB 不会限定使用的连接池，应用可以根据业务特点自行选择连接池实现。

5.4.1.1 连接数配置

比较常见的是应用需要根据自身情况配置合适的连接池大小，以 HikariCP 为例：

maximumPoolSize：连接池最大连接数，配置过大会导致 TiDB 消耗资源维护无用连接，配置过小则会导致应用获取连接变慢，所以需根据应用自身特点配置合适的值。

minimumIdle：连接池最小空闲连接数，主要用于在应用空闲时存留一些连接以应对突发请求，同样是需要根据业务情况进行配置。

应用在使用连接池时，需要注意连接使用完成后归还连接，推荐应用使用对应的连接池相关监控（如 **metricRegistry**），通过监控能及时定位连接池问题。

5.4.1.2 探活配置

连接池维护客户端到 TiDB 的长连接的方式如下：

- v5.4 版本前，TiDB 默认不会主动关闭客户端连接，除非出现报错情况。

- 从 v5.4 起，TiDB 默认会在连接空闲超过 28800 秒（即 8 小时）后，自动关闭客户端连接。你可以使用 TiDB 与 MySQL 兼容的 wait_timeout 变量控制此超时时间，详见 JDBC 查询超时文档。

此外，客户端到 TiDB 之间通常还会有 LVS 或 HAProxy 之类的网络代理。这些代理通常会在连接空闲超过特定时间（由代理的 idle 配置决定）后主动清理连接。除了关注代理的 idle 配置外，连接池还需要进行保活或探测连接。

如果常在 Java 应用中看到以下错误：

The last packet sent successfully to the server was 3600000 milliseconds ago. The driver has not received any packets from the server. com.mysql.jdbc.exceptions.jdbc4.CommunicationsException: Communications link failure

如果 n milliseconds ago 中的 n 如果是 0 或很小的值，则通常是执行的 SQL 导致 TiDB 异常退出引起的报错，推荐查看 TiDB stderr 日志；如果 n 是一个非常大的值（比如这里的 3600000），很可能是因为这个连接空闲太久然后被中间 proxy 关闭了，通常解决方式除了调大 proxy 的 idle 配置，还可以让连接池执行以下操作：

- 每次使用连接前检查连接是否可用。
- 使用单独线程定期检查连接是否可用。
- 定期发送 test query 保活连接。

不同的连接池实现可能会支持其中一种或多种方式，可以查看所使用的连接池文档来寻找对应配置。

5.4.1.3 经验公式

在 HikariCP 的 About Pool Sizing 一文中可以了解到，在完全不知道如何设置数据库连接池大小的时候，可以考虑以以下经验公式为起点，在此基础上，围绕该结果进行尝试，以得到最高性能的连接池大小。

该经验公式描述如下：

connections = ((core_count * 2) + effective_spindle_count)

解释一下参数含义：

- **connections**: 得出的连接数大小。
- **core_count**: CPU 核心数。
- **effective_spindle_count**: 直译为**有效主轴数**，实际上是说你有多少个硬盘（非 SSD），因为每个旋转的硬盘可以被称为是一个旋转轴。例如，你使用的是一个有 16 个磁盘组成的 RAID 阵列的服务器，那么 **effective_spindle_count** 应为 16。此处经验公式，实际上是衡量你的服务器可以管理多少 I/O 并发请求，因为 **HDD** 通常只能串行请求。

要特别说明的是，在这个经验公式的的下方，也看到了一处说明：

A formula which has held up pretty well across a lot of benchmarks for years is that for optimal throughput the number of active connections should be somewhere near ((core_count * 2) + effective_spindle_count). Core count should not include HT threads, even if hyperthreading is enabled. Effective spindle count is zero if the active data set is fully cached, and approaches the actual number of spindles as the cache hit rate falls. ... There hasn't been any analysis so far regarding how well the formula works with SSDs.

这个说明指出：

1. **core_count** 就是_物理核心数_，与你是否开启超线程无关。
2. 数据被全量缓存时，**effective_spindle_count** 应被设置为 0，随着命中率的下降，会更加接近实际的 HDD 个数。
3. **这里没有任何基于 *SSD* 的经验公式。**

这里的说明让你在使用 SSD 时，需探求其他的经验公式。

可以参考 CockroachDB 对数据库连接池中的描述，推荐的连接数大小公式为：

connections = (number of cores * 4)

因此，你在使用 SSD 的情况下可以将连接数设置为 CPU 核心数 * 4。以此来达到初始的连接池最大连接数大小，并以此数据周围进行进一步的调整。

5.4.1.4 调整方向

可以看到，在上方的经验公式中得到的，是一个推荐的初始值，若需得到某台具体机器上的最佳值，需在推荐值周围，通过尝试，得到最佳值。

此最佳值的获取，会有一些基本规律，此处罗列如下：

1. 如果你的网络或存储延迟较大，请增大你的最大连接数，可以进行等待，从而让线程在被阻塞时，其他的线程可继续进行处理。
2. 如果你的服务器上部署了多个服务，并且每个服务拥有独立的连接池时，请关注它们的连接池的最大连接数总和。

5.4.2 连接参数

Java 应用尽管可以选择在不同的框架中封装，但在最底层一般会通过调用 JDBC 来与数据库服务器进行交互。对于 JDBC，需要关注的主要有：API 的使用选择和 API Implementer 的参数配置。

5.4.2.1 JDBC API

对于基本的 JDBC API 使用可以参考 JDBC 官方教程，本文主要强调几个比较重要的 API 选择。

5.4.2.2 使用 Prepare API

对于 OLTP 场景，程序发送给数据库的 SQL 语句在去除参数变化后都是可穷举的某几类，因此建议使用预处理语句 (Prepared Statements) 代替普通的文本执行，并复用预处理语句来直接执行，从而避免 TiDB 重复解析和生成 SQL 执行计划的开销。

目前多数上层框架都会调用 Prepare API 进行 SQL 执行，如果直接使用 JDBC API 进行开发，注意选择使用 Prepare API。

另外需要注意 MySQL Connector/J 实现中默认只会做客户端的语句预处理，会将 ? 在客户端替换后以文本形式发送到服务端，所以除了要使用 Prepare API，还需要

在 JDBC 连接参数中配置 useServerPrepStmts = true，才能在 TiDB 服务器端进行语句预处理（下面参数配置章节有详细介绍）。

5.4.2.3 使用 Batch 批量插入更新

对于批量插入更新，如果插入记录较多，可以选择使用 addBatch/executeBatch API。通过 addBatch 的方式将多条 SQL 的插入更新记录先缓存在客户端，然后在 executeBatch 时一起发送到数据库服务器。

> **注意：**
>
> 对于 MySQL Connector/J 实现，默认 Batch 只是将多次 addBatch 的 SQL 发送时机延迟到调用 executeBatch 的时候，但实际网络发送还是会一条条的发送，通常不会降低与数据库服务器的网络交互次数。
>
> 如果希望 Batch 网络发送，需要在 JDBC 连接参数中配置 rewriteBatchedStatements = true（下面参数配置章节有详细介绍）。

5.4.2.4 使用 StreamingResult 流式获取执行结果

一般情况下，为提升执行效率，JDBC 会默认提前获取查询结果并将其保存在客户端内存中。但在查询返回超大结果集的场景中，客户端会希望数据库服务器减少向客户端一次返回的记录数，等客户端在有限内存处理完一部分后再去向服务器要下一批。

在 JDBC 中通常有以下两种处理方式：

- 方式一：设置 **FetchSize** 为 Integer.MIN_VALUE 让客户端不缓存，客户端通过 StreamingResult 的方式从网络连接上流式读取执行结果。
- 方式二：使用 Cursor Fetch，首先需设置 **FetchSize** 为正整数，且在 JDBC URL 中配置 useCursorFetch = true。

TiDB 同时支持以上两种方式，但更推荐使用第一种将 FetchSize 设置为 Integer.MIN_VALUE 的方式，比第二种功能实现更简单且执行效率更高。

对于第二种方式，TiDB 会先将所有数据加载到 TiDB 节点上，然后根据 FetchSize 依次返回给客户端。因此，通常会比第一种方式使用更多内存。如果将 tidb_enable_tmp_storage_on_oom 设置为 ON，可能会触发落盘临时将结果写入硬盘。

如果系统变量 tidb_enable_lazy_cursor_fetch 设置为 ON，TiDB 将尝试仅在客户端请求数据时读取部分数据，以使用更少的内存。更多信息和使用限制，参见系统变量 tidb_enable_lazy_cursor_fetch 的详细描述。

5.4.2.5 MySQL JDBC 参数

JDBC 实现通常通过 JDBC URL 参数的形式来提供实现相关的配置。这里以 MySQL 官方的 Connector/J 来介绍参数配置（如果使用的是 MariaDB，可以参考 MariaDB 的类似配置）。因为配置项较多，这里主要关注几个可能影响到性能的参数。

5.4.2.5.1 Prepare 相关参数

- **useServerPrepStmts**

  默认情况下，**useServerPrepStmts** 的值为 false，即尽管使用了 Prepare API，也只会在客户端做 “prepare”。因此为了避免服务器重复解析的开销，如果同一条 SQL 语句需要多次使用 Prepare API，则建议设置该选项为 true。

  在 TiDB 监控中可以通过 **Query Summary > CPS By Instance** 查看请求命令类型，如果请求中 COM_QUERY 被 COM_STMT_EXECUTE 或 COM_STMT_PREPARE 代替即生效。

- **cachePrepStmts**

  虽然 useServerPrepStmts = true 能让服务端执行预处理语句，但默认情况下客户端每次执行完后会 close 预处理语句，并不会复用，这样预处理的效率甚至不如文本执行。所以建议开启 useServerPrepStmts = true 后同时配置 cachePrepStmts = true，这会让客户端缓存预处理语句。

在 TiDB 监控中可以通过 **Query Summary > CPS By Instance** 查看请求命令类型，如果请求中 COM_STMT_EXECUTE 数目远远多于 COM_STMT_PREPARE 即生效。

另外，通过 useConfigs = maxPerformance 配置会同时配置多个参数，其中也包括 cachePrepStmts = true。

- **prepStmtCacheSqlLimit**

  在配置 **cachePrepStmts** 后还需要注意 **prepStmtCacheSqlLimit** 配置（默认为 256），该配置控制客户端缓存预处理语句的最大长度，超过该长度将不会被缓存。

  在一些场景 SQL 的长度可能超过该配置，导致预处理 SQL 不能复用，建议根据应用 SQL 长度情况决定是否需要调大该值。

  在 TiDB 监控中通过 **Query Summary > CPS By Instance** 查看请求命令类型，如果已经配置了 cachePrepStmts = true，但 COM_STMT_PREPARE 还是和 COM_STMT_EXECUTE 基本相等且有 COM_STMT_CLOSE，需要检查这个配置项是否设置得太小。

- **prepStmtCacheSize**

  控制缓存的预处理语句数目（默认为 25），如果应用需要预处理的 SQL 种类很多且希望复用预处理语句，可以调大该值。

  和上一条类似，在监控中通过 **Query Summary > CPS By Instance** 查看请求中 COM_STMT_EXECUTE 数目是否远远多于 COM_STMT_PREPARE 来确认是否正常。

5.4.2.5.2 Batch 相关参数

在进行 batch 写入处理时推荐配置 rewriteBatchedStatements = true，在已经使用 addBatch 或 executeBatch 后默认 JDBC 还是会一条条 SQL 发送，例如：

```
pstmt = prepare("INSERT INTO `t` (`a`) VALUES(?)");
pstmt.setInt(1, 10);
pstmt.addBatch();
pstmt.setInt(1, 11);
pstmt.addBatch();
pstmt.setInt(1, 12);
pstmt.executeBatch();
```

虽然使用了 batch 但发送到 TiDB 语句还是单独的多条 insert：

```
INSERT INTO `t` (`a`) VALUES(10);
INSERT INTO `t` (`a`) VALUES(11);
INSERT INTO `t` (`a`) VALUES(12);
```

如果设置 rewriteBatchedStatements = true，发送到 TiDB 的 SQL 将是：

```
INSERT INTO `t` (`a`) VALUES(10),(11),(12);
```

需要注意的是，insert 语句的改写，只能将多个 values 后的值拼接成一整条 SQL，insert 语句如果有其他差异将无法被改写。例如：

```
INSERT INTO `t` (`a`) VALUES (10) ON DUPLICATE KEY UPDATE `a` = 10;
INSERT INTO `t` (`a`) VALUES (11) ON DUPLICATE KEY UPDATE `a` = 11;
INSERT INTO `t` (`a`) VALUES (12) ON DUPLICATE KEY UPDATE `a` = 12;
```

上述 insert 语句将无法被改写成一条语句。该例子中，如果将 SQL 改写成如下形式：

```
INSERT INTO `t` (`a`) VALUES (10) ON DUPLICATE KEY UPDATE `a` = values(`a`);
INSERT INTO `t` (`a`) VALUES (11) ON DUPLICATE KEY UPDATE `a` = values(`a`);
INSERT INTO `t` (`a`) VALUES (12) ON DUPLICATE KEY UPDATE `a` = values(`a`);
```

即可满足改写条件，最终被改写成：

```
INSERT INTO `t` (`a`) VALUES (10), (11), (12) ON DUPLICATE KEY UPDATE `a` = values(`
a`);
```

批量更新时如果有 3 处或 3 处以上更新，则 SQL 语句会改写为 multiple-queries 的形式并发送，这样可以有效减少客户端到服务器的请求开销，但副作用是会产生较大的 SQL 语句，例如这样：

```
UPDATE `t` SET `a` = 10 WHERE `id` = 1;
UPDATE `t` SET `a` = 11 WHERE `id` = 2;
UPDATE `t` SET `a` = 12 WHERE `id` = 3;
```

另外，因为一个客户端 bug，批量更新时如果要配置 rewriteBatchedStatements = true 和 useServerPrepStmts = true，推荐同时配置 allowMultiQueries = true 参数来避免这个 bug。

5.4.2.5.3 集成参数

通过监控可能会发现，虽然业务只向集群进行 insert 操作，却看到有很多多余的 select 语句。通常这是因为 JDBC 发送了一些查询设置类的 SQL 语句（例如 select @@session.transaction_read_only）。这些 SQL 对 TiDB 无用，推荐配置 useConfigs = maxPerformance 来避免额外开销。

useConfigs = maxPerformance 会包含一组配置，可查看 MySQL Connector/J 8.0 版本 或 5.1 版本 来确认当前 MySQL Connector/J 中 maxPerformance 包含的具体配置。

配置后查看监控，可以看到多余语句减少。

5.4.2.5.4 超时参数

TiDB 提供两个与 MySQL 兼容的超时控制参数，wait_timeout 和 max_execution_time。这两个参数分别控制与 Java 应用连接的空闲超时时间和连接中 SQL 执行的超时时间，即控制 TiDB 与 Java 应用的连接最长闲多久和最长忙多久。在 TiDB v5.4 及以上版本中，wait_timeout 参数默认值为 28800 秒，即空闲超时为 8 小时。在 v5.4 之前，wait_timeout 参数的默认值为 0，即没有时间限制。max_execution_time 参数的默认值为 0，即不限制一条 SQL 语句的执行时间。

但是 wait_timeout 的默认值比较大，在事务已启动但未提交或回滚的情况下，你可能需要更细粒度的控制和更短的超时，以避免持有锁的时间过长。此时，你可以使用 TiDB 在 v7.1.8 引入的 tidb_idle_transaction_timeout 控制用户会话中事务的空闲超时。

但在实际生产环境中，空闲连接和一直无限执行的 SQL 对数据库和应用都有不好的影响。你可以通过在应用的连接字符串中配置这两个参数来避免空闲连接和执行时间过长的 SQL 语句。例如，设置 sessionVariables=wait_timeout=3600（1 小时）和 sessionVariables=max_execution_time=300000（5 分钟）。

6 数据库模式设计

6.1 概述

本页概述了平凯数据库中的数据库模式。将从本页开始围绕 Bookshop 这个应用程序来对平凯数据库的设计数据库部分展开介绍。并使用此数据库做后续数据的写入、读取示例。

6.1.1 术语歧义

此处术语会有歧义，为消除歧义，在此作出数据库模式设计文档部分中的术语简要约定：

为避免和通用术语数据库 (Database) 混淆，因此将逻辑对象称为**数据库 (Database)**，TiDB 仍使用原名称，并将 TiDB 的部署实例称为**集群 (Cluster)**。

因为 TiDB 使用与 MySQL 兼容的语法，在此语法下，**模式 (Schema)** 仅代表通用术语定义，并无逻辑对象定义，可参考此官方文档。若你从其他拥有 **Schema** 逻辑对象的数据库（如：PostgreSQL、Oracle、Microsoft SQL Server 等）迁移而来，请注意此区别。

6.1.2 数据库 Database

TiDB 语境中的 Database 或者说数据库，可以认为是表和索引等对象的集合。

TiDB 集群包含一个名为 test 的数据库。但建议你自行创建数据库，而不是使用 test 数据库。

6.1.3 表 Table

TiDB 语境中的 Table 或者说表，从属于某个数据库。

表包含数据**行**。每行数据中的每个值都属于一个特定的**列**。每列都只允许单一数据类型的数据值。列可添加约束来进一步限定。你还可以添加生成列用于计算。

6.1.4 索引 Index

索引是单个表中行的副本，按列或列集排序。TiDB 查询使用索引来更有效的查找表内的数据，同时可以给出特定列的值。每个索引都是从属于某个表的。

索引有两种常见的类型，分别为：

- **Primary Key**: 即主键索引，即标识在主键列上的索引。
- **Secondary Index**: 即二级索引，即在非主键上标识的索引。

  注意：

  TiDB 中，关于 **Primary Key** 的默认定义与 MySQL 常用存储引擎 InnoDB 不一致。**InnoDB** 中，**Primary Key** 的语义为：唯一，不为空，**且为聚簇索引**。

  而在 TiDB 中，**Primary Key** 的定义为：唯一，不为空。但主键不保证为**聚簇索引**。而是由另一组关键字 CLUSTERED、NONCLUSTERED 额外控制 **Primary Key** 是否为聚簇索引，若不指定，则由系统变量 @@global.tidb_enable_clustered_index 影响，具体说明请看聚簇索引。

6.1.4.1 专用索引

TiDB 支持一些特殊场景专用的索引，用以提高特定用例中的查询性能。具体请参考索引和约束。

6.1.5 其他对象

TiDB 支持一些和**表**同级的对象：

- 视图: 视图是一张虚拟表，该虚拟表的结构由创建视图时的 SELECT 语句定义，TiDB 目前不支持物化视图。
- 序列: 创建和存储顺序数据。
- 临时表: 临时表是数据不持久化的表。

6.1.6 访问控制

TiDB 支持基于用户或角色的访问控制。你可以通过角色或直接指向用户，从而授予**用户**查看、修改或删除数据对象和数据模式的权限。

6.1.7 执行数据库模式更改

不推荐使用客户端的 Driver 或 ORM 来执行数据库模式的更改。以经验来看，作为最佳实践，建议使用 MySQL 客户端或使用任意你喜欢的 GUI 客户端来进行数据库模式的更改。本文档中，将在大多数场景下，使用 **MySQL 客户端** 传入 SQL 文件来执行数据库模式的更改。

6.1.8 对象大小限制

具体限制请参考平凯数据库使用限制。

6.2 创建数据库

在这个章节当中，将开始介绍如何使用 SQL 来创建数据库，及创建数据库时应遵守的规则。将在这个章节中围绕 Bookshop 这个应用程序来对 TiDB 的创建数据库部分展开介绍。

> **注意：**
>
> 此处仅对 CREATE DATABASE 语句进行简单描述，详细参考文档（包含其他示例），可参阅 CREATE DATABASE 文档。

6.2.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 在本地快速部署平凯数据库测试集群。
- 阅读数据库模式概览。

6.2.2 什么是数据库

在 TiDB 中数据库对象可以包含**表**、**视图**、**序列**等对象。

6.2.3 创建数据库过程

可使用 CREATE DATABASE 语句来创建数据库。

```
CREATE DATABASE IF NOT EXISTS `bookshop`;
```

此语句会创建一个名为 bookshop 的数据库（如果尚不存在）。请以 root 用户身份执行文件中的建库语句，运行以下命令：

```
mysql
   -u root \
   -h {host} \
   -P {port} \
   -p {password} \
   -e "CREATE DATABASE IF NOT EXISTS bookshop;"
```

要查看集群中的数据库，可在命令行执行一条 SHOW DATABASES 语句：

```
mysql
   -u root \
   -h {host} \
   -P {port} \
   -p {password} \
   -e "SHOW DATABASES;"
```

运行结果为：

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| bookshop           |
| mysql              |
| test               |
+--------------------+
```

6.2.4 数据库创建时应遵守的规则

- 遵循数据库命名规范，给你的数据库起一个有意义的名字。
- test 数据库是 TiDB 提供的一个默认数据库。如果没有必要，尽量不要在生产环境使用它。你可以自行使用 CREATE DATABASE 语句来创建数据库，并且在 SQL 会话中使用 USE {databasename}; 语句来更改当前数据库。
- 使用 root 用户创建数据库、角色、用户等，并只赋予必要的权限。
- 作为通用的规则，不推荐使用 Driver、ORM 进行数据库模式的定义与更改。相反，请使用 **MySQL 命令行客户端**或其他你喜欢的 **MySQL GUI 客户端**来进行操作。

6.2.5 更进一步

至此，你已经准备完毕 bookshop 数据库，可以将**表**添加到该数据库中。

你可继续阅读创建表文档获得相关指引。

6.3 创建表

本文档介绍如何使用 SQL 语句来创建表以及创建表的最佳实践。本文档提供了一个基于平凯数据库的 bookshop 数据库的示例加以说明。

> **注意：**
>
> 此处仅对 CREATE TABLE 语句进行简单描述，详细参考文档（包含其他示例），可参阅 CREATE TABLE 文档。

6.3.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 在本地快速部署平凯数据库测试集群。
- 阅读数据库模式概览。
- 创建一个数据库。

6.3.2 什么是表

表是集群中的一种逻辑对象，它从属于数据库，用于保存从 SQL 中发送的数据。表以行和列的形式组织数据记录,一张表至少有一列。若在表中定义了 n 个列，那么每一行数据都将拥有与这 n 个列中完全一致的字段。

6.3.3 命名表

创建表的第一步，就是给你的表起个名字。请不要使用无意义的表名，将给未来的你或者你的同事带来极大的困扰。推荐你遵循公司或组织的表命名规范。

CREATE TABLE 语句通常采用以下形式：

```
CREATE TABLE {table_name} ( {elements} );
```

参数描述

- {table_name}: 表名。
- {elements}: 以逗号分隔的表元素列表，比如列定义，主键定义等。

假设你需要创建一个表来存储 bookshop 库中的用户信息。

注意，此时因为一个列都没被添加，所以下方这条 SQL 暂时还不能被运行：

```
CREATE TABLE `bookshop`.`users` (
);
```

6.3.4 定义列

列从属于表，每张表都至少有一**列**。**列**通过将每行中的值分成一个个单一数据类型的小单元来为表提供结构。

列定义通常使用以下形式：

```
{column_name} {data_type} {column_qualification}
```

参数描述

- {column_name}：列名。

- {data_type}：列的数据类型。
- {column_qualification}：列的限定条件，如**列级约束**或生成列子句。

可以为 users 表添加一些列，如他们的唯一标识 id，余额 balance 及昵称 nickname。

```
CREATE TABLE `bookshop`.`users` (
  `id` bigint,
  `nickname` varchar(100),
  `balance` decimal(15,2)
);
```

其中，定义了一个字段名为 id，类型为 bigint 的字段。用以表示用户唯一标识。这意味着，所有的用户标识都应该是 bigint 类型的。

而在其后，又定义了一个字段名为 nickname，类型为 varchar，且长度不得超过 100 字符的字段。用以表示用户的昵称。这意味着，所用用户的昵称都是 varchar 类型，且不超过 100 字符的。

最后，又加入了一个字段名为 balance 用以表示用户的余额，类型为 decimal，且其精度为 15，比例为 2。简单的说明一下精度和比例代表的含义，精度代表字段数值的总位数，而比例代表小数点后有多少位。例如: decimal(5,2)，即精度为 5，比例为 2 时，其取值范围为 -999.99 到 999.99。decimal(6,1)，即精度为 6，比例为 1 时，其取值范围为 -99999.9 到 99999.9。decimal 类型为定点数，可精确保存数字，在需要精确数字的场景（如用户财产相关）中，请确保使用定点数类型。

TiDB 支持许多其他的列数据类型，包含整数、浮点数、定点数、时间、枚举等，可参考支持的列的数据类型，并使用与你准备保存在数据库内的数据匹配的**数据类型**。

稍微提升一下复杂度，例如选择定义一张 books 表，这张表将是 bookshop 数据的核心。它包含书的唯一标识、名称、书籍类型（如：杂志、动漫、教辅等）、库存、价格、出版时间等字段。

```
CREATE TABLE `bookshop`.`books` (
  `id` bigint NOT NULL,
  `title` varchar(100),
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humanities & Social Sciences', 'Science & Technology', 'Kids', 'Sports'),
  `published_at` datetime,
  `stock` int,
  `price` decimal(15,2)
);
```

这张表比 users 表包含更多的数据类型：

- int: 推荐使用合适大小的类型，防止使用过量的硬盘甚至影响性能(类型范围过大)或数据溢出(类型范围过小)。
- datetime: 可以使用 **datetime** 类型保存时间值。
- enum: 可以使用 **enum** 类型的枚举来保存有限选择的值。

6.3.5 选择主键

主键是一个或一组列，这个由所有主键列组合起来的值是数据行的唯一标识。

> **注意：**
>
> TiDB 中，关于 **Primary Key** 的默认定义与 MySQL 常用存储引擎 InnoDB 不一致。**InnoDB** 中，**Primary Key** 的语义为：唯一，不为空，**且为聚簇索引**。
>
> 而在 TiDB 中，**Primary Key** 的定义为：唯一，不为空。但主键不保证为**聚簇索引**。而是由另一组关键字 CLUSTERED、NONCLUSTERED 额外控制 **Primary Key** 是否为聚簇索引，若不指定，则由系统变量 @@global.tidb_enable_clustered_index 影响，具体说明请看此文档。

主键在 CREATE TABLE 语句中定义。主键约束要求所有受约束的列仅包含非 NULL 值。

一个表可以没有主键，主键也可以是非整数类型。但此时 TiDB 就会创建一个 _tidb_rowid 作为隐式主键。隐式主键 _tidb_rowid 因为其单调递增的特性，可能在

大批量写入场景下会导致写入热点，如果你写入量密集，可考虑通过 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 两参数控制打散。但这可能导致读放大，请自行取舍。

表的主键为整数类型且使用了 AUTO_INCREMENT 时，无法使用 SHARD_ROW_ID_BITS 消除热点。需解决此热点问题，且无需使用主键的连续和递增时，可使用 AUTO_RANDOM 替换 AUTO_INCREMENT 属性来消除行 ID 的连续性。

更多有关热点问题的处理办法，请参考平凯数据库热点问题处理。

需遵循选择主键时应遵守的规则，举一个 users 表中定义 AUTO_RANDOM 主键的例子：

```
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM,
  `balance` decimal(15,2),
  `nickname` varchar(100),
  PRIMARY KEY (`id`)
);
```

6.3.6 选择聚簇索引

聚簇索引 (clustered index) 是 TiDB 从 v5.0 开始支持的特性，用于控制含有主键的表数据的存储方式。通过使用聚簇索引，TiDB 可以更好地组织数据表，从而提高某些查询的性能。有些数据库管理系统也将聚簇索引称为“索引组织表” (index-organized tables)。

目前 TiDB 中 ***含有主键*** 的表分为以下两类：

- NONCLUSTERED，表示该表的主键为非聚簇索引。在非聚簇索引表中，行数据的键由 TiDB 内部隐式分配的 _tidb_rowid 构成，而主键本质上是唯一索引，因此非聚簇索引表存储一行至少需要两个键值对，分别为：
    - _tidb_rowid（键）- 行数据（值）
    - 主键列数据（键） - _tidb_rowid（值）

- CLUSTERED，表示该表的主键为聚簇索引。在聚簇索引表中，行数据的键由用户给定的主键列数据构成，因此聚簇索引表存储一行至少只要一个键值对，即：
    - 主键列数据（键） - 行数据（值）

如主键中所述，聚簇索引在 TiDB 中，使用关键字 CLUSTERED、NONCLUSTERED 进行控制。

> **注意：**
>
> TiDB 仅支持根据表的主键来进行聚簇操作。聚簇索引启用时，“主键”和“聚簇索引”两个术语在一些情况下可互换使用。主键指的是约束（一种逻辑属性），而聚簇索引描述的是数据存储的物理实现。

需遵循选择聚簇索引时应遵守的规则，假设需要建立一张 books 和 users 之间关联的表，代表用户对某书籍的评分。使用表名 ratings 来创建该表，并使用 book_id 和 user_id 构建复合主键，并在该主键上建立聚簇索引：

```
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint,
  `user_id` bigint,
  `score` tinyint,
  `rated_at` datetime,
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED
);
```

6.3.7 添加列约束

除主键约束外，TiDB 还支持其他的列约束，如：非空约束 NOT NULL、唯一约束 UNIQUE KEY、默认值 DEFAULT 等。完整约束，请查看 TiDB 约束文档。

6.3.7.1 填充默认值

如需在列上设置默认值，请使用 DEFAULT 约束。默认值将可以使你无需指定每一列的值，就可以插入数据。

你可以将 DEFAULT 与支持的 SQL 函数结合使用，将默认值的计算移出应用层，从而节省应用层的资源（当然，计算所消耗的资源并不会凭空消失，只是被转移到了 TiDB 集群中）。常见的，希望实现数据插入时，可默认填充默认的时间。还是使用 ratings 作为示例，可使用以下语句：

```
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint,
  `user_id` bigint,
  `score` tinyint,
  `rated_at` datetime DEFAULT NOW(),
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED
);
```

此外，如需在数据更新时也默认填入当前时间，可使用以下语句（但 ON UPDATE 后仅可填入与当前时间相关的表达式）：

```
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint,
  `user_id` bigint,
  `score` tinyint,
  `rated_at` datetime DEFAULT NOW() ON UPDATE NOW(),
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED
);
```

关于不同数据类型默认值的更多信息，请参阅数据类型的默认值。

6.3.7.2 防止重复

如果你需要防止列中出现重复值，那你可以使用 UNIQUE 约束。

例如，你需要确保用户的昵称唯一，可以这样改写 users 表的创建 SQL：

```
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM,
  `balance` decimal(15,2),
  `nickname` varchar(100) UNIQUE,
  PRIMARY KEY (`id`)
);
```

如果你在 users 表中尝试插入相同的 nickname，将返回错误。

6.3.7.3 防止空值

如果你需要防止列中出现空值，那就可以使用 NOT NULL 约束。

还是使用用户昵称来举例子，除了昵称唯一，还希望昵称不可为空，于是此处可以这样改写 users 表的创建 SQL：

```
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM,
  `balance` decimal(15,2),
  `nickname` varchar(100) UNIQUE NOT NULL,
  PRIMARY KEY (`id`)
);
```

6.3.8 使用 HTAP 能力

> **注意：**
>
> 本指南中有关 HTAP 的步骤仅适用于快速上手体验，不适用于生产环境。
> 如需探索 HTAP 更多功能，请参考深入探索 HTAP。

假设 bookshop 应用程序，有对用户评价的 ratings 表进行 OLAP 分析查询的需求，例如需查询: **书籍的评分，是否和评价的时间具有显著的相关性** 的需求，用以分析用户的书籍评分是否客观。那么会要求查询整个 ratings 表中的 score 和 rated_at 字段。这对普通仅支持的 OLTP 的数据库来说，是一个非常消耗资源的操作。或者使用一些 ETL 或其他数据同步工具，将 OLTP 数据库中的数据，导出到专用的 OLAP 数据库，再进行分析。

这种场景下，TiDB 就是一个比较理想的一站式数据库解决方案，TiDB 是一个 **HTAP (Hybrid Transactional and Analytical Processing)** 数据库，同时支持 OLTP 和 OLAP 场景。

6.3.8.1 同步列存数据

当前，TiDB 支持两种数据分析引擎：**TiFlash** 和 **TiSpark**。大数据场景 (100 T) 下，推荐使用 TiFlash MPP 作为 HTAP 的主要方案，TiSpark 作为补充方案。希望

了解更多关于 TiDB 的 HTAP 能力，可参考以下文章：快速上手 HTAP 和深入探索 HTAP。

此处选用 TiFlash 为 bookshop 数据库的数据分析引擎。

TiFlash 部署完成后并不会自动同步数据，而需要手动指定需要同步的表，开启同步副本仅需一行 SQL，如下所示：

```
ALTER TABLE {table_name} SET TIFLASH REPLICA {count};
```

参数描述

- {table_name}: 表名。
- {count}: 同步副本数，若为 0，则表示删除同步副本。

随后，TiFlash 将同步该表，查询时，TiDB 将会自动基于成本优化，考虑使用 **TiKV (行存)** 或 **TiFlash (列存)** 进行数据查询。当然，除了自动的方法，你也可以直接指定查询是否使用 TiFlash 副本，使用方法可查看使用 TiDB 读取 TiFlash 文档。

6.3.8.2 使用 HTAP 的示例

ratings 表开启 1 个 TiFlash 副本：

```
ALTER TABLE `bookshop`.`ratings` SET TIFLASH REPLICA 1;
```

> **注意：**
>
> 如果你的集群，不包含 TiFlash 节点，此 SQL 语句将会报错：1105 - the tiflash replica count: 1 should be less than the total tiflash server count: 0 你可以在本地快速部署平凯数据库测试集群 来创建一个含有 TiFlash 的集群。

随后正常进行查询即可：

```
SELECT HOUR(`rated_at`), AVG(`score`) FROM `bookshop`.`ratings` GROUP BY HOUR(`rated_at`);
```

也可使用 EXPLAIN ANALYZE 语句查看此语句是否使用了 TiFlash 引擎：

```
EXPLAIN ANALYZE SELECT HOUR(`rated_at`), AVG(`score`) FROM `bookshop`.`ratings`
GROUP BY HOUR(`rated_at`);
```

运行结果为：

```
+----------------------------+----------+---------+-------------+---------------+---------
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
-----------------------------------------------------+-----------------------------------
----------------------------------------------------------------------------------------------
-----+----------+------+
| id                         | estRows   | actRows | task         | access object | execution info


                                  | operator info
                                        | memory   | disk |
+----------------------------+----------+---------+-------------+---------------+---------
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
-----------------------------------------------------+-----------------------------------
----------------------------------------------------------------------------------------------
-----+----------+------+
| Projection_4               | 299821.99 | 24      | root         |               | time:60.8ms, loops:6, Co
ncurrency:5

                                                  | hour(cast(bookshop.ratings.rated_at, time))->Column
#6, Column#5                                                                  | 17.7 KB  | N/A  |
| └─HashAgg_5                | 299821.99 | 24      | root         |               | time:60.7ms, loops:6,
 partial_worker:{wall_time:60.660079ms, concurrency:5, task_num:293, tot_wait:262.53666
9ms, tot_exec:40.171833ms, tot_time:302.827753ms, max:60.636886ms, p95:60.636886m
s}, final_worker:{wall_time:60.701437ms, concurrency:5, task_num:25, tot_wait:303.114278
ms, tot_exec:176.564µs, tot_time:303.297475ms, max:60.69326ms, p95:60.69326ms}  | gro
up by:Column#10, funcs:avg(Column#8)->Column#5, funcs:firstrow(Column#9)->booksh
op.ratings.rated_at                                  | 714.0 KB | N/A  |
|   └─Projection_15          | 300000.00 | 300000  | root         |               | time:58.5ms, loops:
294, Concurrency:5

                                                | cast(bookshop.ratings.score, decimal(8,4) BINA
RY)->Column#8, bookshop.ratings.rated_at, hour(cast(bookshop.ratings.rated_at, time))
->Column#10 | 366.2 KB | N/A  |
|     └─TableReader_10       | 300000.00 | 300000  | root         |               | time:43.5ms, loo
```

```
ps:294, cop_task: {num: 1, max: 43.1ms, proc_keys: 0, rpc_num: 1, rpc_time: 43ms, copr_ca
che_hit_ratio: 0.00}
                                                                                 | data:TableFull
Scan_9                                                                                    | 4.58
 MB  | N/A  |
|        └─TableFullScan_9     | 300000.00 | 300000  | cop[tiflash] | table:ratings | tiflash_task:
{time:5.98ms, loops:8, threads:1}, tiflash_scan:{dtfile:{total_scanned_packs:45, total_skippe
d_packs:1, total_scanned_rows:368640, total_skipped_rows:8192, total_rs_index_load_tim
e: 1ms, total_read_time: 1ms},total_create_snapshot_time:1ms}
                                                    | keep order:false
                                                              | N/A      | N/A  |
+----------------------------+-----------+---------+--------------+---------------+---------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
--------------------------------------------------+---------------------------------------
------------------------------------------------------------------------------------------
-----+----------+------+
```

在出现 cop[tiflash] 字样时，表示该任务发送至 TiFlash 进行处理。

6.3.9 执行 CREATE TABLE 语句

按以上步骤创建所有表后，数据库初始化脚本应该如此所示。若需查看表信息详解，请参阅数据表详解。

如果将数据库初始化脚本命名为 init.sql 并保存，可使用以下语句来执行数据库初始化：

```
mysql
   -u root \
   -h {host} \
   -P {port} \
   -p {password} \
   < init.sql
```

需查看 bookshop 数据库下的所有表，可使用 SHOW TABLES 语句：

```
SHOW TABLES IN `bookshop`;
```

运行结果为：

```
+--------------------+
| Tables_in_bookshop |
+--------------------+
| authors            |
| book_authors       |
| books              |
| orders             |
| ratings            |
| users              |
+--------------------+
```

6.3.10 创建表时应遵守的规则

本小节给出了一些在创建表时应遵守的规则。

6.3.10.1 命名表时应遵守的规则

- 使用**完全限定**的表名称（例如：CREATE TABLE {database_name}.{table_name}）。这是因为你在不指定数据库名称时，TiDB 将使用你 **SQL 会话**中的当前数据库。若你未在 SQL 会话中使用 USE {databasename}; 来指定数据库，TiDB 将会返回错误。
- 请使用有意义的表名，例如，若你需要创建一个用户表，你可以使用名称：user, t_user, users 等，或遵循你公司或组织的命名规范。如果你的公司或组织没有相应的命名规范，可参考表命名规范。请勿使用这样的表名，如：t1, table1 等。
- 多个单词以下划线分隔，不推荐超过 32 个字符。
- 不同业务模块的表单独建立 DATABASE，并增加相应注释。

6.3.10.2 定义列时应遵守的规则

- 查看支持的列的数据类型，并按照数据类型的限制来组织你的数据。为你计划被存在列中的数据选择合适的类型。
- 查看选择主键时应遵守的规则，决定是否使用主键列。
- 查看选择聚簇索引时应遵守的规则，决定是否指定聚簇索引。
- 查看添加列约束，决定是否添加约束到列中。

- 请使用有意义的列名，推荐你遵循公司或组织的表命名规范。如果你的公司或组织没有相应的命名规范，可参考列命名规范。

6.3.10.3 选择主键时应遵守的规则

- 在表内定义一个主键或唯一索引。
- 尽量选择有意义的列作为主键。
- 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。
- 避免在单个单调数据列上定义主键。如果你使用单个单调数据列（例如：AUTO_INCREMENT 的列）来定义主键，有可能会对写性能产生负面影响。可能的话，使用 AUTO_RANDOM 替换 AUTO_INCREMENT（这会失去主键的连续和递增特性）。
- 如果你 ***必须*** 在单个单调数据列上创建索引，且有大量写入的话。请不要将这个单调数据列定义为主键，而是使用 AUTO_RANDOM 创建该表的主键，或使用 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 打散 _tidb_rowid。

6.3.10.4 选择聚簇索引时应遵守的规则

- 遵循选择主键时应遵守的规则：

  聚簇索引将基于主键建立，请遵循选择主键时应遵守的规则，此为选择聚簇索引时应遵守规则的基础。

- 在以下场景中，尽量使用聚簇索引，将带来性能和吞吐量的优势：

  - 插入数据时会减少一次从网络写入索引数据。
  - 等值条件查询仅涉及主键时会减少一次从网络读取数据。
  - 范围条件查询仅涉及主键时会减少多次从网络读取数据。

    - 等值或范围条件查询仅涉及主键的前缀时会减少多次从网络读取数据。
- 在以下场景中，尽量避免使用聚簇索引，将带来性能劣势：
    - 批量插入大量取值相邻的主键时，可能会产生较大的写热点问题，请遵循选择主键时应遵守的规则。
    - 当使用大于 64 位的数据类型作为主键时，可能导致表数据需要占用更多的存储空间。该现象在存在多个二级索引时尤为明显。
- 显式指定是否使用聚簇索引，而非使用系统变量 @@global.tidb_enable_clustered_index 及配置项 alter-primary-key 控制是否使用聚簇索引的默认行为。

6.3.10.5 CREATE TABLE 执行时应遵守的规则

- 不推荐使用客户端的 Driver 或 ORM 来执行数据库模式的更改。基于过往经验，建议使用 MySQL 客户端或使用任意你喜欢的 GUI 客户端来进行数据库模式的更改。本文档中，将在大多数场景下，使用 **MySQL 客户端** 传入 SQL 文件来执行数据库模式的更改。
- 遵循 SQL 开发规范中的建表删表规范，建议业务应用内部封装建表删表语句增加判断逻辑。

6.3.11 更进一步

请注意，到目前为止，创建的所有表都不包含二级索引。添加二级索引的指南，请参考创建二级索引。

6.4 创建二级索引

在这个章节当中，将开始介绍如何使用 SQL 来创建二级索引，及创建二级索引时应遵守的规则。将在这个章节中围绕 Bookshop 这个应用程序来对平凯数据库的创建二级索引部分展开介绍。

6.4.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 在本地快速部署平凯数据库测试集群。
- 阅读数据库模式概览。
- 创建一个数据库。
- 创建表。

6.4.2 什么是二级索引

二级索引是集群中的逻辑对象，你可以简单地认为它就是一种对数据的排序，TiDB 使用这种有序性来加速查询。TiDB 的创建二级索引的操作为在线操作，不会阻塞表中的数据读写。TiDB 会创建表中各行的引用，并按选择的列进行排序。而并非对表本身的数据进行排序。可在二级索引中查看更多信息。二级索引可跟随表进行创建，也可在已有的表上进行添加。

6.4.3 在已有表中添加二级索引

如果需要对已有表中添加二级索引，可使用 CREATE INDEX 语句。在 TiDB 中，CREATE INDEX 为在线操作，不会阻塞表中的数据读写。二级索引创建一般如以下形式：

```
CREATE INDEX {index_name} ON {table_name} ({column_names});
```

参数描述

- {index_name}: 二级索引名。
- {table_name}: 表名。
- {column_names}: 将需要索引的列名列表，以半角逗号分隔。

6.4.4 新建表的同时创建二级索引

如果你希望在创建表的同时，同时创建二级索引，可在 CREATE TABLE 的末尾使用包含 KEY 关键字的子句来创建二级索引：

```
KEY `{index_name}` (`{column_names}`)
```

参数描述

- {index_name}: 二级索引名。
- {column_names}: 将需要索引的列名列表，以半角逗号分隔。

6.4.5 创建二级索引时应遵守的规则

见索引的最佳实践。

6.4.6 例子

假设你希望 bookshop 应用程序有 **查询某个年份出版的所有书籍** 的功能。books 表如下所示:

字段名	类型	含义
id	bigint	书籍的唯一标识
title	varchar(100)	书籍名称
type	enum	书籍类型（如：杂志、动漫、教辅等）
stock	bigint	库存
price	decimal(15,2)	价格
published_at	datetime	出版时间

```
CREATE TABLE `bookshop`.`books` (
  `id` bigint AUTO_RANDOM NOT NULL,
  `title` varchar(100) NOT NULL,
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humanities & Social Sciences', 'Science & Technology', 'Kids', 'Sports') NOT NULL,
  `published_at` datetime NOT NULL,
  `stock` int DEFAULT '0',
  `price` decimal(15,2) DEFAULT '0.0',
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

因此，就需要对 **查询某个年份出版的所有书籍** 的 SQL 进行编写，以 2022 年为例，如下所示：

```
SELECT * FROM `bookshop`.`books` WHERE `published_at` >= '2022-01-01 00:00:00' AND `published_at` < '2023-01-01 00:00:00';
```

可以使用 EXPLAIN 进行 SQL 语句的执行计划检查：

```
EXPLAIN SELECT * FROM `bookshop`.`books` WHERE `published_at` >= '2022-01-01 00:00:00' AND `published_at` < '2023-01-01 00:00:00';
```

运行结果为：

```
+------------------------+----------+-----------+---------------+--------------------------------
--------------------------------------------------------------------------------------------+
| id                     | estRows  | task      | access object | operator info
                                                                                            |
+------------------------+----------+-----------+---------------+--------------------------------
--------------------------------------------------------------------------------------------+
| TableReader_7          | 346.32   | root      |               | data:Selection_6
                                                                                            |
| └─Selection_6          | 346.32   | cop[tikv] |               | ge(bookshop.books.published_at, 2
022-01-01 00:00:00.000000), lt(bookshop.books.published_at, 2023-01-01 00:00:00.00000
0) |
|   └─TableFullScan_5    | 20000.00 | cop[tikv] | table:books   | keep order:false
                                                                                            |
+------------------------+----------+-----------+---------------+--------------------------------
--------------------------------------------------------------------------------------------+
3 rows in set (0.61 sec)
```

可以看到返回的计划中，出现了类似 **TableFullScan** 的字样，这代表 TiDB 准备在这个查询中对 books 表进行全表扫描，这在数据量较大的情况下，几乎是致命的。

在 books 表增加一个 published_at 列的索引：

```
CREATE INDEX `idx_book_published_at` ON `bookshop`.`books` (`bookshop`.`books`.`published_at`);
```

添加索引后，再次运行 EXPLAIN 语句检查执行计划：

```
+-----------------------------+---------+-----------+-------------------------------------------
---------------+-----------------------------------------------------------------+
| id                          | estRows | task      | access object                             | operato
r info                                          |
+-----------------------------+---------+-----------+-------------------------------------------
```

```
---------------+-------------------------------------------------------------------+
| IndexLookUp_10               | 146.01  | root      |                                                   |
                                                  |
| ├─IndexRangeScan_8(Build)    | 146.01  | cop[tikv] | table:books, index:idx_book_publish
ed_at(published_at) | range:[2022-01-01 00:00:00,2023-01-01 00:00:00), keep order:false |
| └─TableRowIDScan_9(Probe)    | 146.01  | cop[tikv] | table:books
      | keep order:false                                          |
+------------------------------+---------+-----------+-----------------------------------
---------------+-------------------------------------------------------------------+
3 rows in set (0.18 sec)
```

可以看到执行计划中没有了 **TableFullScan** 的字样，取而代之的是 **IndexRangeScan**，这代表已经 TiDB 在进行这个查询时准备使用索引。

> **注意：**
>
> 上方执行计划中的的 **TableFullScan**、**IndexRangeScan** 等在 TiDB 内被称为算子。这里对执行计划的解读及算子等不做进一步的展开，若你对此感兴趣，可前往 TiDB 执行计划概览文档查看更多关于执行计划与 TiDB 算子的相关知识。
>
> 执行计划并非每次返回使用的算子都相同，这是由于 TiDB 使用的优化方式为 **基于代价的优化方式 (CBO)**，执行计划不仅与规则相关，还和数据分布相关。你可以前往 SQL 性能调优文档查看更多 TiDB SQL 性能的描述。
>
> TiDB 在查询时，还支持显式地使用索引，你可以使用 Optimizer Hints 或执行计划管理 (SPM) 来人为的控制索引的使用。但如果你不了解它内部发生了什么，请你***暂时先不要使用它***。

可以使用 SHOW INDEXES 语句查询表中的索引：

```
SHOW INDEXES FROM `bookshop`.`books`;
```

运行结果为：

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

```
| Table | Non_unique | Key_name            | Seq_in_index | Column_name  | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression | Clustered |
+-------+------------+---------------------+--------------+--------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+-----------+
| books |          0 | PRIMARY             |            1 | id           | A         |           0 |     NULL | NULL   |      | BTREE      |         |               | YES     | NULL       | YES       |
| books |          1 | idx_book_published_at |            1 | published_at | A         |           0 |     NULL | NULL   |      | BTREE      |         |               | YES     | NULL       | NO        |
+-------+------------+---------------------+--------------+--------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+-----------+
2 rows in set (1.63 sec)
```

6.4.7 更进一步

至此，你已经完成数据库、表及二级索引的创建，接下来，数据库模式已经准备好给你的应用程序提供写入和读取的能力了。

7 数据写入

7.1 插入数据

此页面将展示使用 SQL 语言，配合各种编程语言将数据插入到平凯数据库中。

7.1.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 在本地快速部署平凯数据库测试集群。
- 阅读数据库模式概览，并创建数据库、创建表、创建二级索引。

7.1.2 插入行

假设你需要插入多行数据，那么会有两种插入的办法，假设需要插入 3 个玩家数据：

- 一个**多行插入语句**：

```
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (1, 1000, 1), (2, 230, 2), (3, 300, 5);
```

- 多个**单行插入语句**：

```
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (1, 1000, 1);
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (2, 230, 2);
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (3, 300, 5);
```

一般来说使用一个多行插入语句，会比多个单行插入语句快。

在 SQL 中插入多行数据的示例：

```
CREATE TABLE `player` (`id` INT, `coins` INT, `goods` INT);
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (1, 1000, 1), (2, 230, 2);
```

有关如何使用此 SQL，可查阅在本地快速部署平凯数据库测试集群文档，按文档步骤使用客户端连接到 TiDB 集群后，输入 SQL 语句即可。

在 Java 中插入多行数据的示例：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource
try (Connection connection = ds.getConnection()) {
  connection.setAutoCommit(false);

  PreparedStatement pstmt = connection.prepareStatement("INSERT INTO player (id, coins, goods) VALUES (?, ?, ?)"))

  // first player
  pstmt.setInt(1, 1);
  pstmt.setInt(2, 1000);
  pstmt.setInt(3, 1);
  pstmt.addBatch();

  // second player
  pstmt.setInt(1, 2);
  pstmt.setInt(2, 230);
  pstmt.setInt(3, 2);
  pstmt.addBatch();

  pstmt.executeBatch();
```

```
    connection.commit();
} catch (SQLException e) {
    e.printStackTrace();
}
```

另外，由于 MySQL JDBC Driver 默认设置问题，你需更改部分参数，以获得更好的批量插入性能：

参数	作用	推荐场景	推荐配置
useServerPrepStmts	是否使用服务端开启预处理语句支持	在需要多次使用预处理语句时	true
cachePrepStmts	客户端是否缓存预处理语句	useServerPrepStmts=true 时	true
prepStmtCacheSqlLimit	预处理语句最大大小（默认 256 字符）	预处理语句大于 256 字符时	按实际预处理语句大小配置
prepStmtCacheSize	预处理语句最大缓存数量 （默认 25 条）	预处理语句数量大于 25 条时	按实际预处理语句数量配置
rewriteBatchedStatements	是否重写 Batch 语句	需要批量操作时	true
allowMultiQueries	开启批量操作	因为一个客户端 Bug 在 rewriteBatchedStatements = true 和 useServerPrepStmts = true 时，需设置此项	true

MySQL JDBC Driver 还提供了一个集成配置项：useConfigs。当它配置为 maxPerformance 时，相当于配置了一组配置，以 mysql:mysql-connector-java:8.0.28 为例，useConfigs=maxPerformance 包含：

```
cachePrepStmts=true
cacheCallableStmts=true
cacheServerConfiguration=true
useLocalSessionState=true
elideSetAutoCommits=true
alwaysSendSetIsolation=false
enableQueryTimeouts=false
connectionAttributes=none
useInformationSchema=true
```

你可以自行查看 mysql-connector-java-{version}.jar!/com/mysql/cj/configurations/maxPerformance.properties 来获得对应版本 MySQL JDBC Driver 的 useConfigs=maxPerformance 包含配置。

在此处给出一个较为的通用场景的 JDBC 连接字符串配置，以 Host: 127.0.0.1，Port: 4000，用户名: root，密码: 空，默认数据库: test 为例：

```
jdbc:mysql://127.0.0.1:4000/test?user=root&useConfigs=maxPerformance&useServerPrepStmts=true&prepStmtCacheSqlLimit=2048&prepStmtCacheSize=256&rewriteBatchedStatements=true&allowMultiQueries=true
```

有关 Java 的完整示例，可参阅：

- 平凯数据库和 JDBC 的简单 CRUD 应用程序
- 平凯数据库和 Hibernate 的简单 CRUD 应用程序
- 使用 Spring Boot 构建平凯数据库应用程序

在 Golang 中插入多行数据的示例：

```go
package main

import (
    "database/sql"
    "strings"
```

```
    _ "github.com/go-sql-driver/mysql"
)

type Player struct {
    ID    string
    Coins int
    Goods int
}

func bulkInsertPlayers(db *sql.DB, players []Player, batchSize int) error {
    tx, err := db.Begin()
    if err != nil {
        return err
    }

    stmt, err := tx.Prepare(buildBulkInsertSQL(batchSize))
    if err != nil {
        return err
    }

    defer stmt.Close()

    for len(players) > batchSize {
        if _, err := stmt.Exec(playerToArgs(players[:batchSize])...); err != nil {
            tx.Rollback()
            return err
        }

        players = players[batchSize:]
    }

    if len(players) != 0 {
        if _, err := tx.Exec(buildBulkInsertSQL(len(players)), playerToArgs(players)...); err != nil
{
            tx.Rollback()
            return err
        }
    }

    if err := tx.Commit(); err != nil {
        tx.Rollback()
        return err
    }
```

```
    return nil
}

func playerToArgs(players []Player) []interface{} {
    var args []interface{}
    for _, player := range players {
        args = append(args, player.ID, player.Coins, player.Goods)
    }
    return args
}

func buildBulkInsertSQL(amount int) string {
    return "INSERT INTO player (id, coins, goods) VALUES (?, ?, ?)" + strings.Repeat(",(?,?,?)
", amount-1)
}
```

有关 Golang 的完整示例，可参阅：

- 使用 Go-MySQL-Driver 连接到 TiDB
- 使用 GORM 连接到 TiDB

在 Python 中插入多行数据的示例：

```
import MySQLdb

connection = MySQLdb.connect(
    host="127.0.0.1",
    port=4000,
    user="root",
    password="",
    database="bookshop",
    autocommit=True
)
with get_connection(autocommit=True) as connection:

    with connection.cursor() as cur:
        player_list = random_player(1919)
        for idx in range(0, len(player_list), 114):
            cur.executemany("INSERT INTO player (id, coins, goods) VALUES (%s, %s, %s)", pl
ayer_list[idx:idx + 114])
```

有关 Python 的完整示例，可参阅：

- 使用 PyMySQL 连接到 TiDB
- 使用 mysqlclient 连接到 TiDB
- 使用 MySQL Connector/Python 连接到 TiDB
- 使用 SQLAlchemy 连接到 TiDB
- 使用 Django 连接到 TiDB
- 使用 peewee 连接到 TiDB

7.1.3 批量插入

如果你需要快速地将大量数据导入 TiDB 集群，最好的方式并不是使用 INSERT 语句，这并不是最高效的方法，而且需要你自行处理异常等问题。推荐使用 PingCAP 提供的一系列工具进行数据迁移：

- 数据导出工具：Dumpling。可以导出 MySQL 或 TiDB 的数据到本地或 Amazon S3 中。
- 数据导入工具：TiDB Lightning。可以导入 Dumpling 导出的数据、CSV 文件，或者 Amazon Aurora 生成的 Apache Parquet 文件。同时支持在本地盘或 Amazon S3 云盘读取数据。
- 数据同步工具：TiDB Data Migration。可同步 MySQL、MariaDB、Amazon Aurora 数据库到 TiDB 中。且支持分库分表数据库的迁移。
- 数据备份恢复工具：Backup & Restore (BR)。相对于 Dumpling，BR 更适合***大数据量***的场景。

7.1.4 避免热点

在设计表时需要考虑是否存在大量插入行为，若有，需在表设计期间对热点进行规避。请查看创建表 - 选择主键部分，并遵从选择主键时应遵守的规则。

更多有关热点问题的处理办法，请参考 TiDB 热点问题处理文档。

7.1.5 主键为 AUTO_RANDOM 表插入数据

在插入的表主键为 AUTO_RANDOM 时，这时默认情况下，不能指定主键。例如 bookshop 数据库中，可以看到 users 表 的 id 字段含有 AUTO_RANDOM 属性。

此时，不可使用类似以下 SQL 进行插入：

```
INSERT INTO `bookshop`.`users` (`id`, `balance`, `nickname`) VALUES (1, 0.00, 'nicky');
```

将会产生错误：

```
ERROR 8216 (HY000): Invalid auto random: Explicit insertion on auto_random column is d
isabled. Try to set @@allow_auto_random_explicit_insert = true.
```

这是旨在提示你，不建议在插入时手动指定 AUTO_RANDOM 的列。这时，你有两种解决办法处理此错误：

- (推荐) 插入语句中去除此列，使用 TiDB 帮你初始化的 AUTO_RANDOM 值。这样符合 AUTO_RANDOM 的语义。

  ```
  INSERT INTO `bookshop`.`users` (`balance`, `nickname`) VALUES (0.00, 'nicky');
  ```

- 如果你确认一定需要指定此列，那么可以使用 SET 语句通过更改用户变量的方式，允许在插入时，指定 AUTO_RANDOM 的列。

  ```
  SET @@allow_auto_random_explicit_insert = true;
  INSERT INTO `bookshop`.`users` (`id`, `balance`, `nickname`) VALUES (1, 0.00, 'nic
  ky');
  ```

7.1.6 使用 HTAP

在 TiDB 中，使用 HTAP 能力无需你在插入数据时进行额外操作。不会有任何额外的插入逻辑，由 TiDB 自动进行数据的一致性保证。你只需要在创建表后，开启列存副本同步，就可以直接使用列存副本来加速你的查询。

7.2 更新数据

此页面将展示以下 SQL 语句，配合各种编程语言平凯数据库中的数据进行更新：

- UPDATE: 用于修改指定表中的数据。
- INSERT ON DUPLICATE KEY UPDATE: 用于插入数据，在有主键或唯一键冲突时，更新此数据。注意，***不建议***在有多个唯一键(包含主键)的情况下使用此语句。这是因为此语句在检测到任何唯一键(包括主键) 冲突时，将更新数据。在不止匹配到一行冲突时，将只会更新一行数据。

7.2.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 在本地快速部署平凯数据库测试集群
- 阅读数据库模式概览，并创建数据库、创建表、创建二级索引
- 若需使用 UPDATE 语句更新数据，需先插入数据

7.2.2 使用 UPDATE

需更新表中的现有行，需要使用带有 WHERE 子句的 UPDATE 语句，即需要过滤列进行更新。

> **注意：**
>
> 如果您需要更新大量的行，比如数万甚至更多行，那么建议不要一次性进行完整的更新，而是每次迭代更新一部分，直到所有行全部更新。您可以编写脚本或程序，使用循环完成此操作。 您可参考批量更新获得指引。

7.2.2.1 SQL 语法

在 SQL 中，UPDATE 语句一般为以下形式：

UPDATE {**table**} **SET** {update_column} = {update_value} **WHERE** {filter_column} = {filter_value}

参数	描述
{table}	表名
{update_column}	需更新的列名
{update_value}	需更新的此列的值

参数	描述
{filter_column}	匹配条件过滤器的列名
{filter_value}	匹配条件过滤器的列值

此处仅展示 UPDATE 的简单用法，详细文档可参考 TiDB 的 UPDATE 语法页。

7.2.2.2 UPDATE 最佳实践

以下是更新行时需要遵循的一些最佳实践：

- 始终在更新语句中指定 WHERE 子句。如果 UPDATE 没有 WHERE 子句，TiDB 将更新这个表内的***所有行***。
- 需要更新大量行(数万或更多)的时候，使用批量更新，这是因为 TiDB 单个事务大小限制为 txn-total-size-limit（默认为 100MB），且一次性过多的数据更新，将导致持有锁时间过长（悲观事务），或产生大量冲突（乐观事务）。

7.2.2.3 UPDATE 例子

假设某位作者改名为 Helen Haruki，需要更改 authors 表。假设他的唯一标识 id 为 1，即过滤器应为：id = 1。

在 SQL 中更改作者姓名的示例为：

```
UPDATE `authors` SET `name` = "Helen Haruki" WHERE `id` = 1;
```

在 Java 中更改作者姓名的示例为：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource
try (Connection connection = ds.getConnection()) {
   PreparedStatement pstmt = connection.prepareStatement("UPDATE `authors` SET `na
me` = ? WHERE `id` = ?");
   pstmt.setString(1, "Helen Haruki");
   pstmt.setInt(2, 1);
   pstmt.executeUpdate();
} catch (SQLException e) {
```

```
    e.printStackTrace();
}
```

7.2.3 使用 INSERT ON DUPLICATE KEY UPDATE

如果你需要将新数据插入表中，但如果有任何唯一键（主键也是一种唯一键）发生冲突，则会更新第一条冲突数据，可使用 INSERT ... ON DUPLICATE KEY UPDATE ... 语句进行插入或更新。

7.2.3.1 SQL 语法

在 SQL 中，INSERT ... ON DUPLICATE KEY UPDATE ... 语句一般为以下形式：

```
INSERT INTO {table} ({columns}) VALUES ({values})
   ON DUPLICATE KEY UPDATE {update_column} = {update_value};
```

参数	描述
{table}	表名
{columns}	需插入的列名
{values}	需插入的此列的值
{update_column}	需更新的列名
{update_value}	需更新的此列的值

7.2.3.2 INSERT ON DUPLICATE KEY UPDATE 最佳实践

- 在仅有一个唯一键的表上使用 INSERT ON DUPLICATE KEY UPDATE。此语句在检测到任何 ***唯一键*** (包括主键) 冲突时，将更新数据。在不止匹配到一行冲突时，将只会更新一行数据。因此，除非能保证仅有一行冲突，否则不建议在有多个唯一键的表中使用 INSERT ON DUPLICATE KEY UPDATE 语句。
- 在创建或更新的场景中使用此语句。

7.2.3.3 INSERT ON DUPLICATE KEY UPDATE 例子

例如，需要更新 ratings 表来写入用户对书籍的评价，如果用户还未评价此书籍，将新建一条评价，如果用户已经评价过，那么将会更新他之前的评价。

此处主键为 book_id 和 user_id 的联合主键。user_id 为 1 的用户，给 book_id 为 1000 的书籍，打出的 5 分的评价。

在 SQL 中更新书籍评价的示例为：

```
INSERT INTO `ratings`
  (`book_id`, `user_id`, `score`, `rated_at`)
VALUES
  (1000, 1, 5, NOW())
ON DUPLICATE KEY UPDATE `score` = 5, `rated_at` = NOW();
```

在 Java 中更新书籍评价的示例为：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource

try (Connection connection = ds.getConnection()) {
  PreparedStatement p = connection.prepareStatement("INSERT INTO `ratings` (`book_id`, `user_id`, `score`, `rated_at`)
VALUES (?, ?, ?, NOW()) ON DUPLICATE KEY UPDATE `score` = ?, `rated_at` = NOW()");
  p.setInt(1, 1000);
  p.setInt(2, 1);
  p.setInt(3, 5);
  p.setInt(4, 5);
  p.executeUpdate();
} catch (SQLException e) {
  e.printStackTrace();
}
```

7.2.4 批量更新

需要更新表中多行的数据，可选择使用 UPDATE，并使用 WHERE 子句过滤需要更新的数据。

但如果你需要更新大量行(数万或更多)的时候，建议使用一个迭代，每次都只更新一部分数据，直到更新全部完成。这是因为 TiDB 单个事务大小限制为 txn-total-size-limit（默认为 100MB），且一次性过多的数据更新，将导致持有锁时间过长

（悲观事务），或产生大量冲突（乐观事务）。你可以在程序或脚本中使用循环来完成操作。

本页提供了编写脚本来处理循环更新的示例，该示例演示了应如何进行 SELECT 和 UPDATE 的组合，完成循环更新。

7.2.4.1 编写批量更新循环

首先，你应在你的应用或脚本的循环中，编写一个 SELECT 查询。这个查询的返回值可以作为需要更新的行的主键。需要注意的是，定义这个 SELECT 查询时，需要注意使用 WHERE 子句过滤需要更新的行。

7.2.4.2 例子

假设在过去的一年里，用户在 bookshop 网站进行了大量的书籍打分，但是原本设计为 5 分制的评分导致书籍评分的区分度不够，大量书籍评分集中在 3 分附近，因此，决定将 5 分制改为 10 分制。用来增大书籍评分的区分度。

这时需要对 ratings 表内之前 5 分制的数据进行乘 2 操作，同时需向 ratings 表内添加一个新列，以指示行是否已经被更新了。使用此列，可以在 SELECT 中过滤掉已经更新的行，这将防止脚本崩溃时对行进行多次更新，导致不合理的数据出现。

例如，你可以创建一个名为 ten_point，数据类型为 BOOL 的列作为是否为 10 分制的标识：

```
ALTER TABLE `bookshop`.`ratings` ADD COLUMN `ten_point` BOOL NOT NULL DEFAU
LT FALSE;
```

> **注意：**
>
> 此批量更新程序将使用 **DDL** 语句将进行数据表的模式更改。TiDB 的所有 DDL 变更操作全部都是在线进行的，可查看此处，了解此处使用的 ADD COLUMN 语句。

在 Golang 中，批量更新程序类似于以下内容：

```go
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "strings"
    "time"
)

func main() {
    db, err := sql.Open("mysql", "root:@tcp(127.0.0.1:4000)/bookshop")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    bookID, userID := updateBatch(db, true, 0, 0)
    fmt.Println("first time batch update success")
    for {
        time.Sleep(time.Second)
        bookID, userID = updateBatch(db, false, bookID, userID)
        fmt.Printf("batch update success, [bookID] %d, [userID] %d\n", bookID, userID)
    }
}

// updateBatch select at most 1000 lines data to update score
func updateBatch(db *sql.DB, firstTime bool, lastBookID, lastUserID int64) (bookID, userI
D int64) {
    // select at most 1000 primary keys in five-point scale data
    var err error
    var rows *sql.Rows

    if firstTime {
        rows, err = db.Query("SELECT `book_id`, `user_id` FROM `bookshop`.`ratings` " +
            "WHERE `ten_point` != true ORDER BY `book_id`, `user_id` LIMIT 1000")
    } else {
        rows, err = db.Query("SELECT `book_id`, `user_id` FROM `bookshop`.`ratings` "+
            "WHERE `ten_point` != true AND `book_id` > ? AND `user_id` > ? "+
            "ORDER BY `book_id`, `user_id` LIMIT 1000", lastBookID, lastUserID)
    }

    if err != nil || rows == nil {
```

```
        panic(fmt.Errorf("error occurred or rows nil: %+v", err))
    }

    // joint all id with a list
    var idList []interface{}
    for rows.Next() {
        var tempBookID, tempUserID int64
        if err := rows.Scan(&tempBookID, &tempUserID); err != nil {
            panic(err)
        }
        idList = append(idList, tempBookID, tempUserID)
        bookID, userID = tempBookID, tempUserID
    }

    bulkUpdateSql := fmt.Sprintf("UPDATE `bookshop`.`ratings` SET `ten_point` = true, "+
        "`score` = `score` * 2 WHERE (`book_id`, `user_id`) IN (%s)", placeHolder(len(idList)))
    db.Exec(bulkUpdateSql, idList...)

    return bookID, userID
}

// placeHolder format SQL place holder
func placeHolder(n int) string {
    holderList := make([]string, n/2, n/2)
    for i := range holderList {
        holderList[i] = "(?,?)"
    }
    return strings.Join(holderList, ",")
}
```

每次迭代中，SELECT 按主键顺序进行查询，最多选择 1000 行未更新到 10 分制（ten_point 为 false）数据的主键值。每次 SELECT 都会选择比上一次 SELECT 结果的最大主键还要大的数据，防止重复。然后，使用批量更新的方式，对其 score 列乘 2，并且将 ten_point 设为 true，更新 ten_point 的意义是在于防止更新程序崩溃重启后，反复更新同一行数据，导致数据损坏。每次循环中的 time.Sleep(time.Second) 将使得更新程序暂停 1 秒，防止批量更新程序占用过多的硬件资源。

在 Java (JDBC) 中，批量更新程序类似于以下内容：

Java 代码部分：

```
package com.pingcap.bulkUpdate;

import com.mysql.cj.jdbc.MysqlDataSource;

import java.sql.*;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class BatchUpdateExample {
    static class UpdateID {
        private Long bookID;
        private Long userID;

        public UpdateID(Long bookID, Long userID) {
            this.bookID = bookID;
            this.userID = userID;
        }

        public Long getBookID() {
            return bookID;
        }

        public void setBookID(Long bookID) {
            this.bookID = bookID;
        }

        public Long getUserID() {
            return userID;
        }

        public void setUserID(Long userID) {
            this.userID = userID;
        }

        @Override
        public String toString() {
            return "[bookID] " + bookID + ", [userID] " + userID ;
        }
    }
```

```
public static void main(String[] args) throws InterruptedException {
    // Configure the example database connection.

    // Create a mysql data source instance.
    MysqlDataSource mysqlDataSource = new MysqlDataSource();

    // Set server name, port, database name, username and password.
    mysqlDataSource.setServerName("localhost");
    mysqlDataSource.setPortNumber(4000);
    mysqlDataSource.setDatabaseName("bookshop");
    mysqlDataSource.setUser("root");
    mysqlDataSource.setPassword("");

    UpdateID lastID = batchUpdate(mysqlDataSource, null);

    System.out.println("first time batch update success");
    while (true) {
        TimeUnit.SECONDS.sleep(1);
        lastID = batchUpdate(mysqlDataSource, lastID);
        System.out.println("batch update success, [lastID] " + lastID);
    }
}

public static UpdateID batchUpdate (MysqlDataSource ds, UpdateID lastID) {
    try (Connection connection = ds.getConnection()) {
        UpdateID updateID = null;

        PreparedStatement selectPs;

        if (lastID == null) {
            selectPs = connection.prepareStatement(
                    "SELECT `book_id`, `user_id` FROM `bookshop`.`ratings` " +
                    "WHERE `ten_point` != true ORDER BY `book_id`, `user_id` LIMIT 1000");
        } else {
            selectPs = connection.prepareStatement(
                    "SELECT `book_id`, `user_id` FROM `bookshop`.`ratings` "+
                      "WHERE `ten_point` != true AND `book_id` > ? AND `user_id` > ? "+
                      "ORDER BY `book_id`, `user_id` LIMIT 1000");

            selectPs.setLong(1, lastID.getBookID());
            selectPs.setLong(2, lastID.getUserID());
        }
```

```
            List<Long> idList = new LinkedList<>();
            ResultSet res = selectPs.executeQuery();
            while (res.next()) {
                updateID = new UpdateID(
                        res.getLong("book_id"),
                        res.getLong("user_id")
                );
                idList.add(updateID.getBookID());
                idList.add(updateID.getUserID());
            }

            if (idList.isEmpty()) {
                System.out.println("no data should update");
                return null;
            }

            String updateSQL = "UPDATE `bookshop`.`ratings` SET `ten_point` = true, "+
                    "`score` = `score` * 2 WHERE (`book_id`, `user_id`) IN (" +
                    placeHolder(idList.size() / 2) + ")";
            PreparedStatement updatePs = connection.prepareStatement(updateSQL);
            for (int i = 0; i < idList.size(); i++) {
                updatePs.setLong(i + 1, idList.get(i));
            }
            int count = updatePs.executeUpdate();
            System.out.println("update " + count + " data");

            return updateID;
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return null;
    }

    public static String placeHolder(int n) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < n ; i++) {
            sb.append(i == 0 ? "(?,?)" : ",(?,?)");
        }

        return sb.toString();
    }
}
```

hibernate.cfg.xml 配置部分：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>

        <!-- Database connection settings -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</pr
operty>
        <property name="hibernate.dialect">org.hibernate.dialect.TiDBDialect</propert
y>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:4000/movie</
property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.connection.autocommit">false</property>
        <property name="hibernate.jdbc.batch_size">20</property>

        <!-- Optional: Show SQL output for debugging -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

每次迭代中，SELECT 按主键顺序进行查询，最多选择 1000 行未更新到 10 分制（ten_point 为 false）数据的主键值。每次 SELECT 都会选择比上一次 SELECT 结果的最大主键还要大的数据，防止重复。然后，使用批量更新的方式，对其 score 列乘 2，并且将 ten_point 设为 true，更新 ten_point 的意义是在于防止更新程序崩溃重启后，反复更新同一行数据，导致数据损坏。每次循环中的 TimeUnit.SECONDS.sleep(1); 将使得更新程序暂停 1 秒，防止批量更新程序占用过多的硬件资源。

7.3 删除数据

此页面将使用 DELETE SQL 语句，对平凯数据库中的数据进行删除。如果需要周期性地删除过期数据，可以考虑使用平凯数据库的 TTL 功能。

7.3.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 在本地快速部署平凯数据库测试集群。
- 阅读数据库模式概览，并创建数据库、创建表、创建二级索引。
- 需先插入数据才可删除。

7.3.2 SQL 语法

在 SQL 中，DELETE 语句一般为以下形式：

DELETE FROM {**table**} **WHERE** {**filter**}

参数	描述
{table}	表名
{filter}	过滤器匹配条件

此处仅展示 DELETE 的简单用法，详细文档可参考 TiDB 的 DELETE 语法。

7.3.3 最佳实践

以下是删除行时需要遵循的一些最佳实践：

- 始终在删除语句中指定 WHERE 子句。如果 DELETE 没有 WHERE 子句，TiDB 将删除这个表内的***所有行***。
- 需要删除大量行(数万或更多)的时候，使用批量删除，这是因为 TiDB 单个事务大小限制为 txn-total-size-limit（默认为 100MB）。
- 如果你需要删除表内的所有数据，请勿使用 DELETE 语句，而应该使用 TRUNCATE 语句。

- 查看性能注意事项。
- 在需要大批量删除数据的场景下，非事务批量删除对性能的提升十分明显。但与之相对的，这将丢失删除的事务性，因此**无法**进行回滚，请务必正确进行操作选择。

7.3.4 例子

假设在开发中发现在特定时间段内，发生了业务错误，需要删除这期间内的所有 rating 的数据，例如，2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。此时，可使用 SELECT 语句查看需删除的数据条数：

```
SELECT COUNT(*) FROM `ratings` WHERE `rated_at` >= "2022-04-15 00:00:00" AND `rated_at` <= "2022-04-15 00:15:00";
```

- 若返回数量大于 1 万条，请参考批量删除。
- 若返回数量小于 1 万条，可参考下面的示例进行删除：

在 SQL 中，删除数据的示例如下：

```
DELETE FROM `ratings` WHERE `rated_at` >= "2022-04-15 00:00:00" AND `rated_at` <= "2022-04-15 00:15:00";
```

在 Java 中，删除数据的示例如下：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource

try (Connection connection = ds.getConnection()) {
    String sql = "DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= ? AND `rated_at` <= ?";
    PreparedStatement preparedStatement = connection.prepareStatement(sql);
    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.MILLISECOND, 0);

    calendar.set(2022, Calendar.APRIL, 15, 0, 0, 0);
    preparedStatement.setTimestamp(1, new Timestamp(calendar.getTimeInMillis()));

    calendar.set(2022, Calendar.APRIL, 15, 0, 15, 0);
    preparedStatement.setTimestamp(2, new Timestamp(calendar.getTimeInMillis()));
```

```
    preparedStatement.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
}
```

在 Golang 中，删除数据的示例如下：

```
package main

import (
    "database/sql"
    "fmt"
    "time"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:@tcp(127.0.0.1:4000)/bookshop")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    startTime := time.Date(2022, 04, 15, 0, 0, 0, 0, time.UTC)
    endTime := time.Date(2022, 04, 15, 0, 15, 0, 0, time.UTC)

    bulkUpdateSql := fmt.Sprintf("DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >=
 ? AND  `rated_at` <= ?")
    result, err := db.Exec(bulkUpdateSql, startTime, endTime)
    if err != nil {
        panic(err)
    }
    _, err = result.RowsAffected()
    if err != nil {
        panic(err)
    }
}
```

在 Python 中，删除数据的示例如下：

```
import MySQLdb
import datetime
```

```
import time

connection = MySQLdb.connect(
    host="127.0.0.1",
    port=4000,
    user="root",
    password="",
    database="bookshop",
    autocommit=True
)

with connection:
    with connection.cursor() as cursor:
        start_time = datetime.datetime(2022, 4, 15)
        end_time = datetime.datetime(2022, 4, 15, 0, 15)
        delete_sql = "DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= %s AND `rate
d_at` <= %s"
        affect_rows = cursor.execute(delete_sql, (start_time, end_time))
        print(f'delete {affect_rows} data')
```

注意：

rated_at 字段为日期和时间类型 中的 DATETIME 类型，你可以认为它在 TiDB 保存时，存储为一个字面量，与时区无关。而 TIMESTAMP 类型，将会保存一个时间戳，从而在不同的时区配置时，展示不同的时间字符串。

另外，和 MySQL 一样，TIMESTAMP 数据类型受 2038 年问题的影响。如果存储的值大于 2038，建议使用 DATETIME 类型。

7.3.5 性能注意事项

7.3.5.1 TiDB GC 机制

DELETE 语句运行之后 TiDB 并非立刻删除数据，而是将这些数据标记为可删除。然后等待 TiDB GC (Garbage Collection) 来清理不再需要的旧数据。因此，你的 DELETE 语句***并不会***立即减少磁盘用量。

GC 在默认配置中，为 10 分钟触发一次，每次 GC 都会计算出一个名为 **safe_point** 的时间点，这个时间点前的数据，都不会再被使用到，因此，TiDB 可以安全的对数据进行清除。

GC 的具体实现方案和细节此处不再展开，请参考 GC 机制简介了解更详细的 GC 说明。

7.3.5.2 更新统计信息

TiDB 使用常规统计信息来决定索引的选择，因此，在大批量的数据删除之后，很有可能会导致索引选择不准确的情况发生。你可以使用手动收集的办法，更新统计信息。用以给 TiDB 优化器以更准确的统计信息来提供 SQL 性能优化。

7.3.6 批量删除

需要删除表中多行的数据，可选择 DELETE 示例，并使用 WHERE 子句过滤需要删除的数据。

但如果你需要删除大量行（数万或更多）的时候，建议使用一个迭代，每次都只删除一部分数据，直到删除全部完成。这是因为 TiDB 单个事务大小限制为 txn-total-size-limit（默认为 100MB）。你可以在程序或脚本中使用循环来完成操作。

本页提供了编写脚本来处理循环删除的示例，该示例演示了应如何进行 SELECT 和 DELETE 的组合，完成循环删除。

7.3.6.1 编写批量删除循环

在你的应用或脚本的循环中，编写一个 DELETE 语句，使用 WHERE 子句过滤需要删除的行，并使用 LIMIT 限制单次删除的数据条数。

7.3.6.2 批量删除例子

假设发现在特定时间段内，发生了业务错误，需要删除这期间内的所有 rating 的数据，例如，2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。并且在 15 分钟内，有大于 1 万条数据被写入，此时请使用循环删除的方式进行删除：

在 Java 中，批量删除程序类似于以下内容：

```
package com.pingcap.bulkDelete;

import com.mysql.cj.jdbc.MysqlDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.Calendar;
import java.util.concurrent.TimeUnit;

public class BatchDeleteExample
{
   public static void main(String[] args) throws InterruptedException {
      // Configure the example database connection.

      // Create a mysql data source instance.
      MysqlDataSource mysqlDataSource = new MysqlDataSource();

      // Set server name, port, database name, username and password.
      mysqlDataSource.setServerName("localhost");
      mysqlDataSource.setPortNumber(4000);
      mysqlDataSource.setDatabaseName("bookshop");
      mysqlDataSource.setUser("root");
      mysqlDataSource.setPassword("");

      Integer updateCount = -1;
      while (updateCount != 0) {
         updateCount = batchDelete(mysqlDataSource);
      }
   }

   public static Integer batchDelete (MysqlDataSource ds) {
      try (Connection connection = ds.getConnection()) {
         String sql = "DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= ? AND  `rat
ed_at` <= ? LIMIT 1000";
         PreparedStatement preparedStatement = connection.prepareStatement(sql);
         Calendar calendar = Calendar.getInstance();
         calendar.set(Calendar.MILLISECOND, 0);
```

```
            calendar.set(2022, Calendar.APRIL, 15, 0, 0, 0);
            preparedStatement.setTimestamp(1, new Timestamp(calendar.getTimeInMillis()));

            calendar.set(2022, Calendar.APRIL, 15, 0, 15, 0);
            preparedStatement.setTimestamp(2, new Timestamp(calendar.getTimeInMillis()));

            int count = preparedStatement.executeUpdate();
            System.out.println("delete " + count + " data");

            return count;
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return -1;
    }
}
```

每次迭代中，DELETE 最多删除 1000 行时间段为 2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。

在 Golang 中，批量删除程序类似于以下内容：

```
package main

import (
    "database/sql"
    "fmt"
    "time"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:@tcp(127.0.0.1:4000)/bookshop")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    affectedRows := int64(-1)
```

```
    startTime := time.Date(2022, 04, 15, 0, 0, 0, 0, time.UTC)
    endTime := time.Date(2022, 04, 15, 0, 15, 0, 0, time.UTC)

    for affectedRows != 0 {
        affectedRows, err = deleteBatch(db, startTime, endTime)
        if err != nil {
            panic(err)
        }
    }
}

// deleteBatch delete at most 1000 lines per batch
func deleteBatch(db *sql.DB, startTime, endTime time.Time) (int64, error) {
    bulkUpdateSql := fmt.Sprintf("DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >=
 ? AND  `rated_at` <= ? LIMIT 1000")
    result, err := db.Exec(bulkUpdateSql, startTime, endTime)
    if err != nil {
        return -1, err
    }
    affectedRows, err := result.RowsAffected()
    if err != nil {
        return -1, err
    }

    fmt.Printf("delete %d data\n", affectedRows)
    return affectedRows, nil
}
```

每次迭代中，DELETE 最多删除 1000 行时间段为 2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。

在 Python 中，批量删除程序类似于以下内容：

```
import MySQLdb
import datetime
import time

connection = MySQLdb.connect(
    host="127.0.0.1",
    port=4000,
    user="root",
    password="",
    database="bookshop",
```

```
    autocommit=True
)

with connection:
    with connection.cursor() as cursor:
        start_time = datetime.datetime(2022, 4, 15)
        end_time = datetime.datetime(2022, 4, 15, 0, 15)
        affect_rows = -1
        while affect_rows != 0:
            delete_sql = "DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= %s AND `r
ated_at` <= %s LIMIT 1000"
            affect_rows = cursor.execute(delete_sql, (start_time, end_time))
            print(f'delete {affect_rows} data')
            time.sleep(1)
```

每次迭代中，DELETE 最多删除 1000 行时间段为 2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。

7.3.7 非事务批量删除

TiDB 支持非事务 DML 语句特性。

7.3.7.1 使用前提

在使用非事务批量删除前，请先**仔细**阅读非事务 DML 语句。非事务批量删除，本质是以牺牲事务的原子性、隔离性为代价，增强批量数据处理场景下的性能和易用性。

因此在使用过程中，需要极为小心，否则，因为操作的非事务特性，在误操作时会导致严重的后果（如数据丢失等）。

7.3.7.2 非事务批量删除 SQL 语法

非事务批量删除的 SQL 语法如下：

```
BATCH ON {shard_column} LIMIT {batch_size} {delete_statement};
```

参数	描述
{shard_column}	非事务批量删除的划分列
{batch_size}	非事务批量删除的每批大小
{delete_statement}	删除语句

此处仅展示非事务批量删除的简单用法，详细文档可参考 TiDB 的非事务 DML 语句。

7.3.7.3 非事务批量删除使用示例

以上方批量删除例子场景为例，可使用以下 SQL 语句进行非事务批量删除：

```
BATCH ON `rated_at` LIMIT 1000 DELETE FROM `ratings` WHERE `rated_at` >= "2022-04-15 00:00:00" AND `rated_at` <= "2022-04-15 00:15:00";
```

7.4 使用 TTL (Time to Live) 定期删除过期数据

Time to Live (TTL) 提供了行级别的生命周期控制策略。通过为表设置 TTL 属性，平凯数据库可以周期性地自动检查并清理表中的过期数据。此功能在一些场景可以有效节省存储空间、提升性能。

TTL 常见的使用场景：

- 定期删除验证码、短网址记录
- 定期删除不需要的历史订单
- 自动删除计算的中间结果

TTL 设计的目标是在不影响在线读写负载的前提下，帮助用户周期性且及时地清理不需要的数据。TTL 会以表为单位，并发地分发不同的任务到不同的 TiDB Server 节点上，进行并行删除处理。TTL 并不保证所有过期数据立即被删除，也就是说即使数据过期了，客户端仍然有可能在这之后的一段时间内读到过期的数据，直到其真正的被后台处理任务删除。

7.4.1 语法

你可以通过 CREATE TABLE 或 ALTER TABLE 语句来配置表的 TTL 功能。

7.4.1.1 创建具有 TTL 属性的表

- 创建一个具有 TTL 属性的表：

```
CREATE TABLE t1 (
   id int PRIMARY KEY,
   created_at TIMESTAMP
) TTL = `created_at` + INTERVAL 3 MONTH;
```

上面的例子创建了一张表 t1，并指定了 created_at 为 TTL 的时间列，表示数据的创建时间。同时，它还通过 INTERVAL 3 MONTH 设置了表中行的最长存活时间为 3 个月。超过了此时长的过期数据会在之后被删除。

- 设置 TTL_ENABLE 属性来开启或关闭清理过期数据的功能：

```
CREATE TABLE t1 (
   id int PRIMARY KEY,
   created_at TIMESTAMP
) TTL = `created_at` + INTERVAL 3 MONTH TTL_ENABLE = 'OFF';
```

如果 TTL_ENABLE 被设置成了 OFF，则即使设置了其他 TTL 选项，当前表也不会自动清理过期数据。对于一个设置了 TTL 属性的表，TTL_ENABLE 在缺省条件下默认为 ON。

- 为了与 MySQL 兼容，你也可以使用注释语法来设置 TTL：

```
CREATE TABLE t1 (
   id int PRIMARY KEY,
   created_at TIMESTAMP
) /*T![ttl] TTL = `created_at` + INTERVAL 3 MONTH TTL_ENABLE = 'OFF'*/;
```

在 TiDB 环境中，使用表的 TTL 属性和注释语法来配置 TTL 是等价的。在 MySQL 环境中，会自动忽略注释中的内容，并创建普通的表。

7.4.1.2 修改表的 TTL 属性

- 修改表的 TTL 属性：

  ```
  ALTER TABLE t1 TTL = `created_at` + INTERVAL 1 MONTH;
  ```

  上面的语句既支持修改已配置 TTL 属性的表，也支持为一张非 TTL 的表添加 TTL 属性。

- 单独修改 TTL 表的 TTL_ENABLE 值：

  ```
  ALTER TABLE t1 TTL_ENABLE = 'OFF';
  ```

- 清除一张表的所有 TTL 属性：

  ```
  ALTER TABLE t1 REMOVE TTL;
  ```

7.4.1.3 TTL 和数据类型的默认值

TTL 可以和数据类型的默认值一起使用。以下是两种常见的用法示例：

- 使用 DEFAULT CURRENT_TIMESTAMP 来指定某一列的默认值为该行的创建时间，并用这一列作为 TTL 的时间列，创建时间超过 3 个月的数据将被标记为过期：

  ```
  CREATE TABLE t1 (
    id int PRIMARY KEY,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  ) TTL = `created_at` + INTERVAL 3 MONTH;
  ```

- 指定某一列的默认值为该行的创建时间或更新时间，并用这一列作为 TTL 的时间列，创建时间或更新时间超过 3 个月的数据将被标记为过期：

  ```
  CREATE TABLE t1 (
    id int PRIMARY KEY,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRE
  NT_TIMESTAMP
  ) TTL = `created_at` + INTERVAL 3 MONTH;
  ```

7.4.1.4 TTL 和生成列

TTL 可以和生成列一起使用，用来表达更加复杂的过期规则。例如：

```
CREATE TABLE message (
    id int PRIMARY KEY,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    image bool,
    expire_at TIMESTAMP AS (IF(image,
            created_at + INTERVAL 5 DAY,
            created_at + INTERVAL 30 DAY
    ))
) TTL = `expire_at` + INTERVAL 0 DAY;
```

上述语句的消息以 expire_at 列来作为过期时间，并按照消息类型来设定。如果是图片，则 5 天后过期，不然就 30 天后过期。

TTL 还可以和 JSON 类型 一起使用。例如：

```
CREATE TABLE orders (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    order_info JSON,
    created_at DATE AS (JSON_EXTRACT(order_info, '$.created_at')) VIRTUAL
) TTL = `created_at` + INTERVAL 3 month;
```

7.4.2 TTL 任务

对于每张设置了 TTL 属性的表，TiDB 内部会定期调度后台任务来清理过期的数据。你可以通过给表设置 TTL_JOB_INTERVAL 属性来自定义任务的执行周期，比如通过下面的语句将后台清理任务设置为每 24 小时执行一次：

```
ALTER TABLE orders TTL_JOB_INTERVAL = '24h';
```

TTL_JOB_INTERVAL 的默认值是 1h。

在执行 TTL 任务时，TiDB 会基于 Region 的数量将表拆分为最多 64 个子任务。这些子任务会被分发到不同的 TiDB 节点中执行。你可以通过设置系统变量 tidb_ttl_running_tasks 来限制整个集群中同时执行的 TTL 子任务数量。然而，并非

所有表的 TTL 任务都可以被拆分为子任务。请参考使用限制以了解哪些表的 TTL 任务不能被拆分。

如果想禁止 TTL 任务的执行，除了可以设置表属性 TTL_ENABLE='OFF' 外，也可以通过设置全局变量 tidb_ttl_job_enable 关闭整个集群的 TTL 任务的执行。

```
SET @@global.tidb_ttl_job_enable = OFF;
```

在某些场景下，你可能希望只允许在每天的某个时间段内调度后台的 TTL 任务，此时可以设置全局变量 tidb_ttl_job_schedule_window_start_time 和 tidb_ttl_job_schedule_window_end_time 来指定时间窗口，比如：

```
SET @@global.tidb_ttl_job_schedule_window_start_time = '01:00 +0000';
SET @@global.tidb_ttl_job_schedule_window_end_time = '05:00 +0000';
```

上述语句只允许在 UTC 时间的凌晨 1 点到 5 点调度 TTL 任务。默认情况下的时间窗口设置为 00:00 +0000 到 23:59 +0000，即允许所有时段的任务调度。

7.4.3 TTL 的可观测性

TiDB 会定时采集 TTL 的运行时信息，并在 Grafana 中提供了相关指标的可视化图表。你可以在 TiDB -> TTL 的面板下看到这些信息。指标详情见 TiDB 重要监控指标详解 中的 TTL 部分。

同时，可以通过以下三个系统表获得 TTL 任务执行的更多信息：

- mysql.tidb_ttl_table_status 表中包含了所有 TTL 表的上一次执行与正在执行的 TTL 任务的信息。以其中一行为例：

  ```
  TABLE mysql.tidb_ttl_table_status LIMIT 1\G
  *************************** 1. row ***************************
                    table_id: 85
           parent_table_id: 85
          table_statistics: NULL
              last_job_id: 0b4a6d50-3041-4664-9516-5525ee6d9f90
       last_job_start_time: 2023-02-15 20:43:46
      last_job_finish_time: 2023-02-15 20:44:46
       last_job_ttl_expire: 2023-02-15 19:43:46
  ```

```
            last_job_summary: {"total_rows":4369519,"success_rows":4369519,"error_
rows":0,"total_scan_task":64,"scheduled_scan_task":64,"finished_scan_task":64}
              current_job_id: NULL
        current_job_owner_id: NULL
      current_job_owner_addr: NULL
   current_job_owner_hb_time: NULL
     current_job_start_time: NULL
     current_job_ttl_expire: NULL
         current_job_state: NULL
        current_job_status: NULL
current_job_status_update_time: NULL
1 row in set (0.040 sec)
```

其中列 table_id 为分区表 ID，而 parent_table_id 为表的 ID，与 information_schema.tables 表中的 ID 对应。如果表不是分区表，则 table_id 与 parent_table_id 总是相等。

列 {last, current}_job_{start_time, finish_time, ttl_expire} 分别描述了过去和当前 TTL 任务的开始时间、结束时间和过期时间。last_job_summary 列描述了上一次 TTL 任务的执行情况，包括总行数、成功行数、失败行数。

- mysql.tidb_ttl_task 表中包含了正在执行的 TTL 子任务。单个 TTL 任务会被拆分为多个子任务，该表中记录了正在执行的这些子任务的信息。
- mysql.tidb_ttl_job_history 表中记录了 TTL 任务的执行历史。TTL 任务的历史记录将被保存 90 天。以一行为例：

```
TABLE mysql.tidb_ttl_job_history LIMIT 1\G
*************************** 1. row ***************************
         job_id: f221620c-ab84-4a28-9d24-b47ca2b5a301
       table_id: 85
 parent_table_id: 85
   table_schema: test_schema
     table_name: TestTable
  partition_name: NULL
    create_time: 2023-02-15 17:43:46
    finish_time: 2023-02-15 17:45:46
     ttl_expire: 2023-02-15 16:43:46
   summary_text: {"total_rows":9588419,"success_rows":9588419,"error_rows":0,"t
```

```
otal_scan_task":63,"scheduled_scan_task":63,"finished_scan_task":63}
   expired_rows: 9588419
   deleted_rows: 9588419
error_delete_rows: 0
        status: finished
```

其中列 table_id 为分区表 ID，而 parent_table_id 为表的 ID，与 information_schema.tables 表中的 ID 对应。table_schema、table_name、partition_name 分别对应表示数据库、表名、分区名。create_time、finish_time、ttl_expire 分别表示 TTL 任务的创建时间、结束时间和过期时间。expired_rows 与 deleted_rows 表示过期行数与成功删除的行数。

7.4.4 平凯数据库数据迁移工具兼容性

TTL 功能能够与 TiDB 的迁移、备份、恢复工具一同使用。

工具名称	最低兼容版本	说明
Backup & Restore (BR)	v6.6.0	恢复时会自动将表的 TTL_ENABLE 属性设置为 OFF，关闭 TTL。这样可以防止 TiDB 在备份恢复后立即删除过期的数据。此时你需要手动重新配置 TTL_ENABLE 属性来重新开启各个表的 TTL。
TiDB Lightning	v6.6.0	导入后如果表中有 TTL 属性，会自动将表的 TTL_ENABLE 属性设置为 OFF，关闭 TTL。这样可以防止 TiDB 在导入后立即删除过期的数据。此时你需要手动重新配置 TTL_ENABLE 属性来重新开启各个表的 TTL。
TiCDC	v7.0.0	上游的 TTL 删除将会同步至下游。因此，为了防止重复删

工具名称	最低兼容版本	说明
		除，下游表的 TTL_ENABLE 属性将被强制设置为 OFF。

7.4.5 与平凯数据库其他特性的兼容性

特性名称	说明
FLASHBACK TABLE	FLASHBACK TABLE 语句会将每个表的 TTL_ENABLE 属性强制设置为 OFF。这样可以防止 TiDB 在 FLASHBACK 后立即删除过期的数据。此时你需要手动重新配置 TTL_ENABLE 属性来重新开启各个表的 TTL。
FLASHBACK DATABASE	FLASHBACK DATABASE 语句会将每个表的 TTL_ENABLE 属性强制设置为 OFF。这样可以防止 TiDB 在 FLASHBACK 后立即删除过期的数据。此时你需要手动重新配置 TTL_ENABLE 属性来重新开启各个表的 TTL。
FLASHBACK CLUSTER	FLASHBACK CLUSTER 会将 TIDB_TTL_JOB_ENABLE 系统变量设置为 OFF，同时表的 TTL_ENABLE 属性将保持原样。

7.4.6 使用限制

目前，TTL 特性具有以下限制:

- 不允许在临时表上设置 TTL 属性，包括本地临时表和全局临时表。
- 具有 TTL 属性的表不支持作为外键约束的主表被其他表引用。
- 不保证所有过期数据立即被删除，过期数据被删除的时间取决于后台清理任务的调度周期和调度窗口。
- 对于使用聚簇索引的表，仅支持在以下场景中将 TTL 任务拆分成多个子任务：
    - 主键或者复合主键的第一列为整数或二进制字符串类型。其中，二进制字符串类型主要指下面几种：
        - CHAR(N) CHARACTER SET BINARY
        - VARCHAR(N) CHARACTER SET BINARY
        - BINARY(N)

        - VARBINARY(N)
        - BIT(N)
    - 主键或者复合主键的第一列的字符集为 utf8 或者 utf8mb4，且排序规则设置为 utf8_bin、 utf8mb4_bin 或者 utf8mb4_0900_bin。
- 对于主键第一列的字符集类型是 utf8 或者 utf8mb4 的表，仅会根据 ASCII 可见字符的范围进行子任务拆分。如果大量的主键值具有相同的 ASCII 前缀，可能会造成任务拆分不均匀。
- 对于不支持拆分 TTL 子任务的表，TTL 任务只能在一个 TiDB 节点上按顺序执行。此时如果表中的数据量较大，TTL 任务的执行可能会变得缓慢。

7.4.7 常见问题

- 如何判断删除的速度是否够快，能够保持数据总量相对稳定?

  在 Grafana TiDB 面板中，监控项 TTL Insert Rows Per Hour 记录了前一小时总共插入数据的数量。相应的 TTL Delete Rows Per Hour 记录了前一小时 TTL 任务总共删除的数据总量。如果 TTL Insert Rows Per Hour 长期高于 TTL Delete Rows Per Hour， 说明插入的速度高于删除的速度，数据总量将会上升。例如：

  insert fast example

  值得注意的是，由于 TTL 并不能保证数据立即被删除，且当前插入的数据将会在将来的 TTL 任务中才会被删除，哪怕短时间内 TTL 删除的速度低于插入的速度，也不能说明 TTL 的效率一定过慢。需要结合具体情况分析。

- 如何判断 TTL 任务的瓶颈在扫描还是删除?

  观察面板中 TTL Scan Worker Time By Phase 与 TTL Delete Worker Time By Phase 监控项。如果 scan worker 处于 dispatch 状态的时间有很大占比，且 delete worker 很少处于 idle 状态，那么说明 scan worker 在等待 delete

worker 完成删除工作，如果此时集群资源仍然较为宽松，可以考虑提高 tidb_ttl_delete_worker_count 来提高删除的 worker 数量。例如：

scan fast example

与之相对，如果 scan worker 很少处于 dispatch 的状态，且 delete worker 长期处于 idle 阶段，那么说明 delete worker 闲置，且 scan worker 较为忙碌。例如：

delete fast example

TTL 任务中扫描与删除的占比与机器配置、数据分布都有关系，所以每一时刻的数据只能代表正在执行的 TTL Job 的情况。用户可以通过查询表 mysql.tidb_ttl_job_history 来判断某一时刻运行的 TTL Job 对应哪一张表。

- 如何合理配置 tidb_ttl_scan_worker_count 和 tidb_ttl_delete_worker_count?

    1. 可以参考问题 “如何判断 TTL 任务的瓶颈在扫描还是删除？” 来考虑提升 tidb_ttl_scan_worker_count 还是 tidb_ttl_delete_worker_count。
    2. 如果 TiKV 节点数量较多，提升 tidb_ttl_scan_worker_count 能够使 TTL 任务负载更加均匀。

    由于过高的 TTL worker 数量将会造成较大的压力，所以需要综合观察 TiDB 的 CPU 水平与 TiKV 的磁盘与 CPU 使用量。根据不同场景和需求（需要尽量加速 TTL，或是需要减少 TTL 对其他请求的影响）来调整

tidb_ttl_scan_worker_count 与 tidb_ttl_delete_worker_count，从而提升 TTL 扫描和删除数据的速度，或降低 TTL 任务对性能的影响。

7.5 预处理语句

预处理语句是一种将多个仅有参数不同的 SQL 语句进行模板化的语句，它让 SQL 语句与参数进行了分离。可以用它提升 SQL 语句的：

- 安全性：因为参数和语句已经分离，所以避免了 SQL 注入攻击的风险。
- 性能：因为语句在 TiDB 端被预先解析，后续执行只需要传递参数，节省了完整 SQL 解析、拼接 SQL 语句字符串以及网络传输的代价。

在大部分的应用程序中，SQL 语句是可以被枚举的，可以使用有限个 SQL 语句来完成整个应用程序的数据查询，所以使用预处理语句是最佳实践之一。

7.5.1 SQL 语法

本节将介绍创建、使用及删除预处理语句的 SQL 语法。

7.5.1.1 创建预处理语句

```
PREPARE {prepared_statement_name} FROM '{prepared_statement_sql}';
```

参数	描述
{prepared_statement_name}	预处理语句名称
{prepared_statement_sql}	预处理语句 SQL，以英文半角问号做占位符

你可查看 PREPARE 语句获得更多信息。

7.5.1.2 使用预处理语句

预处理语句仅可使用用户变量作为参数，因此，需先使用 SET 语句设置变量后，供 EXECUTE 语句调用预处理语句。

```
SET @{parameter_name} = {parameter_value};
EXECUTE {prepared_statement_name} USING @{parameter_name};
```

参数	描述
{parameter_name}	用户参数名
{parameter_value}	用户参数值
{prepared_statement_name}	预处理语句名称，需和创建预处理语句中定义的名称一致

你可查看 EXECUTE 语句获得更多信息。

7.5.1.3 删除预处理语句

DEALLOCATE PREPARE {prepared_statement_name};

参数	描述
{prepared_statement_name}	预处理语句名称，需和创建预处理语句中定义的名称一致

你可查看 DEALLOCATE 语句获得更多信息。

7.5.2 例子

本节以使用预处理语句，完成查询数据和插入数据两个场景的示例。

7.5.2.1 查询示例

例如，需要查询 Bookshop 应用 中，id 为 1 的书籍信息。

使用 SQL 查询示例：

```
PREPARE `books_query` FROM 'SELECT * FROM `books` WHERE `id` = ?';
```

运行结果为：

```
Query OK, 0 rows affected (0.01 sec)
```

```
SET @id = 1;
```

运行结果为：

```
Query OK, 0 rows affected (0.04 sec)
```

```
EXECUTE `books_query` USING @id;
```

运行结果为：

```
+---------+--------------------------------+--------+---------------------+-------+--------+
| id      | title                          | type   | published_at        | stock | price  |
+---------+--------------------------------+--------+---------------------+-------+--------+
| 1       | The Adventures of Pierce Wehner | Comics | 1904-06-06 20:46:25 |   586 | 411.66 |
+---------+--------------------------------+--------+---------------------+-------+--------+
1 row in set (0.05 sec)
```

使用 Java 查询示例：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource
try (Connection connection = ds.getConnection()) {
   PreparedStatement preparedStatement = connection.prepareStatement("SELECT * FR
OM `books` WHERE `id` = ?");
   preparedStatement.setLong(1, 1);

   ResultSet res = preparedStatement.executeQuery();
   if(!res.next()) {
      System.out.println("No books in the table with id 1");
   } else {
      // got book's info, which id is 1
      System.out.println(res.getLong("id"));
      System.out.println(res.getString("title"));
      System.out.println(res.getString("type"));
   }
} catch (SQLException e) {
   e.printStackTrace();
}
```

7.5.2.2 插入示例

还是使用 books 表 为例，需要插入一个 title 为 TiDB Developer Guide, type 为 Science & Technology, stock 为 100, price 为 0.0, published_at 为 插入的当前时间 的书籍信息。需要注意的是，books 表的主键包含 AUTO_RANDOM 属性，无需指定它。如果你对插入数据还不了解，可以在插入数据一节了解更多数据插入的相关信息。

使用 SQL 插入数据示例如下：

```
PREPARE `books_insert` FROM 'INSERT INTO `books` (`title`, `type`, `stock`, `price`, `published_at`) VALUES (?, ?, ?, ?, ?);';
```

运行结果为：

```
Query OK, 0 rows affected (0.03 sec)
```

```
SET @title = 'TiDB Developer Guide';
SET @type = 'Science & Technology';
SET @stock = 100;
SET @price = 0.0;
SET @published_at = NOW();
```

运行结果为：

```
Query OK, 0 rows affected (0.04 sec)
```

```
EXECUTE `books_insert` USING @title, @type, @stock, @price, @published_at;
```

运行结果为：

```
Query OK, 1 row affected (0.03 sec)
```

使用 Java 插入数据示例如下：

```
try (Connection connection = ds.getConnection()) {
    String sql = "INSERT INTO `books` (`title`, `type`, `stock`, `price`, `published_at`) VALUES (?, ?, ?, ?, ?);";
    PreparedStatement preparedStatement = connection.prepareStatement(sql);

    preparedStatement.setString(1, "TiDB Developer Guide");
    preparedStatement.setString(2, "Science & Technology");
    preparedStatement.setInt(3, 100);
    preparedStatement.setBigDecimal(4, new BigDecimal("0.0"));
    preparedStatement.setTimestamp(5, new Timestamp(Calendar.getInstance().getTimeInMillis()));

    preparedStatement.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
}
```

可以看到，JDBC 帮你管控了预处理语句的生命周期，而无需你在应用程序里手动使用预处理语句的创建、使用、删除等。但值得注意的是，因为 TiDB 兼容 MySQL 协议，在客户端使用 MySQL JDBC Driver 的过程中，其默认配置并非开启 ***服务端*** 的预处理语句选项，而是使用客户端的预处理语句。你需要关注以下配置项，来获得在 JDBC 下 TiDB 服务端预处理语句的支持，及在你的使用场景下的最佳配置：

参数	作用	推荐场景	推荐配置
useServerPrepStmts	是否使用服务端开启预处理语句支持	在需要多次使用预处理语句时	true
cachePrepStmts	客户端是否缓存预处理语句	useServerPrepStmts=true 时	true
prepStmtCacheSqlLimit	预处理语句最大大小（默认 256 字符）	预处理语句大于 256 字符时	按实际预处理语句大小配置
prepStmtCacheSize	预处理语句最大缓存数量（默认 25 条）	预处理语句数量大于 25 条时	按实际预处理语句数量配置

在此处给出一个较为的通用场景的 JDBC 连接字符串配置，以 Host: 127.0.0.1，Port: 4000，用户: root，密码: 空，默认数据库: test 为例：

```
jdbc:mysql://127.0.0.1:4000/test?user=root&useConfigs=maxPerformance&useServerPrepStmts=true&prepStmtCacheSqlLimit=2048&prepStmtCacheSize=256&rewriteBatchedStatements=true&allowMultiQueries=true
```

你也可以查看插入行一章，来查看是否需要在插入数据场景下更改其他 JDBC 的参数。

有关 Java 的完整示例，可参阅：

- 平凯数据库和 JDBC 的简单 CRUD 应用程序
- 平凯数据库和 Hibernate 的简单 CRUD 应用程序
- 使用 Spring Boot 构建平凯数据库应用程序

8 数据读取

8.1 单表查询

在这个章节当中，将开始介绍如何使用 SQL 来对数据库中的数据进行查询。

8.1.1 开始之前

下面将围绕 Bookshop 这个应用程序来对 TiDB 的数据查询部分展开介绍。

在阅读本章节之前，你需要做以下准备工作：

1. 构建 TiDB 集群（推荐使用 TiUP）。
2. 导入 Bookshop 应用程序的表结构和示例数据。
3. 连接到 TiDB。

8.1.2 简单的查询

在 Bookshop 应用程序的数据库当中，authors 表存放了作家们的基础信息，可以通过 SELECT ... FROM ... 语句将数据从数据库当中调取出去。

在 MySQL Client 等客户端输入并执行如下 SQL 语句：

```
SELECT id, name FROM authors;
```

输出结果如下：

```
+------------+-------------------------+
| id         | name                    |
+------------+-------------------------+
|       6357 | Adelle Bosco            |
|     345397 | Chanelle Koepp          |
|     807584 | Clementina Ryan         |
|     839921 | Gage Huel               |
|     850070 | Ray Armstrong           |
|     850362 | Ford Waelchi            |
|     881210 | Jayme Gutkowski         |
|    1165261 | Allison Kuvalis         |
|    1282036 | Adela Funk              |
```

```
...
| 4294957408 | Lyla Nitzsche          |
+------------+------------------------+
20000 rows in set (0.05 sec)
```

在 Java 语言当中，可以通过声明一个 Author 类来定义如何存放作者的基础信息，根据数据的类型和取值范围从 Java 语言当中选择合适的数据类型来存放对应的数据，例如：

- 使用 Int 类型变量存放 int 类型的数据。
- 使用 Long 类型变量存放 bigint 类型的数据。
- 使用 Short 类型变量存放 tinyint 类型的数据。
- 使用 String 类型变量存放 varchar 类型的数据。
- ...

```
public class Author {
    private Long id;
    private String name;
    private Short gender;
    private Short birthYear;
    private Short deathYear;

    public Author() {}

    // Skip the getters and setters.
}

public class AuthorDAO {

    // Omit initialization of instance variables.

    public List<Author> getAuthors() throws SQLException {
        List<Author> authors = new ArrayList<>();

        try (Connection conn = ds.getConnection()) {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT id, name FROM authors");
            while (rs.next()) {
                Author author = new Author();
                author.setId(rs.getLong("id"));
                author.setName(rs.getString("name"));
```

```
                authors.add(author);
            }
        }
        return authors;
    }
}
```

- 在获得数据库连接之后，你可以通过 conn.createStatement() 语句创建一个 Statement 实例对象。
- 然后调用 stmt.executeQuery("query_sql") 方法向 TiDB 发起一个数据库查询请求。
- 数据库返回的查询结果将会存放到 ResultSet 当中，通过遍历 ResultSet 对象可以将返回结果映射到此前准备的 Author 类对象当中。

8.1.3 对结果进行筛选

查询得到的结果非常多，但是并不都是你想要的？可以通过 WHERE 语句对查询的结果进行过滤，从而找到想要查询的部分。

例如，想要查找众多作家当中找出在 1998 年出生的作家：

在 SQL 中，可以使用 WHERE 子句添加筛选的条件：

```
SELECT * FROM authors WHERE birth_year = 1998;
```

对于 Java 程序而言，可以通过同一个 SQL 来处理带有动态参数的数据查询请求。

将参数拼接到 SQL 语句当中也许是一种方法，但是这可能不是一个好的主意，因为这会给应用程序带来潜在的 SQL 注入风险。

在处理这类查询时，应该使用 PreparedStatement 来替代普通的 Statement。

```
public List<Author> getAuthorsByBirthYear(Short birthYear) throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        PreparedStatement stmt = conn.prepareStatement("""
```

```
        SELECT * FROM authors WHERE birth_year = ?;
        """);
        stmt.setShort(1, birthYear);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            authors.add(author);
        }
    }
    return authors;
}
```

8.1.4 对结果进行排序

使用 ORDER BY 语句可以让查询结果按照期望的方式进行排序。

例如，可以通过下面的 SQL 语句对 authors 表的数据按照 birth_year 列进行降序 (DESC) 排序，从而得到最年轻的作家列表。

```
SELECT id, name, birth_year
FROM authors
ORDER BY birth_year DESC;
```

```
public List<Author> getAuthorsSortByBirthYear() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
            SELECT id, name, birth_year
            FROM authors
            ORDER BY birth_year DESC;
            """);

        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            author.setBirthYear(rs.getShort("birth_year"));
```

```
            authors.add(author);
        }
    }
    return authors;
}
```

查询结果如下：

```
+----------+----------------------+------------+
| id       | name                 | birth_year |
+----------+----------------------+------------+
| 83420726 | Terrance Dach        | 2000       |
| 57938667 | Margarita Christiansen | 2000     |
| 77441404 | Otto Dibbert         | 2000       |
| 61338414 | Danial Cormier       | 2000       |
| 49680887 | Alivia Lemke         | 2000       |
| 45460101 | Itzel Cummings       | 2000       |
| 38009380 | Percy Hodkiewicz     | 2000       |
| 12943560 | Hulda Hackett        | 2000       |
| 1294029  | Stanford Herman      | 2000       |
| 111453184 | Jeffrey Brekke      | 2000       |
...
300000 rows in set (0.23 sec)
```

8.1.5 限制查询结果数量

如果希望 TiDB 只返回部分结果，可以使用 LIMIT 语句限制查询结果返回的记录数。

```
SELECT id, name, birth_year
FROM authors
ORDER BY birth_year DESC
LIMIT 10;
```

```
public List<Author> getAuthorsWithLimit(Integer limit) throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        PreparedStatement stmt = conn.prepareStatement("""
            SELECT id, name, birth_year
            FROM authors
```

```
            ORDER BY birth_year DESC
            LIMIT ?;
            """);
        stmt.setInt(1, limit);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            author.setBirthYear(rs.getShort("birth_year"));
            authors.add(author);
        }
    }
    return authors;
}
```

查询结果如下：

```
+-----------+-----------------------+------------+
| id        | name                  | birth_year |
+-----------+-----------------------+------------+
| 83420726  | Terrance Dach         | 2000       |
| 57938667  | Margarita Christiansen | 2000       |
| 77441404  | Otto Dibbert          | 2000       |
| 61338414  | Danial Cormier        | 2000       |
| 49680887  | Alivia Lemke          | 2000       |
| 45460101  | Itzel Cummings        | 2000       |
| 38009380  | Percy Hodkiewicz      | 2000       |
| 12943560  | Hulda Hackett         | 2000       |
| 1294029   | Stanford Herman       | 2000       |
| 111453184 | Jeffrey Brekke        | 2000       |
+-----------+-----------------------+------------+
10 rows in set (0.11 sec)
```

通过观察查询结果你会发现，在使用 LIMIT 语句之后，查询的时间明显缩短，这是 TiDB 对 LIMIT 子句进行优化后的结果，你可以通过 TopN 和 Limit 下推章节了解更多细节。

8.1.6 聚合查询

如果你想要关注数据整体的情况，而不是部分数据，你可以通过使用 GROUP BY 语句配合聚合函数，构建一个聚合查询来帮助你对数据的整体情况有一个更好的了解。

比如说，你希望知道哪些年出生的作家比较多，你可以将作家基本信息按照 birth_year 列进行分组，然后分别统计在当年出生的作家数量：

```
SELECT birth_year, COUNT(DISTINCT id) AS author_count
FROM authors
GROUP BY birth_year
ORDER BY author_count DESC;
```

```
public class AuthorCount {
    private Short birthYear;
    private Integer authorCount;

    public AuthorCount() {}

     // Skip the getters and setters.
}

public List<AuthorCount> getAuthorCountsByBirthYear() throws SQLException {
    List<AuthorCount> authorCounts = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
            SELECT birth_year, COUNT(DISTINCT id) AS author_count
            FROM authors
            GROUP BY birth_year
            ORDER BY author_count DESC;
            """);

        while (rs.next()) {
            AuthorCount authorCount = new AuthorCount();
            authorCount.setBirthYear(rs.getShort("birth_year"));
            authorCount.setAuthorCount(rs.getInt("author_count"));
            authorCounts.add(authorCount);
        }
```

```
    }
    return authorCount;
}
```

查询结果如下：

```
+------------+--------------+
| birth_year | author_count |
+------------+--------------+
|       1932 |          317 |
|       1947 |          290 |
|       1939 |          282 |
|       1935 |          289 |
|       1968 |          291 |
|       1962 |          261 |
|       1961 |          283 |
|       1986 |          289 |
|       1994 |          280 |
...
|       1972 |          306 |
+------------+--------------+
71 rows in set (0.00 sec)
```

除了 COUNT 函数外，TiDB 还支持了其他聚合函数。详情请参考 GROUP BY 聚合函数。

8.2 多表连接查询

很多时候，应用程序需要在一个查询当中使用到多张表的数据，这个时候可以通过 JOIN 语句将两张或多张表的数据组合在一起。

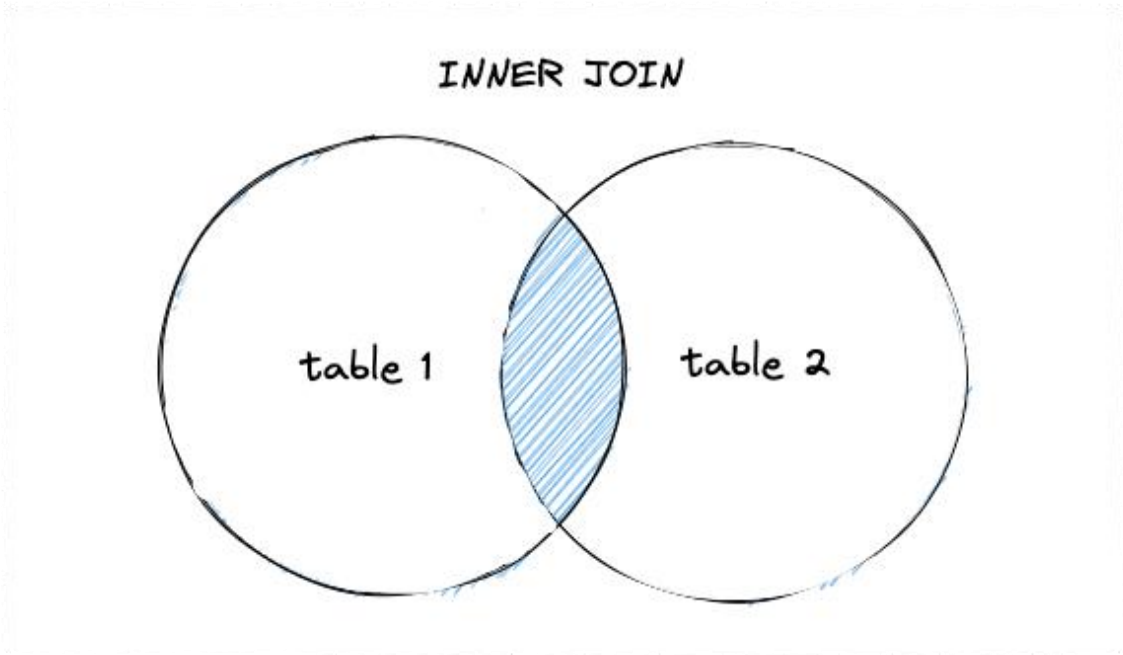
8.2.1 Join 类型

此节将详细叙述 Join 的连接类型。

8.2.1.1 内连接 INNER JOIN

内连接的连接结果只返回匹配连接条件的行。

例如，想要知道编写过最多书的作家是谁，需要将作家基础信息表 authors 与书籍作者表 book_authors 进行连接。



Inner Join

在下面的 SQL 语句当中，通过关键字 JOIN 声明要将左表 authors 和右表 book_authors 的数据行以内连接的方式进行连接，连接条件为 a.id = ba.author_id，那么连接的结果集当中将只会包含满足连接条件的行。假设有一个作家没有编写过任何书籍，那么他在 authors 表当中的记录将无法满足连接条件，因此也不会出现在结果集当中。

```
SELECT ANY_VALUE(a.id) AS author_id, ANY_VALUE(a.name) AS author_name, COUNT(b
a.book_id) AS books
FROM authors a
JOIN book_authors ba ON a.id = ba.author_id
GROUP BY ba.author_id
ORDER BY books DESC
LIMIT 10;
```

查询结果如下：

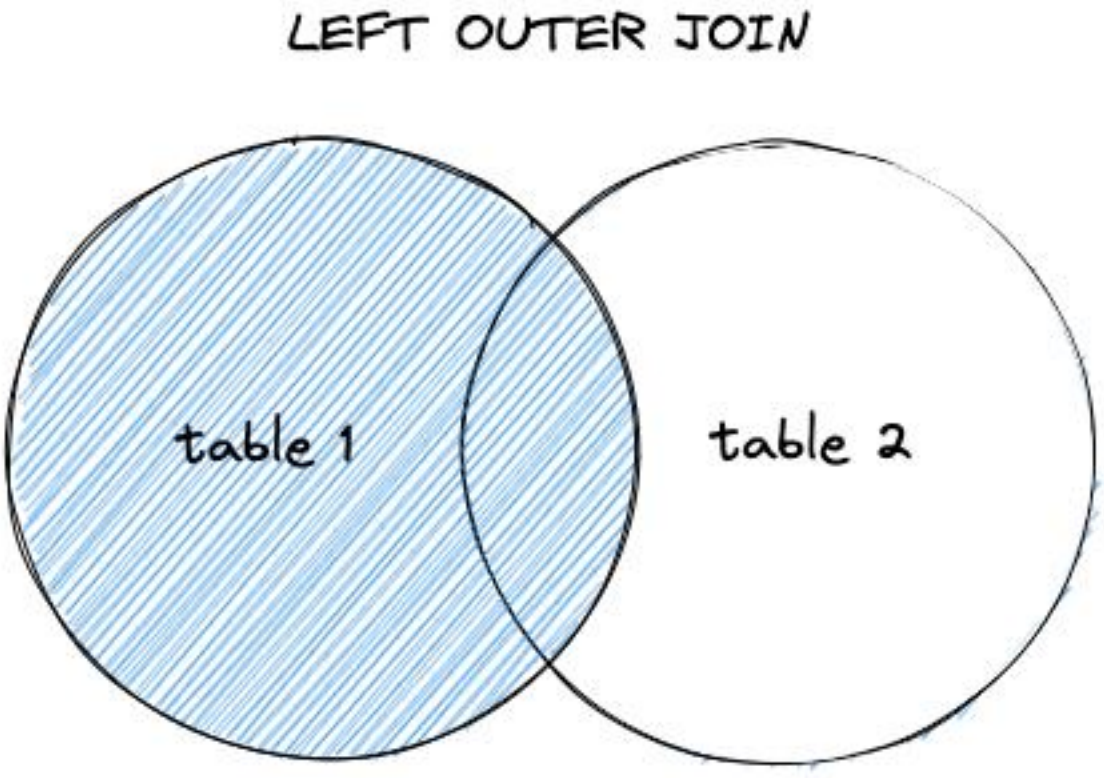
```
+------------+---------------+-------+
| author_id  | author_name    | books |
+------------+---------------+-------+
|  431192671 | Emilie Cassin  |     7 |
|  865305676 | Nola Howell    |     7 |
|  572207928 | Lamar Koch     |     6 |
| 3894029860 | Elijah Howe    |     6 |
| 1150614082 | Cristal Stehr  |     6 |
| 4158341032 | Roslyn Rippin  |     6 |
| 2430691560 | Francisca Hahn |     6 |
| 3346415350 | Leta Weimann   |     6 |
| 1395124973 | Albin Cole     |     6 |
| 2768150724 | Caleb Wyman    |     6 |
+------------+---------------+-------+
10 rows in set (0.01 sec)
```

在 Java 中内连接的示例如下：

```
public List<Author> getTop10AuthorsOrderByBooks() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
        SELECT ANY_VALUE(a.id) AS author_id, ANY_VALUE(a.name) AS author_name, COU
NT(ba.book_id) AS books
        FROM authors a
        JOIN book_authors ba ON a.id = ba.author_id
        GROUP BY ba.author_id
        ORDER BY books DESC
        LIMIT 10;
        """);
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("author_id"));
            author.setName(rs.getString("author_name"));
            author.setBooks(rs.getInt("books"));
            authors.add(author);
        }
    }
    return authors;
}
```

8.2.1.2 左外连接 LEFT OUTER JOIN

左外连接会返回左表中的所有数据行，以及右表当中能够匹配连接条件的值，如果在右表当中没有找到能够匹配的行，则使用 NULL 填充。



Left Outer Join

在一些情况下，希望使用多张表来完成数据的查询，但是并不希望因为不满足连接条件而导致数据集变小。

例如，在 Bookshop 应用的首页，希望展示一个带有平均评分的最新书籍列表。在这种情况下，最新的书籍可能是还没有经过任何人评分的，如果使用内连接就会导致这些无人评分的书籍信息被过滤掉，而这并不是期望的结果。

在下面的 SQL 语句当中，通过 LEFT JOIN 关键字声明左表 books 将以左外连接的方式与右表 ratings 进行连接，从而确保 books 表当中的所有记录都能得到返回。

```
SELECT b.id AS book_id, ANY_VALUE(b.title) AS book_title, AVG(r.score) AS average_scor
e
FROM books b
LEFT JOIN ratings r ON b.id = r.book_id
GROUP BY b.id
ORDER BY b.published_at DESC
LIMIT 10;
```

查询结果如下：

```
+------------+--------------------------------+---------------+
| book_id    | book_title                     | average_score |
+------------+--------------------------------+---------------+
| 3438991610 | The Documentary of lion        |        2.7619 |
| 3897175886 | Torey Kuhn                     |        3.0000 |
| 1256171496 | Elmo Vandervort                |        2.5500 |
| 1036915727 | The Story of Munchkin          |        2.0000 |
|  270254583 | Tate Kovacek                   |        2.5000 |
| 1280950719 | Carson Damore                  |        3.2105 |
| 1098041838 | The Documentary of grasshopper |        2.8462 |
| 1476566306 | The Adventures of Vince Sanford |        2.3529 |
| 4036300890 | The Documentary of turtle      |        2.4545 |
| 1299849448 | Antwan Olson                   |        3.0000 |
+------------+--------------------------------+---------------+
10 rows in set (0.30 sec)
```

看起来最新出版的书籍已经有了很多评分，为了验证上面所说的，通过 SQL 语句把 **The Documentary of lion** 这本书的所有评分给删掉：

```
DELETE FROM ratings WHERE book_id = 3438991610;
```

再次查询，你会发现 **The Documentary of lion** 这本书依然出现在结果集当中，但是通过右表 ratings 的 score 列计算得到的 average_score 列被填上了 NULL。

```
+------------+--------------------------------+---------------+
| book_id    | book_title                     | average_score |
+------------+--------------------------------+---------------+
| 3438991610 | The Documentary of lion        |          NULL |
| 3897175886 | Torey Kuhn                     |        3.0000 |
| 1256171496 | Elmo Vandervort                |        2.5500 |
| 1036915727 | The Story of Munchkin          |        2.0000 |
|  270254583 | Tate Kovacek                   |        2.5000 |
| 1280950719 | Carson Damore                  |        3.2105 |
```

```
| 1098041838 | The Documentary of grasshopper  |       2.8462 |
| 1476566306 | The Adventures of Vince Sanford |       2.3529 |
| 4036300890 | The Documentary of turtle       |       2.4545 |
| 1299849448 | Antwan Olson                    |       3.0000 |
+------------+---------------------------------+--------------+
10 rows in set (0.30 sec)
```

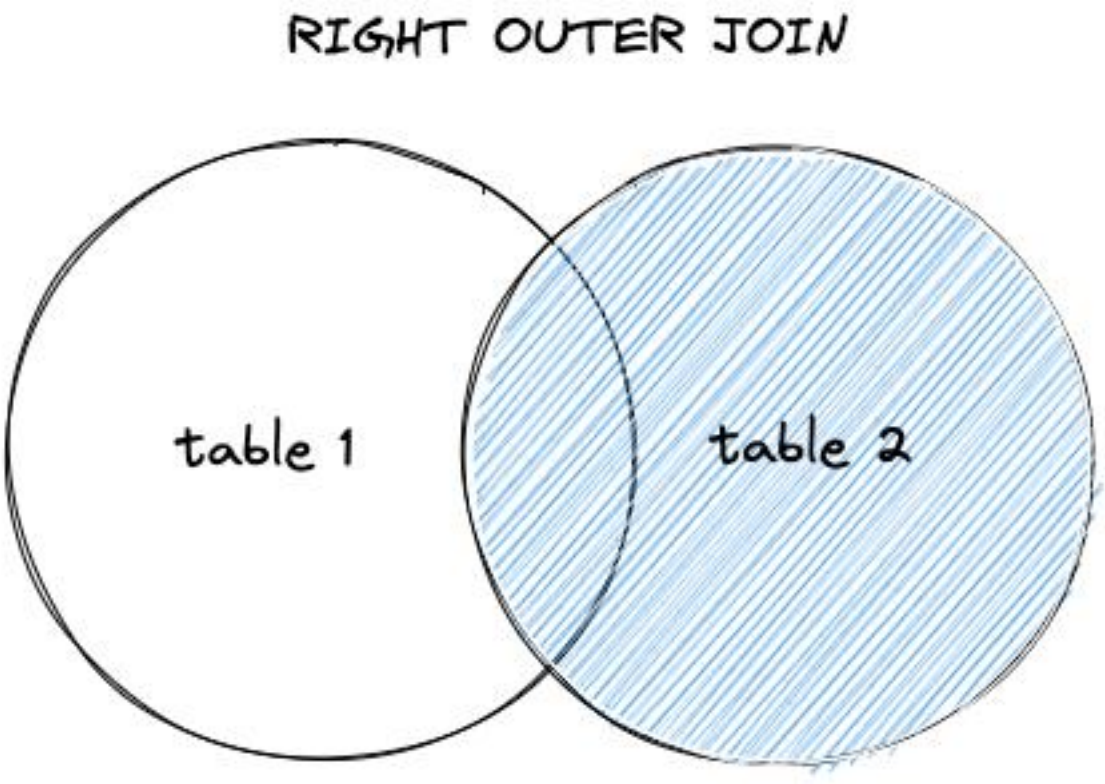
如果改成使用的是内连接 JOIN 结果会怎样？这就交给你来尝试了。

在 Java 中左外连接的示例如下：

```
public List<Book> getLatestBooksWithAverageScore() throws SQLException {
    List<Book> books = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
        SELECT b.id AS book_id, ANY_VALUE(b.title) AS book_title, AVG(r.score) AS average_
score
        FROM books b
        LEFT JOIN ratings r ON b.id = r.book_id
        GROUP BY b.id
        ORDER BY b.published_at DESC
        LIMIT 10;
        """);
        while (rs.next()) {
            Book book = new Book();
            book.setId(rs.getLong("book_id"));
            book.setTitle(rs.getString("book_title"));
            book.setAverageScore(rs.getFloat("average_score"));
            books.add(book);
        }
    }
    return books;
}
```

8.2.1.3 右外连接 RIGHT OUTER JOIN

右外连接返回右表中的所有记录，以及左表当中能够匹配连接条件的值，没有匹配的值则使用 NULL 填充。



Right Outer Join

8.2.1.4 交叉连接 CROSS JOIN

当连接条件恒成立时，两表之间的内连接称为交叉连接（又被称为“笛卡尔连接”）。交叉连接会把左表的每一条记录和右表的所有记录相连接，如果左表的记录数为 m，右表的记录数为 n，则结果集中会产生 m * n 条记录。

8.2.1.5 左半连接 LEFT SEMI JOIN

TiDB 在 SQL 语法层面上不支持 LEFT SEMI JOIN table_name，但是在执行计划层面，子查询相关的优化会将 semi join 作为改写后的等价 JOIN 查询默认的连接方式。

8.2.2 隐式连接

在显式声明连接的 JOIN 语句作为 SQL 标准出现之前，在 SQL 语句当中可以通过 FROM t1, t2 子句来连接两张或多张表，通过 WHERE t1.id = t2.id 子句来指定连接的

条件。你可以将其理解为隐式声明的连接，隐式连接会使用内连接的方式进行连接。

8.2.3 Join 相关算法

TiDB 支持下列三种常规的表连接算法，优化器会根据所连接表的数据量等因素来选择合适的 Join 算法去执行。你可以通过 EXPLAIN 语句来查看查询使用了何种算法进行 Join。

- Index Join
- Hash Join
- Merge Join

如果发现 TiDB 的优化器没有按照最佳的 Join 算法去执行。你也可以通过 Optimizer Hints 强制 TiDB 使用更好的 Join 算法去执行。

例如，假设上文当中的左连接查询的示例 SQL 使用 Hash Join 算法执行更快，而优化器并没有选择这种算法，你可以在 SELECT 关键字后面加上 Hint /*+ HASH_JOIN(b, r) */（注意：如果表名添加了别名，Hint 当中也应该使用表别名）。

```
EXPLAIN SELECT /*+ HASH_JOIN(b, r) */ b.id AS book_id, ANY_VALUE(b.title) AS book_title, AVG(r.score) AS average_score
FROM books b
LEFT JOIN ratings r ON b.id = r.book_id
GROUP BY b.id
ORDER BY b.published_at DESC
LIMIT 10;
```

Join 算法相关的 Hints：

- MERGE_JOIN(t1_name [, tl_name ...])
- INL_JOIN(t1_name [, tl_name ...])
- INL_HASH_JOIN(t1_name [, tl_name ...])
- HASH_JOIN(t1_name [, tl_name ...])

8.2.4 Join 顺序

在实际的业务场景中，多个表的 Join 语句是很常见的，而 Join 的执行效率和各个表参与 Join 的顺序有关。TiDB 使用 Join Reorder 算法来确定多个表进行 Join 的顺序。

当优化器选择的 Join 顺序并不够好时，你可以使用 STRAIGHT_JOIN 语法让 TiDB 强制按照 FROM 子句中所使用的表的顺序做联合查询。

```
EXPLAIN SELECT *
FROM authors a STRAIGHT_JOIN book_authors ba STRAIGHT_JOIN books b
WHERE b.id = ba.book_id AND ba.author_id = a.id;
```

关于该算法的实现细节和限制你可以通过查看 Join Reorder 算法简介章节进行了解。

8.2.5 扩展阅读

- 用 EXPLAIN 查看 JOIN 查询的执行计划
- Join Reorder 算法简介

8.3 子查询

本章将介绍 TiDB 中的子查询功能。

8.3.1 概述

子查询是嵌套在另一个查询中的 SQL 表达式，借助子查询，可以在一个查询当中使用另外一个查询的查询结果。

下面将以 Bookshop 应用为例对子查询展开介绍：

8.3.2 子查询语句

通常情况下，子查询语句分为如下几种形式：

- 标量子查询（Scalar Subquery），如 SELECT (SELECT s1 FROM t2) FROM t1。
- 派生表（Derived Tables），如 SELECT t1.s1 FROM (SELECT s1 FROM t2) t1。

- 存在性测试（Existential Test），如 WHERE NOT EXISTS(SELECT ... FROM t2)，WHERE t1.a IN (SELECT ... FROM t2)。
- 集合比较（Quantified Comparison），如 WHERE t1.a = ANY(SELECT ... FROM t2)。
- 作为比较运算符操作数的子查询，如 WHERE t1.a > (SELECT ... FROM t2)。

8.3.3 子查询的分类

一般来说，可以将子查询分为关联子查询（Correlated Subquery）和无关联子查询 (Self-contained Subquery) 两大类，TiDB 对于这两类子查询的处理方式是不一样的。

判断是否为关联子查询的依据在于子查询当中是否引用了外层查询的列。

8.3.3.1 无关联子查询

对于将子查询作为比较运算符 (> / >=/ < / <= / = / !=) 操作数的这类无关联子查询而言，内层子查询只需要进行一次查询，TiDB 在生成执行计划阶段会将内层子查询改写为常量。

例如，想要查找 authors 表当中年龄大于总体平均年龄的作家，可以通过将子查询作为比较操作符的操作数来实现：

```
SELECT * FROM authors a1 WHERE (IFNULL(a1.death_year, YEAR(NOW())) - a1.birth_yea
r) > (
  SELECT
    AVG(IFNULL(a2.death_year, YEAR(NOW())) - a2.birth_year) AS average_age
  FROM
    authors a2
)
```

在 TiDB 执行上述查询的时候会先执行一次内层子查询：

```
SELECT AVG(IFNULL(a2.death_year, YEAR(NOW())) - a2.birth_year) AS average_age FRO
M authors a2;
```

假设查询得到的结果为 34，即总体平均年龄为 34，34 将作为常量替换掉原来的子查询。

```
SELECT * FROM authors a1
WHERE (IFNULL(a1.death_year, YEAR(NOW())) - a1.birth_year) > 34;
```

运行结果为：

```
+--------+------------------+--------+------------+------------+
| id     | name             | gender | birth_year | death_year |
+--------+------------------+--------+------------+------------+
| 13514  | Kennith Kautzer  | 1      | 1956       | 2018       |
| 13748  | Dillon Langosh   | 1      | 1985       | NULL       |
| 99184  | Giovanny Emmerich | 1      | 1954       | 2012       |
| 180191 | Myrtie Robel     | 1      | 1958       | 2009       |
| 200969 | Iva Renner       | 0      | 1977       | NULL       |
| 209671 | Abraham Ortiz    | 0      | 1943       | 2016       |
| 229908 | Wellington Wiza  | 1      | 1932       | 1969       |
| 306642 | Markus Crona     | 0      | 1969       | NULL       |
| 317018 | Ellis McCullough | 0      | 1969       | 2014       |
| 322369 | Mozelle Hand     | 0      | 1942       | 1977       |
| 325946 | Elta Flatley     | 0      | 1933       | 1986       |
| 361692 | Otho Langosh     | 1      | 1931       | 1997       |
| 421294 | Karelle VonRueden | 0      | 1977       | NULL       |
...
```

对于存在性测试和集合比较两种情况下的无关联列子查询，TiDB 会将其进行改写和等价替换以获得更好的执行性能，你可以通过阅读子查询相关的优化章节来了解更多的实现细节。

8.3.4 关联子查询

对于关联子查询而言，由于内层的子查询引用外层查询的列，子查询需要对外层查询得到的每一行都执行一遍，也就是说假设外层查询得到一千万的结果，那么子查询也会被执行一千万次，这会导致查询需要消耗更多的时间和资源。

因此在处理过程中，TiDB 会尝试对关联子查询去关联，以从执行计划层面上提高查询效率。

例如，假设想要查找那些大于其它相同性别作家的平均年龄的的作家，SQL 语句可以这样写：

```
SELECT * FROM authors a1 WHERE (IFNULL(a1.death_year, YEAR(NOW())) - a1.birth_yea
r) > (
  SELECT
    AVG(
      IFNULL(a2.death_year, YEAR(NOW())) - IFNULL(a2.birth_year, YEAR(NOW()))
    ) AS average_age
  FROM
    authors a2
  WHERE a1.gender = a2.gender
);
```

TiDB 在处理该 SQL 语句是会将其改写为等价的 Join 查询：

```
SELECT *
FROM
  authors a1,
  (
    SELECT
      gender, AVG(
        IFNULL(a2.death_year, YEAR(NOW())) - IFNULL(a2.birth_year, YEAR(NOW()))
      ) AS average_age
    FROM
      authors a2
    GROUP BY gender
  ) a2
WHERE
  a1.gender = a2.gender
  AND (IFNULL(a1.death_year, YEAR(NOW())) - a1.birth_year) > a2.average_age;
```

作为最佳实践，在实际开发当中，建议在明确知道有更好的等价写法时，尽量避免通过关联子查询来进行查询。

8.3.5 扩展阅读

- 子查询相关的优化
- 关联子查询去关联
- TiDB 中的子查询优化技术

8.4 分页查询

当查询结果数据量较大时，往往希望以“分页”的方式返回所需要的部分。

8.4.1 对查询结果进行分页

在 TiDB 当中，可以利用 LIMIT 语句来实现分页功能，常规的分页语句写法如下所示：

```
SELECT * FROM table_a t ORDER BY gmt_modified DESC LIMIT offset, row_count;
```

offset 表示起始记录数，row_count 表示每页记录数。除此之外，TiDB 也支持 LIMIT row_count OFFSET offset 语法。

除非明确要求不要使用任何排序来随机展示数据，使用分页查询语句时都应该通过 ORDER BY 语句指定查询结果的排序方式。

例如，在 Bookshop 应用当中，希望将最新书籍列表以分页的形式返回给用户。通过 LIMIT 0, 10 语句，便可以得到列表第 1 页的书籍信息，每页中最多有 10 条记录。获取第 2 页信息，则改成可以改成 LIMIT 10, 10，如此类推。

```
SELECT *
FROM books
ORDER BY published_at DESC
LIMIT 0, 10;
```

在使用 Java 开发应用程序时，后端程序从前端接收到的参数页码 page_number 和每页的数据条数 page_size，而不是起始记录数 offset，因此在进行数据库查询前需要对其进行一些转换。

```
public List<Book> getLatestBooksPage(Long pageNumber, Long pageSize) throws SQL
Exception {
    pageNumber = pageNumber < 1L ? 1L : pageNumber;
    pageSize = pageSize < 10L ? 10L : pageSize;
    Long offset = (pageNumber - 1) * pageSize;
    Long limit = pageSize;
    List<Book> books = new ArrayList<>();
```

```
    try (Connection conn = ds.getConnection()) {
        PreparedStatement stmt = conn.prepareStatement("""
        SELECT id, title, published_at
        FROM books
        ORDER BY published_at DESC
        LIMIT ?, ?;
        """);
        stmt.setLong(1, offset);
        stmt.setLong(2, limit);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Book book = new Book();
            book.setId(rs.getLong("id"));
            book.setTitle(rs.getString("title"));
            book.setPublishedAt(rs.getDate("published_at"));
            books.add(book);
        }
    }
    return books;
}
```

8.4.2 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 LIMIT 语法中的 offset，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。

下面将介绍一种更为高效的分页批处理方案：

使用 SQL 实现分页批处理，可以按照如下步骤进行：

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。

```
SELECT
   floor((t.row_num - 1) / 1000) + 1 AS page_num,
   min(t.id) AS start_key,
   max(t.id) AS end_key,
   count(*) AS page_size
FROM (
   SELECT id, row_number() OVER (ORDER BY id) AS row_num
   FROM books
) t
GROUP BY page_num
ORDER BY page_num;
```

查询结果如下：

```
+----------+-----------+-----------+-----------+
| page_num | start_key  | end_key    | page_size |
+----------+-----------+-----------+-----------+
|       1 |    268996 |  213168525 |     1000 |
|       2 |  213210359 |  430012226 |     1000 |
|       3 |  430137681 |  647846033 |     1000 |
|       4 |  647998334 |  848878952 |     1000 |
|       5 |  848899254 | 1040978080 |     1000 |
...
|      20 | 4077418867 | 4294004213 |     1000 |
+----------+-----------+-----------+-----------+
20 rows in set (0.01 sec)
```

接下来，只需要使用 WHERE id BETWEEN start_key AND end_key 语句查询每个分片的数据即可。修改数据时，也可以借助上面计算好的分片信息，实现高效的数据更新。

例如，假如想要删除第 1 页上的所有书籍的基本信息，可以将上表第 1 页所对应的 start_key 和 end_key 填入 SQL 语句当中。

```
DELETE FROM books
WHERE
   id BETWEEN 268996 AND 213168525
ORDER BY id;
```

在 Java 语言当中，可以定义一个 PageMeta 类来存储分页元信息。

```
public class PageMeta<K> {
    private Long pageNum;
    private K startKey;
    private K endKey;
    private Long pageSize;

    // Skip the getters and setters.

}
```

定义一个 getPageMetaList() 方法获取到分页元信息列表，然后定义一个可以根据页面元信息批量删除数据的方法 deleteBooksByPageMeta()。

```
public class BookDAO {
    public List<PageMeta<Long>> getPageMetaList() throws SQLException {
        List<PageMeta<Long>> pageMetaList = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("""
            SELECT
                floor((t.row_num - 1) / 1000) + 1 AS page_num,
                min(t.id) AS start_key,
                max(t.id) AS end_key,
                count(*) AS page_size
            FROM (
                SELECT id, row_number() OVER (ORDER BY id) AS row_num
                FROM books
            ) t
            GROUP BY page_num
            ORDER BY page_num;
            """);
            while (rs.next()) {
                PageMeta<Long> pageMeta = new PageMeta<>();
                pageMeta.setPageNum(rs.getLong("page_num"));
                pageMeta.setStartKey(rs.getLong("start_key"));
                pageMeta.setEndKey(rs.getLong("end_key"));
                pageMeta.setPageSize(rs.getLong("page_size"));
                pageMetaList.add(pageMeta);
            }
        }
        return pageMetaList;
    }
```

```
    public void deleteBooksByPageMeta(PageMeta<Long> pageMeta) throws SQLExcept
ion {
        try (Connection conn = ds.getConnection()) {
            PreparedStatement stmt = conn.prepareStatement("DELETE FROM books WHERE
 id >= ? AND id <= ?");
            stmt.setLong(1, pageMeta.getStartKey());
            stmt.setLong(2, pageMeta.getEndKey());
            stmt.executeUpdate();
        }
    }
}
```

如果想要删除第 1 页的数据，可以这样写：

```
List<PageMeta<Long>> pageMetaList = bookDAO.getPageMetaList();
if (pageMetaList.size() > 0) {
    bookDAO.deleteBooksByPageMeta(pageMetaList.get(0));
}
```

如果希望通过分页分批地删除所有书籍数据，可以这样写：

```
List<PageMeta<Long>> pageMetaList = bookDAO.getPageMetaList();
pageMetaList.forEach((pageMeta) -> {
    try {
        bookDAO.deleteBooksByPageMeta(pageMeta);
    } catch (SQLException e) {
        e.printStackTrace();
    }
});
```

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。

8.4.3 复合主键表的分页批处理

8.4.3.1 非聚簇索引表

对于非聚簇索引表（又被称为“非索引组织表”）而言，可以使用隐藏字段 _tidb_rowid 作为分页的 key，分页的方法与单列主键表中所介绍的方法相同。

建议：

你可以通过 SHOW CREATE TABLE users; 语句查看表主键是否使用了聚簇索引。

例如：

```
SELECT
    floor((t.row_num - 1) / 1000) + 1 AS page_num,
    min(t._tidb_rowid) AS start_key,
    max(t._tidb_rowid) AS end_key,
    count(*) AS page_size
FROM (
    SELECT _tidb_rowid, row_number() OVER (ORDER BY _tidb_rowid) AS row_num
    FROM users
) t
GROUP BY page_num
ORDER BY page_num;
```

查询结果如下：

```
+----------+-----------+---------+-----------+
| page_num | start_key | end_key | page_size |
+----------+-----------+---------+-----------+
|        1 |         1 |    1000 |      1000 |
|        2 |      1001 |    2000 |      1000 |
|        3 |      2001 |    3000 |      1000 |
|        4 |      3001 |    4000 |      1000 |
|        5 |      4001 |    5000 |      1000 |
|        6 |      5001 |    6000 |      1000 |
|        7 |      6001 |    7000 |      1000 |
|        8 |      7001 |    8000 |      1000 |
|        9 |      8001 |    9000 |      1000 |
|       10 |      9001 |    9990 |       990 |
+----------+-----------+---------+-----------+
10 rows in set (0.00 sec)
```

8.4.3.2 聚簇索引表

对于聚簇索引表（又被称为“索引组织表”），可以利用 concat 函数将多个列的值连接起来作为一个 key，然后使用窗口函数获取分页信息。

需要注意的是，这时候 key 是一个字符串，你必须确保这个字符串长度总是相等的，才能够通过 min 和 max 聚合函数得到分页内正确的 start_key 和 end_key。如果进行字符串连接的字段长度不固定，你可以通过 LPAD 函数进行补全。

例如，想要对 ratings 表里的数据进行分页批处理。

先可以通过下面的 SQL 语句来在制造元信息表。因为组成 key 的 book_id 列和 user_id 列都是 bigint 类型，转换为字符串是并不是等宽的，因此需要根据 bigint 类型的最大位数 19，使用 LPAD 函数在长度不够时用 0 补齐。

```
SELECT
   floor((t1.row_num - 1) / 10000) + 1 AS page_num,
   min(mvalue) AS start_key,
   max(mvalue) AS end_key,
   count(*) AS page_size
FROM (
   SELECT
      concat('(', LPAD(book_id, 19, 0), ',', LPAD(user_id, 19, 0), ')') AS mvalue,
      row_number() OVER (ORDER BY book_id, user_id) AS row_num
   FROM ratings
) t1
GROUP BY page_num
ORDER BY page_num;
```

注意：

该 SQL 会以全表扫描 (TableFullScan) 方式执行，当数据量较大时，查询速度会变慢，此时可以使用 TiFlash 进行加速。

查询结果如下：

```
+----------+-----------------------------------------+-----------------------------------------
---+-----------+
| page_num | start_key                               | end_key                                 | page_size |
+----------+-----------------------------------------+-----------------------------------------
---+-----------+
|        1 | (0000000000000268996,0000000000092104804) | (0000000000140982742,0000
000000374645100) |     10000 |
|        2 | (0000000000140982742,0000000000456757551) | (0000000000287195082,0000
000000404053200550) |     10000 |
|        3 | (0000000000287196791,0000000000191962769) | (0000000000434010216,0000
```

```
000000237646714) |     10000 |
|       4 | (0000000000434010216,0000000000375066168) | (0000000000578893327,0000
000002167504460) |     10000 |
|       5 | (0000000000578893327,0000000002457322286) | (0000000000718287668,0000
000001502744628) |     10000 |
...
|      29 | (0000000004002523918,0000000000902930986) | (0000000004147203315,0000
000004090920746) |     10000 |
|      30 | (0000000004147421329,0000000000319181561) | (0000000004294004213,0000
000003586311166) |      9972 |
+----------+-------------------------------------------+-------------------------------------
---+-----------+
30 rows in set (0.28 sec)
```

假如想要删除第 1 页上的所有评分记录，可以将上表第 1 页所对应的 start_key 和 end_key 填入 SQL 语句当中。

```sql
SELECT * FROM ratings
WHERE
  (book_id > 268996 AND book_id < 140982742)
  OR (
    book_id = 268996 AND user_id >= 92104804
  )
  OR (
    book_id = 140982742 AND user_id <= 374645100
  )
ORDER BY book_id, user_id;
```

8.5 视图

本章将介绍 TiDB 中的视图功能。

8.5.1 概述

TiDB 支持视图，视图是一张虚拟表，该虚拟表的结构由创建视图时的 SELECT 语句定义。

- 通过视图可以对用户只暴露安全的字段及数据，进而保证底层表的敏感字段及数据的安全。
- 将频繁出现的复杂查询定义为视图，可以使复杂查询更加简单便捷。

8.5.2 创建视图

在 TiDB 当中，可以通过 CREATE VIEW 语句来将某个较为复杂的查询定义为视图，其语法如下：

```
CREATE VIEW view_name AS query;
```

请注意，创建的视图名称不能与已有的视图或表重名。

例如，在多表连接查询 章节当中，通过 JOIN 语句连接 books 表和 ratings 表查询到了带有平均评分的书籍列表。为了方便后续查询，可以将该查询语句定义为一个视图，SQL 语句如下所示：

```
CREATE VIEW book_with_ratings AS
SELECT b.id AS book_id, ANY_VALUE(b.title) AS book_title, AVG(r.score) AS average_scor
e
FROM books b
LEFT JOIN ratings r ON b.id = r.book_id
GROUP BY b.id;
```

8.5.3 查询视图

视图创建完成后，便可以使用 SELECT 语句像查询一般数据表一样查询视图。

```
SELECT * FROM book_with_ratings LIMIT 10;
```

TiDB 在执行查询视图语句时，会将视图展开成创建视图时定义的 SELECT 语句，进而执行展开后的查询语句。

8.5.4 更新视图

目前 TiDB 中的视图不支持 ALTER VIEW view_name AS query; 语法，你可以通过以下两种方式实现视图的 “更新”：

- 先 DROP VIEW view_name; 语句删除旧视图，再通过 CREATE VIEW view_name AS query; 语句创建新视图的方式来更新视图。
- 使用 CREATE OR REPLACE VIEW view_name AS query; 语句覆盖已存在的同名视图。

```
CREATE OR REPLACE VIEW book_with_ratings AS
SELECT b.id AS book_id, ANY_VALUE(b.title), ANY_VALUE(b.published_at) AS book_title,
AVG(r.score) AS average_score
FROM books b
LEFT JOIN ratings r ON b.id = r.book_id
GROUP BY b.id;
```

8.5.5 获取视图相关信息

8.5.5.1 使用 SHOW CREATE TABLE|VIEW view_name 语句

```
SHOW CREATE VIEW book_with_ratings\G
```

运行结果为：

```
*************************** 1. row ***************************
                View: book_with_ratings
         Create View: CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`%` SQL SECURIT
Y DEFINER VIEW `book_with_ratings` (`book_id`, `ANY_VALUE(b.title)`, `book_title`, `averag
e_score`) AS SELECT `b`.`id` AS `book_id`,ANY_VALUE(`b`.`title`) AS `ANY_VALUE(b.title)`,A
NY_VALUE(`b`.`published_at`) AS `book_title`,AVG(`r`.`score`) AS `average_score` FROM `b
ookshop`.`books` AS `b` LEFT JOIN `bookshop`.`ratings` AS `r` ON `b`.`id`=`r`.`book_id` GR
OUP BY `b`.`id`
character_set_client: utf8mb4
collation_connection: utf8mb4_general_ci
1 row in set (0.00 sec)
```

8.5.5.2 查询 INFORMATION_SCHEMA.VIEWS 表

```
SELECT * FROM information_schema.views WHERE TABLE_NAME = 'book_with_ratings'\
G
```

运行结果为：

```
*************************** 1. row ***************************
       TABLE_CATALOG: def
        TABLE_SCHEMA: bookshop
          TABLE_NAME: book_with_ratings
     VIEW_DEFINITION: SELECT `b`.`id` AS `book_id`,ANY_VALUE(`b`.`title`) AS `ANY_VALUE
(b.title)`,ANY_VALUE(`b`.`published_at`) AS `book_title`,AVG(`r`.`score`) AS `average_score`
FROM `bookshop`.`books` AS `b` LEFT JOIN `bookshop`.`ratings` AS `r` ON `b`.`id`=`r`.`boo
k_id` GROUP BY `b`.`id`
        CHECK_OPTION: CASCADED
        IS_UPDATABLE: NO
             DEFINER: root@%
```

```
      SECURITY_TYPE: DEFINER
CHARACTER_SET_CLIENT: utf8mb4
COLLATION_CONNECTION: utf8mb4_general_ci
1 row in set (0.00 sec)
```

8.5.6 删除视图

通过 DROP VIEW view_name; 语句可以删除已经创建的视图。

```
DROP VIEW book_with_ratings;
```

8.5.7 局限性

关于局限性，你可以通过阅读参考文档当中的视图章节进行了解。

8.5.8 扩展阅读

- 视图
- CREATE VIEW 语句
- DROP VIEW 语句
- 用 EXPLAIN 查看带视图的 SQL 执行计划
- TiFlink: 使用 TiKV 和 Flink 实现强一致的物化视图

8.6 临时表

临时表可以被认为是一种复用查询结果的技术。

假设希望知道 Bookshop 应用当中最年长的作家们的一些情况，可能需要编写多个查询，而这些查询都需要使用到这个最年长作家列表。可以通过下面的 SQL 语句从 authors 表当中找出最年长的前 50 位作家作为研究对象。

```
SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
FROM authors a
ORDER BY age DESC
LIMIT 50;
```

查询结果如下：

```
+------------+--------------------+------+
| id         | name               | age  |
```

```
+------------+--------------------+------+
| 4053452056 | Dessie Thompson    |   80 |
| 2773958689 | Pedro Hansen       |   80 |
| 4005636688 | Wyatt Keeling      |   80 |
| 3621155838 | Colby Parker       |   80 |
| 2738876051 | Friedrich Hagenes  |   80 |
| 2299112019 | Ray Macejkovic     |   80 |
| 3953661843 | Brandi Williamson  |   80 |
...
| 4100546410 | Maida Walsh        |   80 |
+------------+--------------------+------+
50 rows in set (0.01 sec)
```

在找到这 50 位最年长的作家后，希望缓存这个查询结果，以便后续的查询能够方便地使用到这组数据。如果使用一般的数据库表进行存储的话，在创建这些表时，需要考虑如何避免不同会话之间的表重名问题，而且可能在一批查询结束之后就不再需要这些表了，还需要及时地对这些中间结果表进行清理。

8.6.1 创建临时表

为了满足这类缓存中间结果的需求，TiDB 在 v5.3.0 版本中引入了临时表功能，对于临时表当中的本地临时表而言，TiDB 将会在会话结束的一段时间后自动清理这些已经没用的临时表，用户无需担心中间结果表的增多会带来管理上的麻烦。

8.6.1.1 临时表类型

TiDB 的临时表分为本地临时表和全局临时表：

- 本地临时表的表定义和表内数据只对当前会话可见，适用于暂存会话内的中间数据。
- 全局临时表的表定义对整个 TiDB 集群可见，表内数据只对当前事务可见，适用于暂存事务内的中间数据。

8.6.1.2 创建本地临时表

在创建本地临时表前，你需要给当前数据库用户添加上 CREATE TEMPORARY TABLES 权限。

在 SQL 中，通过 CREATE TEMPORARY TABLE <table_name> 语句创建临时表，默认临时表的类型为本地临时表，它只能被当前会话所访问。

```
CREATE TEMPORARY TABLE top_50_eldest_authors (
    id BIGINT,
    name VARCHAR(255),
    age INT,
    PRIMARY KEY(id)
);
```

在创建完临时表后，你可以通过 INSERT INTO table_name SELECT ... 语句，将上述查询得到的结果导入到刚刚创建的临时表当中。

```
INSERT INTO top_50_eldest_authors
SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
FROM authors a
ORDER BY age DESC
LIMIT 50;
```

运行结果为：

```
Query OK, 50 rows affected (0.03 sec)
Records: 50  Duplicates: 0  Warnings: 0
```

在 Java 中创建本地临时表的示例如下：

```
public List<Author> getTop50EldestAuthorInfo() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("""
            CREATE TEMPORARY TABLE top_50_eldest_authors (
                id BIGINT,
                name VARCHAR(255),
                age INT,
                PRIMARY KEY(id)
            );
        """);

        stmt.executeUpdate("""
            INSERT INTO top_50_eldest_authors
```

```
            SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
            FROM authors a
            ORDER BY age DESC
            LIMIT 50;
        """);

        ResultSet rs = stmt.executeQuery("""
            SELECT id, name FROM top_50_eldest_authors;
        """);

        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            authors.add(author);
        }
    }
    return authors;
}
```

8.6.1.3 创建全局临时表

在 SQL 中，你可以通过加上 GLOBAL 关键字来声明你所创建的是全局临时表。创建全局临时表时必须在末尾 ON COMMIT DELETE ROWS 修饰，这表明该全局数据表的所有数据行将在事务结束后被删除。

```
CREATE GLOBAL TEMPORARY TABLE IF NOT EXISTS top_50_eldest_authors_global (
    id BIGINT,
    name VARCHAR(255),
    age INT,
    PRIMARY KEY(id)
) ON COMMIT DELETE ROWS;
```

在对全局临时表导入数据时，你需要特别注意，你必须通过 BEGIN 显式声明事务的开始。否则导入的数据在 INSERT INTO 语句执行后就清除掉，因为 Auto Commit 模式下，INSERT INTO 语句的执行结束，事务会自动被提交，事务结束，全局临时表的数据便被清空了。

在 Java 中使用全局临时表时，你需要将 Auto Commit 模式先关闭。在 Java 语言当中，你可以通过 conn.setAutoCommit(false); 语句来实现，当你使用完成后，可以通过 conn.commit(); 显式地提交事务。事务在提交或取消后，在事务过程中对全局临时表添加的数据将会被清除。

```
public List<Author> getTop50EldestAuthorInfo() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        conn.setAutoCommit(false);

        Statement stmt = conn.createStatement();
        stmt.executeUpdate("""
            CREATE GLOBAL TEMPORARY TABLE IF NOT EXISTS top_50_eldest_authors (
                id BIGINT,
                name VARCHAR(255),
                age INT,
                PRIMARY KEY(id)
            ) ON COMMIT DELETE ROWS;
        """);

        stmt.executeUpdate("""
            INSERT INTO top_50_eldest_authors
            SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
            FROM authors a
            ORDER BY age DESC
            LIMIT 50;
        """);

        ResultSet rs = stmt.executeQuery("""
            SELECT id, name FROM top_50_eldest_authors;
        """);

        conn.commit();
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            authors.add(author);
        }
    }
    return authors;
}
```

8.6.2 查看临时表信息

通过 SHOW [FULL] TABLES 语句可以查看到已经创建的全局临时表，但是无法看到本地临时表的信息，TiDB 暂时也没有类似的 information_schema.INNODB_TEMP_TABLE_INFO 系统表存放临时表的信息。

例如，你可以在 table 列表当中查看到全局临时表 top_50_eldest_authors_global，但是无法查看到 top_50_eldest_authors 表。

```
+------------------------------+------------+
| Tables_in_bookshop           | Table_type |
+------------------------------+------------+
| authors                      | BASE TABLE |
| book_authors                 | BASE TABLE |
| books                        | BASE TABLE |
| orders                       | BASE TABLE |
| ratings                      | BASE TABLE |
| top_50_eldest_authors_global | BASE TABLE |
| users                        | BASE TABLE |
+------------------------------+------------+
9 rows in set (0.00 sec)
```

8.6.3 查询临时表

在临时表准备就绪之后，你便可以像对一般数据表一样对临时表进行查询：

```
SELECT * FROM top_50_eldest_authors;
```

你可以通过表连接将临时表中的数据引用到你的查询当中：

```
EXPLAIN SELECT ANY_VALUE(ta.id) AS author_id, ANY_VALUE(ta.age), ANY_VALUE(ta.na
me), COUNT(*) AS books
FROM top_50_eldest_authors ta
LEFT JOIN book_authors ba ON ta.id = ba.author_id
GROUP BY ta.id;
```

与视图有所不同，在对临时表进行查询时，不会再执行导入数据时所使用的原始查询，而是直接从临时表中获取数据。在一些情况下，这会帮助你提高查询的效率。

8.6.4 删除临时表

本地临时表会在**会话**结束后连同数据和表结构都进行自动清理。全局临时表在**事务**结束后会自动清除数据，但是表结构依然保留，需要手动删除。

你可以通过 DROP TABLE 或 DROP TEMPORARY TABLE 语句手动删除**本地临时表**。例如：

```
DROP TEMPORARY TABLE top_50_eldest_authors;
```

你还可以通过 DROP TABLE 或 DROP GLOBAL TEMPORARY TABLE 语句手动删除**全局临时表**。例如：

```
DROP GLOBAL TEMPORARY TABLE top_50_eldest_authors_global;
```

8.6.5 限制

关于 TiDB 在临时表功能上的一些限制，你可以通过阅读参考文档中的临时表与其他平凯数据库功能的兼容性限制小节进行了解。

8.6.6 扩展阅读

- 临时表

8.7 公共表表达式 (CTE)

由于业务的客观复杂性，有时候会写出长达 2000 行的单条 SQL 语句，其中包含大量的聚合和多层子查询嵌套，维护此类 SQL 堪称开发人员的噩梦。

在前面的小节当中已经介绍了如何使用视图简化查询，也介绍了如何使用临时表来缓存中间查询结果。

在这一小节当中，将介绍 TiDB 当中的公共表表达式（CTE）语法，它是一种更加便捷的复用查询结果的方法。

TiDB 从 5.1 版本开始支持 ANSI SQL 99 标准的 CTE 及其递归的写法，极大提升开发人员和 DBA 编写复杂业务逻辑 SQL 的效率，增强代码的可维护性。

8.7.1 基本使用

公共表表达式 (CTE) 是一个临时的中间结果集，能够在 SQL 语句中引用多次，提高 SQL 语句的可读性与执行效率。在 TiDB 中可以通过 WITH 语句使用公共表表达式。

公共表表达式可以分为非递归和递归两种类型。

8.7.1.1 非递归的 CTE

非递归的 CTE 使用如下语法进行定义：

```
WITH <query_name> AS (
    <query_definition>
)
SELECT ... FROM <query_name>;
```

例如，假设还想知道最年长的 50 位作家分别编写过多少书籍。

在 SQL 中，可以将临时表小节当中的例子改为以下 SQL 语句：

```
WITH top_50_eldest_authors_cte AS (
    SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
    FROM authors a
    ORDER BY age DESC
    LIMIT 50
)
SELECT
    ANY_VALUE(ta.id) AS author_id,
    ANY_VALUE(ta.age) AS author_age,
    ANY_VALUE(ta.name) AS author_name,
    COUNT(*) AS books
FROM top_50_eldest_authors_cte ta
LEFT JOIN book_authors ba ON ta.id = ba.author_id
GROUP BY ta.id;
```

查询结果如下：

```
+-----------+-----------+--------------------+-------+
| author_id | author_age | author_name        | books |
```

```
+------------+-----------+--------------------+-------+
| 1238393239 |        80 | Araceli Purdy      |     1 |
|  817764631 |        80 | Ivory Davis        |     3 |
| 3093759193 |        80 | Lysanne Harris     |     1 |
| 2299112019 |        80 | Ray Macejkovic     |     4 |
...
+------------+-----------+--------------------+-------+
50 rows in set (0.01 sec)
```

在 Java 中的示例如下：

```
public List<Author> getTop50EldestAuthorInfoByCTE() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
            WITH top_50_eldest_authors_cte AS (
                SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
                FROM authors a
                ORDER BY age DESC
                LIMIT 50
            )
            SELECT
                ANY_VALUE(ta.id) AS author_id,
                ANY_VALUE(ta.name) AS author_name,
                ANY_VALUE(ta.age) AS author_age,
                COUNT(*) AS books
            FROM top_50_eldest_authors_cte ta
            LEFT JOIN book_authors ba ON ta.id = ba.author_id
            GROUP BY ta.id;
        """);
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("author_id"));
            author.setName(rs.getString("author_name"));
            author.setAge(rs.getShort("author_age"));
            author.setBooks(rs.getInt("books"));
            authors.add(author);
        }
    }
    return authors;
}
```

这时，可以发现名为 “Ray Macejkovic” 的作者写了 4 本书，继续通过 CTE 查询来了解这 4 本书的销量和评分：

```
WITH books_authored_by_rm AS (
    SELECT *
    FROM books b
    LEFT JOIN book_authors ba ON b.id = ba.book_id
    WHERE author_id = 2299112019
), books_with_average_ratings AS (
    SELECT
        b.id AS book_id,
        AVG(r.score) AS average_rating
    FROM books_authored_by_rm b
    LEFT JOIN ratings r ON b.id = r.book_id
    GROUP BY b.id
), books_with_orders AS (
    SELECT
        b.id AS book_id,
        COUNT(*) AS orders
    FROM books_authored_by_rm b
    LEFT JOIN orders o ON b.id = o.book_id
    GROUP BY b.id
)
SELECT
    b.id AS `book_id`,
    b.title AS `book_title`,
    br.average_rating AS `average_rating`,
    bo.orders AS `orders`
FROM
    books_authored_by_rm b
    LEFT JOIN books_with_average_ratings br ON b.id = br.book_id
    LEFT JOIN books_with_orders bo ON b.id = bo.book_id
;
```

查询结果如下：

```
+------------+------------------------+----------------+--------+
| book_id    | book_title             | average_rating | orders |
+------------+------------------------+----------------+--------+
|  481008467 | The Documentary of goat |         2.0000 |     16 |
| 2224531102 | Brandt Skiles          |         2.7143 |     17 |
| 2641301356 | Sheridan Bashirian     |         2.4211 |     12 |
| 4154439164 | Karson Streich         |         2.5833 |     19 |
```

```
+------------+-----------------------+---------------+--------+
4 rows in set (0.06 sec)
```

在这个 SQL 语句，定义了三个 CTE 块，CTE 块之间使用 , 进行分隔。

先在 CTE 块 books_authored_by_rm 当中将该作者（作者 ID 为 2299112019）所编写的书查出来，然后在 books_with_average_ratings 和 books_with_orders 中分别查出这些书的平均评分和订单数，最后通过 JOIN 语句进行汇总。

值得注意的是，books_authored_by_rm 中的查询只会执行一次，TiDB 会开辟一块临时空间对查询的结果进行缓存，当 books_with_average_ratings 和 books_with_orders 引用时会直接从该临时空间当中获取数据。

建议：

当默认的 CTE 查询执行效率不高时，你可以使用 MERGE() hint，将 CTE 子查询拓展到外部查询，以此提高执行效率。

8.7.1.2 递归的 CTE

递归的公共表表达式可以使用如下语法进行定义：

```
WITH RECURSIVE <query_name> AS (
    <query_definition>
)
SELECT ... FROM <query_name>;
```

比较经典的例子是通过递归的 CTE 生成一组斐波那契数：

```
WITH RECURSIVE fibonacci (n, fib_n, next_fib_n) AS
(
  SELECT 1, 0, 1
  UNION ALL
  SELECT n + 1, next_fib_n, fib_n + next_fib_n FROM fibonacci WHERE n < 10
)
SELECT * FROM fibonacci;
```

查询结果如下：

```
+------+-------+------------+
| n    | fib_n | next_fib_n |
```

```
+------+-------+------------+
|    1 |     0 |          1 |
|    2 |     1 |          1 |
|    3 |     1 |          2 |
|    4 |     2 |          3 |
|    5 |     3 |          5 |
|    6 |     5 |          8 |
|    7 |     8 |         13 |
|    8 |    13 |         21 |
|    9 |    21 |         34 |
|   10 |    34 |         55 |
+------+-------+------------+
10 rows in set (0.00 sec)
```

8.7.2 扩展阅读

- WITH

8.8 读取副本数据

8.8.1 Follower Read

本章将介绍使用 Follower Read 在特定情况下加速查询的方法。

8.8.1.1 简介

在 TiDB 当中，数据是以 Region 为单位，分散在集群中所有的节点上进行存储的。一个 Region 可以存在多个副本，副本又分为一个 leader 和多个 follower。当 leader 上的数据发生变化时，TiDB 会将数据同步更新到 follower。

默认情况下，TiDB 只会在同一个 Region 的 leader 上读写数据。当系统中存在读取热点 Region 导致 leader 资源紧张成为整个系统读取瓶颈时，启用 Follower Read 功能可明显降低 leader 的负担，并且通过在多个 follower 之间均衡负载，显著地提升整体系统的吞吐能力。

8.8.1.2 何时使用

8.8.1.2.1 优化读热点

你可以在 TiDB Dashboard 流量可视化页面当中通过可视化的方法分析你的应用程序是否存在热点 Region。你可以通过将「指标选择框」选择到 Read (bytes) 或 Read (keys) 查看是否存在读取热点 Region。

如果发现确实存在热点问题，你可以通过阅读 TiDB 热点问题处理章节进行逐一排查，以便从应用程序层面上避免热点的产生。

如果读取热点的确无法避免或者改动的成本很大，你可以尝试通过 Follower Read 功能将读取请求更好的负载均衡到 follower region。

8.8.1.2.2 优化跨数据中心部署的延迟

如果 TiDB 集群是跨地区或跨数据中心部署的，一个 Region 的不同副本分布在不同的地区或数据中心，此时可以通过配置 Follower Read 为 closest-adaptive 或 closest-replicas 让 TiDB 优先从当前的数据中心执行读操作，这样可以大幅降低读操作的延迟和流量开销。具体原理可参考 Follower Read。

8.8.1.3 开启 Follower Read

在 SQL 中，你可以将变量 tidb_replica_read 的值（默认为 leader）设置为 follower、leader-and-follower、prefer-leader、closest-replicas 或 closest-adaptive 开启 TiDB 的 Follower Read 功能：

```
SET [GLOBAL] tidb_replica_read = 'follower';
```

你可以通过 Follower Read 使用方式 了解该变量的更多细节。

在 Java 语言当中，可以定义一个 FollowerReadHelper 类用于开启 Follower Read 功能：

```
public enum FollowReadMode {
    LEADER("leader"),
    FOLLOWER("follower"),
    LEADER_AND_FOLLOWER("leader-and-follower"),
    CLOSEST_REPLICA("closest-replica"),
    CLOSEST_ADAPTIVE("closest-adaptive"),
    PREFER_LEADER("prefer-leader");

    private final String mode;

    FollowReadMode(String mode) {
        this.mode = mode;
    }

    public String getMode() {
        return mode;
    }
}

public class FollowerReadHelper {

    public static void setSessionReplicaRead(Connection conn, FollowReadMode mode) t
hrows SQLException {
        if (mode == null) mode = FollowReadMode.LEADER;
        PreparedStatement stmt = conn.prepareStatement(
            "SET @@tidb_replica_read = ?;"
        );
        stmt.setString(1, mode.getMode());
        stmt.execute();
    }

    public static void setGlobalReplicaRead(Connection conn, FollowReadMode mode) th
rows SQLException {
        if (mode == null) mode = FollowReadMode.LEADER;
        PreparedStatement stmt = conn.prepareStatement(
            "SET GLOBAL @@tidb_replica_read = ?;"
        );
        stmt.setString(1, mode.getMode());
        stmt.execute();
    }

}
```

在需要使用从 Follower 节点读取数据时，通过 setSessionReplicaRead(conn, FollowReadMode.LEADER_AND_FOLLOWER) 方法在当前 Session 开启能够在 Leader 节点和 Follower 节点进行负载均衡的 Follower Read 功能，当连接断开时，会恢复到原来的模式。

```
public static class AuthorDAO {

    // Omit initialization of instance variables...

    public void getAuthorsByFollowerRead() throws SQLException {
        try (Connection conn = ds.getConnection()) {
            // Enable the follower read feature.
            FollowerReadHelper.setSessionReplicaRead(conn, FollowReadMode.LEADER_AND
_FOLLOWER);

            // Read the authors list for 100000 times.
            Random random = new Random();
            for (int i = 0; i < 100000; i++) {
                Integer birthYear = 1920 + random.nextInt(100);
                List<Author> authors = this.getAuthorsByBirthYear(birthYear);
                System.out.println(authors.size());
            }
        }
    }

    public List<Author> getAuthorsByBirthYear(Integer birthYear) throws SQLException {
        List<Author> authors = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            PreparedStatement stmt = conn.prepareStatement("SELECT id, name FROM auth
ors WHERE birth_year = ?");
            stmt.setInt(1, birthYear);
            ResultSet rs = stmt.executeQuery();
            while (rs.next()) {
                Author author = new Author();
                author.setId( rs.getLong("id"));
                author.setName(rs.getString("name"));
                authors.add(author);
            }
        }
        return authors;
    }
}
```

8.8.1.4 扩展阅读

- Follower Read
- 平凯数据库热点问题处理
- TiDB Dashboard 流量可视化页面

8.8.2 Stale Read

Stale Read 是一种读取历史数据版本的机制，通过 Stale Read 功能，你能从指定时间点或时间范围内读取对应的历史数据，从而在数据强一致需求没那么高的场景降低读取数据的延迟。当使用 Stale Read 时，TiDB 默认会随机选择一个副本来读取数据，因此能利用所有保存有副本的节点的处理能力。

在实际的使用当中，请根据具体的场景判断是否适合在 TiDB 当中开启 Stale Read 功能。如果你的应用程序不能容忍读到非实时的数据，请勿使用 Stale Read，否则读到的数据可能不是最新成功写入的数据。

TiDB 提供了语句级别、事务级别、会话级别三种级别的 Stale Read 功能，接下来将逐一进行介绍：

8.8.2.1 引入

在 Bookshop 应用程序当中，你可以通过下面的 SQL 语句查询出最新出版的书籍以及它们的价格：

```
SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
```

运行结果为：

```
+------------+-----------------------------+----------------------+--------+
| id         | title                       | type                 | price  |
+------------+-----------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 100.00 |
| 1064253862 | Collin Rolfson              | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat      | Magazine             | 159.75 |
|  893930596 | Myrl Hills                  | Education & Reference | 356.85 |
```

```
| 3062833277 | Keven Wyman               | Life                | 477.91 |
+------------+---------------------------+---------------------+--------+
5 rows in set (0.02 sec)
```

看到此时（2022-04-20 15:20:00）的列表中，**The Story of Droolius Caesar** 这本小说的价格为 100.0 元。

于此同时，卖家发现这本书很受欢迎，于是他通过下面的 SQL 语句将这本书的价格高到了 150.0 元。

```
UPDATE books SET price = 150 WHERE id = 3181093216;
```

运行结果为：

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

当再次查询最新书籍列表时，发现这本书确实涨价了。

```
+------------+---------------------------+---------------------+--------+
| id         | title                     | type                | price  |
+------------+---------------------------+---------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel               | 150.00 |
| 1064253862 | Collin Rolfson            | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat    | Magazine            | 159.75 |
|  893930596 | Myrl Hills                | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman               | Life                | 477.91 |
+------------+---------------------------+---------------------+--------+
5 rows in set (0.01 sec)
```

如果不要求必须使用最新的数据，可以让 TiDB 通过 Stale Read 功能直接返回可能已经过期的历史数据，避免使用强一致性读时数据同步带来的延迟。

假设在 Bookshop 应用程序当中，在用户浏览书籍列表页时，不对书籍价格的实时性进行要求，只有用户在点击查看书籍详情页或下单时才去获取实时的价格信息，可以借助 Stale Read 能力来进一步提升应用的吞吐量。

8.8.2.2 语句级别

在 SQL 中，你可以在上述价格的查询语句当中添加上 AS OF TIMESTAMP <datetime> 语句查看到固定时间点之前这本书的价格。

```
SELECT id, title, type, price FROM books AS OF TIMESTAMP '2022-04-20 15:20:00' ORDER BY published_at DESC LIMIT 5;
```

运行结果为：

```
+------------+-----------------------------+----------------------+--------+
| id         | title                       | type                 | price  |
+------------+-----------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 100.00 |
| 1064253862 | Collin Rolfson              | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat      | Magazine             | 159.75 |
|  893930596 | Myrl Hills                  | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman                 | Life                 | 477.91 |
+------------+-----------------------------+----------------------+--------+
5 rows in set (0.01 sec)
```

除了指定精确的时间点外，你还可以通过：

- AS OF TIMESTAMP NOW() - INTERVAL 10 SECOND 表示读取 10 秒前最新的数据。
- AS OF TIMESTAMP TIDB_BOUNDED_STALENESS('2016-10-08 16:45:26', '2016-10-08 16:45:29') 表示读取在 2016 年 10 月 8 日 16 点 45 分 26 秒到 29 秒的时间范围内尽可能新的数据。
- AS OF TIMESTAMP TIDB_BOUNDED_STALENESS(NOW() - INTERVAL 20 SECOND, NOW()) 表示读取 20 秒前到现在的时间范围内尽可能新的数据。

需要注意的是，设定的时间戳或时间戳的范围不能过早或晚于当前时间。此外 NOW() 默认精确到秒，当精度要求较高时，需要添加参数，例如 NOW(3) 精确到毫秒。详情请参考 MySQL 文档。

过期的数据在 TiDB 当中会由垃圾回收器进行回收，数据在被清除之前会被保留一小段时间，这段时间被称为 GC Life Time (默认 10 分钟)。每次进行 GC 时，将以当前时间减去该时间周期的值作为 **GC Safe Point**。如果尝试读取 GC Safe Point 之前数据，TiDB 会报如下错误：

ERROR 9006 (HY000): GC life time is shorter than transaction duration...

如果给出的时间戳是一个未来的时间节点，TiDB 会报如下错误：

ERROR 9006 (HY000): cannot set read timestamp to a future time.

在 Java 中的示例如下：

```
public class BookDAO {

  // Omit some code...

  public List<Book> getTop5LatestBooks() throws SQLException {
    List<Book> books = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
      Statement stmt = conn.createStatement();
      ResultSet rs = stmt.executeQuery("""
      SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
      """);
      while (rs.next()) {
        Book book = new Book();
        book.setId(rs.getLong("id"));
        book.setTitle(rs.getString("title"));
        book.setType(rs.getString("type"));
        book.setPrice(rs.getDouble("price"));
        books.add(book);
      }
    }
    return books;
  }

  public void updateBookPriceByID(Long id, Double price) throws SQLException {
    try (Connection conn = ds.getConnection()) {
      PreparedStatement stmt = conn.prepareStatement("""
      UPDATE books SET price = ? WHERE id = ?;
      """);
      stmt.setDouble(1, price);
      stmt.setLong(2, id);
      int affects = stmt.executeUpdate();
      if (affects == 0) {
        throw new SQLException("Failed to update the book with id: " + id);
      }
    }
  }
```

```
    public List<Book> getTop5LatestBooksWithStaleRead(Integer seconds) throws SQLEx
ception {
        List<Book> books = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            PreparedStatement stmt = conn.prepareStatement("""
            SELECT id, title, type, price FROM books AS OF TIMESTAMP NOW() - INTERVAL ?
SECOND ORDER BY published_at DESC LIMIT 5;
            """);
            stmt.setInt(1, seconds);
            ResultSet rs = stmt.executeQuery();
            while (rs.next()) {
                Book book = new Book();
                book.setId(rs.getLong("id"));
                book.setTitle(rs.getString("title"));
                book.setType(rs.getString("type"));
                book.setPrice(rs.getDouble("price"));
                books.add(book);
            }
        } catch (SQLException e) {
            if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 1105) {
                System.out.println("WARN: cannot set read timestamp to a future time.");
            } else if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 9006) {
                System.out.println("WARN: GC life time is shorter than transaction duration.");
            } else {
                throw e;
            }
        }
        return books;
    }
}

List<Book> top5LatestBooks = bookDAO.getTop5LatestBooks();

if (top5LatestBooks.size() > 0) {
    System.out.println("The latest book price (before update): " + top5LatestBooks.get(0).g
etPrice());

    Book book = top5LatestBooks.get(0);
    bookDAO.updateBookPriceByID(book.getId(), book.price + 10);

    top5LatestBooks = bookDAO.getTop5LatestBooks();
    System.out.println("The latest book price (after update): " + top5LatestBooks.get(0).get
Price());
```

```
    // Use the stale read.
    top5LatestBooks = bookDAO.getTop5LatestBooksWithStaleRead(5);
    System.out.println("The latest book price (maybe stale): " + top5LatestBooks.get(0).get
Price());

    // Try to stale read the data at the future time.
    bookDAO.getTop5LatestBooksWithStaleRead(-5);

    // Try to stale read the data before 20 minutes.
    bookDAO.getTop5LatestBooksWithStaleRead(20 * 60);
}
```

通过结果可以看到通过 Stale Read 读取到了更新之前的价格 100.00 元。

```
The latest book price (before update): 100.00
The latest book price (after update): 150.00
The latest book price (maybe stale): 100.00
WARN: cannot set read timestamp to a future time.
WARN: GC life time is shorter than transaction duration.
```

8.8.2.3 事务级别

通过 START TRANSACTION READ ONLY AS OF TIMESTAMP 语句，你可以开启一个基于历史时间的只读事务，该事务基于所提供的历史时间来读取历史数据。

在 SQL 中的示例如下：

```
START TRANSACTION READ ONLY AS OF TIMESTAMP NOW() - INTERVAL 5 SECOND;
```

尝试通过 SQL 查询最新书籍的价格，发现 **The Story of Droolius Caesar** 这本书的价格还是更新之前的价格 100.0 元。

```
SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
```

运行结果为：

```
+------------+-----------------------------+----------------------+--------+
| id         | title                       | type                 | price  |
```

```
+------------+----------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 100.00 |
| 1064253862 | Collin Rolfson               | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat       | Magazine             | 159.75 |
|  893930596 | Myrl Hills                   | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman                  | Life                 | 477.91 |
+------------+----------------------------+----------------------+--------+
5 rows in set (0.01 sec)
```

随后通过 COMMIT; 语句提交事务，当事务结束后，又可以重新读取到最新数据：

```
+------------+----------------------------+----------------------+--------+
| id         | title                      | type                 | price  |
+------------+----------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 150.00 |
| 1064253862 | Collin Rolfson               | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat       | Magazine             | 159.75 |
|  893930596 | Myrl Hills                   | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman                  | Life                 | 477.91 |
+------------+----------------------------+----------------------+--------+
5 rows in set (0.01 sec)
```

在 Java 中，可以先定义一个事务的工具类，将开启事务级别 Stale Read 的命令封装成工具方法。

```
public static class StaleReadHelper {

    public static void startTxnWithStaleRead(Connection conn, Integer seconds) throws S
QLException {
        conn.setAutoCommit(false);
        PreparedStatement stmt = conn.prepareStatement(
            "START TRANSACTION READ ONLY AS OF TIMESTAMP NOW() - INTERVAL ? SEC
OND;"
        );
        stmt.setInt(1, seconds);
        stmt.execute();
    }

}
```

然后在 BookDAO 类当中定义一个通过事务开启 Stale Read 功能的方法，在方法内查询最新的书籍列表，但是不再在查询语句中添加 AS OF TIMESTAMP。

```
public class BookDAO {

    // Omit some code...

    public List<Book> getTop5LatestBooksWithTxnStaleRead(Integer seconds) throws SQ
LException {
        List<Book> books = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            // Start a read only transaction.
            TxnHelper.startTxnWithStaleRead(conn, seconds);

            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("""
            SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
            """);
            while (rs.next()) {
                Book book = new Book();
                book.setId(rs.getLong("id"));
                book.setTitle(rs.getString("title"));
                book.setType(rs.getString("type"));
                book.setPrice(rs.getDouble("price"));
                books.add(book);
            }

            // Commit transaction.
            conn.commit();
        } catch (SQLException e) {
            if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 1105) {
                System.out.println("WARN: cannot set read timestamp to a future time.");
            } else if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 9006) {
                System.out.println("WARN: GC life time is shorter than transaction duration.");
            } else {
                throw e;
            }
        }
        return books;
    }
}

List<Book> top5LatestBooks = bookDAO.getTop5LatestBooks();

if (top5LatestBooks.size() > 0) {
    System.out.println("The latest book price (before update): " + top5LatestBooks.get(0).g
etPrice());
```

```
    Book book = top5LatestBooks.get(0);
    bookDAO.updateBookPriceByID(book.getId(), book.price + 10);

    top5LatestBooks = bookDAO.getTop5LatestBooks();
    System.out.println("The latest book price (after update): " + top5LatestBooks.get(0).get
Price());

    // Use the stale read.
    top5LatestBooks = bookDAO.getTop5LatestBooksWithTxnStaleRead(5);
    System.out.println("The latest book price (maybe stale): " + top5LatestBooks.get(0).get
Price());

    // After the stale read transaction is committed.
    top5LatestBooks = bookDAO.getTop5LatestBooks();
    System.out.println("The latest book price (after the transaction commit): " + top5Latest
Books.get(0).getPrice());
}
```

输出结果：

```
The latest book price (before update): 100.00
The latest book price (after update): 150.00
The latest book price (maybe stale): 100.00
The latest book price (after the transaction commit): 150
```

通过 SET TRANSACTION READ ONLY AS OF TIMESTAMP 语句，你可以将当前事务或下一个事务设置为基于指定历史时间的只读事务。该事务将会基于所提供的历史时间来读取历史数据。

例如，可以通过下面这个 SQL 将已开启的事务切换到只读模式，通过 AS OF TIMESTAMP 语句开启能够读取 5 秒前的历史数据 Stale Read 功能。

```
SET TRANSACTION READ ONLY AS OF TIMESTAMP NOW() - INTERVAL 5 SECOND;
```

可以先定义一个事务的工具类，将开启事务级别 Stale Read 的命令封装成工具方法。

```
public static class TxnHelper {

    public static void setTxnWithStaleRead(Connection conn, Integer seconds) throws SQ
LException {
        PreparedStatement stmt = conn.prepareStatement(
            "SET TRANSACTION READ ONLY AS OF TIMESTAMP NOW() - INTERVAL ? SECON
D;"
        );
        stmt.setInt(1, seconds);
        stmt.execute();
    }

}
```

然后在 BookDAO 类当中定义一个通过事务开启 Stale Read 功能的方法，在方法内查询最新的书籍列表，但是不再在查询语句中添加 AS OF TIMESTAMP。

```
public class BookDAO {

    // Omit some code...

    public List<Book> getTop5LatestBooksWithTxnStaleRead2(Integer seconds) throws S
QLException {
        List<Book> books = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            // Start a read only transaction.
            conn.setAutoCommit(false);
            StaleReadHelper.setTxnWithStaleRead(conn, seconds);

            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("""
            SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
            """);
            while (rs.next()) {
                Book book = new Book();
                book.setId(rs.getLong("id"));
                book.setTitle(rs.getString("title"));
                book.setType(rs.getString("type"));
                book.setPrice(rs.getDouble("price"));
                books.add(book);
            }

            // Commit transaction.
```

```
            conn.commit();
        } catch (SQLException e) {
            if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 1105) {
                System.out.println("WARN: cannot set read timestamp to a future time.");
            } else if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 9006) {
                System.out.println("WARN: GC life time is shorter than transaction duration.");
            } else {
                throw e;
            }
        }
        return books;
    }
}
```

8.8.2.4 会话级别

为支持读取历史版本数据，TiDB 从 5.4 版本起引入了一个新的系统变量 tidb_read_staleness。系统变量 tidb_read_staleness 用于设置当前会话允许读取的历史数据范围，其数据类型为 int，作用域为 SESSION。

在会话中开启 Stale Read：

```
SET @@tidb_read_staleness="-5";
```

比如，如果该变量的值设置为 -5，TiDB 会在 5 秒时间范围内，保证 TiKV 或者 TiFlash 拥有对应历史版本数据的情况下，选择尽可能新的一个时间戳。

关闭会话当中的 Stale Read：

```
set @@tidb_read_staleness="";
```

在 Java 中示例如下：

```
public static class StaleReadHelper{

    public static void enableStaleReadOnSession(Connection conn, Integer seconds) thro
ws SQLException {
        PreparedStatement stmt = conn.prepareStatement(
            "SET @@tidb_read_staleness= ?;"
```

```
        );
        stmt.setString(1, String.format("-%d", seconds));
        stmt.execute();
    }

    public static void disableStaleReadOnSession(Connection conn) throws SQLException
{
        PreparedStatement stmt = conn.prepareStatement(
            "SET @@tidb_read_staleness=\"\";"
        );
        stmt.execute();
    }

}
```

8.8.2.5 扩展阅读

- Stale Read 功能的使用场景
- 使用 AS OF TIMESTAMP 语法读取历史数据
- 通过系统变量 tidb_read_staleness 读取历史数据

8.9 HTAP 查询

HTAP 是 Hybrid Transactional / Analytical Processing 的缩写。传统意义上，数据库往往专为交易或者分析场景设计，因而数据平台往往需要被切分为 Transactional Processing 和 Analytical Processing 两个部分，而数据需要从交易库复制到分析型数据库以便快速响应分析查询。而 TiDB 数据库则可以同时承担交易和分析两种职能，这大大简化了数据平台的建设，也能让用户使用更新鲜的数据进行分析。

在 TiDB 当中，同时拥有面向在线事务处理的行存储引擎 TiKV 与面向实时分析场景的列存储引擎 TiFlash 两套存储引擎。数据在行存 (Row-Store) 与列存 (Columnar-Store) 同时存在，自动同步，保持强一致性。行存为在线事务处理 OLTP 提供优化，列存则为在线分析处理 OLAP 提供性能优化。

在创建数据库章节当中，已经介绍了如何开启 TiDB 的 HTAP 能力。下面将进一步介绍如何使用 HTAP 能力更快地分析数据。

8.9.1 数据准备

在开始之前，你可以通过 tiup demo 命令导入更加大量的示例数据，例如：

```
tiup demo bookshop prepare --users=200000 --books=500000 --authors=100000 --ratings=1000000 --orders=1000000 --host 127.0.0.1 --port 4000 --drop-tables
```

8.9.2 窗口函数

在使用数据库时，除了希望它能够存储想要记录的数据，能够实现诸如下单买书、给书籍评分等业务功能外，可能还需要对已有的数据进行分析，以便根据数据作出进一步的运营和决策。

在单表读取章节当中，已经介绍了如何使用聚合查询来分析数据的整体情况，在更为复杂的使用场景下，你可能希望多个聚合查询的结果汇总在一个查询当中。例如：你想要对某一本书的订单量的历史趋势有所了解，就需要在每个月都对所有订单数据进行一次聚合求 sum，然后将 sum 结果汇总在一起才能够得到历史的趋势变化数据。

为了方便用户进行此类分析，TiDB 从 3.0 版本开始便支持了窗口函数功能，窗口函数为每一行数据提供了跨行数据访问的能力，不同于常规的聚合查询，窗口函数在对数据行进行聚合时不会导致结果集被合并成单行数据。

与聚合函数类似，窗口函数在使用时也需要搭配一套固定的语法：

```
SELECT
   window_function() OVER ([partition_clause] [order_clause] [frame_clause]) AS alias
FROM
   table_name
```

8.9.2.1 ORDER BY 子句

例如：可以利用聚合窗口函数 sum() 函数的累加效果来实现对某一本书的订单量的历史趋势的分析:

```
WITH orders_group_by_month AS (
  SELECT DATE_FORMAT(ordered_at, '%Y-%c') AS month, COUNT(*) AS orders
  FROM orders
  WHERE book_id = 3461722937
  GROUP BY 1
)
SELECT
month,
SUM(orders) OVER(ORDER BY month ASC) as acc
FROM orders_group_by_month
ORDER BY month ASC;
```

sum() 函数会在 OVER 子句当中通过 ORDER BY 子句指定的排序方式按顺序对数据进行累加，累加的结果如下：

```
+---------+-------+
| month   | acc   |
+---------+-------+
| 2011-5  |     1 |
| 2011-8  |     2 |
| 2012-1  |     3 |
| 2012-2  |     4 |
| 2013-1  |     5 |
| 2013-3  |     6 |
| 2015-11 |     7 |
| 2015-4  |     8 |
| 2015-8  |     9 |
| 2017-11 |    10 |
| 2017-5  |    11 |
| 2019-5  |    13 |
| 2020-2  |    14 |
+---------+-------+
13 rows in set (0.01 sec)
```

将得到的数据通过一个横轴为时间，纵轴为累计订单量的折线图进行可视化，便可以轻松地通过折线图的斜率变化宏观地了解到这本书的历史订单的增长趋势。

8.9.2.2 PARTITION BY 子句

把需求变得更复杂一点，假设想要分析不同类型书的历史订单增长趋势，并且希望将这些数据通过同一个多系列折线图进行呈现。

可以利用 PARTITION BY 子句根据书的类型进行分组，对不同类型的书籍分别统计它们的订单历史订单累计量。

```
WITH orders_group_by_month AS (
   SELECT
      b.type AS book_type,
      DATE_FORMAT(ordered_at, '%Y-%c') AS month,
      COUNT(*) AS orders
   FROM orders o
   LEFT JOIN books b ON o.book_id = b.id
   WHERE b.type IS NOT NULL
   GROUP BY book_type, month
), acc AS (
   SELECT
      book_type,
      month,
      SUM(orders) OVER(PARTITION BY book_type ORDER BY book_type, month ASC)
as acc
   FROM orders_group_by_month
   ORDER BY book_type, month ASC
)
SELECT * FROM acc;
```

查询结果如下：

```
+----------------------------+---------+------+
| book_type                  | month   | acc  |
+----------------------------+---------+------+
| Magazine                   | 2011-10 |    1 |
| Magazine                   | 2011-8  |    2 |
| Magazine                   | 2012-5  |    3 |
| Magazine                   | 2013-1  |    4 |
| Magazine                   | 2013-6  |    5 |
...
| Novel                      | 2011-3  |   13 |
| Novel                      | 2011-4  |   14 |
| Novel                      | 2011-6  |   15 |
| Novel                      | 2011-8  |   17 |
| Novel                      | 2012-1  |   18 |
| Novel                      | 2012-2  |   20 |
...
| Sports                     | 2021-4  |   49 |
| Sports                     | 2021-7  |   50 |
```

```
| Sports                     | 2022-4 |   51 |
+----------------------------+---------+------+
1500 rows in set (1.70 sec)
```

8.9.2.3 非聚合窗口函数

除此之外，TiDB 还提供了一些非聚合的窗口函数，可以借助这些函数实现更加丰富分析查询。

例如，在前面的分页查询章节当中，已经介绍了如何巧妙地利用 row_number() 函数实现高效的分页批处理能力。

8.9.3 混合负载

当将 TiDB 应用于在线实时分析处理的混合负载场景时，开发人员只需要提供一个入口，TiDB 将自动根据业务类型选择不同的处理引擎。

8.9.3.1 开启列存副本

TiDB 默认使用的存储引擎 TiKV 是行存的，你可以通过阅读开启 HTAP 能力章节，在进行后续步骤前，先通过如下 SQL 对 books 与 orders 表添加 TiFlash 列存副本：

```
ALTER TABLE books SET TIFLASH REPLICA 1;
ALTER TABLE orders SET TIFLASH REPLICA 1;
```

通过执行下面的 SQL 语句可以查看到 TiDB 创建列存副本的进度：

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'booksho
p' and TABLE_NAME = 'books';
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'booksho
p' and TABLE_NAME = 'orders';
```

当 PROGRESS 列为 1 时表示同步进度完成度达到 100%，AVAILABLE 列为 1 表示副本当前可用。

```
+--------------+------------+----------+--------------+----------------+-----------+-------
---+
| TABLE_SCHEMA | TABLE_NAME | TABLE_ID | REPLICA_COUNT | LOCATION_LABELS | AVA
ILABLE | PROGRESS |
+--------------+------------+----------+--------------+----------------+-----------+-------
---+
```

```
| bookshop    | books     |     143 |          1 |             |       1 |       1 |
+-------------+-----------+---------+------------+-------------+---------+---------+
1 row in set (0.07 sec)
+-------------+-----------+---------+------------+-------------+---------+---------+
| TABLE_SCHEMA | TABLE_NAME | TABLE_ID | REPLICA_COUNT | LOCATION_LABELS | AVAILABLE | PROGRESS |
+-------------+-----------+---------+------------+-------------+---------+---------+
| bookshop    | orders    |     147 |          1 |             |       1 |       1 |
+-------------+-----------+---------+------------+-------------+---------+---------+
1 row in set (0.07 sec)
```

副本添加完成之后，你可以通过使用 EXPLAIN 语句查看上面窗口函数示例 SQL 的执行计划。你会发现执行计划当中已经出现了 cop[tiflash] 字样，说明 TiFlash 引擎已经开始发挥作用了。

再次执行示例 SQL，查询结果如下：

```
+----------------------------+---------+------+
| book_type                  | month   | acc  |
+----------------------------+---------+------+
| Magazine                   | 2011-10 |    1 |
| Magazine                   | 2011-8  |    2 |
| Magazine                   | 2012-5  |    3 |
| Magazine                   | 2013-1  |    4 |
| Magazine                   | 2013-6  |    5 |
...
| Novel                      | 2011-3  |   13 |
| Novel                      | 2011-4  |   14 |
| Novel                      | 2011-6  |   15 |
| Novel                      | 2011-8  |   17 |
| Novel                      | 2012-1  |   18 |
| Novel                      | 2012-2  |   20 |
...
| Sports                     | 2021-4  |   49 |
| Sports                     | 2021-7  |   50 |
| Sports                     | 2022-4  |   51 |
+----------------------------+---------+------+
1500 rows in set (0.79 sec)
```

通过对比前后两次的执行结果，你会发现使用 TiFlash 处理有查询速度有了较为明显的提升（当数据量更大时，提升会更为显著）。这是因为在使用窗口函数时往往需要对某些列的数据进行全表扫描，相比行存的 TiKV，列存的 TiFlash 更加适合来处理这类分析型任务的负载。而对于 TiKV 来说，如果能够通过主键或索引快速地将所要查询的行数减少，往往查询速度也会非常快，而且所消耗的资源一般相对 TiFlash 而言会更少。

8.9.3.2 指定查询引擎

尽管 TiDB 会使用基于成本的优化器（CBO）自动地根据代价估算选择是否使用 TiFlash 副本。但是在实际使用当中，如果你非常确定查询的类型，推荐你使用 Optimizer Hints 明确的指定查询所使用的执行引擎，避免因为优化器的优化结果不同，导致应用程序性能出现波动。

你可以像下面的 SQL 一样在 SELECT 语句中通过 Hint /*+ read_from_storage(engine_name[table_name]) */ 指定查询时需要使用的查询引擎。

注意：

1. 如果你的表使用了别名，你应该将 Hints 当中的 table_name 替代为 alias_name，否则 Hints 会失效。
2. 另外，对公共表表达式设置 read_from_storage Hint 是不起作用的。

```
WITH orders_group_by_month AS (
  SELECT
    /*+ read_from_storage(tikv[o]) */
    b.type AS book_type,
    DATE_FORMAT(ordered_at, '%Y-%c') AS month,
    COUNT(*) AS orders
  FROM orders o
  LEFT JOIN books b ON o.book_id = b.id
  WHERE b.type IS NOT NULL
  GROUP BY book_type, month
), acc AS (
  SELECT
    book_type,
    month,
    SUM(orders) OVER(PARTITION BY book_type ORDER BY book_type, month ASC)
```

```
as acc
   FROM orders_group_by_month mo
   ORDER BY book_type, month ASC
)
SELECT * FROM acc;
```

如果你通过 EXPLAIN 语句查看上面 SQL 的执行计划，你会发现 task 列中会同时出现 cop[tiflash] 和 cop[tikv]，这意味着 TiDB 在处理这个查询的时候会同时调度行存查询引擎和列存查询引擎来完成查询任务。需要指出的是，因为 tiflash 和 tikv 存储引擎通常属于不同的计算节点，所以两种查询类型互相之间不受影响。

你可以通过阅读使用 TiDB 读取 TiFlash 小节进一步了解 TiDB 如何选择使用 TiFlash 作为查询引擎。

8.9.4 扩展阅读

- HTAP 快速上手指南
- HTAP 深入探索指南
- 窗口函数
- 使用 TiFlash

9 向量搜索

9.1 向量搜索概述

TiDB 向量搜索提供了一种高级的语义搜索功能，可以在文档、图像、音频和视频等多种数据类型之间进行相似度搜索。TiDB 向量搜索的 SQL 语法与 MySQL 兼容，熟悉 MySQL 的开发人员可以基于该功能轻松构建人工智能 (AI) 应用。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.1.1 概念

向量搜索是一种优先考虑数据语义以提供相关结果的搜索方法。

与传统的全文搜索（主要依赖于精确的关键词匹配和词频）不同，向量搜索通过将不同类型的数据（如文本、图像或音频）转换为高维向量，并根据这些向量之间的相似度来进行查询。这种搜索方法能够捕捉数据的语义特征和上下文信息，从而更准确地理解用户意图。

即使搜索的词语与数据库中的内容不完全匹配，向量搜索仍然可以通过对数据语义的理解，找到与用户意图相符合的结果。

例如，搜索 “一种会游泳的动物” 时，全文搜索只会返回包含这些精确关键词的结果，而向量搜索可以返回其他游泳动物的结果，如鱼或鸭子，即使这些结果并未包含精确的关键词。

9.1.1.1 向量嵌入

向量嵌入 (vector embedding) 也称为嵌入 (embedding) ，是在高维空间中用于表示现实世界对象的数字序列。它可以捕捉文档、图像、音频和视频等非结构化数据的语义特征和上下文。

向量嵌入在机器学习中至关重要，是语义相似性搜索的基础。

TiDB 专门引入了向量数据类型以及向量搜索索引，用于优化向量嵌入的存储和检索，增强其在人工智能领域的应用。你可以使用向量类型在 TiDB 中存储向量嵌入，并执行向量搜索查询，找到语义上最相关的数据。

9.1.1.2 嵌入模型

嵌入模型是将数据转换为向量嵌入的算法。

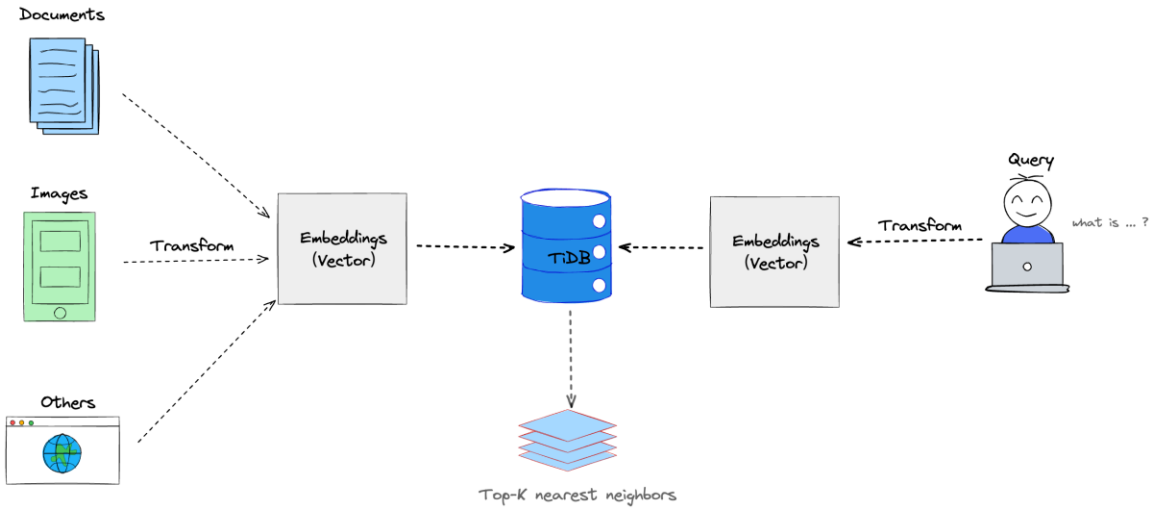
选择一种合适的嵌入模型对于确保语义搜索结果的准确性和相关性至关重要。对于非结构化的文本数据，你可以在 Massive Text Embedding Benchmark (MTEB) Leaderboard 上找到性能最佳的文本嵌入模型。

如需了解如何为特定数据类型生成向量嵌入，请参阅相关嵌入模型的教程或示例。

9.1.2 工作原理

在你将原始数据转换为向量并存储在 TiDB 中后，你的应用程序就可以开始利用这些向量来执行向量搜索查询，找到与用户查询语义或上下文最相关的数据。

TiDB 向量搜索 (Vector Search) 通过使用距离函数来计算给定向量与数据库中存储的向量之间的距离，从而识别前 k 个近邻（KNN）向量。其中，与给定向量距离最小的向量即代表最相似的数据。



The Schematic TiDB Vector Search

TiDB 作为一款关系型数据库，在引入了向量搜索功能后，支持将数据及其对应的向量表示（向量嵌入）存储在同一个数据库中。你可以选择以下任一种存储方式：

- 将数据和对应的向量表示存储在同一张表的不同列中。
- 将数据和对应的向量表示分别存储在不同的表中。在进行搜索时，通过 JOIN 查询将这些表关联起来。

9.1.3 使用场景

9.1.3.1 检索增强生成 (Retrieval-Augmented Generation, RAG)

检索增强生成（RAG）是一种优化大型语言模型（LLM）输出的架构。通过使用向量搜索，RAG 应用程序可以在数据库中存储向量嵌入，并在 LLM 生成回复时检索相关文档作为附加上下文，从而提高回复的质量和相关性。

9.1.3.2 语义搜索

语义搜索是一种根据查询的含义而不是简单地匹配关键词来返回结果的搜索技术。它将不同语言和各种类型的数据（如文本、图像、音频）的含义转换为向量嵌入。然后，向量搜索算法会使用这些向量嵌入来查找满足用户查询的最相关数据。

9.1.3.3 推荐引擎

推荐引擎是一种推荐系统，它会主动向用户推荐与他们高度相关且个性化的内容、产品或服务。为了实现这一目标，推荐引擎会创建反映用户行为和偏好的向量嵌入。这些嵌入可以帮助系统识别其他用户曾经互动过或感兴趣的类似项目，从而提高推荐内容与用户的相关性，并增加吸引用户的可能性。

9.1.4 另请参阅

要开始使用 TiDB 向量搜索，请参阅以下文档：

- 使用 Python 开始向量搜索
- 使用 SQL 开始向量搜索

9.2 快速入门

9.2.1 使用 SQL 开始向量搜索

TiDB 扩展了 MySQL 语法以支持向量搜索，并引入了向量数据类型和多个向量函数。

本文将展示如何使用 SQL 语句在 TiDB 中进行向量搜索。在本文中，你将使用 MySQL 命令行客户端完成以下任务：

- 连接到 TiDB 集群
- 创建向量表
- 存储向量嵌入
- 执行向量搜索查询

警告：

向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.2.1.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 MySQL 命令行客户端 (MySQL CLI)
- 准备一个 TiDB 集群

如果你还没有 TiDB 集群，可以按照以下方式创建：

参考部署本地测试集群或部署正式集群，创建本地集群。

9.2.1.2 快速开始

9.2.1.2.1 第 1 步：连接到平凯数据库集群

在本地部署的集群启动后，在终端中执行你的集群连接命令：

以下为 macOS 上的连接命令示例：

```
mysql --comments --host 127.0.0.1 --port 4000 -u root
```

9.2.1.2.2 第 2 步：创建向量表

创建表时，你可以使用 VECTOR 数据类型声明指定列为向量列。

例如，要创建一个带有三维 VECTOR 列的 embedded_documents 表，可以使用 MySQL CLI 执行以下 SQL 语句：

```
USE test;
CREATE TABLE embedded_documents (
  id       INT       PRIMARY KEY,
  -- document 列存储文档的原始内容
  document  TEXT,
  -- embedding 列存储文档的向量表示
  embedding VECTOR(3)
);
```

预期输出如下：

```
Query OK, 0 rows affected (0.27 sec)
```

9.2.1.2.3 第 3 步：向表中插入向量

向 embedded_documents 表中插入三行，每一行包含数据和数据的向量嵌入：

```
INSERT INTO embedded_documents
VALUES
    (1, 'dog', '[1,2,1]'),
    (2, 'fish', '[1,2,4]'),
    (3, 'tree', '[1,0,0]');
```

预期输出如下：

```
Query OK, 3 rows affected (0.15 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

> **Note**
>
> 为了方便展示，本示例简化了向量的维数，仅使用三维向量。
>
> 在实际应用中，嵌入模型通常会生成数百或数千维的向量。

9.2.1.2.4 第 4 步：查询向量表

要验证上一步中的三行数据是否已正确插入，可以查询 embedded_documents 表：

```
SELECT * FROM embedded_documents;
```

预期输出如下：

```
+----+----------+-----------+
| id | document | embedding |
+----+----------+-----------+
|  1 | dog      | [1,2,1]   |
|  2 | fish     | [1,2,4]   |
|  3 | tree     | [1,0,0]   |
+----+----------+-----------+
3 rows in set (0.15 sec)
```

9.2.1.2.5 第 5 步：执行向量搜索查询

与全文搜索类似，在使用向量搜索时，你需要提供搜索词。

在本例中，搜索词是“一种会游泳的动物”，假设其对应的向量是 [1,2,3]。在实际应用中，你需要使用嵌入模型将用户的搜索词转换为向量。

执行以下 SQL 语句后，TiDB 会计算 [1,2,3] 与表中各向量之间的余弦距离 (vec_cosine_distance)，然后对这些距离进行排序并输出表中最接近搜索向量（余弦距离最小）的前三个文档。

```
SELECT id, document, vec_cosine_distance(embedding, '[1,2,3]') AS distance
FROM embedded_documents
ORDER BY distance
LIMIT 3;
```

预期输出如下：

```
+----+----------+--------------------+
| id | document | distance           |
+----+----------+--------------------+
|  2 | fish     | 0.00853986601633272 |
|  1 | dog      | 0.12712843905603044 |
|  3 | tree     |  0.7327387580875756 |
+----+----------+--------------------+
3 rows in set (0.15 sec)
```

搜索结果中的三个词按它们与查询向量的距离排序：距离越小，对应的 document 越相关。

因此，从输出结果来看，会游泳的动物很可能是一条鱼 (fish)，或者是一只有游泳天赋的狗 (dog)。

9.2.1.3 另请参阅

- 向量数据类型
- 向量搜索索引

9.2.2 使用 Python 开始向量搜索

本文将展示如何开发一个简单的 AI 应用，这个 AI 应用实现了简单的**语义搜索**功能。不同于传统的关键字搜索，语义搜索可以智能地理解你的输入，返回更相关的结果。例如，在“狗”、“鱼”和“树”这三条内容中搜索“一种会游泳的动物”时，语义搜索会将“鱼”作为最相关的结果返回。

在本文中，你将使用 TiDB 向量搜索、Python、TiDB Vector Python SDK 和 AI 大模型完成这个 AI 应用的开发。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.2.2.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Python 3.8 或更高版本
- 在你的机器上安装 Git
- 准备一个 TiDB 集群

如果你还没有 TiDB 集群，可以按照以下方式创建：

参考部署本地测试集群或部署正式集群，创建本地集群。

9.2.2.2 快速开始

以下为从零开始构建这个应用的详细步骤，你也可以从 pingcap/tidb-vector-python 开源代码库获取到完整代码，直接运行示例。

9.2.2.2.1 第 1 步：新建一个 Python 项目

在你的本地目录中，新建一个 Python 项目和一个名为 example.py 的文件：

```
mkdir python-client-quickstart
cd python-client-quickstart
touch example.py
```

9.2.2.2.2 第 2 步：安装所需的依赖

在该项目的目录下，运行以下命令安装所需的软件包：

```
pip install sqlalchemy pymysql sentence-transformers tidb-vector python-dotenv
```

- tidb-vector：用于与 TiDB 向量搜索交互的 Python 客户端。
- sentence-transformers：提供预训练模型的 Python 库，用于从文本生成向量嵌入。

9.2.2.2.3 第 3 步：配置平凯数据库集群的连接字符串

对于本地部署的 TiDB，请在 Python 项目的根目录下新建一个 .env 文件，将以下内容复制到 .env 文件中，并根据集群的连接参数修改环境变量值为 TiDB 实际对应的值：

```
TIDB_DATABASE_URL="mysql+pymysql://<USER>:<PASSWORD>@<HOST>:<PORT>/<DATABASE>"
### 例如：TIDB_DATABASE_URL="mysql+pymysql://root@127.0.0.1:4000/test"
```

如果你在本机运行 TiDB，<HOST> 默认为 127.0.0.1。<PASSWORD> 初始密码为空，若你是第一次启动集群，则无需带上此字段。

以下为各参数的解释：

- <USER>：连接 TiDB 集群的用户名。
- <PASSWORD>：连接 TiDB 集群的密码。
- <HOST>：TiDB 集群的主机号。
- <PORT>：TiDB 集群的端口。
- <DATABASE>：要连接的数据库名称。

9.2.2.2.4 第 4 步：初始化嵌入模型

嵌入模型用于将数据转换为向量嵌入。本示例将使用预训练模型 **msmarco-MiniLM-L12-cos-v5** 将文本数据转换为向量嵌入。该模型为一个轻量级模型，由 sentence-transformers 库提供，可将文本数据转换为 384 维的向量嵌入。

将以下代码复制到 example.py 文件中，完成模型的设置。这段代码初始化了一个 SentenceTransformer 实例，并定义了一个 text_too_embedding() 函数用于将文本数据转换为向量数据。

```
from sentence_transformers import SentenceTransformer

print("Downloading and loading the embedding model...")
embed_model = SentenceTransformer("sentence-transformers/msmarco-MiniLM-L12-cos-v5", trust_remote_code=True)
embed_model_dims = embed_model.get_sentence_embedding_dimension()

def text_to_embedding(text):
    """Generates vector embeddings for the given text."""
    embedding = embed_model.encode(text)
    return embedding.tolist()
```

9.2.2.2.5 第 5 步：连接到平凯数据库集群

使用 TiDBVectorClient 类连接到 TiDB 集群，并创建一个包含向量列的表 embedded_documents。

> **Note**
>
> 请确保你创建的表中向量列的维度与嵌入模型生成的向量维度一致。例如，**msmarco-MiniLM-L12-cos-v5** 模型生成的向量有 384 个维度，embedded_documents 的向量列维度也应为 384。

```
import os
from tidb_vector.integrations import TiDBVectorClient
from dotenv import load_dotenv

### 从 .env 文件加载连接配置信息
load_dotenv()
```

```
vector_store = TiDBVectorClient(
    # embedded_documents 表将用于存储向量数据
    table_name='embedded_documents',
    # 指定 TiDB 集群的连接字符串
    connection_string=os.environ.get('TIDB_DATABASE_URL'),
    # 指定嵌入模型生成的向量的维度
    vector_dimension=embed_model_dims,
    # 如果表已经存在，则重新创建该表
    drop_existing_table=True,
)
```

9.2.2.2.6 第 6 步：将文本数据转换为向量嵌入，并向表中插入数据

准备一些文本数据，比如 "dog"、"fish" 和 "tree"。以下代码将使用 text_to_embedding() 函数将这些文本数据转换为向量嵌入，然后将向量嵌入插入到 embedded_documents 表中：

```
documents = [
    {
        "id": "f8e7dee2-63b6-42f1-8b60-2d46710c1971",
        "text": "dog",
        "embedding": text_to_embedding("dog"),
        "metadata": {"category": "animal"},
    },
    {
        "id": "8dde1fbc-2522-4ca2-aedf-5dcb2966d1c6",
        "text": "fish",
        "embedding": text_to_embedding("fish"),
        "metadata": {"category": "animal"},
    },
    {
        "id": "e4991349-d00b-485c-a481-f61695f2b5ae",
        "text": "tree",
        "embedding": text_to_embedding("tree"),
        "metadata": {"category": "plant"},
    },
]

vector_store.insert(
    ids=[doc["id"] for doc in documents],
```

```
    texts=[doc["text"] for doc in documents],
    embeddings=[doc["embedding"] for doc in documents],
    metadatas=[doc["metadata"] for doc in documents],
)
```

9.2.2.2.7 第 7 步：执行语义搜索

查询一个与已有文档 documents 中任何单词都不匹配的关键词，比如 “a swimming animal”。

以下的代码会再次使用 text_to_embedding() 函数将查询文本转换为向量嵌入，然后使用该嵌入进行查询，找出最匹配的前三个词。

```
def print_result(query, result):
  print(f"Search result (\"{query}\"):")
  for r in result:
    print(f"- text: \"{r.document}\", distance: {r.distance}")

query = "a swimming animal"
query_embedding = text_to_embedding(query)
search_result = vector_store.query(query_embedding, k=3)
print_result(query, search_result)
```

运行 example.py 文件，输出结果如下：

```
Search result ("a swimming animal"):
- text: "fish", distance: 0.4562914811223072
- text: "dog", distance: 0.6469335836410557
- text: "tree", distance: 0.798545178640937
```

搜索结果中的三个词按它们与查询向量的距离排序：距离越小，对应的 document 越相关。

因此，从输出结果来看，会游泳的动物很可能是一条鱼 (fish)，或者是一只有游泳天赋的狗 (dog)。

9.2.2.3 另请参阅

- 向量数据类型
- 向量搜索索引

9.3 集成

9.3.1 向量搜索集成概览

本文档介绍了 TiDB 向量搜索支持的 AI 框架、嵌入模型和对象关系映射 (ORM) 库。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.3.1.1 AI 框架

TiDB 目前支持以下 AI 框架。基于这些 AI 框架，你可以使用 TiDB 向量搜索轻松构建 AI 应用程序。

AI 框架	教程
Langchain	在 LangChain 中使用平凯数据库向量搜索
LlamaIndex	在 LlamaIndex 中使用平凯数据库向量搜索

此外，你还可以使用 TiDB 完成多种其它需求，例如将 TiDB 用于 AI 应用程序的文档存储和知识图谱存储等。

9.3.1.2 嵌入模型和服务

TiDB 向量搜索支持存储高达 16383 维的向量，可适应大多数嵌入模型。

你可以使用自行部署的开源嵌入模型或第三方嵌入模型提供商的嵌入 API 来生成向量。

下表列出了部分主流嵌入模型服务提供商和相应的集成教程。

嵌入模型服务提供商	教程
Jina AI	结合 Jina AI 嵌入模型 API 使用平凯数据库向量搜索

9.3.1.3 对象关系映射 (ORM) 库

你可以将 TiDB 向量搜索功能与 ORM 库结合使用，以便与 TiDB 数据库交互。

下表列出了支持的 ORM 库和相应的使用教程：

语言	ORM/客户端	安装说明	教程
Python	TiDB Vector Client	pip install tidb-vector[client]	使用 Python 开始向量搜索
	SQLAlchemy	pip install tidb-vector	在 SQLAlchemy 中使用 TiDB 向量搜索
	peewee	pip install tidb-vector	在 peewee 中使用 TiDB 向量搜索
	Django	pip install django-tidb[vector]	在 Django 中使用 TiDB 向量搜索

9.3.2 AI 框架

9.3.2.1 在 LlamaIndex 中使用平凯数据库向量搜索

本文档将展示如何在 LlamaIndex 中使用 TiDB 向量搜索。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。
>
> **Tip**
>
> 你可以在 Jupyter Notebook 上查看完整的示例代码，或直接在 Colab 在线环境中运行示例代码。

9.3.2.1.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Python 3.8 或更高版本
- 在你的机器上安装 Jupyter Notebook
- 在你的机器上安装 Git

- 准备一个 TiDB 集群

如果你还没有 TiDB 集群，可以按照以下方式创建：

参考部署本地测试集群或部署正式集群，创建本地集群。

9.3.2.1.2 快速开始

本节将详细介绍如何将 TiDB 的向量搜索功能与 LlamaIndex 结合使用，以实现语义搜索。

第 1 步：新建 Jupyter Notebook 文件

在根目录下，新建一个名为 integrate_with_llamaindex.ipynb 的 Jupyter Notebook 文件：

```
touch integrate_with_llamaindex.ipynb
```

第 2 步：安装所需的依赖

在你的项目目录下，运行以下命令安装所需的软件包：

```
pip install llama-index-vector-stores-tidbvector
pip install llama-index
```

在 Jupyter Notebook 中打开 integrate_with_llamaindex.ipynb 文件，添加以下代码以导入所需的软件包：

```
import textwrap

from llama_index.core import SimpleDirectoryReader, StorageContext
from llama_index.core import VectorStoreIndex
from llama_index.vector_stores.tidbvector import TiDBVectorStore
```

第 3 步：配置环境变量

本文档使用 OpenAI 作为嵌入模型生成向量嵌入。在此步骤中，你需要提供集群的连接字符串和 OpenAI API 密钥。

运行以下代码，配置环境变量。代码运行后，系统会提示输入连接字符串和 OpenAI API 密钥：

```
#### Use getpass to securely prompt for environment variables in your terminal.
import getpass
import os

#### Connection string format: "mysql+pymysql://<USER>:<PASSWORD>@<HOST>:4000/<DB>?ssl_ca=/etc/ssl/cert.pem&ssl_verify_cert=true&ssl_verify_identity=true"
tidb_connection_string = getpass.getpass("TiDB Connection String:")
os.environ["OPENAI_API_KEY"] = getpass.getpass("OpenAI API Key:")
```

以 macOS 为例，集群的连接字符串如下所示：

```
TIDB_DATABASE_URL="mysql+pymysql://<USERNAME>:<PASSWORD>@<HOST>:<PORT>/<DATABASE_NAME>"
#### 例如： TIDB_DATABASE_URL="mysql+pymysql://root@127.0.0.1:4000/test"
```

请替换连接字符串中的参数为你的 TiDB 实际对应的值。如果你在本机运行 TiDB，默认 <HOST> 地址为 127.0.0.1。<PASSWORD> 初始密码为空，若你是第一次启动集群，则无需带上此字段。

以下为各参数的解释：

- <USERNAME>：连接 TiDB 集群的用户名。
- <PASSWORD>：连接 TiDB 集群的密码。
- <HOST>：TiDB 集群的主机地址。
- <PORT>：TiDB 集群的端口号。
- <DATABASE>：要连接的数据库名称。

第 4 步：加载样本文档

4.1 下载样本文档

在你的项目目录中创建一个名为 data/paul_graham/ 的目录，然后从 run-llama/llama_index GitHub 代码库中下载样本文档 paul_graham_essay.txt：

```
mkdir -p 'data/paul_graham/'
wget 'https://raw.githubusercontent.com/run-llama/llama_index/main/docs/docs/examples/data/paul_graham/paul_graham_essay.txt' -O 'data/paul_graham/paul_graham_essay.txt'
```

4.2 加载文档

使用 SimpleDirectoryReader 从 data/paul_graham/paul_graham_essay.txt 中加载示例文档：

```
documents = SimpleDirectoryReader("./data/paul_graham").load_data()
print("Document ID:", documents[0].doc_id)

for index, document in enumerate(documents):
  document.metadata = {"book": "paul_graham"}
```

第 5 步：生成并存储文档向量

5.1 初始化平凯数据库向量存储

以下代码将在 TiDB 中创建一个 paul_graham_test 表，该表针对向量搜索进行了优化。

```
tidbvec = TiDBVectorStore(
  connection_string=tidb_connection_url,
  table_name="paul_graham_test",
  distance_strategy="cosine",
  vector_dimension=1536,
  drop_existing_table=False,
)
```

执行成功后，你可以直接查看和访问 TiDB 数据库中的 paul_graham_test 表。

5.2 生成并存储向量嵌入

以下代码将解析文档以生成向量嵌入，并将向量嵌入存储到 TiDB 向量存储中。

```
storage_context = StorageContext.from_defaults(vector_store=tidbvec)
index = VectorStoreIndex.from_documents(
  documents, storage_context=storage_context, show_progress=True
)
```

预期输出如下：

```
Parsing nodes: 100%|██████████| 1/1 [00:00<00:00,  8.76it/s]
Generating embeddings: 100%|██████████| 21/21 [00:02<00:00,  8.22it/s]
```

第 6 步：执行向量搜索

以下代码将基于 TiDB 向量存储创建一个查询引擎，并执行语义相似性搜索。

```
query_engine = index.as_query_engine()
response = query_engine.query("What did the author do?")
print(textwrap.fill(str(response), 100))
```

> **注意：**
>
> TiDBVectorStore 只支持 default 查询模式。

预期输出如下：

```
The author worked on writing, programming, building microcomputers, giving talks at co
nferences,
publishing essays online, developing spam filters, painting, hosting dinner parties, and p
urchasing
a building for office use.
```

第 7 步：使用元数据过滤器进行搜索

为了优化搜索，你可以使用元数据过滤器来筛选出符合特定条件的近邻结果。

使用 book != “paul_graham” 过滤器查询

以下示例的查询将排除掉 book 元数据字段为 paul_graham 的结果：

```
from llama_index.core.vector_stores.types import (
  MetadataFilter,
  MetadataFilters,
)

query_engine = index.as_query_engine(
  filters=MetadataFilters(
    filters=[
      MetadataFilter(key="book", value="paul_graham", operator="!="),
    ]
```

```
  ),
  similarity_top_k=2,
)
response = query_engine.query("What did the author learn?")
print(textwrap.fill(str(response), 100))
```

预期输出如下：

```
Empty Response
```

Query with book == "paul_graham" filter

以下示例的查询将筛选出 book 元数据字段为 paul_graham 的结果：

```
from llama_index.core.vector_stores.types import (
  MetadataFilter,
  MetadataFilters,
)

query_engine = index.as_query_engine(
  filters=MetadataFilters(
    filters=[
      MetadataFilter(key="book", value="paul_graham", operator="=="),
    ]
  ),
  similarity_top_k=2,
)
response = query_engine.query("What did the author learn?")
print(textwrap.fill(str(response), 100))
```

预期输出如下：

```
The author learned programming on an IBM 1401 using an early version of Fortran in 9th
 grade, then
later transitioned to working with microcomputers like the TRS-80 and Apple II. Addition
ally, the
author studied philosophy in college but found it unfulfilling, leading to a switch to study
ing AI.
Later on, the author attended art school in both the US and Italy, where they observed a l
ack of
substantial teaching in the painting department.
```

第 8 步：删除文档

从索引中删除第一个文档：

```
tidbvec.delete(documents[0].doc_id)
```

检查文档是否已被删除：

```
query_engine = index.as_query_engine()
response = query_engine.query("What did the author learn?")
print(textwrap.fill(str(response), 100))
```

预期输出如下：

```
Empty Response
```

9.3.2.1.3 另请参阅

- 向量数据类型
- 向量搜索索引

9.3.2.2 在 LangChain 中使用平凯数据库向量搜索

本文档将展示如何在 LangChain 中使用 TiDB 向量搜索。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。
>
> **Tip**
>
> 你可以在 Jupyter Notebook 上查看完整的示例代码，也可以直接在 Colab 在线环境中运行示例代码。

9.3.2.2.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Python 3.8 或更高版本
- 在你的机器上安装 Jupyter Notebook

- 在你的机器上安装 Git
- 准备一个 TiDB 集群

如果你还没有 TiDB 集群，可以按照以下方式创建：

参考部署本地测试集群或部署正式集群，创建本地集群。

9.3.2.2.2 快速开始

本节将详细介绍如何将 TiDB 的向量搜索功能与 LangChain 结合使用，以实现语义搜索。

第 1 步：新建 Jupyter Notebook 文件

在根目录下，新建一个名为 integrate_with_langchain.ipynb 的 Jupyter Notebook 文件：

```
touch integrate_with_langchain.ipynb
```

第 2 步：安装所需的依赖

在你的项目目录下，运行以下命令安装所需的软件包：

```
pip install langchain langchain-community
pip install langchain-openai
pip install pymysql
pip install tidb-vector
```

在 Jupyter Notebook 中打开 integrate_with_langchain.ipynb 文件，添加以下代码以导入所需的软件包：

```
from langchain_community.document_loaders import TextLoader
from langchain_community.vectorstores import TiDBVectorStore
from langchain_openai import OpenAIEmbeddings
from langchain_text_splitters import CharacterTextSplitter
```

第 3 步：配置环境变量

本文档使用 OpenAI 作为嵌入模型生成向量嵌入。在此步骤中，你需要提供集群的连接字符串和 OpenAI API 密钥。

运行以下代码，配置环境变量。代码运行后，系统会提示输入连接字符串和 OpenAI API 密钥：

```
#### Use getpass to securely prompt for environment variables in your terminal.
import getpass
import os

#### Connection string format: "mysql+pymysql://<USER>:<PASSWORD>@<HOST>:4000/<DB>?ssl_ca=/etc/ssl/cert.pem&ssl_verify_cert=true&ssl_verify_identity=true"
tidb_connection_string = getpass.getpass("TiDB Connection String:")
os.environ["OPENAI_API_KEY"] = getpass.getpass("OpenAI API Key:")
```

以 macOS 为例，集群的连接字符串如下所示：

```
TIDB_DATABASE_URL="mysql+pymysql://<USERNAME>:<PASSWORD>@<HOST>:<PORT>/<DATABASE_NAME>"
#### 例如：TIDB_DATABASE_URL="mysql+pymysql://root@127.0.0.1:4000/test"
```

请替换连接字符串中的参数为你的 TiDB 实际对应的值。如果你在本机运行 TiDB，默认 <HOST> 地址为 127.0.0.1。<PASSWORD> 初始密码为空，若你是第一次启动集群，则无需带上此字段。

以下为各参数的解释：

- <USERNAME>：连接 TiDB 集群的用户名。
- <PASSWORD>：连接 TiDB 集群的密码。
- <HOST>：TiDB 集群的主机地址。
- <PORT>：TiDB 集群的端口号。
- <DATABASE>：要连接的数据库名称。

第 4 步：加载样本文档

4.1 下载样本文档

在你的项目目录中创建一个名为 data/how_to/ 的目录，然后从 langchain-ai/langchain 代码库中下载样本文档 state_of_the_union.txt。

```
mkdir -p 'data/how_to/'
wget 'https://raw.githubusercontent.com/langchain-ai/langchain/master/docs/docs/how_to/state_of_the_union.txt' -O 'data/how_to/state_of_the_union.txt'
```

4.2 加载并分割文档

从 data/how_to/state_of_the_union.txt 中加载示例文档，并使用 CharacterTextSplitter 将其分割成每块约 1000 个字符的文本块。

```
loader = TextLoader("data/how_to/state_of_the_union.txt")
documents = loader.load()
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
docs = text_splitter.split_documents(documents)
```

第 5 步：生成并存储文档向量

TiDB 支持使用余弦距离 (cosine) 和欧式距离 (L2) 来评估向量之间的相似性。在存储向量时，默认使用余弦距离。

以下代码将在 TiDB 中创建一个 embedded_documents 表，该表针对向量搜索进行了优化。

```
embeddings = OpenAIEmbeddings()
vector_store = TiDBVectorStore.from_documents(
    documents=docs,
    embedding=embeddings,
    table_name="embedded_documents",
    connection_string=tidb_connection_string,
    distance_strategy="cosine",  # default, another option is "l2"
)
```

成功执行后，你可以直接查看和访问 TiDB 数据库中的 embedded_documents 表。

第 6 步：执行向量搜索

本节将展示如何在 state_of_the_union.txt 文档中查询 “What did the president say about Ketanji Brown Jackson”。

```
query = "What did the president say about Ketanji Brown Jackson"
```

方式一：使用 similarity_search_with_score()

similarity_search_with_score() 方法用于计算文档内容与查询语句之间的向量距离。该距离是一个相似度的得分，其计算方式由所选的 distance_strategy 决定。该方法会返回得分最低的前 k 个文档。得分越低，说明文档与你的查询语句之间的相似度越高。

```
docs_with_score = vector_store.similarity_search_with_score(query, k=3)
for doc, score in docs_with_score:
  print("-" * 80)
  print("Score: ", score)
  print(doc.page_content)
  print("-" * 80)
```

预期输出

```
--------------------------------------------------------------------------------
Score:  0.18472413652518527
Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Votin
g Rights Act. And while you're at it, pass the Disclose Act so Americans can know who is f
unding our elections.

Tonight, I'd like to honor someone who has dedicated his life to serve this country: Justic
e Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the Un
ited States Supreme Court. Justice Breyer, thank you for your service.

One of the most serious constitutional responsibilities a President has is nominating som
eone to serve on the United States Supreme Court.

And I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brow
n Jackson. One of our nation's top legal minds, who will continue Justice Breyer's legacy
of excellence.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

Score: 0.21757513022785557
A former top litigator in private practice. A former federal public defender. And from a fa mily of public school educators and police officers. A consensus builder. Since she's been nominated, she's received a broad range of support—from the Fraternal Order of Police to former judges appointed by Democrats and Republicans.

And if we are to advance liberty and justice, we need to secure the Border and fix the im migration system.

We can do both. At our border, we've installed new technology like cutting-edge scanner s to better detect drug smuggling.

We've set up joint patrols with Mexico and Guatemala to catch more human traffickers.

We're putting in place dedicated immigration judges so families fleeing persecution and violence can have their cases heard faster.

We're securing commitments and supporting partners in South and Central America to h ost more refugees and secure their own borders.

Score: 0.22676987253721725
And for our LGBTQ+ Americans, let's finally get the bipartisan Equality Act to my desk. Th e onslaught of state laws targeting transgender Americans and their families is wrong.

As I said last year, especially to our younger transgender Americans, I will always have yo ur back as your President, so you can be yourself and reach your God-given potential.

While it often appears that we never agree, that isn't true. I signed 80 bipartisan bills into law last year. From preventing government shutdowns to protecting Asian-Americans fr om still-too-common hate crimes to reforming military justice.

And soon, we'll strengthen the Violence Against Women Act that I first wrote three deca des ago. It is important for us to show the nation that we can come together and do big things.

So tonight I'm offering a Unity Agenda for the Nation. Four big things we can do togethe r.

First, beat the opioid epidemic.

方式二：使用 similarity_search_with_relevance_scores() 方法

similarity_search_with_relevance_scores() 方法会返回相关性得分最高的前 k 个文档。分数越高，说明文档内容与你的查询语句之间的相似度越高。

```
docs_with_relevance_score = vector_store.similarity_search_with_relevance_scores(query, k=2)
for doc, score in docs_with_relevance_score:
    print("-" * 80)
    print("Score: ", score)
    print(doc.page_content)
    print("-" * 80)
```

预期输出

```
--------------------------------------------------------------------------------
Score:  0.8152758634748147
Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Votin
g Rights Act. And while you're at it, pass the Disclose Act so Americans can know who is f
unding our elections.

Tonight, I'd like to honor someone who has dedicated his life to serve this country: Justic
e Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the Un
ited States Supreme Court. Justice Breyer, thank you for your service.

One of the most serious constitutional responsibilities a President has is nominating som
eone to serve on the United States Supreme Court.

And I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brow
n Jackson. One of our nation's top legal minds, who will continue Justice Breyer's legacy
of excellence.
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Score:  0.7824248697721444
A former top litigator in private practice. A former federal public defender. And from a fa
mily of public school educators and police officers. A consensus builder. Since she's been
 nominated, she's received a broad range of support—from the Fraternal Order of Police
to former judges appointed by Democrats and Republicans.

And if we are to advance liberty and justice, we need to secure the Border and fix the im
migration system.

We can do both. At our border, we've installed new technology like cutting-edge scanner
```

s to better detect drug smuggling.

We've set up joint patrols with Mexico and Guatemala to catch more human traffickers.

We're putting in place dedicated immigration judges so families fleeing persecution and violence can have their cases heard faster.

We're securing commitments and supporting partners in South and Central America to h ost more refugees and secure their own borders.

用作检索器

在 Langchain 中，检索器是一个接口，用于响应非结构化查询，检索相关文档。相比于向量存储，检索器可以为你提供更多的功能。以下代码演示了如何将 TiDB 向量存储用作检索器。

```
retriever = vector_store.as_retriever(
  search_type="similarity_score_threshold",
  search_kwargs={"k": 3, "score_threshold": 0.8},
)
docs_retrieved = retriever.invoke(query)
for doc in docs_retrieved:
  print("-" * 80)
  print(doc.page_content)
  print("-" * 80)
```

预期输出如下：

```
--------------------------------------------------------------------------------
Tonight. I call on the Senate to: Pass the Freedom to Vote Act. Pass the John Lewis Votin g Rights Act. And while you're at it, pass the Disclose Act so Americans can know who is f unding our elections.

Tonight, I'd like to honor someone who has dedicated his life to serve this country: Justic e Stephen Breyer—an Army veteran, Constitutional scholar, and retiring Justice of the Un ited States Supreme Court. Justice Breyer, thank you for your service.

One of the most serious constitutional responsibilities a President has is nominating som eone to serve on the United States Supreme Court.

And I did that 4 days ago, when I nominated Circuit Court of Appeals Judge Ketanji Brow
```

n Jackson. One of our nation's top legal minds, who will continue Justice Breyer's legacy of excellence.

移除向量存储

要删除现有的 TiDB 向量存储，可以使用 drop_vectorstore() 方法：

```
vector_store.drop_vectorstore()
```

9.3.2.2.3 使用元数据过滤器进行搜索

为了优化搜索，你可以使用元数据过滤器来筛选出符合特定条件的近邻结果。

支持的元数据类型

在 TiDB 向量存储中，每个文档都可以与元数据配对。元数据的结构是 JSON 对象中的键值对 (key-value pairs) 形式。键 (key) 的类型是字符串，而值 (value) 可以是以下任何类型：

- 字符串
- 数值：整数或浮点数
- Boolean：true 或 false

例如，下面是一个有效的元数据格式：

```
{
  "page": 12,
  "book_title": "Siddhartha"
}
```

元数据过滤器语法

可用的过滤器包括：

- $or：选择符合任意一个指定条件的向量。
- $and：选择符合所有指定条件的向量。
- $eq：等于指定值。

- $ne：不等于指定值。
- $gt：大于指定值。
- $gte：大于或等于指定值。
- $lt：小于指定值。
- $lte：小于或等于指定值。
- $in：在指定的值数组中。
- $nin：不在指定值数组中。

假如一个文档的元数据如下：

```
{
  "page": 12,
  "book_title": "Siddhartha"
}
```

以下元数据筛选器均可匹配到该文档：

```
{ "page": 12 }
```

```
{ "page": { "$eq": 12 } }
```

```
{
  "page": {
    "$in": [11, 12, 13]
  }
}
```

```
{ "page": { "$nin": [13] } }
```

```
{ "page": { "$lt": 11 } }
```

```
{
  "$or": [{ "page": 11 }, { "page": 12 }],
  "$and": [{ "page": 12 }, { "page": 13 }]
}
```

TiDB 会将元数据过滤器中的每个键值对视为一个独立的过滤条件，并使用 AND 逻辑操作符将这些条件组合起来。

示例

以下示例代码向 TiDBVectorStore 添加了两个文档，并为每个文档添加了一个 title 字段作为元数据：

```
vector_store.add_texts(
    texts=[
        "TiDB Vector offers advanced, high-speed vector processing capabilities, enhancing AI workflows with efficient data handling and analytics support.",
        "TiDB Vector, starting as low as $10 per month for basic usage",
    ],
    metadatas=[
        {"title": "TiDB Vector functionality"},
        {"title": "TiDB Vector Pricing"},
    ],
)
```

预期输出如下：

```
[UUID('c782cb02-8eec-45be-a31f-fdb78914f0a7'),
 UUID('08dcd2ba-9f16-4f29-a9b7-18141f8edae3')]
```

使用元数据过滤器进行相似性搜索：

```
docs_with_score = vector_store.similarity_search_with_score(
    "Introduction to TiDB Vector", filter={"title": "TiDB Vector functionality"}, k=4
)
for doc, score in docs_with_score:
    print("-" * 80)
    print("Score: ", score)
    print(doc.page_content)
    print("-" * 80)
```

预期输出如下：

```
--------------------------------------------------------------------------------
Score:  0.12761409169211535
TiDB Vector offers advanced, high-speed vector processing capabilities, enhancing AI wo
rkflows with efficient data handling and analytics support.
--------------------------------------------------------------------------------
```

9.3.2.2.4 进阶用法示例：旅行代理

本节演示如何将 Langchain 和 TiDB 向量搜索相结合，应用于旅行代理的场景。该场景的目标是为客户创建个性化的旅行报告，帮助他们找到具备特定设施（例如干净的休息室和素食选项）的机场。

该示例包括两个主要步骤：

1. 对机场介绍中进行语义搜索，以找出符合所需设施的机场代码。
2. 执行 SQL 查询，将这些代码与航线信息相结合，以便突出显示符合用户偏好的航空公司和目的地。

准备数据

首先，创建一个表来存储机场航线数据：

```
#### 创建表格以存储飞行计划数据。
vector_store.tidb_vector_client.execute(
    """CREATE TABLE airplan_routes (
        id INT AUTO_INCREMENT PRIMARY KEY,
        airport_code VARCHAR(10),
        airline_code VARCHAR(10),
        destination_code VARCHAR(10),
        route_details TEXT,
        duration TIME,
        frequency INT,
        airplane_type VARCHAR(50),
        price DECIMAL(10, 2),
        layover TEXT
    );"""
)

#### 在 airplan_routes 和向量表中插入一些样本数据。
vector_store.tidb_vector_client.execute(
    """INSERT INTO airplan_routes (
        airport_code,
        airline_code,
        destination_code,
        route_details,
        duration,
```

```
        frequency,
        airplane_type,
        price,
        layover
    ) VALUES
    ('JFK', 'DL', 'LAX', 'Non-stop from JFK to LAX.', '06:00:00', 5, 'Boeing 777', 299.99, 'None
'),
    ('LAX', 'AA', 'ORD', 'Direct LAX to ORD route.', '04:00:00', 3, 'Airbus A320', 149.99, 'None
'),
    ('EFGH', 'UA', 'SEA', 'Daily flights from SFO to SEA.', '02:30:00', 7, 'Boeing 737', 129.99, '
None');
    """
)
vector_store.add_texts(
    texts=[
        "Clean lounges and excellent vegetarian dining options. Highly recommended.",
        "Comfortable seating in lounge areas and diverse food selections, including vegetari
an.",
        "Small airport with basic facilities.",
    ],
    metadatas=[
        {"airport_code": "JFK"},
        {"airport_code": "LAX"},
        {"airport_code": "EFGH"},
    ],
)
```

预期输出如下：

```
[UUID('6dab390f-acd9-4c7d-b252-616606fbc89b'),
 UUID('9e811801-0e6b-4893-8886-60f4fb67ce69'),
 UUID('f426747c-0f7b-4c62-97ed-3eeb7c8dd76e')]
```

执行语义搜索

以下代码可以搜索到有清洁设施和素食选择的机场：

```
retriever = vector_store.as_retriever(
    search_type="similarity_score_threshold",
    search_kwargs={"k": 3, "score_threshold": 0.85},
)
semantic_query = "Could you recommend a US airport with clean lounges and good veg
etarian dining options?"
reviews = retriever.invoke(semantic_query)
```

```
for r in reviews:
    print("-" * 80)
    print(r.page_content)
    print(r.metadata)
    print("-" * 80)
```

预期输出如下：

```
--------------------------------------------------------------------------------
Clean lounges and excellent vegetarian dining options. Highly recommended.
{'airport_code': 'JFK'}
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Comfortable seating in lounge areas and diverse food selections, including vegetarian.
{'airport_code': 'LAX'}
--------------------------------------------------------------------------------
```

检索详细的机场信息

从搜索结果中提取机场代码，查询数据库中的详细航线信息：

```
#### Extracting airport codes from the metadata
airport_codes = [review.metadata["airport_code"] for review in reviews]

#### Executing a query to get the airport details
search_query = "SELECT * FROM airplan_routes WHERE airport_code IN :codes"
params = {"codes": tuple(airport_codes)}

airport_details = vector_store.tidb_vector_client.execute(search_query, params)
airport_details.get("result")
```

预期输出如下：

```
[(1, 'JFK', 'DL', 'LAX', 'Non-stop from JFK to LAX.', datetime.timedelta(seconds=21600), 5, '
Boeing 777', Decimal('299.99'), 'None'),
 (2, 'LAX', 'AA', 'ORD', 'Direct LAX to ORD route.', datetime.timedelta(seconds=14400), 3, '
Airbus A320', Decimal('149.99'), 'None')]
```

简化流程

你也可以使用单个 SQL 查询来简化整个流程：

```
search_query = f"""
    SELECT
```

```
        VEC_Cosine_Distance(se.embedding, :query_vector) as distance,
        ar.*,
        se.document as airport_review
    FROM
        airplan_routes ar
    JOIN
        {TABLE_NAME} se ON ar.airport_code = JSON_UNQUOTE(JSON_EXTRACT(se.meta, '
$.airport_code'))
    ORDER BY distance ASC
    LIMIT 5;
"""
query_vector = embeddings.embed_query(semantic_query)
params = {"query_vector": str(query_vector)}
airport_details = vector_store.tidb_vector_client.execute(search_query, params)
airport_details.get("result")
```

预期输出如下：

```
[(0.1219207353407008, 1, 'JFK', 'DL', 'LAX', 'Non-stop from JFK to LAX.', datetime.timedelt
a(seconds=21600), 5, 'Boeing 777', Decimal('299.99'), 'None', 'Clean lounges and excellen
t vegetarian dining options. Highly recommended.'),
 (0.14613754359804654, 2, 'LAX', 'AA', 'ORD', 'Direct LAX to ORD route.', datetime.timede
lta(seconds=14400), 3, 'Airbus A320', Decimal('149.99'), 'None', 'Comfortable seating in lo
unge areas and diverse food selections, including vegetarian.'),
 (0.19840519342700513, 3, 'EFGH', 'UA', 'SEA', 'Daily flights from SFO to SEA.', datetime.ti
medelta(seconds=9000), 7, 'Boeing 737', Decimal('129.99'), 'None', 'Small airport with bas
ic facilities.')]
```

清理数据

最后，删除创建的表，清理资源：

```
vector_store.tidb_vector_client.execute("DROP TABLE airplan_routes")
```

预期输出如下：

```
{'success': True, 'result': 0, 'error': None}
```

9.3.2.2.5 另请参阅

- 向量数据类型
- 向量搜索索引

9.3.3 嵌入模型/服务

9.3.3.1 结合 Jina AI 嵌入模型 API 使用平凯数据库向量搜索

本文档将展示如何使用 Jina AI 为文本数据生成向量嵌入，然后将向量嵌入存储在 TiDB 中，并根据向量嵌入搜索相似文本。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.3.3.1.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Python 3.8 或更高版本
- 在你的机器上安装 Git
- 准备一个 TiDB 集群

如果你还没有 TiDB 集群，可以按照以下方式创建：

参考部署本地测试集群或部署正式集群，创建本地集群。

9.3.3.1.2 运行示例应用程序

您可以通过以下步骤快速了解如何结合 Jina AI 嵌入模型 API 使用 TiDB 向量搜索。

第 1 步：克隆示例代码仓库

将 tidb-vector-python 仓库克隆到本地：

git clone https://github.com/pingcap/tidb-vector-python.git

第 2 步：创建虚拟环境

为你的项目创建虚拟环境：

```
cd tidb-vector-python/examples/jina-ai-embeddings-demo
python3 -m venv .venv
source .venv/bin/activate
```

第 3 步：安装所需的依赖

安装项目所需的依赖：

```
pip install -r requirements.txt
```

第 4 步：配置环境变量

从 Jina AI Embeddings API 页面获取 Jina AI API 密钥，然后配置环境变量。

对于本地部署的 TiDB，你可以通过在终端中直接设置环境变量以连接 TiDB 集群：

```
export JINA_API_KEY="****"
export TIDB_DATABASE_URL="mysql+pymysql://<USERNAME>:<PASSWORD>@<HOST>:<PORT>/<DATABASE>"
#### 例如：export TIDB_DATABASE_URL="mysql+pymysql://root@127.0.0.1:4000/test"
```

请替换命令中的参数为你的 TiDB 实际对应的值。如果你在本机运行 TiDB，<HOST> 默认为 127.0.0.1。<PASSWORD> 初始密码为空，若你是第一次启动集群，则无需带上此字段。

以下为各参数的解释：

- <USERNAME>：连接 TiDB 集群的用户名。
- <PASSWORD>：连接 TiDB 集群的密码。
- <HOST>：TiDB 集群的主机地址。
- <PORT>：TiDB 集群的端口号。
- <DATABASE>：要连接的数据库名称。

第 5 步：运行示例应用程序

```
python jina-ai-embeddings-demo.py
```

示例输出：

```
- Inserting Data to TiDB...
  - Inserting: Jina AI offers best-in-class embeddings, reranker and prompt optimizer, ena
bling advanced multimodal AI.
  - Inserting: TiDB is an open-source MySQL-compatible database that supports Hybrid T
ransactional and Analytical Processing (HTAP) workloads.
- List All Documents and Their Distances to the Query:
  - distance: 0.3585317326132522
    content: Jina AI offers best-in-class embeddings, reranker and prompt optimizer, enabl
ing advanced multimodal AI.
  - distance: 0.10858102967720984
    content: TiDB is an open-source MySQL-compatible database that supports Hybrid Tra
nsactional and Analytical Processing (HTAP) workloads.
- The Most Relevant Document and Its Distance to the Query:
  - distance: 0.10858102967720984
    content: TiDB is an open-source MySQL-compatible database that supports Hybrid Tra
nsactional and Analytical Processing (HTAP) workloads.
```

9.3.3.1.3 示例代码片段

通过 Jina AI 获取嵌入信息

定义一个 generate_embeddings 函数，用于调用 Jina AI 的嵌入 API：

```
import os
import requests
import dotenv

dotenv.load_dotenv()

JINAAI_API_KEY = os.getenv('JINAAI_API_KEY')

def generate_embeddings(text: str):
    JINAAI_API_URL = 'https://api.jina.ai/v1/embeddings'
    JINAAI_HEADERS = {
        'Content-Type': 'application/json',
        'Authorization': f'Bearer {JINAAI_API_KEY}'
    }
    JINAAI_REQUEST_DATA = {
        'input': [text],
        'model': 'jina-embeddings-v2-base-en'  # with dimension 768.
    }
    response = requests.post(JINAAI_API_URL, headers=JINAAI_HEADERS, json=JINAAI_RE
```

```
QUEST_DATA)
    return response.json()['data'][0]['embedding']
```

连接到平凯数据库集群

通过 SQLAlchemy 连接 TiDB 集群：

```
import os
import dotenv

from tidb_vector.sqlalchemy import VectorType
from sqlalchemy.orm import Session, declarative_base

dotenv.load_dotenv()

TIDB_DATABASE_URL = os.getenv('TIDB_DATABASE_URL')
assert TIDB_DATABASE_URL is not None
engine = create_engine(url=TIDB_DATABASE_URL, pool_recycle=300)
```

定义向量表结构

创建一张 jinaai_tidb_demo_documents 表，其中包含一个 content 列用于存储文本，一个 content_vec 向量列用于存储向量嵌入：

```
from sqlalchemy import Column, Integer, String, create_engine
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Document(Base):
    __tablename__ = "jinaai_tidb_demo_documents"

    id = Column(Integer, primary_key=True)
    content = Column(String(255), nullable=False)
    content_vec = Column(
        # DIMENSIONS is determined by the embedding model,
        # for Jina AI's jina-embeddings-v2-base-en model it's 768.
        VectorType(dim=768)
    )
```

注意：

- 向量列的维度必须与嵌入模型生成的向量嵌入维度相同。

- 在本例中，jina-embeddings-v2-base-en 模型生成的向量嵌入维度为 768。

使用 Jina AI 生成向量嵌入并存入平凯数据库

使用 Jina AI 嵌入 API 为每条文本生成向量嵌入，并将这些向量存储在 TiDB 中：

```
TEXTS = [
  'Jina AI offers best-in-class embeddings, reranker and prompt optimizer, enabling advanced multimodal AI.',
  'TiDB is an open-source MySQL-compatible database that supports Hybrid Transactional and Analytical Processing (HTAP) workloads.',
]
data = []

for text in TEXTS:
  # 通过 Jina AI API 生成文本的向量嵌入
  embedding = generate_embeddings(text)
  data.append({
    'text': text,
    'embedding': embedding
  })

with Session(engine) as session:
  print('- Inserting Data to TiDB...')
  for item in data:
    print(f'  - Inserting: {item["text"]}')
    session.add(Document(
      content=item['text'],
      content_vec=item['embedding']
    ))
  session.commit()
```

使用 Jina AI 生成的向量嵌入在平凯数据库中执行语义搜索

通过 Jina AI 的嵌入 API 生成查询文本的向量嵌入，然后根据**查询文本的向量嵌入**和**向量表中各个向量嵌入**之间的余弦距离搜索最相关的 document：

```
query = 'What is TiDB?'
#### 通过 Jina AI API 生成查询文本的向量嵌入
query_embedding = generate_embeddings(query)

with Session(engine) as session:
```

```
print('- The Most Relevant Document and Its Distance to the Query:')
doc, distance = session.query(
    Document,
    Document.content_vec.cosine_distance(query_embedding).label('distance')
).order_by(
    'distance'
).limit(1).first()
print(f'  - distance: {distance}\n'
      f'    content: {doc.content}')
```

9.3.3.1.4 另请参阅

- 向量数据类型
- 向量搜索索引

9.3.4 ORM 库

9.3.4.1 在 SQLAlchemy 中使用平凯数据库向量搜索

本文档将展示如何使用 SQLAlchemy 与 TiDB 向量搜索进行交互，以及如何存储向量和执行向量搜索查询。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.3.4.1.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Python 3.8 或更高版本
- 在你的机器上安装 Git
- 准备一个 TiDB 集群

如果你还没有 TiDB 集群，可以按照以下方式创建：

参考部署本地测试集群或部署正式集群，创建本地集群。

9.3.4.1.2 运行示例应用程序

你可以通过以下步骤快速了解如何在 SQLAlchemy 中使用 TiDB 向量搜索。

第 1 步：克隆示例代码仓库

将 tidb-vector-python 仓库克隆到本地：

```
git clone https://github.com/pingcap/tidb-vector-python.git
```

第 2 步：创建虚拟环境

为你的项目创建虚拟环境：

```
cd tidb-vector-python/examples/orm-sqlalchemy-quickstart
python3 -m venv .venv
source .venv/bin/activate
```

第 3 步：安装所需的依赖

安装示例项目所需的依赖：

```
pip install -r requirements.txt
```

你也可以直接为项目安装以下依赖项：

```
pip install pymysql python-dotenv sqlalchemy tidb-vector
```

第 4 步：配置环境变量

对于本地部署的 TiDB，请在 Python 项目的根目录下新建一个 .env 文件，将以下内容复制到 .env 文件中，并根据集群的启动参数修改环境变量值为 TiDB 实际对应的值：

```
TIDB_DATABASE_URL=mysql+pymysql://<USERNAME>:<PASSWORD>@<HOST>:<PORT>/<DATABASE>
#### 例如：TIDB_DATABASE_URL="mysql+pymysql://root@127.0.0.1:4000/test"
```

如果你在本机运行 TiDB，<HOST> 默认为 127.0.0.1。<PASSWORD> 初始密码为空，若你是第一次启动集群，则无需带上此字段。

以下为各参数的解释：

- <USERNAME>：连接 TiDB 集群的用户名。
- <PASSWORD>：连接 TiDB 集群的密码。
- <HOST>：TiDB 集群的主机。
- <PORT>：TiDB 集群的端口。
- <DATABASE>：要连接的数据库名称。

第 5 步：运行示例应用程序

```
python sqlalchemy-quickstart.py
```

输出示例：

```
Get 3-nearest neighbor documents:
  - distance: 0.00853986601633272
    document: fish
  - distance: 0.12712843905603044
    document: dog
  - distance: 0.7327387580875756
    document: tree
Get documents within a certain distance:
  - distance: 0.00853986601633272
    document: fish
  - distance: 0.12712843905603044
    document: dog
```

9.3.4.1.3 示例代码片段

你可以参考以下示例代码片段来完成自己的应用程序开发。

创建向量表

连接到平凯数据库集群

```
import os
import dotenv

from sqlalchemy import Column, Integer, create_engine, Text
from sqlalchemy.orm import declarative_base, Session
from tidb_vector.sqlalchemy import VectorType
```

```
dotenv.load_dotenv()

tidb_connection_string = os.environ['TIDB_DATABASE_URL']
engine = create_engine(tidb_connection_string)
```

定义向量列

创建一个表格，其中包含一个向量数据类型的 embedding 列，用于存储三维向量。

```
Base = declarative_base()

class Document(Base):
    __tablename__ = 'sqlalchemy_demo_documents'
    id = Column(Integer, primary_key=True)
    content = Column(Text)
    embedding = Column(VectorType(3))
```

存储包含向量的 document

```
with Session(engine) as session:
    session.add(Document(content="dog", embedding=[1, 2, 1]))
    session.add(Document(content="fish", embedding=[1, 2, 4]))
    session.add(Document(content="tree", embedding=[1, 0, 0]))
    session.commit()
```

搜索近邻向量

可以选择使用余弦距离 (CosineDistance) 函数，查询与向量 [1, 2, 3] 语义最接近的前 3 个 document。

```
with Session(engine) as session:
    distance = Document.embedding.cosine_distance([1, 2, 3]).label('distance')
    results = session.query(
        Document, distance
    ).order_by(distance).limit(3).all()
```

搜索一定距离内的向量

可以选择使用余弦距离 (CosineDistance) 函数，查询与向量 [1, 2, 3] 的余弦距离小于 0.2 的向量。

```
with Session(engine) as session:
    distance = Document.embedding.cosine_distance([1, 2, 3]).label('distance')
    results = session.query(
        Document, distance
    ).filter(distance < 0.2).order_by(distance).limit(3).all()
```

9.3.4.1.4 另请参阅

- 向量数据类型
- 向量搜索索引

9.3.4.2 在 peewee 中使用平凯数据库向量搜索

本文档将展示如何使用 peewee 与 TiDB 向量搜索进行交互，以及如何存储向量和执行向量搜索查询。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.3.4.2.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Python 3.8 或更高版本
- 在你的机器上安装 Git
- 准备一个 TiDB 集群

如果你还没有 TiDB 集群，可以按照以下方式创建：

参考部署本地测试集群或部署正式集群，创建本地集群。

9.3.4.2.2 运行示例应用程序

你可以通过以下步骤快速了解如何在 peewee 中使用 TiDB 向量搜索。

第 1 步：克隆示例代码仓库

将 tidb-vector-python 仓库克隆到本地：

```
git clone https://github.com/pingcap/tidb-vector-python.git
```

第 2 步：创建虚拟环境

为你的项目创建虚拟环境：

```
cd tidb-vector-python/examples/orm-peewee-quickstart
python3 -m venv .venv
source .venv/bin/activate
```

第 3 步：安装所需的依赖

安装项目所需的依赖：

```
pip install -r requirements.txt
```

你也可以直接为项目安装以下依赖项：

```
pip install peewee pymysql python-dotenv tidb-vector
```

第 4 步：配置环境变量

对于本地部署的 TiDB，请在 Python 项目的根目录下新建一个 .env 文件，将以下内容复制到 .env 文件中，并根据集群的连接参数修改环境变量值为 TiDB 实际对应的值：

```
TIDB_HOST=127.0.0.1
TIDB_PORT=4000
TIDB_USERNAME=root
TIDB_PASSWORD=
TIDB_DATABASE=test
```

如果你在本机运行 TiDB，TIDB_HOST 默认为 127.0.0.1。TIDB_PASSWORD 初始密码为空，若你是第一次启动集群，则无需带上此字段。

以下为各参数的解释：

- TIDB_HOST：TiDB 集群的主机号。
- TIDB_PORT：TiDB 集群的端口号。
- TIDB_USERNAME：连接 TiDB 集群的用户名。
- TIDB_PASSWORD：连接 TiDB 集群的密码。
- TIDB_DATABASE：要连接的数据库名称。

第 5 步：运行示例应用程序

```
python peewee-quickstart.py
```

输出示例：

```
Get 3-nearest neighbor documents:
  - distance: 0.00853986601633272
    document: fish
  - distance: 0.12712843905603044
    document: dog
  - distance: 0.7327387580875756
    document: tree
Get documents within a certain distance:
  - distance: 0.00853986601633272
    document: fish
  - distance: 0.12712843905603044
    document: dog
```

9.3.4.2.3 示例代码片段

你可以参考以下示例代码片段来完成自己的应用程序开发。

创建向量表

连接到平凯数据库集群

```
import os
import dotenv

from peewee import Model, MySQLDatabase, SQL, TextField
from tidb_vector.peewee import VectorField

dotenv.load_dotenv()
```

```
#### Using `pymysql` as the driver.
connect_kwargs = {
    'ssl_verify_cert': True,
    'ssl_verify_identity': True,
}

#### Using `mysqlclient` as the driver.
#### connect_kwargs = {
####     'ssl_mode': 'VERIFY_IDENTITY',
####     'ssl': {
####         # Root certificate default path.
####         'ca': os.environ.get('TIDB_CA_PATH', '/path/to/ca.pem'),
####     },
#### }

db = MySQLDatabase(
    database=os.environ.get('TIDB_DATABASE', 'test'),
    user=os.environ.get('TIDB_USERNAME', 'root'),
    password=os.environ.get('TIDB_PASSWORD', ''),
    host=os.environ.get('TIDB_HOST', 'localhost'),
    port=int(os.environ.get('TIDB_PORT', '4000')),
    **connect_kwargs,
)
```

定义向量列

创建一个表格，其中包含一个向量数据类型的 embedding 列，用于存储三维向量。

```
class Document(Model):
    class Meta:
        database = db
        table_name = 'peewee_demo_documents'

    content = TextField()
    embedding = VectorField(3)
```

存储包含向量的 document

```
Document.create(content='dog', embedding=[1, 2, 1])
Document.create(content='fish', embedding=[1, 2, 4])
Document.create(content='tree', embedding=[1, 0, 0])
```

搜索近邻向量

可以选择使用余弦距离 (CosineDistance) 函数，查询与向量 [1, 2, 3] 语义最接近的前 3 个 document。

```
distance = Document.embedding.cosine_distance([1, 2, 3]).alias('distance')
results = Document.select(Document, distance).order_by(distance).limit(3)
```

搜索一定距离内的向量

可以选择使用余弦距离 (CosineDistance) 函数，查询与向量 [1, 2, 3] 的余弦距离小于 0.2 的向量。

```
distance_expression = Document.embedding.cosine_distance([1, 2, 3])
distance = distance_expression.alias('distance')
results = Document.select(Document, distance).where(distance_expression < 0.2).order_by(distance).limit(3)
```

9.3.4.2.4 另请参阅

- 向量数据类型
- 向量搜索索引

9.3.4.3 在 Django ORM 中使用平凯数据库向量搜索

本文档将展示如何使用 Django ORM 与 TiDB 向量搜索进行交互，以及如何存储向量和执行向量搜索查询。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.3.4.3.1 前置需求

为了能够顺利完成本文中的操作，你需要提前：

- 在你的机器上安装 Python 3.8 或更高版本
- 在你的机器上安装 Git

- 准备一个 TiDB 集群

如果你还没有 TiDB 集群，可以按照以下任一种方式创建：

参考部署本地测试集群或部署正式集群，创建本地集群。

9.3.4.3.2 运行示例应用程序

你可以通过以下步骤快速了解如何在 Django ORM 中使用 TiDB 向量搜索。

第 1 步：克隆示例代码仓库

将 tidb-vector-python 仓库克隆到本地：

```
git clone https://github.com/pingcap/tidb-vector-python.git
```

第 2 步：创建虚拟环境

为你的项目创建虚拟环境：

```
cd tidb-vector-python/examples/orm-django-quickstart
python3 -m venv .venv
source .venv/bin/activate
```

第 3 步：安装所需的依赖

安装示例项目所需的依赖：

```
pip install -r requirements.txt
```

你也可以直接为项目安装以下依赖项：

```
pip install Django django-tidb mysqlclient numpy python-dotenv
```

如果遇到 mysqlclient 安装问题，请参阅 mysqlclient 官方文档。

什么是 django-tidb?

django-tidb 是一个为 Django 提供的 TiDB 适配器。通过该适配器，Django ORM 实现了对 TiDB 特有的功能（如，向量搜索）的支持，并解决了 TiDB 和 Django 之间的兼容性问题。

安装 django-tidb 时，选择与你的 Django 版本匹配的版本。例如，如果你使用的是 django==4.2.*，则应安装 django-tidb==4.2.*，其中 minor 版本号不需要完全相同。建议使用最新的 minor 版本。

更多信息，请参考 django-tidb 仓库。

第 4 步：配置环境变量

对于本地部署的 TiDB，请在 Python 项目的根目录下新建一个 .env 文件，将以下内容复制到 .env 文件中，并根据集群的连接参数修改环境变量值为 TiDB 实际对应的值：

```
TIDB_HOST=127.0.0.1
TIDB_PORT=4000
TIDB_USERNAME=root
TIDB_PASSWORD=
TIDB_DATABASE=test
```

如果你在本机运行 TiDB，TIDB_HOST 默认为 127.0.0.1。TIDB_PASSWORD 初始密码为空，若你是第一次启动集群，则无需带上此字段。

以下为各参数的解释：

- TIDB_HOST：TiDB 集群的主机号。
- TIDB_PORT：TiDB 集群的端口号。
- TIDB_USERNAME：连接 TiDB 集群的用户名。
- TIDB_PASSWORD：连接 TiDB 集群的密码。
- TIDB_DATABASE：要连接的数据库名称。

第 5 步：运行示例应用程序

迁移数据库模式：

```
python manage.py migrate
```

运行 Django 开发服务器：

```
python manage.py runserver
```

打开浏览器，访问 http://127.0.0.1:8000 查看该示例程序的可视化界面。以下为该程序可用的 API 路径：

API 路径	描述
POST: /insert_documents	插入含有向量的 document。
GET: /get_nearest_neighbors_documents	获取距离最近的 3 个 document。
GET: /get_documents_within_distance	获取处于给定距离内的所有 document。

9.3.4.3.3 示例代码片段

你可以参考以下示例代码片段来完成自己的应用程序开发。

连接到平凯数据库集群

打开 sample_project/settings.py 文件，添加以下配置：

```
dotenv.load_dotenv()

DATABASES = {
    "default": {
        # https://github.com/pingcap/django-tidb
        "ENGINE": "django_tidb",
        "HOST": os.environ.get("TIDB_HOST", "127.0.0.1"),
        "PORT": int(os.environ.get("TIDB_PORT", 4000)),
        "USER": os.environ.get("TIDB_USERNAME", "root"),
        "PASSWORD": os.environ.get("TIDB_PASSWORD", ""),
        "NAME": os.environ.get("TIDB_DATABASE", "test"),
        "OPTIONS": {
            "charset": "utf8mb4",
        },
    }
}

TIDB_CA_PATH = os.environ.get("TIDB_CA_PATH", "")
if TIDB_CA_PATH:
    DATABASES["default"]["OPTIONS"]["ssl_mode"] = "VERIFY_IDENTITY"
    DATABASES["default"]["OPTIONS"]["ssl"] = {
        "ca": TIDB_CA_PATH,
    }
```

你可以在项目的根目录下创建一个 .env 文件，在文件中添加环境变量 TIDB_HOST、TIDB_PORT、TIDB_USERNAME、TIDB_PASSWORD、TIDB_DATABASE 和 TIDB_CA_PATH，并根据你的 TiDB 集群的实际值来设置这些变量的值。

创建向量表

定义向量列

tidb-django 提供了一个 VectorField，可以在表中用来表示和存储向量类型。

创建一个表，其中包含一个向量数据类型的 embedding 列，用于存储三维向量。

```
class Document(models.Model):
  content = models.TextField()
  embedding = VectorField(dimensions=3)
```

存储包含向量的 document

```
Document.objects.create(content="dog", embedding=[1, 2, 1])
Document.objects.create(content="fish", embedding=[1, 2, 4])
Document.objects.create(content="tree", embedding=[1, 0, 0])
```

搜索近邻向量

TiDB 向量支持以下距离函数：

- L1Distance
- L2Distance
- CosineDistance
- NegativeInnerProduct

可以选择使用余弦距离 (CosineDistance) 函数，查询与向量 [1, 2, 3] 语义最接近的前 3 个 document。

```
results = Document.objects.annotate(
  distance=CosineDistance('embedding', [1, 2, 3])
).order_by('distance')[:3]
```

搜索一定距离内的向量

可以选择使用余弦距离 (CosineDistance) 函数，查询与向量 [1, 2, 3] 的余弦距离小于 0.2 的向量。

```
results = Document.objects.annotate(
  distance=CosineDistance('embedding', [1, 2, 3])
).filter(distance__lt=0.2).order_by('distance')[:3]
```

9.3.4.3.4 另请参阅

- 向量数据类型
- 向量搜索索引

9.4 优化向量搜索性能

在 TiDB 中，你可以通过向量搜索功能进行近似最近邻（Approximate Nearest Neighbor，简称 ANN）搜索，查找与给定的图像、文档等相似的结果。为了提升查询性能，请参考以下最佳实践。

> **警告：**
>
> 向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.4.1 为向量列添加向量搜索索引

向量搜索索引可显著提高向量搜索查询的性能，通常能提高 10 倍或更多，而召回率仅略有下降。

9.4.2 确保向量索引已完全构建

当插入大批量向量数据时，可能会有部分数据处于 Delta 层等待后续的持久化，这一部分的数据会在持久化后才会开始构建向量索引。在向量搜索索引完全构建好后，向量搜索性能才能达到最佳水平。要查看索引构建进度，可参阅查看索引构建进度。

9.4.3 减少向量维数或缩短嵌入时间

随着向量维度增加，向量搜索索引和查询的计算复杂度会显著增加，因为需要进行更多的浮点数比较运算。

为了优化性能，可以考虑尽可能地减少向量的维数。这通常需要切换到另一种嵌入模型。在切换模型时，你需要评估改变嵌入模型对向量查询准确性的影响。

一些嵌入模型，如 OpenAI text-embedding-3-large，支持缩短向量嵌入，即在不丢失向量表示的概念特征的情况下，从向量序列末尾移除一些数字。你也可以使用这种嵌入模型来减少向量维数。

9.4.4 在结果输出中排除向量列

向量嵌入数据通常很大，而且只在搜索过程中使用。通过从查询结果中排除向量列，可以显著减少 TiDB 服务器和 SQL 客户端之间传输的数据量，从而提高查询性能。

要从结果输出中排除向量列，请在 SELECT 语句中明确指定需要检索的列，而不是使用 SELECT * 检索所有列。

9.4.5 预热索引

当访问一个从未被使用过或长时间未被使用过的索引（冷访问）时，TiDB 需要从云存储或磁盘（而不是内存）加载整个索引。这个过程需要一定的时间，往往会导致较高的查询延迟。此外，如果集群长时间（比如数小时）内没有进行 SQL 查询，计算资源就会被回收，这样下次访问时就会变成冷访问。

要避免这种查询延迟，可在实际工作负载前，使用类似的向量搜索查询对索引进行预热。

9.5 向量搜索限制

本文档介绍 TiDB 向量搜索的已知限制。

警告：

向量搜索目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化。

9.5.1 向量数据类型限制

- 向量最大支持 16383 维。
- 向量数据中不支持 NaN、Infinity 和 -Infinity 浮点数。
- 向量列不能作为主键或者主键的一部分。
- 向量列不能作为唯一索引或者唯一索引的一部分。
- 向量列不能作为分区键或者分区键的一部分。
- 向量数据类型不支持存储双精度浮点数。当向 TiDB 中的向量列插入或存储双精度浮点数时，TiDB 会将这些双精度浮点数自动转换为单精度浮点数。
- 目前 TiDB 不支持将向量类型的列修改为其他数据类型（如 JSON、VARCHAR 等）。

9.5.2 向量搜索索引限制

参考向量搜索索引 - 使用限制。

9.5.3 工具兼容性

- 确保使用 BR v7.1.8 及以上版本进行备份与恢复。不支持将带有向量数据类型的表恢复至 v7.1.8 之前的 TiDB 集群。
- TiDB Data Migration (DM) 不支持迁移或同步 MySQL 9.0 的向量数据类型到 TiDB。
- TiCDC 在同步向量数据到不支持向量数据类型的下游时会修改数据类型。详情参考向量数据类型兼容性说明。

9.5.4 反馈

我们非常重视您的反馈意见。如果在开发的过程中遇到问题，可以在 AskTUG 上进行提问，寻求帮助。

10 事务

10.1 事务概览

TiDB 支持完整的分布式事务，提供乐观事务与悲观事务（TiDB 3.0 中引入）两种事务模型。本文主要介绍涉及到事务的语句、乐观事务和悲观事务、事务的隔离级别，以及乐观事务应用端重试和错误处理。

10.1.1 拓展学习视频

TiDB 特有功能与事务控制 - TiDB v6：了解可用于应用程序的 TiDB 独特功能，如 AUTO_RANDOM 及 AUTO_INCREMENT 特别注意事项、全局临时表、如何使用 TiFlash 启用 HTAP 以及放置策略等。

10.1.2 通用语句

本章介绍在 TiDB 中如何使用事务。 将使用下面的示例来演示一个简单事务的控制流程：

Bob 要给 Alice 转账 20 元钱，当中至少包括两个操作：

- Bob 账户减少 20 元。
- Alice 账户增加 20 元。

事务可以确保以上两个操作要么都执行成功，要么都执行失败，不会出现钱平白消失或出现的情况。

使用 bookshop 数据库中的 users 表，在表中插入一些示例数据：

```
INSERT INTO users (id, nickname, balance)
  VALUES (2, 'Bob', 200);
INSERT INTO users (id, nickname, balance)
  VALUES (1, 'Alice', 100);
```

现在，运行以下事务并解释每个语句的含义：

```
BEGIN;
  UPDATE users SET balance = balance - 20 WHERE nickname = 'Bob';
```

```
  UPDATE users SET balance = balance + 20 WHERE nickname= 'Alice';
COMMIT;
```

上述事务成功后，表应如下所示：

```
+----+--------------+---------+
| id | account_name | balance |
+----+--------------+---------+
|  1 | Alice        |  120.00 |
|  2 | Bob          |  180.00 |
+----+--------------+---------+
```

10.1.2.1 开启事务

要显式地开启一个新事务，既可以使用 BEGIN 语句，也可以使用 START TRANSACTION 语句，两者效果相同。语法：

```
BEGIN;
```

```
START TRANSACTION;
```

TiDB 的默认事务模式是悲观事务，你也可以明确指定开启乐观事务：

```
BEGIN OPTIMISTIC;
```

开启悲观事务：

```
BEGIN PESSIMISTIC;
```

如果执行以上语句时，当前 Session 正处于一个事务的中间过程，那么系统会先自动提交当前事务，再开启一个新的事务。

10.1.2.2 提交事务

COMMIT 语句用于提交 TiDB 在当前事务中进行的所有修改。语法：

```
COMMIT;
```

启用乐观事务前，请确保应用程序可正确处理 COMMIT 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

10.1.2.3 回滚事务

ROLLBACK 语句用于回滚并撤销当前事务的所有修改。语法：

```
ROLLBACK;
```

回到之前转账示例，使用 ROLLBACK 回滚整个事务之后，Alice 和 Bob 的余额都未发生改变，当前事务的所有修改一起被取消。

```
TRUNCATE TABLE `users`;

INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (1, 'Alice', 100), (2, 'Bob', 200);

SELECT * FROM `users`;
+----+-------------+---------+
| id | nickname    | balance |
+----+-------------+---------+
|  1 | Alice       |  100.00 |
|  2 | Bob         |  200.00 |
+----+-------------+---------+

BEGIN;
  UPDATE `users` SET `balance` = `balance` - 20 WHERE `nickname`='Bob';
  UPDATE `users` SET `balance` = `balance` + 20 WHERE `nickname`='Alice';
ROLLBACK;

SELECT * FROM `users`;
+----+-------------+---------+
| id | nickname    | balance |
+----+-------------+---------+
|  1 | Alice       |  100.00 |
|  2 | Bob         |  200.00 |
+----+-------------+---------+
```

如果客户端连接中止或关闭，也会自动回滚该事务。

10.1.3 事务隔离级别

事务隔离级别是数据库事务处理的基础，**ACID** 中的 **“I”**，即 Isolation，指的就是事务的隔离性。

SQL-92 标准定义了 4 种隔离级别：读未提交 (READ UNCOMMITTED)、读已提交 (READ COMMITTED)、可重复读 (REPEATABLE READ)、串行化 (SERIALIZABLE)。详见下表：

Isolation Level	Dirty Write	Dirty Read	Fuzzy Read	Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not possible	Possible	Possible
REPEATABLE READ	Not Possible	Not possible	Not possible	Possible
SERIALIZABLE	Not Possible	Not possible	Not possible	Not possible

TiDB 语法上支持设置 READ COMMITTED 和 REPEATABLE READ 两种隔离级别：

```
mysql> SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
ERROR 8048 (HY000): The isolation level 'READ-UNCOMMITTED' is not supported. Set t
idb_skip_isolation_level_check=1 to skip this error
mysql> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.00 sec)

mysql> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
ERROR 8048 (HY000): The isolation level 'SERIALIZABLE' is not supported. Set tidb_skip
_isolation_level_check=1 to skip this error
```

TiDB 实现了快照隔离 (Snapshot Isolation, SI) 级别的一致性。为与 MySQL 保持一致，又称其为“可重复读”。该隔离级别不同于 ANSI 可重复读隔离级别 和 MySQL 可重复读隔离级别。更多细节请阅读 TiDB 事务隔离级别。

10.2 乐观事务和悲观事务

简单的讲，乐观事务模型就是直接提交，遇到冲突就回滚，悲观事务模型就是在真正提交事务前，先尝试对需要修改的资源上锁，只有在确保事务一定能够执行成功后，才开始提交。

对于乐观事务模型来说，比较适合冲突率不高的场景，因为直接提交大概率会成功，冲突是小概率事件，但是一旦遇到事务冲突，回滚的代价会比较大。

悲观事务的好处是对于冲突率高的场景，提前上锁的代价小于事后回滚的代价，而且还能以比较低的代价解决多个并发事务互相冲突导致谁也成功不了的场景。不过悲观事务在冲突率不高的场景并没有乐观事务处理高效。

从应用端实现的复杂度而言，悲观事务更直观，更容易实现。而乐观事务需要复杂的应用端重试机制来保证。

下面用 bookshop 数据库中的表实现一个购书的例子来演示乐观事务和悲观事务的区别以及优缺点。购书流程主要包括：

1. 更新库存数量
2. 创建订单
3. 付款

这三个操作需要保证全部成功或者全部失败，并且在并发情况下要保证不超卖。

10.2.1 悲观事务

下面代码以悲观事务的方式，用两个线程模拟了两个用户并发买同一本书的过程，书店剩余 10 本，Bob 购买了 6 本，Alice 购买了 4 本。两个人几乎同一时间完成订单，最终，这本书的剩余库存为零。

当使用多个线程模拟多用户同时插入的情况时，需要使用一个线程安全的连接对象，这里使用 Java 当前较流行的连接池 HikariCP。

Golang 的 sql.DB 是并发安全的，无需引入外部包。

封装一个用于适配 TiDB 事务的工具包 util，编写以下代码备用：

```
package util

import (
    "context"
    "database/sql"
)
```

```
type TiDBSqlTx struct {
    *sql.Tx
    conn       *sql.Conn
    pessimistic bool
}

func TiDBSqlBegin(db *sql.DB, pessimistic bool) (*TiDBSqlTx, error) {
    ctx := context.Background()
    conn, err := db.Conn(ctx)
    if err != nil {
        return nil, err
    }
    if pessimistic {
        _, err = conn.ExecContext(ctx, "set @@tidb_txn_mode=?", "pessimistic")
    } else {
        _, err = conn.ExecContext(ctx, "set @@tidb_txn_mode=?", "optimistic")
    }
    if err != nil {
        return nil, err
    }
    tx, err := conn.BeginTx(ctx, nil)
    if err != nil {
        return nil, err
    }
    return &TiDBSqlTx{
        conn:       conn,
        Tx:         tx,
        pessimistic: pessimistic,
    }, nil
}

func (tx *TiDBSqlTx) Commit() error {
    defer tx.conn.Close()
    return tx.Tx.Commit()
}

func (tx *TiDBSqlTx) Rollback() error {
    defer tx.conn.Close()
    return tx.Tx.Rollback()
}
```

使用 Python 的 mysqlclient Driver 开启多个连接对象进行交互，线程之间不共享连接，以保证其线程安全。

10.2.1.1 1. 编写悲观事务示例

配置文件

在 Java 中，如果你使用 Maven 作为包管理，在 pom.xml 中的 <dependencies> 节点中，加入以下依赖来引入 HikariCP，同时设定打包目标，及 JAR 包启动的主类，完整的 pom.xml 如下所示:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.pingcap</groupId>
  <artifactId>plain-java-txn</artifactId>
  <version>0.0.1</version>

  <name>plain-java-jdbc</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>
```

```
  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.28</version>
  </dependency>

  <dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>5.0.1</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>3.3.0</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
          <manifest>
            <mainClass>com.pingcap.txn.TxnExample</mainClass>
          </manifest>
        </archive>

      </configuration>
      <executions>
        <execution>
          <id>make-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
```

```
  </build>

</project>
```

代码

随后编写代码：

```
package com.pingcap.txn;

import com.zaxxer.hikari.HikariDataSource;

import java.math.BigDecimal;
import java.sql.*;
import java.util.Arrays;
import java.util.concurrent.*;

public class TxnExample {
    public static void main(String[] args) throws SQLException, InterruptedException {
        System.out.println(Arrays.toString(args));
        int aliceQuantity = 0;
        int bobQuantity = 0;

        for (String arg: args) {
            if (arg.startsWith("ALICE_NUM")) {
                aliceQuantity = Integer.parseInt(arg.replace("ALICE_NUM=", ""));
            }

            if (arg.startsWith("BOB_NUM")) {
                bobQuantity = Integer.parseInt(arg.replace("BOB_NUM=", ""));
            }
        }

        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:mysql://localhost:4000/bookshop?useServerPrepStmts=true&ca
chePrepStmts=true");
        ds.setUsername("root");
        ds.setPassword("");

        // prepare data
        Connection connection = ds.getConnection();
        createBook(connection, 1L, "Designing Data-Intensive Application", "Science & Tech
nology",
```

```
            Timestamp.valueOf("2018-09-01 00:00:00"), new BigDecimal(100), 10);
        createUser(connection, 1L, "Bob", new BigDecimal(10000));
        createUser(connection, 2L, "Alice", new BigDecimal(10000));

        CountDownLatch countDownLatch = new CountDownLatch(2);
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        final int finalBobQuantity = bobQuantity;
        threadPool.execute(() -> {
            buy(ds, 1, 1000L, 1L, 1L, finalBobQuantity);
            countDownLatch.countDown();
        });
        final int finalAliceQuantity = aliceQuantity;
        threadPool.execute(() -> {
            buy(ds, 2, 1001L, 1L, 2L, finalAliceQuantity);
            countDownLatch.countDown();
        });

        countDownLatch.await(5, TimeUnit.SECONDS);
    }

    public static void createUser(Connection connection, Long id, String nickname, BigDec
imal balance) throws SQLException  {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (?, ?, ?)");
        insert.setLong(1, id);
        insert.setString(2, nickname);
        insert.setBigDecimal(3, balance);
        insert.executeUpdate();
    }

    public static void createBook(Connection connection, Long id, String title, String type,
Timestamp publishedAt, BigDecimal price, Integer stock) throws SQLException {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `books` (`id`, `title`, `type`, `published_at`, `price`, `stock`) values
(?, ?, ?, ?, ?, ?)");
        insert.setLong(1, id);
        insert.setString(2, title);
        insert.setString(3, type);
        insert.setTimestamp(4, publishedAt);
        insert.setBigDecimal(5, price);
        insert.setInt(6, stock);
```

```
        insert.executeUpdate();
    }

    public static void buy (HikariDataSource ds, Integer threadID,
                            Long orderID, Long bookID, Long userID, Integer quantity) {
        String txnComment = "/* txn " + threadID + " */ ";

        try (Connection connection = ds.getConnection()) {
            try {
                connection.setAutoCommit(false);
                connection.createStatement().executeUpdate(txnComment + "begin pessimisti
c");

                // waiting for other thread ran the 'begin pessimistic' statement
                TimeUnit.SECONDS.sleep(1);

                BigDecimal price = null;

                // read price of book
                PreparedStatement selectBook = connection.prepareStatement(txnComment +
 "select price from books where id = ? for update");
                selectBook.setLong(1, bookID);
                ResultSet res = selectBook.executeQuery();
                if (!res.next()) {
                    throw new RuntimeException("book not exist");
                } else {
                    price = res.getBigDecimal("price");
                }

                // update book
                String updateBookSQL = "update `books` set stock = stock - ? where id = ? an
d stock - ? >= 0";
                PreparedStatement updateBook = connection.prepareStatement(txnComment
+ updateBookSQL);
                updateBook.setInt(1, quantity);
                updateBook.setLong(2, bookID);
                updateBook.setInt(3, quantity);
                int affectedRows = updateBook.executeUpdate();

                if (affectedRows == 0) {
                    // stock not enough, rollback
                    connection.createStatement().executeUpdate(txnComment + "rollback");
                    return;
```

```
            }

            // insert order
            String insertOrderSQL = "insert into `orders` (`id`, `book_id`, `user_id`, `quality`)
values (?, ?, ?, ?)";
            PreparedStatement insertOrder = connection.prepareStatement(txnComment
+ insertOrderSQL);
            insertOrder.setLong(1, orderID);
            insertOrder.setLong(2, bookID);
            insertOrder.setLong(3, userID);
            insertOrder.setInt(4, quantity);
            insertOrder.executeUpdate();

            // update user
            String updateUserSQL = "update `users` set `balance` = `balance` - ? where id
= ?";
            PreparedStatement updateUser = connection.prepareStatement(txnComment
+ updateUserSQL);
            updateUser.setBigDecimal(1, price.multiply(new BigDecimal(quantity)));
            updateUser.setLong(2, userID);
            updateUser.executeUpdate();

            connection.createStatement().executeUpdate(txnComment + "commit");
        } catch (Exception e) {
            connection.createStatement().executeUpdate(txnComment + "rollback");
            e.printStackTrace();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```

首先编写一个封装了所需的数据库操作的 helper.go 文件：

```
package main

import (
  "context"
  "database/sql"
  "fmt"
  "time"
```

```
    "github.com/go-sql-driver/mysql"
    "github.com/pingcap-inc/tidb-example-golang/util"
    "github.com/shopspring/decimal"
)

type TxnFunc func(txn *util.TiDBSqlTx) error

const (
    ErrWriteConflict      = 9007 // Transactions in TiKV encounter write conflicts.
    ErrInfoSchemaChanged  = 8028 // table schema changes
    ErrForUpdateCantRetry = 8002 // "SELECT FOR UPDATE" commit conflict
    ErrTxnRetryable       = 8022 // The transaction commit fails and has been rolled back
)

const retryTimes = 5

var retryErrorCodeSet = map[uint16]interface{}{
    ErrWriteConflict:      nil,
    ErrInfoSchemaChanged:  nil,
    ErrForUpdateCantRetry: nil,
    ErrTxnRetryable:       nil,
}

func runTxn(db *sql.DB, optimistic bool, optimisticRetryTimes int, txnFunc TxnFunc) {
    txn, err := util.TiDBSqlBegin(db, !optimistic)
    if err != nil {
        panic(err)
    }

    err = txnFunc(txn)
    if err != nil {
        txn.Rollback()
        if mysqlErr, ok := err.(*mysql.MySQLError); ok && optimistic && optimisticRetryTim
es != 0 {
            if _, retryableError := retryErrorCodeSet[mysqlErr.Number]; retryableError {
                fmt.Printf("[runTxn] got a retryable error, rest time: %d\n", optimisticRetryTimes
-1)
                runTxn(db, optimistic, optimisticRetryTimes-1, txnFunc)
                return
            }
        }

        fmt.Printf("[runTxn] got an error, rollback: %+v\n", err)
```

```
    } else {
        err = txn.Commit()
        if mysqlErr, ok := err.(*mysql.MySQLError); ok && optimistic && optimisticRetryTim
es != 0 {
            if _, retryableError := retryErrorCodeSet[mysqlErr.Number]; retryableError {
                fmt.Printf("[runTxn] got a retryable error, rest time: %d\n", optimisticRetryTimes
-1)
                runTxn(db, optimistic, optimisticRetryTimes-1, txnFunc)
                return
            }
        }

        if err == nil {
            fmt.Println("[runTxn] commit success")
        }
    }
}

func prepareData(db *sql.DB, optimistic bool) {
    runTxn(db, optimistic, retryTimes, func(txn *util.TiDBSqlTx) error {
        publishedAt, err := time.Parse("2006-01-02 15:04:05", "2018-09-01 00:00:00")
        if err != nil {
            return err
        }

        if err = createBook(txn, 1, "Designing Data-Intensive Application",
            "Science & Technology", publishedAt, decimal.NewFromInt(100), 10); err != nil {
            return err
        }

        if err = createUser(txn, 1, "Bob", decimal.NewFromInt(10000)); err != nil {
            return err
        }

        if err = createUser(txn, 2, "Alice", decimal.NewFromInt(10000)); err != nil {
            return err
        }

        return nil
    })
}

func buyPessimistic(db *sql.DB, goroutineID, orderID, bookID, userID, amount int) {
```

```
    txnComment := fmt.Sprintf("/* txn %d */ ", goroutineID)
    if goroutineID != 1 {
        txnComment = "\t" + txnComment
    }

    fmt.Printf("\nuser %d try to buy %d books(id: %d)\n", userID, amount, bookID)

    runTxn(db, false, retryTimes, func(txn *util.TiDBSqlTx) error {
        time.Sleep(time.Second)

        // read the price of book
        selectBookForUpdate := "select `price` from books where id = ? for update"
        bookRows, err := txn.Query(selectBookForUpdate, bookID)
        if err != nil {
            return err
        }
        fmt.Println(txnComment + selectBookForUpdate + " successful")
        defer bookRows.Close()

        price := decimal.NewFromInt(0)
        if bookRows.Next() {
            err = bookRows.Scan(&price)
            if err != nil {
                return err
            }
        } else {
            return fmt.Errorf("book ID not exist")
        }
        bookRows.Close()

        // update book
        updateStock := "update `books` set stock = stock - ? where id = ? and stock - ? >=
0"
        result, err := txn.Exec(updateStock, amount, bookID, amount)
        if err != nil {
            return err
        }
        fmt.Println(txnComment + updateStock + " successful")

        affected, err := result.RowsAffected()
        if err != nil {
            return err
        }
```

```
        if affected == 0 {
            return fmt.Errorf("stock not enough, rollback")
        }

        // insert order
        insertOrder := "insert into `orders` (`id`, `book_id`, `user_id`, `quality`) values (?, ?, ?, ?)
"
        if _, err := txn.Exec(insertOrder,
            orderID, bookID, userID, amount); err != nil {
            return err
        }
        fmt.Println(txnComment + insertOrder + " successful")

        // update user
        updateUser := "update `users` set `balance` = `balance` - ? where id = ?"
        if _, err := txn.Exec(updateUser,
            price.Mul(decimal.NewFromInt(int64(amount))), userID); err != nil {
            return err
        }
        fmt.Println(txnComment + updateUser + " successful")

        return nil
    })
}

func buyOptimistic(db *sql.DB, goroutineID, orderID, bookID, userID, amount int) {
    txnComment := fmt.Sprintf("/* txn %d */ ", goroutineID)
    if goroutineID != 1 {
        txnComment = "\t" + txnComment
    }

    fmt.Printf("\nuser %d try to buy %d books(id: %d)\n", userID, amount, bookID)

    runTxn(db, true, retryTimes, func(txn *util.TiDBSqlTx) error {
        time.Sleep(time.Second)

        // read the price and stock of book
        selectBookForUpdate := "select `price`, `stock` from books where id = ? for update"
        bookRows, err := txn.Query(selectBookForUpdate, bookID)
        if err != nil {
            return err
        }
```

```
    fmt.Println(txnComment + selectBookForUpdate + " successful")
    defer bookRows.Close()

    price, stock := decimal.NewFromInt(0), 0
    if bookRows.Next() {
        err = bookRows.Scan(&price, &stock)
        if err != nil {
            return err
        }
    } else {
        return fmt.Errorf("book ID not exist")
    }
    bookRows.Close()

    if stock < amount {
        return fmt.Errorf("book not enough")
    }

    // update book
    updateStock := "update `books` set stock = stock - ? where id = ? and stock - ? >=
0"
    result, err := txn.Exec(updateStock, amount, bookID, amount)
    if err != nil {
        return err
    }
    fmt.Println(txnComment + updateStock + " successful")

    affected, err := result.RowsAffected()
    if err != nil {
        return err
    }

    if affected == 0 {
        return fmt.Errorf("stock not enough, rollback")
    }

    // insert order
    insertOrder := "insert into `orders` (`id`, `book_id`, `user_id`, `quality`) values (?, ?, ?, ?)
"
    if _, err := txn.Exec(insertOrder,
        orderID, bookID, userID, amount); err != nil {
        return err
    }
```

```
        fmt.Println(txnComment + insertOrder + " successful")

        // update user
        updateUser := "update `users` set `balance` = `balance` - ? where id = ?"
        if _, err := txn.Exec(updateUser,
            price.Mul(decimal.NewFromInt(int64(amount))), userID); err != nil {
            return err
        }
        fmt.Println(txnComment + updateUser + " successful")

        return nil
    })
}

func createBook(txn *util.TiDBSqlTx, id int, title, bookType string,
    publishedAt time.Time, price decimal.Decimal, stock int) error {
    _, err := txn.ExecContext(context.Background(),
        "INSERT INTO `books` (`id`, `title`, `type`, `published_at`, `price`, `stock`) values (?, ?, ?,
 ?, ?, ?)",
        id, title, bookType, publishedAt, price, stock)
    return err
}

func createUser(txn *util.TiDBSqlTx, id int, nickname string, balance decimal.Decimal) err
or {
    _, err := txn.ExecContext(context.Background(),
        "INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (?, ?, ?)",
        id, nickname, balance)
    return err
}
```

再编写一个包含 main 函数的 txn.go 来调用 helper.go，同时处理传入的命令行参数：

```
package main

import (
    "database/sql"
    "flag"
    "fmt"
    "sync"
)
```

```
func main() {
    optimistic, alice, bob := parseParams()

    openDB("mysql", "root:@tcp(127.0.0.1:4000)/bookshop?charset=utf8mb4", func(db *s
ql.DB) {
        prepareData(db, optimistic)
        buy(db, optimistic, alice, bob)
    })
}

func buy(db *sql.DB, optimistic bool, alice, bob int) {
    buyFunc := buyOptimistic
    if !optimistic {
        buyFunc = buyPessimistic
    }

    wg := sync.WaitGroup{}
    wg.Add(1)
    go func() {
        defer wg.Done()
        buyFunc(db, 1, 1000, 1, 1, bob)
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        buyFunc(db, 2, 1001, 1, 2, alice)
    }()

    wg.Wait()
}

func openDB(driverName, dataSourceName string, runnable func(db *sql.DB)) {
    db, err := sql.Open(driverName, dataSourceName)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    runnable(db)
}

func parseParams() (optimistic bool, alice, bob int) {
```

```
    flag.BoolVar(&optimistic, "o", false, "transaction is optimistic")
    flag.IntVar(&alice, "a", 4, "Alice bought num")
    flag.IntVar(&bob, "b", 6, "Bob bought num")

    flag.Parse()

    fmt.Println(optimistic, alice, bob)

    return optimistic, alice, bob
}
```

Golang 的例子中，已经包含乐观事务。

```
import time

import MySQLdb
import os
import datetime
from threading import Thread

REPEATABLE_ERROR_CODE_SET = {
    9007,  # Transactions in TiKV encounter write conflicts.
    8028,  # table schema changes
    8002,  # "SELECT FOR UPDATE" commit conflict
    8022   # The transaction commit fails and has been rolled back
}


def create_connection():
    return MySQLdb.connect(
        host="127.0.0.1",
        port=4000,
        user="root",
        password="",
        database="bookshop",
        autocommit=False
    )


def prepare_data() -> None:
    connection = create_connection()
    with connection:
        with connection.cursor() as cursor:
```

```
        cursor.execute("INSERT INTO `books` (`id`, `title`, `type`, `published_at`, `price`, `sto
ck`) "
                       "values (%s, %s, %s, %s, %s, %s)",
                       (1, "Designing Data-Intensive Application", "Science & Technology",
                        datetime.datetime(2018, 9, 1), 100, 10))

        cursor.executemany("INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (%
s, %s, %s)",
                           [(1, "Bob", 10000), (2, "ALICE", 10000)])
        connection.commit()


def buy_optimistic(thread_id: int, order_id: int, book_id: int, user_id: int, amount: int,
                   optimistic_retry_times: int = 5) -> None:
    connection = create_connection()

    txn_log_header = f"/* txn {thread_id} */"
    if thread_id != 1:
        txn_log_header = "\t" + txn_log_header

    with connection:
        with connection.cursor() as cursor:
            cursor.execute("BEGIN OPTIMISTIC")
            print(f'{txn_log_header} BEGIN OPTIMISTIC')
            time.sleep(1)

            try:
                # read the price of book
                select_book_for_update = "SELECT `price`, `stock` FROM books WHERE id = %s
FOR UPDATE"
                cursor.execute(select_book_for_update, (book_id,))
                book = cursor.fetchone()
                if book is None:
                    raise Exception("book_id not exist")
                price, stock = book
                print(f'{txn_log_header} {select_book_for_update} successful')

                if stock < amount:
                    raise Exception("book not enough, rollback")

                # update book
                update_stock = "update `books` set stock = stock - %s where id = %s and stoc
k - %s >= 0"
```

```
            rows_affected = cursor.execute(update_stock, (amount, book_id, amount))
            print(f'{txn_log_header} {update_stock} successful')

            if rows_affected == 0:
                raise Exception("stock not enough, rollback")

            # insert order
            insert_order = "insert into `orders` (`id`, `book_id`, `user_id`, `quality`) values (%
s, %s, %s, %s)"
            cursor.execute(insert_order, (order_id, book_id, user_id, amount))
            print(f'{txn_log_header} {insert_order} successful')

            # update user
            update_user = "update `users` set `balance` = `balance` - %s where id = %s"
            cursor.execute(update_user, (amount * price, user_id))
            print(f'{txn_log_header} {update_user} successful')

        except Exception as err:
            connection.rollback()

            print(f'something went wrong: {err}')
        else:
            # important here! you need deal the Exception from the TiDB
            try:
                connection.commit()
            except MySQLdb.MySQLError as db_err:
                code, desc = db_err.args
                if code in REPEATABLE_ERROR_CODE_SET and optimistic_retry_times > 0:
                    print(f'retry, rest {optimistic_retry_times - 1} times, for {code} {desc}')
                    buy_optimistic(thread_id, order_id, book_id, user_id, amount, optimistic_re
try_times - 1)


def buy_pessimistic(thread_id: int, order_id: int, book_id: int, user_id: int, amount: int) ->
None:
    connection = create_connection()

    txn_log_header = f"/* txn {thread_id} */"
    if thread_id != 1:
        txn_log_header = "\t" + txn_log_header

    with connection:
        with connection.cursor() as cursor:
```

```
        cursor.execute("BEGIN PESSIMISTIC")
        print(f'{txn_log_header} BEGIN PESSIMISTIC')
        time.sleep(1)

        try:
            # read the price of book
            select_book_for_update = "SELECT `price` FROM books WHERE id = %s FOR UP
DATE"
            cursor.execute(select_book_for_update, (book_id,))
            book = cursor.fetchone()
            if book is None:
                raise Exception("book_id not exist")
            price = book[0]
            print(f'{txn_log_header} {select_book_for_update} successful')

            # update book
            update_stock = "update `books` set stock = stock - %s where id = %s and stoc
k - %s >= 0"
            rows_affected = cursor.execute(update_stock, (amount, book_id, amount))
            print(f'{txn_log_header} {update_stock} successful')

            if rows_affected == 0:
                raise Exception("stock not enough, rollback")

            # insert order
            insert_order = "insert into `orders` (`id`, `book_id`, `user_id`, `quality`) values (%
s, %s, %s, %s)"
            cursor.execute(insert_order, (order_id, book_id, user_id, amount))
            print(f'{txn_log_header} {insert_order} successful')

            # update user
            update_user = "update `users` set `balance` = `balance` - %s where id = %s"
            cursor.execute(update_user, (amount * price, user_id))
            print(f'{txn_log_header} {update_user} successful')

        except Exception as err:
            connection.rollback()
            print(f'something went wrong: {err}')
        else:
            connection.commit()


optimistic = os.environ.get('OPTIMISTIC')
```

```
alice = os.environ.get('ALICE')
bob = os.environ.get('BOB')

if not (optimistic and alice and bob):
    raise Exception("please use \"OPTIMISTIC=<is_optimistic> ALICE=<alice_num> "
                    "BOB=<bob_num> python3 txn_example.py\" to start this script")

prepare_data()

if bool(optimistic) is True:
    buy_func = buy_optimistic
else:
    buy_func = buy_pessimistic

bob_thread = Thread(target=buy_func, kwargs={
    "thread_id": 1, "order_id": 1000, "book_id": 1, "user_id": 1, "amount": int(bob)})
alice_thread = Thread(target=buy_func, kwargs={
    "thread_id": 2, "order_id": 1001, "book_id": 1, "user_id": 2, "amount": int(alice)})

bob_thread.start()
alice_thread.start()
bob_thread.join(timeout=10)
alice_thread.join(timeout=10)
```

Python 的例子中，已经包含乐观事务。

10.2.1.2 2. 运行不涉及超卖的例子

运行示例程序：

在 Java 中运行示例程序：

```
mvn clean package
java -jar target/plain-java-txn-0.0.1-jar-with-dependencies.jar ALICE_NUM=4 BOB_NUM
=6
```

在 Golang 中运行示例程序：

```
go build -o bin/txn
./bin/txn -a 4 -b 6
```

在 Python 中运行示例程序：

```
OPTIMISTIC=False ALICE=4 BOB=6 python3 txn_example.py
```

SQL 日志：

```
/* txn 1 */BEGIN PESSIMISTIC
   /* txn 2 */BEGIN PESSIMISTIC
   /* txn 2 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
   /* txn 2 */UPDATE `books` SET `stock` = `stock` - 4 WHERE `id` = 1 AND `stock` - 4 >
= 0
   /* txn 2 */INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1001, 1,
1, 4)
   /* txn 2 */UPDATE `users` SET `balance` = `balance` - 400.0 WHERE `id` = 2
   /* txn 2 */COMMIT
/* txn 1 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
/* txn 1 */UPDATE `books` SET `stock` = `stock` - 6 WHERE `id` = 1 AND `stock` - 6 >=
0
/* txn 1 */INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1000, 1, 1,
6)
/* txn 1 */UPDATE `users` SET `balance` = `balance` - 600.0 WHERE `id` = 1
/* txn 1 */COMMIT
```

最后，检验一下订单创建、用户余额扣减、图书库存扣减情况，都符合预期。

```
mysql> SELECT * FROM `books`;
+----+-----------------------------------+--------------------+--------------------+----
---+--------+
| id | title                             | type               | published_at       | stock | price  |
+----+-----------------------------------+--------------------+--------------------+----
---+--------+
|  1 | Designing Data-Intensive Application | Science & Technology | 2018-09-01 00:00:00
|     0 | 100.00 |
+----+-----------------------------------+--------------------+--------------------+----
---+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM orders;
+------+---------+---------+---------+---------------------+
| id   | book_id | user_id | quality | ordered_at          |
+------+---------+---------+---------+---------------------+
| 1000 |       1 |       1 |       6 | 2022-04-19 10:58:12 |
```

```
| 1001 |      1 |      1 |      4 | 2022-04-19 10:58:11 |
+------+--------+--------+--------+---------------------+
2 rows in set (0.01 sec)

mysql> SELECT * FROM users;
+----+---------+----------+
| id | balance | nickname |
+----+---------+----------+
|  1 | 9400.00 | Bob      |
|  2 | 9600.00 | Alice    |
+----+---------+----------+
2 rows in set (0.00 sec)
```

10.2.1.3 3. 运行防止超卖的例子

可以再把难度加大，如果图书的库存剩余 10 本，Bob 购买 7 本，Alice 购买 4 本，两人几乎同时下单，结果会是怎样？继续复用上个例子里的代码来解决这个需求，只不过把 Bob 购买数量从 6 改成 7：

运行示例程序：

在 Java 中运行示例程序：

```
mvn clean package
java -jar target/plain-java-txn-0.0.1-jar-with-dependencies.jar ALICE_NUM=4 BOB_NUM
=7
```

在 Golang 中运行示例程序：

```
go build -o bin/txn
./bin/txn -a 4 -b 7
```

在 Python 中运行示例程序：

```
OPTIMISTIC=False ALICE=4 BOB=7 python3 txn_example.py
```

```
/* txn 1 */BEGIN PESSIMISTIC
  /* txn 2 */BEGIN PESSIMISTIC
  /* txn 2 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
  /* txn 2 */UPDATE `books` SET `stock` = `stock` - 4 WHERE `id` = 1 AND `stock` - 4 >
```

```
= 0
  /* txn 2 */INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) values (1001, 1, 1,
4)
  /* txn 2 */UPDATE `users` SET `balance` = `balance` - 400.0 WHERE `id` = 2
  /* txn 2 */COMMIT
/* txn 1 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
/* txn 1 */UPDATE `books` SET `stock` = `stock` - 7 WHERE `id` = 1 AND `stock` - 7 >=
0
/* txn 1 */ROLLBACK
```

由于 txn 2 抢先获得锁资源，更新了 stock，txn 1 里面 affected_rows 返回值为 0，进入了 rollback 流程。

再检验一下订单创建、用户余额扣减、图书库存扣减情况。Alice 下单 4 本书成功，Bob 下单 7 本书失败，库存剩余 6 本符合预期。

```
mysql> SELECT * FROM books;
+----+-----------------------------------+--------------------+--------------------+----
---+--------+
| id | title                             | type               | published_at       | stock | price  |
+----+-----------------------------------+--------------------+--------------------+----
---+--------+
|  1 | Designing Data-Intensive Application | Science & Technology | 2018-09-01 00:00:00
|     6 | 100.00 |
+----+-----------------------------------+--------------------+--------------------+----
---+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM orders;
+------+---------+---------+---------+---------------------+
| id   | book_id | user_id | quality | ordered_at          |
+------+---------+---------+---------+---------------------+
| 1001 |       1 |       1 |       4 | 2022-04-19 11:03:03 |
+------+---------+---------+---------+---------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM users;
+----+----------+----------+
| id | balance  | nickname |
+----+----------+----------+
|  1 | 10000.00 | Bob      |
|  2 |  9600.00 | Alice    |
```

```
+----+----------+----------+
2 rows in set (0.01 sec)
```

10.2.2 乐观事务

下面代码以乐观事务的方式，用两个线程模拟了两个用户并发买同一本书的过程，和悲观事务的示例一样。书店剩余 10 本，Bob 购买了 6 本，Alice 购买了 4 本。两个人几乎同一时间完成订单，最终，这本书的剩余库存为零。

10.2.2.1 1. 编写乐观事务示例

使用 Java 编写乐观事务示例：

代码编写

```
package com.pingcap.txn.optimistic;

import com.zaxxer.hikari.HikariDataSource;

import java.math.BigDecimal;
import java.sql.*;
import java.util.Arrays;
import java.util.concurrent.*;

public class TxnExample {
    public static void main(String[] args) throws SQLException, InterruptedException {
        System.out.println(Arrays.toString(args));
        int aliceQuantity = 0;
        int bobQuantity = 0;

        for (String arg: args) {
            if (arg.startsWith("ALICE_NUM")) {
                aliceQuantity = Integer.parseInt(arg.replace("ALICE_NUM=", ""));
            }

            if (arg.startsWith("BOB_NUM")) {
                bobQuantity = Integer.parseInt(arg.replace("BOB_NUM=", ""));
            }
        }
```

```
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:mysql://localhost:4000/bookshop?useServerPrepStmts=true&cachePrepStmts=true");
        ds.setUsername("root");
        ds.setPassword("");

        // prepare data
        Connection connection = ds.getConnection();
        createBook(connection, 1L, "Designing Data-Intensive Application", "Science & Technology",
                Timestamp.valueOf("2018-09-01 00:00:00"), new BigDecimal(100), 10);
        createUser(connection, 1L, "Bob", new BigDecimal(10000));
        createUser(connection, 2L, "Alice", new BigDecimal(10000));

        CountDownLatch countDownLatch = new CountDownLatch(2);
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        final int finalBobQuantity = bobQuantity;
        threadPool.execute(() -> {
            buy(ds, 1, 1000L, 1L, 1L, finalBobQuantity, 5);
            countDownLatch.countDown();
        });
        final int finalAliceQuantity = aliceQuantity;
        threadPool.execute(() -> {
            buy(ds, 2, 1001L, 1L, 2L, finalAliceQuantity, 5);
            countDownLatch.countDown();
        });

        countDownLatch.await(5, TimeUnit.SECONDS);
    }

    public static void createUser(Connection connection, Long id, String nickname, BigDecimal balance) throws SQLException  {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (?, ?, ?)");
        insert.setLong(1, id);
        insert.setString(2, nickname);
        insert.setBigDecimal(3, balance);
        insert.executeUpdate();
    }

    public static void createBook(Connection connection, Long id, String title, String type, Timestamp publishedAt, BigDecimal price, Integer stock) throws SQLException {
```

```
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `books` (`id`, `title`, `type`, `published_at`, `price`, `stock`) values
(?, ?, ?, ?, ?, ?)");
        insert.setLong(1, id);
        insert.setString(2, title);
        insert.setString(3, type);
        insert.setTimestamp(4, publishedAt);
        insert.setBigDecimal(5, price);
        insert.setInt(6, stock);

        insert.executeUpdate();
    }

    public static void buy (HikariDataSource ds, Integer threadID, Long orderID, Long boo
kID,
                            Long userID, Integer quantity, Integer retryTimes) {
        String txnComment = "/* txn " + threadID + " */ ";

        try (Connection connection = ds.getConnection()) {
            try {

                connection.setAutoCommit(false);
                connection.createStatement().executeUpdate(txnComment + "begin optimistic
");

                // waiting for other thread ran the 'begin optimistic' statement
                TimeUnit.SECONDS.sleep(1);

                BigDecimal price = null;

                // read price of book
                PreparedStatement selectBook = connection.prepareStatement(txnComment +
 "SELECT * FROM books where id = ? for update");
                selectBook.setLong(1, bookID);
                ResultSet res = selectBook.executeQuery();
                if (!res.next()) {
                    throw new RuntimeException("book not exist");
                } else {
                    price = res.getBigDecimal("price");
                    int stock = res.getInt("stock");
                    if (stock < quantity) {
                        throw new RuntimeException("book not enough");
                    }
```

```
            }

            // update book
            String updateBookSQL = "update `books` set stock = stock - ? where id = ? an
d stock - ? >= 0";
            PreparedStatement updateBook = connection.prepareStatement(txnComment
+ updateBookSQL);
            updateBook.setInt(1, quantity);
            updateBook.setLong(2, bookID);
            updateBook.setInt(3, quantity);
            updateBook.executeUpdate();

            // insert order
            String insertOrderSQL = "insert into `orders` (`id`, `book_id`, `user_id`, `quality`)
values (?, ?, ?, ?)";
            PreparedStatement insertOrder = connection.prepareStatement(txnComment
+ insertOrderSQL);
            insertOrder.setLong(1, orderID);
            insertOrder.setLong(2, bookID);
            insertOrder.setLong(3, userID);
            insertOrder.setInt(4, quantity);
            insertOrder.executeUpdate();

            // update user
            String updateUserSQL = "update `users` set `balance` = `balance` - ? where id
= ?";
            PreparedStatement updateUser = connection.prepareStatement(txnComment
+ updateUserSQL);
            updateUser.setBigDecimal(1, price.multiply(new BigDecimal(quantity)));
            updateUser.setLong(2, userID);
            updateUser.executeUpdate();

            connection.createStatement().executeUpdate(txnComment + "commit");
        } catch (Exception e) {
            connection.createStatement().executeUpdate(txnComment + "rollback");
            System.out.println("error occurred: " + e.getMessage());

            if (e instanceof SQLException sqlException) {
                switch (sqlException.getErrorCode()) {
                    // You can get all error codes at https://docs.pingcap.com/tidb/stable/err
or-codes
                    case 9007: // Transactions in TiKV encounter write conflicts.
                    case 8028: // table schema changes
```

```
                    case 8002: // "SELECT FOR UPDATE" commit conflict
                    case 8022: // The transaction commit fails and has been rolled back
                        if (retryTimes != 0) {
                            System.out.println("rest " + retryTimes + " times. retry for " + e.getM
essage());
                            buy(ds, threadID, orderID, bookID, userID, quantity, retryTimes - 1);
                        }
                }
            }
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```

配置更改

此处，需将 pom.xml 中启动类：

<**mainClass**>com.pingcap.txn.TxnExample</**mainClass**>

更改为：

<**mainClass**>com.pingcap.txn.optimistic.TxnExample</**mainClass**>

来指向乐观事务的例子。

Golang 在编写悲观事务示例章节中的例子已经支持了乐观事务，无需更改，可直接使用。

Python 在编写悲观事务示例章节中的例子已经支持了乐观事务，无需更改，可直接使用。

10.2.2.2 2. 运行不涉及超卖的例子

运行示例程序：

在 Java 中运行示例程序：

```
mvn clean package
java -jar target/plain-java-txn-0.0.1-jar-with-dependencies.jar ALICE_NUM=4 BOB_NUM
=6
```

在 Golang 中运行示例程序：

```
go build -o bin/txn
./bin/txn -a 4 -b 6 -o true
```

在 Python 中运行示例程序：

```
OPTIMISTIC=True ALICE=4 BOB=6 python3 txn_example.py
```

SQL 语句执行过程：

```
   /* txn 2 */BEGIN OPTIMISTIC
/* txn 1 */BEGIN OPTIMISTIC
   /* txn 2 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
   /* txn 2 */UPDATE `books` SET `stock` = `stock` - 4 WHERE `id` = 1 AND `stock` - 4 >
= 0
   /* txn 2 */INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1001, 1,
1, 4)
   /* txn 2 */UPDATE `users` SET `balance` = `balance` - 400.0 WHERE `id` = 2
   /* txn 2 */COMMIT
/* txn 1 */SELECT * FROM `books` WHERE `id` = 1 for UPDATE
/* txn 1 */UPDATE `books` SET `stock` = `stock` - 6 WHERE `id` = 1 AND `stock` - 6 >=
0
/* txn 1 */INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1000, 1, 1,
6)
/* txn 1 */UPDATE `users` SET `balance` = `balance` - 600.0 WHERE `id` = 1
retry 1 times for 9007 Write conflict, txnStartTS=432618733006225412, conflictStartTS=
432618733006225411, conflictCommitTS=432618733006225414, key={tableID=126, han
dle=1} primary={tableID=114, indexID=1, indexValues={1, 1000, }} [try again later]
/* txn 1 */BEGIN OPTIMISTIC
/* txn 1 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
/* txn 1 */UPDATE `books` SET `stock` = `stock` - 6 WHERE `id` = 1 AND `stock` - 6 >=
0
/* txn 1 */INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1000, 1, 1,
6)
```

```
/* txn 1 */UPDATE `users` SET `balance` = `balance` - 600.0 WHERE `id` = 1
/* txn 1 */COMMIT
```

在乐观事务模式下，由于中间状态不一定正确，不能像悲观事务模式一样，通过 affected_rows 来判断某个语句是否执行成功。需要把事务看做一个整体，通过最终的 COMMIT 语句是否返回异常来判断当前事务是否发生写冲突。

从上面 SQL 日志可以看出，由于两个事务并发执行，并且对同一条记录做了修改，txn 1 COMMIT 之后抛出了 9007 Write conflict 异常。对于乐观事务写冲突，在应用端可以进行安全的重试，重试一次之后提交成功，最终执行结果符合预期：

```
mysql> SELECT * FROM books;
+----+-----------------------------------+--------------------+---------------------+-------+--------+
| id | title                             | type               | published_at        | stock | price  |
+----+-----------------------------------+--------------------+---------------------+-------+--------+
|  1 | Designing Data-Intensive Application | Science & Technology | 2018-09-01 00:00:00 |     0 | 100.00 |
+----+-----------------------------------+--------------------+---------------------+-------+--------+
1 row in set (0.01 sec)

mysql> SELECT * FROM orders;
+------+---------+---------+---------+---------------------+
| id   | book_id | user_id | quality | ordered_at          |
+------+---------+---------+---------+---------------------+
| 1000 |       1 |       1 |       6 | 2022-04-19 03:18:19 |
| 1001 |       1 |       1 |       4 | 2022-04-19 03:18:17 |
+------+---------+---------+---------+---------------------+
2 rows in set (0.01 sec)

mysql> SELECT * FROM users;
+----+---------+----------+
| id | balance | nickname |
+----+---------+----------+
|  1 | 9400.00 | Bob      |
|  2 | 9600.00 | Alice    |
+----+---------+----------+
2 rows in set (0.00 sec)
```

10.2.2.3 3. 运行防止超卖的例子

再来看一下用乐观事务防止超卖的例子，如果图书的库存剩余 10 本，Bob 购买 7 本，Alice 购买 4 本，两人几乎同时下单，结果会是怎样？继续复用乐观事务例子里的代码来解决这个需求，只不过把 Bob 购买数量从 6 改成 7：

运行示例程序：

在 Java 中运行示例程序：

```
mvn clean package
java -jar target/plain-java-txn-0.0.1-jar-with-dependencies.jar ALICE_NUM=4 BOB_NUM
=7
```

在 Golang 中运行示例程序：

```
go build -o bin/txn
./bin/txn -a 4 -b 7 -o true
```

在 Python 中运行示例程序：

```
OPTIMISTIC=True ALICE=4 BOB=7 python3 txn_example.py
```

```
/* txn 1 */BEGIN OPTIMISTIC
   /* txn 2 */BEGIN OPTIMISTIC
   /* txn 2 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
   /* txn 2 */UPDATE `books` SET `stock` = `stock` - 4 WHERE `id` = 1 AND `stock` - 4 >
= 0
   /* txn 2 */INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1001, 1,
1, 4)
   /* txn 2 */UPDATE `users` SET `balance` = `balance` - 400.0 WHERE `id` = 2
   /* txn 2 */COMMIT
/* txn 1 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
/* txn 1 */UPDATE `books` SET `stock` = `stock` - 7 WHERE `id` = 1 AND `stock` - 7 >=
0
/* txn 1 */INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1000, 1, 1,
7)
/* txn 1 */UPDATE `users` SET `balance` = `balance` - 700.0 WHERE `id` = 1
retry 1 times for 9007 Write conflict, txnStartTS=432619094333980675, conflictStartTS=
```

```
432619094333980676, conflictCommitTS=432619094333980678, key={tableID=126, han
dle=1} primary={tableID=114, indexID=1, indexValues={1, 1000, }} [try again later]
/* txn 1 */BEGIN OPTIMISTIC
/* txn 1 */SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
Fail -> out of stock
/* txn 1 */ROLLBACK
```

从上面的 SQL 日志可以看出，第一次执行由于写冲突，txn 1 在应用端进行了重试，从获取到的最新快照对比发现，剩余库存不够，应用端抛出 out of stock 异常结束。

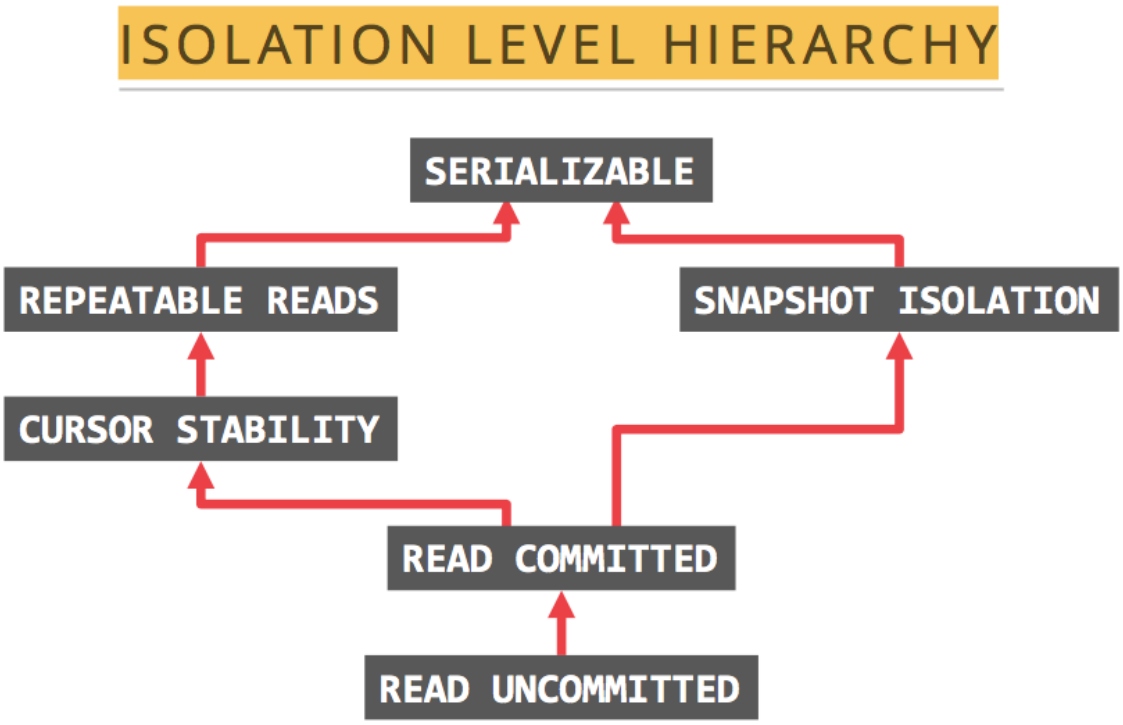
```
mysql> SELECT * FROM books;
+----+-------------------------------------+--------------------+--------------------+-------+--------+
| id | title                               | type               | published_at       | stock | price  |
+----+-------------------------------------+--------------------+--------------------+-------+--------+
|  1 | Designing Data-Intensive Application | Science & Technology | 2018-09-01 00:00:00 |     6 | 100.00 |
+----+-------------------------------------+--------------------+--------------------+-------+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM orders;
+------+---------+---------+---------+---------------------+
| id   | book_id | user_id | quality | ordered_at          |
+------+---------+---------+---------+---------------------+
| 1001 |       1 |       1 |       4 | 2022-04-19 03:41:16 |
+------+---------+---------+---------+---------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM users;
+----+----------+----------+
| id | balance  | nickname |
+----+----------+----------+
|  1 | 10000.00 | Bob      |
|  2 |  9600.00 | Alice    |
+----+----------+----------+
2 rows in set (0.00 sec)
```

10.3 事务限制

本章将简单介绍 TiDB 中的事务限制。

10.3.1 隔离级别

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



隔离级别

10.3.2 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。

10.3.3 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。

举个例子，假设你正在为医院写一个医生轮班管理程序。医院通常会同时要求几位医生待命，但底线是至少有一位医生在待命。医生可以放弃他们的班次（例如，如果他们自己生病了），只要至少有一个同事在这一班中继续工作。

现在出现这样一种情况，Alice 和 Bob 是两位值班医生。两人都感到不适，所以他们都决定请假。不幸的是，他们恰好在同一时间点击按钮下班。下面用程序来模拟一下这个过程。

Java 程序示例如下：

```
package com.pingcap.txn.write.skew;

import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;
```

```
public class EffectWriteSkew {
    public static void main(String[] args) throws SQLException, InterruptedException {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:mysql://localhost:4000/test?useServerPrepStmts=true&cachePre
pStmts=true");
        ds.setUsername("root");

        // prepare data
        Connection connection = ds.getConnection();
        createDoctorTable(connection);
        createDoctor(connection, 1, "Alice", true, 123);
        createDoctor(connection, 2, "Bob", true, 123);
        createDoctor(connection, 3, "Carol", false, 123);

        Semaphore txn1Pass = new Semaphore(0);
        CountDownLatch countDownLatch = new CountDownLatch(2);
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        threadPool.execute(() -> {
            askForLeave(ds, txn1Pass, 1, 1);
            countDownLatch.countDown();
        });

        threadPool.execute(() -> {
            askForLeave(ds, txn1Pass, 2, 2);
            countDownLatch.countDown();
        });

        countDownLatch.await();
    }

    public static void createDoctorTable(Connection connection) throws SQLException {
        connection.createStatement().executeUpdate("CREATE TABLE `doctors` (" +
                "    `id` int NOT NULL," +
                "    `name` varchar(255) DEFAULT NULL," +
                "    `on_call` tinyint DEFAULT NULL," +
                "    `shift_id` int DEFAULT NULL," +
                "    PRIMARY KEY (`id`)," +
                "    KEY `idx_shift_id` (`shift_id`)" +
                "  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin");
    }

    public static void createDoctor(Connection connection, Integer id, String name, Boole
```

```
an onCall, Integer shiftID) throws SQLException {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `doctors` (`id`, `name`, `on_call`, `shift_id`) VALUES (?, ?, ?, ?)");
        insert.setInt(1, id);
        insert.setString(2, name);
        insert.setBoolean(3, onCall);
        insert.setInt(4, shiftID);
        insert.executeUpdate();
    }

    public static void askForLeave(HikariDataSource ds, Semaphore txn1Pass, Integer txnI
D, Integer doctorID) {
        try(Connection connection = ds.getConnection()) {
            try {
                connection.setAutoCommit(false);

                String comment = txnID == 2 ? "    " : "" + "/* txn #{txn_id} */ ";
                connection.createStatement().executeUpdate(comment + "BEGIN");

                // Txn 1 should be waiting until txn 2 is done.
                if (txnID == 1) {
                    txn1Pass.acquire();
                }

                PreparedStatement currentOnCallQuery = connection.prepareStatement(com
ment +
                        "SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = ? AND `s
hift_id` = ?");
                currentOnCallQuery.setBoolean(1, true);
                currentOnCallQuery.setInt(2, 123);
                ResultSet res = currentOnCallQuery.executeQuery();

                if (!res.next()) {
                    throw new RuntimeException("error query");
                } else {
                    int count = res.getInt("count");
                    if (count >= 2) {
                        // If current on-call doctor has 2 or more, this doctor can leave
                        PreparedStatement insert = connection.prepareStatement( comment +
                                "UPDATE `doctors` SET `on_call` = ? WHERE `id` = ? AND `shift_id` = ?
");
                        insert.setBoolean(1, false);
                        insert.setInt(2, doctorID);
```

```
                    insert.setInt(3, 123);
                    insert.executeUpdate();

                    connection.commit();
                } else {
                    throw new RuntimeException("At least one doctor is on call");
                }
            }

            // Txn 2 is done. Let txn 1 run again.
            if (txnID == 2) {
                txn1Pass.release();
            }
        } catch (Exception e) {
            // If got any error, you should roll back, data is priceless
            connection.rollback();
            e.printStackTrace();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```

在 Golang 中，首先，封装一个用于适配 TiDB 事务的工具包 util，随后编写以下代码：

```
package main

import (
  "database/sql"
  "fmt"
  "sync"

  "github.com/pingcap-inc/tidb-example-golang/util"

  _ "github.com/go-sql-driver/mysql"
)

func main() {
  openDB("mysql", "root:@tcp(127.0.0.1:4000)/test", func(db *sql.DB) {
    writeSkew(db)
  })
```

```
}

func openDB(driverName, dataSourceName string, runnable func(db *sql.DB)) {
    db, err := sql.Open(driverName, dataSourceName)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    runnable(db)
}

func writeSkew(db *sql.DB) {
    err := prepareData(db)
    if err != nil {
        panic(err)
    }

    waitingChan, waitGroup := make(chan bool), sync.WaitGroup{}

    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        err = askForLeave(db, waitingChan, 1, 1)
        if err != nil {
            panic(err)
        }
    }()

    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        err = askForLeave(db, waitingChan, 2, 2)
        if err != nil {
            panic(err)
        }
    }()

    waitGroup.Wait()
}

func askForLeave(db *sql.DB, waitingChan chan bool, goroutineID, doctorID int) error {
    txnComment := fmt.Sprintf("/* txn %d */ ", goroutineID)
```

```
    if goroutineID != 1 {
        txnComment = "\t" + txnComment
    }

    txn, err := util.TiDBSqlBegin(db, true)
    if err != nil {
        return err
    }
    fmt.Println(txnComment + "start txn")

    // Txn 1 should be waiting until txn 2 is done.
    if goroutineID == 1 {
        <-waitingChan
    }

    txnFunc := func() error {
        queryCurrentOnCall := "SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_ca
ll` = ? AND `shift_id` = ?"
        rows, err := txn.Query(queryCurrentOnCall, true, 123)
        if err != nil {
            return err
        }
        defer rows.Close()
        fmt.Println(txnComment + queryCurrentOnCall + " successful")

        count := 0
        if rows.Next() {
            err = rows.Scan(&count)
            if err != nil {
                return err
            }
        }
        rows.Close()

        if count < 2 {
            return fmt.Errorf("at least one doctor is on call")
        }

        shift := "UPDATE `doctors` SET `on_call` = ? WHERE `id` = ? AND `shift_id` = ?"
        _, err = txn.Exec(shift, false, doctorID, 123)
        if err == nil {
            fmt.Println(txnComment + shift + " successful")
        }
```

```
        return err
    }

    err = txnFunc()
    if err == nil {
        txn.Commit()
        fmt.Println("[runTxn] commit success")
    } else {
        txn.Rollback()
        fmt.Printf("[runTxn] got an error, rollback: %+v\n", err)
    }

    // Txn 2 is done. Let txn 1 run again.
    if goroutineID == 2 {
        waitingChan <- true
    }

    return nil
}

func prepareData(db *sql.DB) error {
    err := createDoctorTable(db)
    if err != nil {
        return err
    }

    err = createDoctor(db, 1, "Alice", true, 123)
    if err != nil {
        return err
    }
    err = createDoctor(db, 2, "Bob", true, 123)
    if err != nil {
        return err
    }
    err = createDoctor(db, 3, "Carol", false, 123)
    if err != nil {
        return err
    }
    return nil
}

func createDoctorTable(db *sql.DB) error {
    _, err := db.Exec("CREATE TABLE IF NOT EXISTS `doctors` (" +
```

```
        "    `id` int NOT NULL," +
        "    `name` varchar(255) DEFAULT NULL," +
        "    `on_call` tinyint DEFAULT NULL," +
        "    `shift_id` int DEFAULT NULL," +
        "    PRIMARY KEY (`id`)," +
        "    KEY `idx_shift_id` (`shift_id`)" +
        "  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin")
    return err
}

func createDoctor(db *sql.DB, id int, name string, onCall bool, shiftID int) error {
    _, err := db.Exec("INSERT INTO `doctors` (`id`, `name`, `on_call`, `shift_id`) VALUES (?, ?,
 ?, ?)",
        id, name, onCall, shiftID)
    return err
}
```

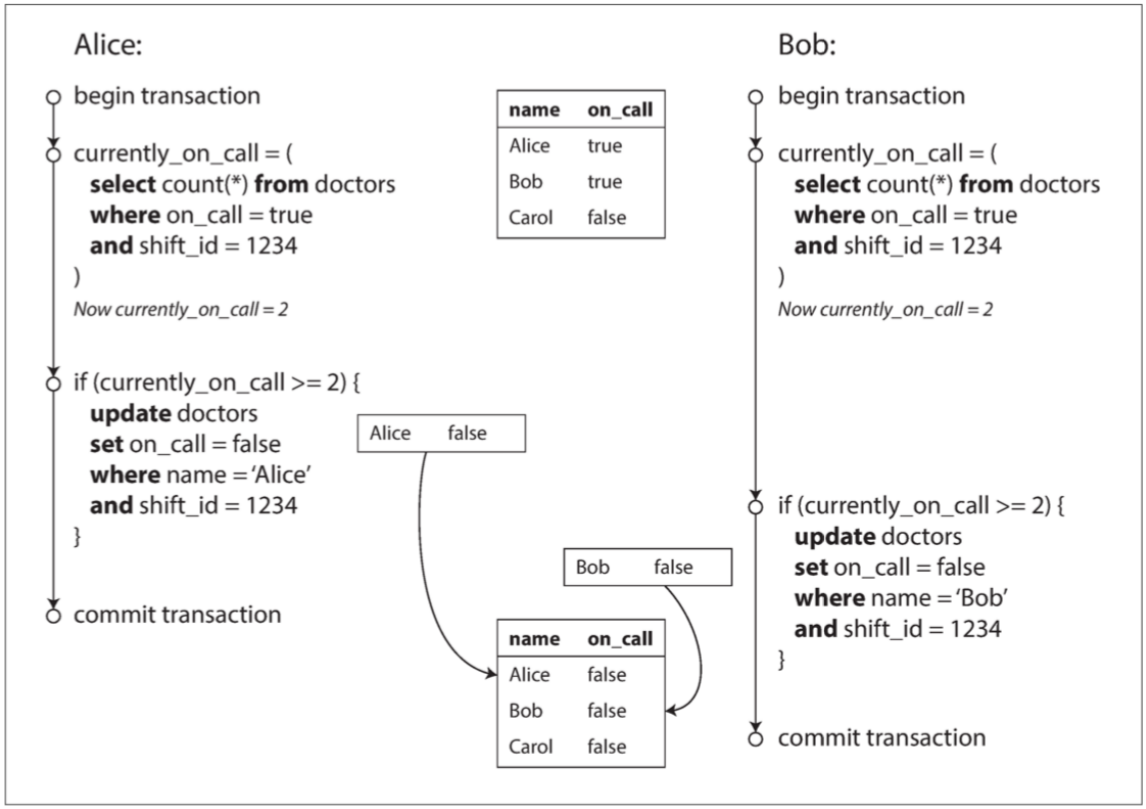
SQL 日志：

```
/* txn 1 */BEGIN
    /* txn 2 */BEGIN
    /* txn 2 */SELECT COUNT(*) as `count` FROM `doctors` WHERE `on_call` = 1 AND `shi
ft_id` = 123
    /* txn 2 */UPDATE `doctors` SET `on_call` = 0 WHERE `id` = 2 AND `shift_id` = 123
    /* txn 2 */COMMIT
/* txn 1 */SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = 1 and `shift_i
d` = 123
/* txn 1 */UPDATE `doctors` SET `on_call` = 0 WHERE `id` = 1 AND `shift_id` = 123
/* txn 1 */COMMIT
```

执行结果：

```
mysql> SELECT * FROM doctors;
+----+-------+---------+----------+
| id | name  | on_call | shift_id |
+----+-------+---------+----------+
|  1 | Alice |       0 |      123 |
|  2 | Bob   |       0 |      123 |
|  3 | Carol |       0 |      123 |
+----+-------+---------+----------+
```

在两个事务中，应用首先检查是否有两个或以上的医生正在值班；如果是的话，它就假定一名医生可以安全地休班。由于数据库使用快照隔离，两次检查都返回 2，所以两个事务都进入下一个阶段。Alice 更新自己的记录休班了，而 Bob 也做了一样的事情。两个事务都成功提交了，现在没有医生值班了。违反了至少有一名医生在值班的要求。下图(引用自《Designing Data-Intensive Application》)说明了实际发生的情况：



Write Skew

现在更改示例程序，使用 SELECT FOR UPDATE 来克服写偏斜问题：

Java 中使用 SELECT FOR UPDATE 来克服写偏斜问题的示例如下：

```
package com.pingcap.txn.write.skew;

import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;
```

```
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class EffectWriteSkew {
    public static void main(String[] args) throws SQLException, InterruptedException {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:mysql://localhost:4000/test?useServerPrepStmts=true&cachePrepStmts=true");
        ds.setUsername("root");

        // prepare data
        Connection connection = ds.getConnection();
        createDoctorTable(connection);
        createDoctor(connection, 1, "Alice", true, 123);
        createDoctor(connection, 2, "Bob", true, 123);
        createDoctor(connection, 3, "Carol", false, 123);

        Semaphore txn1Pass = new Semaphore(0);
        CountDownLatch countDownLatch = new CountDownLatch(2);
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        threadPool.execute(() -> {
            askForLeave(ds, txn1Pass, 1, 1);
            countDownLatch.countDown();
        });

        threadPool.execute(() -> {
            askForLeave(ds, txn1Pass, 2, 2);
            countDownLatch.countDown();
        });

        countDownLatch.await();
    }

    public static void createDoctorTable(Connection connection) throws SQLException {
        connection.createStatement().executeUpdate("CREATE TABLE `doctors` (" +
                "    `id` int NOT NULL," +
                "    `name` varchar(255) DEFAULT NULL," +
```

```
            "    `on_call` tinyint DEFAULT NULL," +
            "    `shift_id` int DEFAULT NULL," +
            "    PRIMARY KEY (`id`)," +
            "    KEY `idx_shift_id` (`shift_id`)" +
            "  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin");
    }

    public static void createDoctor(Connection connection, Integer id, String name, Boole
an onCall, Integer shiftID) throws SQLException {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `doctors` (`id`, `name`, `on_call`, `shift_id`) VALUES (?, ?, ?, ?)");
        insert.setInt(1, id);
        insert.setString(2, name);
        insert.setBoolean(3, onCall);
        insert.setInt(4, shiftID);
        insert.executeUpdate();
    }

    public static void askForLeave(HikariDataSource ds, Semaphore txn1Pass, Integer txnI
D, Integer doctorID) {
        try(Connection connection = ds.getConnection()) {
            try {
                connection.setAutoCommit(false);

                String comment = txnID == 2 ? "    " : "" + "/* txn #{txn_id} */ ";
                connection.createStatement().executeUpdate(comment + "BEGIN");

                // Txn 1 should be waiting until txn 2 is done.
                if (txnID == 1) {
                    txn1Pass.acquire();
                }

                PreparedStatement currentOnCallQuery = connection.prepareStatement(com
ment +
                        "SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = ? AND `s
hift_id` = ? FOR UPDATE");
                currentOnCallQuery.setBoolean(1, true);
                currentOnCallQuery.setInt(2, 123);
                ResultSet res = currentOnCallQuery.executeQuery();

                if (!res.next()) {
                    throw new RuntimeException("error query");
                } else {
```

```
                int count = res.getInt("count");
                if (count >= 2) {
                    // If current on-call doctor has 2 or more, this doctor can leave
                    PreparedStatement insert = connection.prepareStatement( comment +
                            "UPDATE `doctors` SET `on_call` = ? WHERE `id` = ? AND `shift_id` = ?
");
                    insert.setBoolean(1, false);
                    insert.setInt(2, doctorID);
                    insert.setInt(3, 123);
                    insert.executeUpdate();

                    connection.commit();
                } else {
                    throw new RuntimeException("At least one doctor is on call");
                }
            }

            // Txn 2 is done. Let txn 1 run again.
            if (txnID == 2) {
                txn1Pass.release();
            }
        } catch (Exception e) {
            // If got any error, you should roll back, data is priceless
            connection.rollback();
            e.printStackTrace();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
  }
}
```

Golang 中使用 SELECT FOR UPDATE 来克服写偏斜问题的示例如下：

```
package main

import (
  "database/sql"
  "fmt"
  "sync"

  "github.com/pingcap-inc/tidb-example-golang/util"
```

```
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    openDB("mysql", "root:@tcp(127.0.0.1:4000)/test", func(db *sql.DB) {
        writeSkew(db)
    })
}

func openDB(driverName, dataSourceName string, runnable func(db *sql.DB)) {
    db, err := sql.Open(driverName, dataSourceName)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    runnable(db)
}

func writeSkew(db *sql.DB) {
    err := prepareData(db)
    if err != nil {
        panic(err)
    }

    waitingChan, waitGroup := make(chan bool), sync.WaitGroup{}

    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        err = askForLeave(db, waitingChan, 1, 1)
        if err != nil {
            panic(err)
        }
    }()

    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        err = askForLeave(db, waitingChan, 2, 2)
        if err != nil {
            panic(err)
        }
```

```
    }()

    waitGroup.Wait()
}

func askForLeave(db *sql.DB, waitingChan chan bool, goroutineID, doctorID int) error {
    txnComment := fmt.Sprintf("/* txn %d */ ", goroutineID)
    if goroutineID != 1 {
        txnComment = "\t" + txnComment
    }

    txn, err := util.TiDBSqlBegin(db, true)
    if err != nil {
        return err
    }
    fmt.Println(txnComment + "start txn")

    // Txn 1 should be waiting until txn 2 is done.
    if goroutineID == 1 {
        <-waitingChan
    }

    txnFunc := func() error {
        queryCurrentOnCall := "SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_ca
ll` = ? AND `shift_id` = ?"
        rows, err := txn.Query(queryCurrentOnCall, true, 123)
        if err != nil {
            return err
        }
        defer rows.Close()
        fmt.Println(txnComment + queryCurrentOnCall + " successful")

        count := 0
        if rows.Next() {
            err = rows.Scan(&count)
            if err != nil {
                return err
            }
        }
        rows.Close()

        if count < 2 {
            return fmt.Errorf("at least one doctor is on call")
```

```
        }

        shift := "UPDATE `doctors` SET `on_call` = ? WHERE `id` = ? AND `shift_id` = ?"
        _, err = txn.Exec(shift, false, doctorID, 123)
        if err == nil {
            fmt.Println(txnComment + shift + " successful")
        }
        return err
    }

    err = txnFunc()
    if err == nil {
        txn.Commit()
        fmt.Println("[runTxn] commit success")
    } else {
        txn.Rollback()
        fmt.Printf("[runTxn] got an error, rollback: %+v\n", err)
    }

    // Txn 2 is done. Let txn 1 run again.
    if goroutineID == 2 {
        waitingChan <- true
    }

    return nil
}

func prepareData(db *sql.DB) error {
    err := createDoctorTable(db)
    if err != nil {
        return err
    }

    err = createDoctor(db, 1, "Alice", true, 123)
    if err != nil {
        return err
    }
    err = createDoctor(db, 2, "Bob", true, 123)
    if err != nil {
        return err
    }
    err = createDoctor(db, 3, "Carol", false, 123)
    if err != nil {
```

```
        return err
    }
    return nil
}

func createDoctorTable(db *sql.DB) error {
    _, err := db.Exec("CREATE TABLE IF NOT EXISTS `doctors` (" +
        "    `id` int NOT NULL," +
        "    `name` varchar(255) DEFAULT NULL," +
        "    `on_call` tinyint DEFAULT NULL," +
        "    `shift_id` int DEFAULT NULL," +
        "    PRIMARY KEY (`id`)," +
        "    KEY `idx_shift_id` (`shift_id`)" +
        "  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin")
    return err
}

func createDoctor(db *sql.DB, id int, name string, onCall bool, shiftID int) error {
    _, err := db.Exec("INSERT INTO `doctors` (`id`, `name`, `on_call`, `shift_id`) VALUES (?, ?,
 ?, ?)",
        id, name, onCall, shiftID)
    return err
}
```

SQL 日志：

```
/* txn 1 */BEGIN
    /* txn 2 */BEGIN
    /* txn 2 */SELECT COUNT(*) AS `count` FROM `doctors` WHERE on_call = 1 AND `shi
ft_id` = 123 FOR UPDATE
    /* txn 2 */UPDATE `doctors` SET on_call = 0 WHERE `id` = 2 AND `shift_id` = 123
    /* txn 2 */COMMIT
/* txn 1 */SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = 1 FOR UPD
ATE
At least one doctor is on call
/* txn 1 */ROLLBACK
```

执行结果：

```
mysql> SELECT * FROM doctors;
+----+-------+---------+----------+
| id | name  | on_call | shift_id |
```

```
+----+-------+---------+----------+
| 1 | Alice |       1 |      123 |
| 2 | Bob   |       0 |      123 |
| 3 | Carol |       0 |      123 |
+----+-------+---------+----------+
```

10.3.4 对 savepoint 和嵌套事务的支持

注意：

TiDB 从 v6.2.0 版本开始支持 savepoint 特性。因此低于 v6.2.0 版本的 TiDB 不支持 PROPAGATION_NESTED 传播行为。建议升级至 v6.2.0 及之后版本。如无法升级 TiDB 版本，且基于 Java Spring 框架的应用使用了 PROPAGATION_NESTED 传播行为，需要在应用端做出调整，将嵌套事务的逻辑移除。

Spring 支持的 PROPAGATION_NESTED 传播行为会启动一个嵌套的事务，它是当前事务之上独立启动的一个子事务。嵌套事务开始时会记录一个 savepoint，如果嵌套事务执行失败，事务将会回滚到 savepoint 的状态。嵌套事务是外层事务的一部分，它将会在外层事务提交时一起被提交。下面案例展示了 savepoint 机制：

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

10.3.5 大事务限制

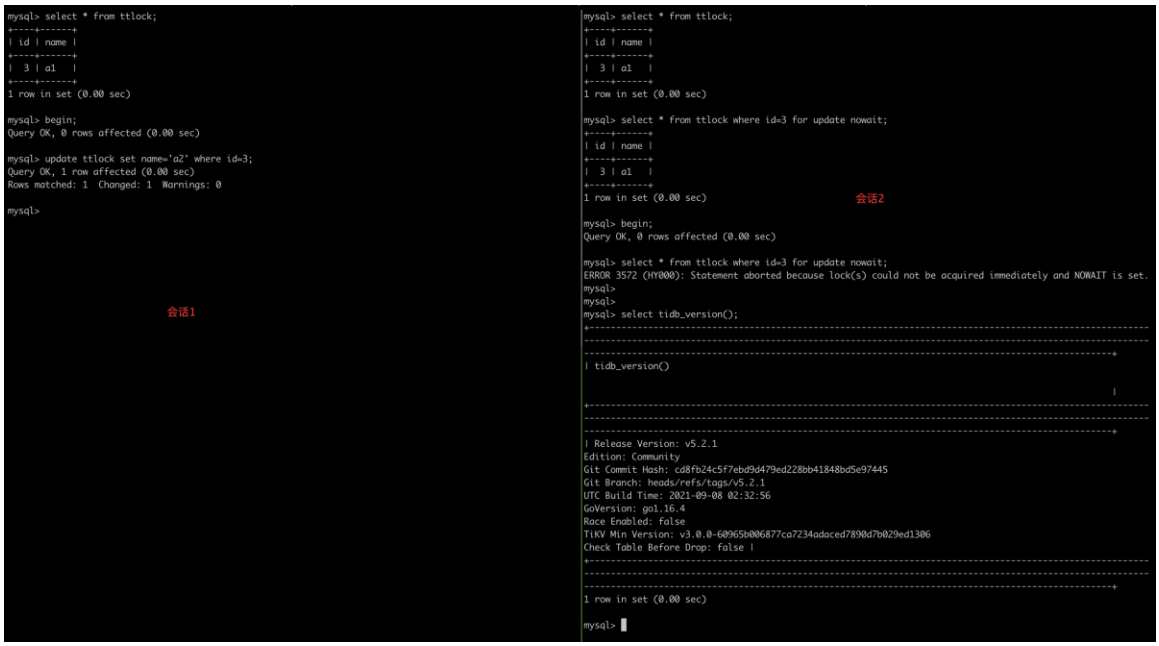
基本原则是要限制事务的大小。TiDB 对单个事务的大小有限制，这层限制是在 KV 层面。反映在 SQL 层面的话，简单来说一行数据会映射为一个 KV entry，每多一个索引，也会增加一个 KV entry。所以这个限制反映在 SQL 层面是：

- 最大单行记录容量为 120 MiB。
    - TiDB v4.0.10 及更高的 v4.0.x 版本、v5.0.0 及更高的版本可通过 tidb-server 配置项 performance.txn-entry-size-limit 调整，低于 TiDB v4.0.10 的版本支持的单行容量为 6 MiB。
    - 从 v7.1.8 开始，你可以使用 tidb_txn_entry_size_limit 系统变量动态修改该配置项的值。
- 支持的最大单个事务容量为 1 TiB。
    - TiDB v4.0 及更高版本可通过 tidb-server 配置项 performance.txn-total-size-limit 调整，低于 TiDB v4.0 的版本支持的最大单个事务容量为 100 MiB。
    - 在 v6.5.0 及之后的版本中，不再推荐使用配置项 performance.txn-total-size-limit。更多详情请参考 performance.txn-total-size-limit。

另外注意，无论是大小限制还是行数限制，还要考虑事务执行过程中，TiDB 做编码以及事务额外 Key 的开销。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。

10.3.6 自动提交的 SELECT FOR UPDATE 语句不会等锁

自动提交下的 SELECT FOR UPDATE 目前不会加锁。效果如下图所示：



TiDB 中的情况

这是已知的与 MySQL 不兼容的地方。

可以通过使用显式的 BEGIN;COMMIT; 解决该问题。

10.4 事务错误处理

本章介绍使用事务时可能会遇到的错误和处理办法。

10.4.1 死锁

如果应用程序遇到下面错误时，说明遇到了死锁问题：

ERROR 1213: Deadlock found **when** trying **to** get **lock**; try restarting **transaction**

当两个及以上的事务，双方都在等待对方释放已经持有的锁或因为加锁顺序不一致，造成循环等待锁资源，就会出现“死锁”。这里以 bookshop 数据库中的 books 表为示例演示死锁：

先给 books 表中写入 2 条数据：

INSERT INTO books (**id**, title, stock, published_at) **VALUES** (1, 'book-1', 10, now()), (2, 'book-2', 10, now());

在 TiDB 悲观事务模式下，用 2 个客户端分别执行以下语句，就会遇到死锁：

客户端-A	客户端-B
BEGIN;	
	BEGIN;
UPDATE books SET stock=stock-1 WHERE id=1;	
	UPDATE books SET stock=stock-1 WHERE id=2;
UPDATE books SET stock=stock-1 WHERE id=2; – 执行会被阻塞	
	UPDATE books SET stock=stock-1 WHERE id=1; – 遇到 Deadlock 错误

在客户端-B 遇到死锁错误后，TiDB 会自动 ROLLBACK 客户端-B 中的事务，然后客户端-A 中购买 id=2 的操作就会执行成功，再执行 COMMIT 即可完成购买的事务流程。

10.4.1.1 解决方案 1：避免死锁

为了应用程序有更好的性能，可以通过调整业务逻辑或者 Schema 设计，尽量从应用层面避免死锁。例如上面示例中，如果客户端-B 也和客户端-A 用同样的购买顺序，即都先买 id=1 的书，再买 id=2 的书，就可以避免死锁了：

客户端-A	客户端-B
BEGIN;	
	BEGIN;
UPDATE books SET stock=stock-1 WHERE id=1;	
	UPDATE books SET stock=stock-1 WHERE id=1; – 执行会被阻塞，当事务 A 完成后再继续执行
UPDATE books SET stock=stock-1 WHERE id=2;	
	UPDATE books SET stock=stock-1 WHERE id=2;
COMMIT;	
	COMMIT;

或者直接用 1 条 SQL 购买 2 本书，也能避免死锁，而且执行效率更高：

```
UPDATE books SET stock=stock-1 WHERE id IN (1, 2);
```

10.4.1.2 解决方案 2：减小事务粒度

如果每次购书都是一个单独的事务，也能避免死锁。但需要权衡的是，事务粒度太小不符合性能上的最佳实践。

10.4.1.3 解决方案 3：使用乐观事务

乐观事务模型下，并不会有死锁问题，但应用端需要加上乐观事务在失败后的重试逻辑，具体重试逻辑见应用端重试和错误处理。

10.4.1.4 解决方案 4：重试

正如错误信息中提示的那样，在应用代码中加入重试逻辑即可。具体重试逻辑见应用端重试和错误处理。

10.4.2 应用端重试和错误处理

尽管 TiDB 尽可能地与 MySQL 兼容，但其分布式系统的性质导致了某些差异，其中之一就是事务模型。

开发者用来与数据库通信的 Adapter 和 ORM 都是为 MySQL 和 Oracle 等传统数据库量身定制的，在这些数据库中，提交很少在默认隔离级别失败，因此不需要重试机制。对于这些客户端，当提交失败时，它们会因错误而中止，因为这在这些数据库中被呈现为罕见的异常。

与 MySQL 等传统数据库不同的是，在 TiDB 中，如果采用乐观事务模型，想要避免提交失败，需要在自己的应用程序的业务逻辑中添加机制来处理相关的异常。

下面的类似 Python 的伪代码展示了如何实现应用程序级的重试。 它不要求您的驱动程序或 ORM 来实现高级重试处理逻辑，因此可以在任何编程语言或环境中使用。

特别是，您的重试逻辑必须：

- 如果失败重试的次数达到 max_retries 限制，则抛出错误
- 使用 try ... catch ... 语句捕获 SQL 执行异常，当遇到下面这些错误时进行失败重试，遇到其它错误则进行回滚。详细信息请参考：错误码与故障诊断。
    - Error 8002: can not retry select for update statement：SELECT FOR UPDATE 写入冲突报错。
    - Error 8022: Error: KV error safe to retry：事务提交失败报错。
    - Error 8028: Information schema is changed during the execution of the statement：表的 Schema 结构因为完成了 DDL 变更，导致事务提交时报错。
    - Error 9007: Write conflict：写冲突报错，一般是采用乐观事务模式时，多个事务都对同一行数据进行修改时遇到的写冲突报错。
- 在 try 块结束时使用 COMMIT 提交事务：

```
while True:
    n++
    if n == max_retries:
        raise("did not succeed within #{n} retries")
    try:
        connection.execute("your sql statement here")
        connection.exec('COMMIT')
        break
    catch error:
        if (error.code != "9007" && error.code != "8028" && error.code != "8002" && error.
code != "8022"):
            raise error
        else:
            connection.exec('ROLLBACK')

            # Capture the error types that require application-side retry,
            # wait for a short period of time,
            # and exponentially increase the wait time for each transaction failure
            sleep_ms = int(((1.5 ** n) + rand) * 100)
            sleep(sleep_ms) # make sure your sleep() takes milliseconds
```

注意：

如果你经常遇到 Error 9007: Write conflict 错误，你可能需要进一步评估你的 Schema 设计和数据存取模型，找到冲突的根源并从设计上避免冲突。关于如何定位和解决事务冲突，请参考平凯数据库锁冲突问题处理。

10.4.3 推荐阅读

- 平凯数据库锁冲突问题处理
- 乐观事务模型下写写冲突问题排查

11 优化 SQL 性能

11.1 概览

本章内容描述了如何在 TiDB 中优化 SQL 语句的性能。为了获得更好的性能，你可以从以下方面入手：

- SQL 性能调优。
- Schema 设计：根据你的业务负载类型，为了避免事务冲突或者是热点，你可能需要对表的 Schema 做出一些调整。

11.1.1 SQL 性能调优

为了让 SQL 语句的性能更好，可以遵循以下原则：

- 扫描的数据越少越好，最好能只扫描需要的数据，避免扫描多余的数据。
- 使用合适的索引，对于 SQL 中的 WHERE 子句中的 Column，需要保证有相应索引，否则这将是一个全表扫的语句，性能会很差。
- 使用合适的 Join 类型。根据查询中各个表的大小和关联性，选择合适的 Join 类型也会非常重要。一般情况下，TiDB 的 cost-based 优化器会自动选择最优的 Join 类型。但在少数情况下，用户手动指定 Join 类型可能会更好。
- 使用合适的存储引擎。对于 OLTP 和 OLAP 混合类型的负载，推荐使用 TiFlash 查询引擎，具体可以参考 HTAP 查询。

11.1.2 Schema 设计

如果根据 SQL 性能调优调优后任然无法获得较好的性能，你可能需要检查你的 Schema 设计和数据读取模型，来确保避免以下问题：

- 事务冲突。关于如何定位和解决事务冲突，请参考平凯数据库锁冲突问题处理。
- 热点。关于如何定位和解决热点，请参考平凯数据库热点问题处理。

11.1.3 推荐阅读

- SQL 性能调优。

11.2 SQL 性能调优

本章介绍常见的 SQL 性能调优，你将会了解导致 SQL 执行慢的常见的原因。

11.2.1 准备工作

在开始之前，你可以通过 tiup demo 命令导入示例数据：

```
tiup demo bookshop prepare --books 1000000 --host 127.0.0.1 --port 4000
```

11.2.2 问题：全表扫描

慢查询最常见的原因就是 SELECT 语句执行是全表扫描，或者是用了不合适的索引。

当基于不在主键或任何二级索引中的列从大表中检索少量行时，通常会获得较差的性能：

```
SELECT * FROM books WHERE title = 'Marian Yost';
```

```
+------------+-------------+----------------------+--------------------+-------+--------+
| id         | title       | type                 | published_at       | stock | price  |
+------------+-------------+----------------------+--------------------+-------+--------+
| 65670536   | Marian Yost | Arts                 | 1950-04-09 06:28:58 | 542   | 435.01 |
| 1164070689 | Marian Yost | Education & Reference | 1916-05-27 12:15:35 | 216   | 328.18 |
| 1414277591 | Marian Yost | Arts                 | 1932-06-15 09:18:14 | 303   | 496.52 |
| 2305318593 | Marian Yost | Arts                 | 2000-08-15 19:40:58 | 398   | 402.90 |
```

```
| 2638226326 | Marian Yost | Sports              | 1952-04-02 12:40:37 | 191   | 174.64 |
+------------+-------------+---------------------+---------------------+-------+--------+
5 rows in set
Time: 0.582s
```

可以使用 EXPLAIN 来查看这个查询的执行计划，看看为什么查询这么慢：

```
EXPLAIN SELECT * FROM books WHERE title = 'Marian Yost';

+-------------------+-----------+----------+--------------+-------------------------------
-----------+
| id                | estRows   | task     | access object | operator info                 |
+-------------------+-----------+----------+--------------+-------------------------------
-----------+
| TableReader_7     | 1.27      | root     |               | data:Selection_6              |
| └─Selection_6     | 1.27      | cop[tikv] |              | eq(bookshop.books.title, "Marian Yos
t") |
|   └─TableFullScan_5 | 1000000.00 | cop[tikv] | table:books  | keep order:false
     |
+-------------------+-----------+----------+--------------+-------------------------------
-----------+
```

从执行计划中的 **TableFullScan_5** 可以看出，TiDB 将会对表 books 进行全表扫描，然后对每一行都判断 title 是否满足条件。**TableFullScan_5** 的 estRows 值为 1000000.00，说明优化器估计这个全表扫描会扫描 1000000.00 行数据。

更多关于 EXPLAIN 的使用介绍，可以阅读使用 EXPLAIN 解读执行计划。

11.2.2.1 解决方案：使用索引过滤数据

为了加速上面的查询，可以在 books.title 列创建一个索引：

```
CREATE INDEX title_idx ON books (title);
```

现在再执行这个查询将会快很多：

```
SELECT * FROM books WHERE title = 'Marian Yost';

+------------+-------------+---------------------+---------------------+-------+--------+
| id         | title       | type                | published_at        | stock | price  |
+------------+-------------+---------------------+---------------------+-------+--------+
| 1164070689 | Marian Yost | Education & Reference | 1916-05-27 12:15:35 | 216   | 328.1
8 |
```

```
| 1414277591 | Marian Yost | Arts              | 1932-06-15 09:18:14 | 303   | 496.52 |
| 2305318593 | Marian Yost | Arts              | 2000-08-15 19:40:58 | 398   | 402.90 |
| 2638226326 | Marian Yost | Sports            | 1952-04-02 12:40:37 | 191   | 174.64 |
| 65670536   | Marian Yost | Arts              | 1950-04-09 06:28:58 | 542   | 435.01 |
+------------+-------------+--------------------+---------------------+-------+--------+
5 rows in set
Time: 0.007s
```

可以使用 EXPLAIN 来查看这个查询的执行计划，看看为什么查询变快了：

```
EXPLAIN SELECT * FROM books WHERE title = 'Marian Yost';
```

```
+-------------------------+---------+-----------+---------------------------------+-----
-------------------------------------------+
| id                      | estRows | task      | access object                   | operator info
                          |
+-------------------------+---------+-----------+---------------------------------+-----
-------------------------------------------+
| IndexLookUp_10          | 1.27    | root      |                                 |
                          |
| ├─IndexRangeScan_8(Build) | 1.27    | cop[tikv] | table:books, index:title_idx(title) | rang
e:["Marian Yost","Marian Yost"], keep order:false |
| └─TableRowIDScan_9(Probe) | 1.27    | cop[tikv] | table:books                     | keep ord
er:false                                 |
+-------------------------+---------+-----------+---------------------------------+-----
-------------------------------------------+
```

从执行计划中的 **IndexLookUp_10** 可以看出，TiDB 将会通过索引 title_idx 来查询数据，其 estRows 值为 1.27，说明优化器估计只会扫描 1.27 行数据，远远小于之前全表扫的 1000000.00 行数据。

IndexLookUp_10 执行计划的执行流程是先用 **IndexRangeScan_8** 算子通过 title_idx 索引获取符合条件的索引数据，然后 **TableRowIDScan_9** 再更据索引数据里面的 Row ID 回表查询相应的行数据。

更多关于 TiDB 执行计划的内容，可以阅读平凯数据库执行计划概览。

11.2.2.2 解决方案：使用索引查询数据

上述解决方案中，需要先读取索引信息，再回表查询对应的行数据。但如果索引数据中包含了 SQL 查询所需的所有信息，就可以省去回表查询这个步骤。

例如下面查询中，仅需要根据 title 查询对应的 price：

```
SELECT title, price FROM books WHERE title = 'Marian Yost';
```

```
+------------+--------+
| title      | price  |
+------------+--------+
| Marian Yost | 435.01 |
| Marian Yost | 328.18 |
| Marian Yost | 496.52 |
| Marian Yost | 402.90 |
| Marian Yost | 174.64 |
+------------+--------+
5 rows in set
Time: 0.007s
```

由于索引 title_idx 仅包含 title 列的信息，所以 TiDB 还是需要扫描索引数据，然后回表查询 price 数据：

```
EXPLAIN SELECT title, price FROM books WHERE title = 'Marian Yost';
```

```
+-------------------------+---------+-----------+---------------------------------+-----
-----------------------------------------------+
| id                      | estRows | task      | access object                   | operator info
                              |
+-------------------------+---------+-----------+---------------------------------+-----
-----------------------------------------------+
| IndexLookUp_10          | 1.27    | root      |                                 |
                              |
| ├─IndexRangeScan_8(Build) | 1.27    | cop[tikv] | table:books, index:title_idx(title) | rang
e:["Marian Yost","Marian Yost"], keep order:false |
| └─TableRowIDScan_9(Probe) | 1.27    | cop[tikv] | table:books                     | keep ord
er:false                                 |
+-------------------------+---------+-----------+---------------------------------+-----
-----------------------------------------------+
```

删除 title_idx 索引，并新建一个 title_price_idx 索引：

```
ALTER TABLE books DROP INDEX title_idx;

CREATE INDEX title_price_idx ON books (title, price);
```

现在，price 数据已经存储在索引 title_price_idx 中了，所以下面查询仅需扫描索引数据，无需回表查询了。这种索引通常被叫做覆盖索引：

```
EXPLAIN SELECT title, price FROM books WHERE title = 'Marian Yost';

+-------------------+---------+-----------+-----------------------------------------------+-----------------------------------------------+
| id                | estRows | task      | access object                                 | operator info                                 |
+-------------------+---------+-----------+-----------------------------------------------+-----------------------------------------------+
| IndexReader_6     | 1.27    | root      |                                               | index:IndexRangeScan_5                        |
| └─IndexRangeScan_5 | 1.27    | cop[tikv] | table:books, index:title_price_idx(title, price) | range:["Marian Yost","Marian Yost"], keep order:false |
+-------------------+---------+-----------+-----------------------------------------------+-----------------------------------------------+
```

现在这条查询的速度将会更快：

```
SELECT title, price FROM books WHERE title = 'Marian Yost';

+------------+--------+
| title      | price  |
+------------+--------+
| Marian Yost | 174.64 |
| Marian Yost | 328.18 |
| Marian Yost | 402.90 |
| Marian Yost | 435.01 |
| Marian Yost | 496.52 |
+------------+--------+
5 rows in set
Time: 0.004s
```

由于后面的示例还会用到这个库，删除 title_price_idx 索引。

```
ALTER TABLE books DROP INDEX title_price_idx;
```

11.2.2.3 解决方案：使用主键查询数据

如果查询中使用主键过滤数据，这条查询的执行速度会非常快，例如表 books 的主键是列 id，使用列 id 来查询数据：

```
SELECT * FROM books WHERE id = 896;
```

```
+-----+---------------+--------------------+-------------------+-------+--------+
| id  | title         | type               | published_at      | stock | price  |
+-----+---------------+--------------------+-------------------+-------+--------+
| 896 | Kathryne Doyle | Science & Technology | 1969-03-18 01:34:15 | 468   | 281.32 |
+-----+---------------+--------------------+-------------------+-------+--------+
1 row in set
Time: 0.004s
```

使用 EXPLAIN 查看执行计划:

```
EXPLAIN SELECT * FROM books WHERE id = 896;
```

```
+------------+---------+------+--------------+--------------+
| id         | estRows | task | access object | operator info |
+------------+---------+------+--------------+--------------+
| Point_Get_1 | 1.00    | root | table:books   | handle:896    |
+------------+---------+------+--------------+--------------+
```

Point_Get，又名 “点查”，它的执行速度也非常快。

11.2.3 选择合适的 Join 执行计划

见 JOIN 查询的执行计划。

11.2.4 推荐阅读

- 使用 EXPLAIN 解读执行计划。
- 用 EXPLAIN 查看索引查询的执行计划。

11.3 性能调优最佳实践

本章将介绍在使用 TiDB 数据库的一些最佳实践。

11.3.1 DML 最佳实践

以下将介绍使用 TiDB 的 DML 时所涉及到的最佳实践。

11.3.1.1 使用单个语句多行数据操作

当需要修改多行数据时，推荐使用单个 SQL 多行数据的语句：

```
INSERT INTO t VALUES (1, 'a'), (2, 'b'), (3, 'c');

DELETE FROM t WHERE id IN (1, 2, 3);
```

不推荐使用多个 SQL 单行数据的语句：

```
INSERT INTO t VALUES (1, 'a');
INSERT INTO t VALUES (2, 'b');
INSERT INTO t VALUES (3, 'c');

DELETE FROM t WHERE id = 1;
DELETE FROM t WHERE id = 2;
DELETE FROM t WHERE id = 3;
```

11.3.1.2 使用 PREPARE

当需要多次执行某个 SQL 语句时，推荐使用 PREPARE 语句，可以避免重复解析 SQL 语法的开销。

在 Golang 中使用 PREPARE 语句：

```
func BatchInsert(db *sql.DB) error {
  stmt, err := db.Prepare("INSERT INTO t (id) VALUES (?), (?), (?), (?), (?)")
  if err != nil {
    return err
  }
  for i := 0; i < 1000; i += 5 {
    values := []interface{}{i, i + 1, i + 2, i + 3, i + 4}
    _, err = stmt.Exec(values...)
    if err != nil {
      return err
    }
  }
```

```
  return nil
}
```

在 Java 中使用 PREPARE 语句：

```
public void batchInsert(Connection connection) throws SQLException {
  PreparedStatement statement = connection.prepareStatement(
      "INSERT INTO `t` (`id`) VALUES (?), (?), (?), (?), (?)");
  for (int i = 0; i < 1000; i ++) {
    statement.setInt(i % 5 + 1, i);

    if (i % 5 == 4) {
      statement.executeUpdate();
    }
  }
}
```

在 Python 中使用 PREPARE 语句时，并不需要显式指定。在你使用参数化查询时，mysqlclient 等 Driver 将自动转用执行计划。

注意不要重复执行 PREPARE 语句，否则并不能提高执行效率。

11.3.1.3 避免查询不必要的信息

如非必要，不要总是用 SELECT * 返回所以列的数据，下面查询是低效的：

```
SELECT * FROM books WHERE title = 'Marian Yost';
```

应该仅查询需要的列信息，例如：

```
SELECT title, price FROM books WHERE title = 'Marian Yost';
```

11.3.1.4 使用批量删除

当需要删除大量的数据，推荐使用批量删除，见批量删除

11.3.1.5 使用批量更新

当需要更新大量的数据时，推荐使用批量更新，见批量更新

11.3.1.6 使用 TRUNCATE 语句代替 DELETE 全表数据

当需要删除一个表的所有数据时，推荐使用 TRUNCATE 语句：

```
TRUNCATE TABLE t;
```

不推荐使用 DELETE 全表数据：

```
DELETE FROM t;
```

11.3.2 DDL 最佳实践

以下将介绍使用 TiDB 的 DDL 时所涉及到的最佳实践。

11.3.2.1 主键选择的最佳实践

见选择主键时应遵守的规则。

11.3.3 索引的最佳实践

见索引的最佳实践。

11.3.3.1 添加索引性能最佳实践

TiDB 支持在线添加索引操作，可通过 ADD INDEX 或 CREATE INDEX 完成索引添加操作。添加索引不会阻塞表中的数据读写。可以通过修改下面的系统变量来调整 DDL 操作 re-organize 阶段的并行度与回填索引的单批数量大小：

- tidb_ddl_reorg_worker_cnt
- tidb_ddl_reorg_batch_size

为了减少对在线业务的影响，添加索引的默认速度会比较保守。当添加索引的目标列仅涉及查询负载，或者与线上负载不直接相关时，可以适当调大上述变量来加速添加索引：

```
SET @@global.tidb_ddl_reorg_worker_cnt = 16;
SET @@global.tidb_ddl_reorg_batch_size = 4096;
```

当添加索引操作的目标列被频繁更新（包含 UPDATE、INSERT 和 DELETE）时，调大上述配置会造成较为频繁的写冲突，使得在线负载较大；同时添加索引操作也可能

由于不断地重试，需要很长的时间才能完成。此时建议调小上述配置来避免和在线业务的写冲突：

```
SET @@global.tidb_ddl_reorg_worker_cnt = 4;
SET @@global.tidb_ddl_reorg_batch_size = 128;
```

11.3.4 事务冲突

关于如何定位和解决事务冲突，请参考平凯数据库锁冲突问题处理。

11.3.5 Java 数据库应用开发最佳实践

TiDB 最佳实践系列（五）Java 数据库应用开发指南。

11.3.5.1 推荐阅读

- TiDB 最佳实践系列（一）高并发写入常见热点问题及规避方法。

11.4 索引的最佳实践

本章会介绍在 TiDB 中使用索引的一些最佳实践。

11.4.1 准备工作

本章内容将会用 bookshop 数据库中的 books 表作为示例。

```
CREATE TABLE `books` (
  `id` bigint AUTO_RANDOM NOT NULL,
  `title` varchar(100) NOT NULL,
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humanities & Social Sciences', 'Science & Technology', 'Kids', 'Sports') NOT NULL,
  `published_at` datetime NOT NULL,
  `stock` int DEFAULT '0',
  `price` decimal(15,2) DEFAULT '0.0',
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

11.4.2 创建索引的最佳实践

- 建立你需要使用的数据的所有列的组合索引，这种优化技巧被称为覆盖索引优化 (covering index optimization)。**覆盖索引优化**将使得 TiDB 可以直接在索引上得到该查询所需的所有数据，可以大幅提升性能。

- 避免创建你不需要的二级索引，有用的二级索引能加速查询，但是要注意新增一个索引是有副作用的。每增加一个索引，在插入一条数据的时候，就要额外新增一个 Key-Value，所以索引越多，写入越慢，并且空间占用越大。另外过多的索引也会影响优化器运行时间，并且不合适的索引会误导优化器。所以索引并不是越多越好。

- 根据具体的业务特点创建合适的索引。原则上需要对查询中需要用到的列创建索引，目的是提高性能。下面几种情况适合创建索引：

    - 区分度比较大的列，通过索引能显著地减少过滤后的行数。例如推荐在人的身份证号码这一列上创建索引，但不推荐在人的性别这一列上创建索引。
    - 有多个查询条件时，可以选择组合索引，注意需要把等值条件的列放在组合索引的前面。这里举一个例子，假设常用的查询是 SELECT * FROM t where c1 = 10 and c2 = 100 and c3 > 10，那么可以考虑建立组合索引 Index cidx (c1, c2, c3)，这样可以用查询条件构造出一个索引前缀进行 Scan。

- 请使用有意义的二级索引名，推荐你遵循公司或组织的表命名规范。如果你的公司或组织没有相应的命名规范，可参考索引命名规范。

11.4.3 使用索引的最佳实践

- 建立索引的目的是为了加速查询，所以请确保索引能在一些查询中被用上。如果一个索引不会被任何查询语句用到，那这个索引是没有意义的，请删除这个索引。

- 使用组合索引时，需要满足最左前缀原则。

  例如假设在列 title, published_at 上新建一个组合索引索引：

  ```
  CREATE INDEX title_published_at_idx ON books (title, published_at);
  ```

  下面这个查询依然能用上这个组合索引：

  ```
  SELECT * FROM books WHERE title = 'database';
  ```

  但下面这个查询由于未指定组合索引中最左边第一列的条件，所以无法使用组合索引：

  ```
  SELECT * FROM books WHERE published_at = '2018-08-18 21:42:08';
  ```

- 在查询条件中使用索引列作为条件时，不要在索引列上做计算，函数，或者类型转换的操作，会导致优化器无法使用该索引。

  例如假设在时间类型的列 published_at 上新建一个索引：

  ```
  CREATE INDEX published_at_idx ON books (published_at);
  ```

  但下面查询是无法使用 published_at 上的索引的：

  ```
  SELECT * FROM books WHERE YEAR(published_at)=2022;
  ```

  可以改写成下面查询，避免在索引列上做函数计算后，即可使用 published_at 上的索引:

  ```
  SELECT * FROM books WHERE published_at >= '2022-01-01' AND published_at < '2023-01-01';
  ```

  也可以使用表达式索引，例如对查询条件中的 YEAR(published_at) 创建一个表达式索引：

  ```
  CREATE INDEX published_year_idx ON books ((YEAR(published_at)));
  ```

  然后通过 SELECT * FROM books WHERE YEAR(published_at)=2022; 查询就能使用 published_year_idx 索引来加速查询了。

注意：

表达式索引目前是 TiDB 的实验特性，需要在 TiDB 配置文件中开启表达式索引特性，详情可以参考表达式索引文档。

- 尽量使用覆盖索引，即索引列包含查询列，避免总是 SELECT * 查询所有列的语句。

  例如下面查询只需扫描索引 title_published_at_idx 数据即可获取查询列的数据：

  ```
  SELECT title, published_at FROM books WHERE title = 'database';
  ```

  但下面查询语句虽然能用上组合索引 (title, published_at)，但会多一个回表查询非索引列数据的额外开销，回表查询是指根据索引数据中存储的引用（一般是主键信息），到表中查询相应行的数据。

  ```
  SELECT * FROM books WHERE title = 'database';
  ```

- 查询条件使用 !=，NOT IN 时，无法使用索引。例如下面查询无法使用任何索引：

  ```
  SELECT * FROM books WHERE title != 'database';
  ```

- 使用 LIKE 时如果条件是以通配符 % 开头，也无法使用索引。例如下面查询无法使用任何索引：

  ```
  SELECT * FROM books WHERE title LIKE '%database';
  ```

- 当查询条件有多个索引可供使用，但你知道用哪一个索引是最优的时，推荐使用优化器 Hint 来强制优化器使用这个索引，这样可以避免优化器因为统计信息不准或其他问题时，选错索引。

  例如下面查询中，假设在列 id 和列 title 上都各自有索引 id_idx 和 title_idx，你知道 id_idx 的过滤性更好，就可以在 SQL 中使用 USE INDEX Hint 来强制优化器使用 id_idx 索引。

```
SELECT * FROM t USE INDEX(id_idx) WHERE id = 1 and title = 'database';
```

- 查询条件使用 IN 表达式时，后面匹配的条件数量建议不要超过 300 个，否则执行效率会较差。

11.5 其他优化

11.5.1 避免隐式类型转换

本章内容将介绍 TiDB 中的隐式类型转换规则、可能带来的后果及避免方法。

11.5.1.1 转换规则

当 SQL 中谓词两侧的数据类型不一致时，TiDB 将隐式地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（NULL 安全的 <=> 相等比较运算符除外，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。

11.5.1.2 隐式类型转换引起的后果

隐式类型转换增强了人机交互的易用性，但在应用代码中，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
- 精度丢失

11.5.1.2.1 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.acc
ount_id), 6.010000000009801e+15) |
|   └─TableFullScan_5   | 10000785000.00 | cop[tikv] | table:account | keep order:false
                           |
+-----------------------+---------------+----------+--------------+-------------------------------------
-----------------------------------+
3 rows in set (0.00 sec)
```

运行结果简述：从以上执行计划中，可见 Cast 算子。

11.5.1.2.2 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 '11112222111111112
00000';
+-----------------------+---------+----------+--------------+---------------------------
-----------------------------------------------------+
| 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) |
|   └─TableFullScan_5   | 1.00    | cop[tikv] | table:t1     | keep order:false, stats:pseudo
                                                      |
+-----------------------+---------+----------+--------------+---------------------------
-----------------------------------------------------+
3 rows in set (0.00 sec)
```

运行结果简述：从以上执行计划中，可见 Cast 算子。

```
SELECT * FROM `t1` WHERE `a` BETWEEN '12123123' AND '1111222211111111200000';
+----------------------+
| a                    |
+----------------------+
| 111122221111111222211 |
+----------------------+
1 row in set (0.01 sec)
```

运行结果简述：以上执行出现了错误结果。

11.5.2 唯一序列号生成方案

本章将介绍唯一序列号生成方案，为自行生成唯一 ID 的开发者提供帮助。

11.5.2.1 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，自增列并未拥有业务属性，仅仅代表了这一行数据，即被作为无业务含义的代理主键使用。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，用户将难以通过自增列来获取序列号中所需的数字自增值。

11.5.2.2 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。从 TiDB v4.0 版本开始提供序列功能，详情请参考 CREATE SEQUENCE。

11.5.2.3 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	delta seconds	worker node id	sequencs
1bit	28bits	22bits	13bits

- 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。

11.5.2.4 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	当前序列已被分配出去的最大值
STEP	int	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。例如，经过位反转后，ID 00000010100101000001111010011100 变为 00111001011110000010100101000000，ID 11111111111111111111111111111101 变为 10111111111111111111111111111111。

12 故障诊断

12.1 SQL 或事务问题

本章介绍在开发应用过程中可能遇到的常见问题的诊断处理方法。

12.1.1 SQL 操作常见问题

如果你想提高 SQL 的性能，可以阅读 SQL 性能优化来避免一些常见的性能问题。然后如果依然存在性能问题，推荐阅读：

- 分析慢查询
- 使用 Top SQL 定位系统资源消耗过多的查询

如果你遇到了一些关于 SQL 操作的问题，可以阅读 SQL 操作常见问题。

12.1.2 事务错误处理

见事务错误处理。

12.1.3 推荐阅读

- 不支持的功能特性
- 集群管理 FAQ
- 产品 FAQ

12.2 结果集不稳定

本章将叙述结果集不稳定错误的处理办法。

12.2.1 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，所以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.courscore) 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
```

运行结果简述：上例为 sql_mode 设置了 ONLY_FULL_GROUP_BY 的效果。

12.2.2 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，需明确地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;
+------------+-------------+------------------------+-----------+
| class      | stuname     | course                 | courscore |
+------------+-------------+------------------------+-----------+
| 2018_CS_01 | MonkeyDLuffy | PrinciplesofDatabase   |      60.5 |
| 2018_CS_01 | MonkeyDLuffy | English                |      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  | AdvancedMathematics    |      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  | DiscreteMathematics    |     14.0 |
| 2018_CS_03 | PatrickStar  | PrinciplesofDatabase   |      9.0 |
| 2018_CS_03 | PatrickStar  | English                |     60.0 |
| 2018_CS_03 | PatrickStar  | OpTableTennis          |     12.0 |
| 2018_CS_03 | PatrickStar  | OpPiano                |     99.0 |
| 2018_CS_03 | SpongeBob    | FundamentalsofCompiling |     43.0 |
| 2018_CS_03 | SpongeBob    | OperatingSystem        |     95.0 |
| 2018_CS_03 | SpongeBob    | PrincipleofStatistics  |     90.0 |
| 2018_CS_03 | SpongeBob    | ProbabilityTheory      |     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)
```

当遇到相同的 order by 值时，排序结果不稳定。为减少随机性，应当尽可能保持 order by 值的唯一性。不能保证唯一的继续加，保证 order by 的字段组合是唯一时，结果才能唯一。

12.2.3 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() 在不加 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，造成结果集不稳定：

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 ',')                              |
+-----------------------------------------------------------------------+
| 20000200992,20000200993,20000200994,20000200995,20000200996,2000020 0... |
+-----------------------------------------------------------------------+
```

   第二次查询：

```
mysql> select GROUP_CONCAT( customer_id SEPARATOR ',' ) FROM customer where customer_id like '200002%';
+-----------------------------------------------------------------------+
| GROUP_CONCAT(customer_id  SEPARATOR ',')                              |
+-----------------------------------------------------------------------+
| 20000203040,20000203041,20000203042,20000203043,20000203044,2000020 3... |
+-----------------------------------------------------------------------+
```

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 ',')                              |
+-----------------------------------------------------------------------+
| 20000200000,20000200001,20000200002,20000200003,20000200004,2000020 0... |
+-----------------------------------------------------------------------+
```

   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 ',')                              |
+-----------------------------------------------------------------------+
| 20000200000,20000200001,20000200002,20000200003,20000200004,2000020
0... |
+-----------------------------------------------------------------------+
```

12.2.4 select * from t limit n 的结果不稳定

返回结果与数据在存储节点 (TiKV) 上的分布有关。如果进行了多次查询，存储节点 (TiKV) 不同存储单元 (Region) 返回结果的速度不同，会造成结果不稳定。

12.3 平凯数据库中的各种超时

本章将介绍 TiDB 中的各种超时，为排查错误提供依据。

12.3.1 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。

- TiDB v4.0 之前的版本：

  默认情况下，TiDB 可以确保每个 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。

- TiDB v4.0 及之后的版本：

  正在运行的事务，如果持续时间不超过 24 小时，在运行期间 GC 会被阻塞，不会出现 GC life time is shorter than transaction duration 报错。

如果你确定在临时特殊场景中需要更长的读取时间，可以通过以下方式调大 MVCC 版本保留时间：

- TiDB v5.0 之前的版本 ： 调整 mysql.tidb 表中的 tikv_gc_life_time。
- TiDB v5.0 及之后的版本：调整系统变量 tidb_gc_life_time。

需要注意的是，此变量的配置是立刻影响全局的，调大它会增加当前所有快照的生命时长，调小它也会立即缩短所有快照的生命时长。过多的 MVCC 版本会影响 TiDB 的集群性能，因此在使用后，需要及时把此变量调整回之前的设置。

> **Tip:**
>
> 特别地，在 Dumpling 备份时，如果导出的数据量少于 1 TB 且导出的 TiDB 版本为 v4.0.0 或更新版本，并且 Dumpling 可以访问 TiDB 集群的 PD 地址以及 INFORMATION_SCHEMA.CLUSTER_INFO 表，Dumpling 会自动调整 GC 的 safe point 从而阻塞 GC 且不会对原集群造成影响。以下场景除外：
>
> - 数据量非常大（超过 1 TB）。
> - Dumpling 无法直接连接到 PD，例如 TiDB 集群运行在 Kubernetes 上且与 Dumpling 分离。
>
> 在这些场景中，你必须使用 tikv_gc_life_time 提前手动调长 GC 时间，以避免因为导出过程中发生 GC 导致导出失败。详见 TiDB 工具 Dumpling 的手动设置 TiDB GC 时间。

更多关于 GC 的信息，请参考 GC 机制简介文档。

12.3.2 事务超时

在事务已启动但未提交或回滚的情况下，你可能需要更细粒度的控制和更短的超时，以避免持有锁的时间过长。此时，你可以使用 TiDB 在 v7.1.8 引入的 tidb_idle_transaction_timeout 控制用户会话中事务的空闲超时。

垃圾回收 (GC) 不会影响到正在执行的事务。但悲观事务的运行仍有上限，有基于事务超时的限制（TiDB 配置文件 [performance] 类别下的 max-txn-ttl 修改，默认为 60 分钟）和基于事务使用内存的限制。

形如 INSERT INTO t10 SELECT * FROM t1 的 SQL 语句，不会受到 GC 的影响，但超过了 max-txn-ttl 的时间后，会由于超时而回滚。

12.3.3 SQL 执行时间超时

TiDB 还提供了一个系统变量来限制单条 SQL 语句的执行时间，仅对“只读”语句生效：max_execution_time，它的默认值为 0，表示无限制。max_execution_time 的单位为 ms，但实际精度在 100ms 级别，而非更准确的毫秒级别。

12.3.4 JDBC 查询超时

MySQL jdbc 的查询超时设置 setQueryTimeout() 对 TiDB 不起作用。这是因为现实客户端感知超时时，向数据库发送一个 KILL 命令。但是由于 tidb-server 是负载均衡的，为防止在错误的 tidb-server 上终止连接，tidb-server 不会执行这个 KILL。这时就要用 MAX_EXECUTION_TIME 实现查询超时的效果。

TiDB 提供了三个与 MySQL 兼容的超时控制参数：

- **wait_timeout**，控制与 Java 应用连接的非交互式空闲超时时间。在 TiDB v5.4 及以上版本中，默认值为 28800 秒，即空闲超时为 8 小时。在 v5.4 之前，默认值为 0，即没有时间限制。
- **interactive_timeout**，控制与 Java 应用连接的交互式空闲超时时间，默认值为 8 小时。
- **max_execution_time**，控制连接中 SQL 执行的超时时间，仅对“只读”语句生效，默认值是 0，即允许连接无限忙碌（一个 SQL 语句执行无限的长的时间）。

但在实际生产环境中，空闲连接和一直无限执行的 SQL 对数据库和应用都有不好的影响。你可以通过在应用的连接字符串中配置这两个 session 级的变量来避免空闲连接和执行时间过长的 SQL 语句。例如，设置 sessionVariables=wait_timeout=3600（1 小时）和 sessionVariables=max_execution_time=300000（5 分钟）。

13 引用文档

13.1 Bookshop 应用

Bookshop 是一个虚拟的在线书店应用，你可以在 Bookshop 当中便捷地购买到各种类别的书，也可以对你看过的书进行点评。

为了方便你阅读应用开发指南中的内容，我们将以 Bookshop 应用的数据表结构和数据为基础来编写示例 SQL。本章节将为你介绍如何导入该应用的表结构和数据，以及其数据表结构的定义。

13.1.1 导入表结构和数据

你可以通过 TiUP 导入 Bookshop 应用的表结构和数据。

13.1.1.1 通过 tiup demo 命令行导入

如果你使用 TiUP 部署 TiDB 集群或者你可以直接连接到你的 TiDB 服务器，你可以通过如下命令快速生成并导入 Bookshop 应用的示例数据：

tiup demo bookshop prepare

该命令默认会连接到 127.0.0.1 地址上的 4000 端口，使用 root 用户名进行无密码登录，默认在名为 bookshop 的数据库中创建表结构。

13.1.1.1.1 配置连接信息

你可以通过如下参数修改默认的连接信息：

参数	简写	默认值	解释
--host	-H	127.0.0.1	数据库地址
--port	-P	4000	数据库端口
--user	-U	root	数据库用户
--password	-p	无	数据库用户密码
--db	-D	bookshop	数据库名称

例如，你想要连接到数据库，可以使用如下命令指定连接信息进行连接：

tiup demo bookshop prepare -U <username> -H <endpoint> -P 4000 -p <password>

13.1.1.1.2 设置数据量

另外，你还可以通过如下参数指定各个数据库表生成的数据量：

参数	默认值	解释
--users	10000	指定在 users 表生成的数据行数
--authors	20000	指定在 authors 表生成的数据行数
--books	20000	指定在 books 表生成的数据行数
--orders	300000	指定在 orders 表生成的数据行数
--ratings	300000	指定在 ratings 表生成的数据行数

例如，以下命令通过 --users 参数指定生成 20 万行用户信息，通过 --books 参数指定生成 50 万行书籍的基本信息，通过 --authors 参数指定生成 10 万的作者信息，通过 --ratings 参数指定生成 100 万的评分记录，通过 --orders 参数指定生成 100 万的订单记录。

tiup demo bookshop prepare --users=200000 --books=500000 --authors=100000 --ratings=1000000 --orders=1000000 --drop-tables

通过 --drop-tables 参数你可以删除原有的表结构，更多的参数说明你可以通过命令 tiup demo bookshop --help 进行了解。

13.1.1.2 查看数据导入情况

导入完成后，你可以通过下面的 SQL 语句查看各个表的数据量信息：

```
SELECT
   CONCAT(table_schema,'.',table_name) AS 'Table Name',
   table_rows AS 'Number of Rows',
   CONCAT(ROUND(data_length/(1024*1024*1024),4),'G') AS 'Data Size',
   CONCAT(ROUND(index_length/(1024*1024*1024),4),'G') AS 'Index Size',
   CONCAT(ROUND((data_length+index_length)/(1024*1024*1024),4),'G') AS 'Total'
FROM
```

```
    information_schema.TABLES
WHERE table_schema LIKE 'bookshop';
```

运行结果为：

```
+----------------------+---------------+-----------+------------+---------+
| Table Name           | Number of Rows | Data Size | Index Size | Total   |
+----------------------+---------------+-----------+------------+---------+
| bookshop.orders      |       1000000 | 0.0373G   | 0.0075G    | 0.0447G |
| bookshop.book_authors |       1000000 | 0.0149G   | 0.0149G    | 0.0298G |
| bookshop.ratings     |       4000000 | 0.1192G   | 0.1192G    | 0.2384G |
| bookshop.authors     |        100000 | 0.0043G   | 0.0000G    | 0.0043G |
| bookshop.users       |        195348 | 0.0048G   | 0.0021G    | 0.0069G |
| bookshop.books       |       1000000 | 0.0546G   | 0.0000G    | 0.0546G |
+----------------------+---------------+-----------+------------+---------+
6 rows in set (0.03 sec)
```

13.1.2 数据表详解

以下将详细介绍 Bookshop 应用程序的数据库表结构：

13.1.2.1 books 表

该表用于存储书籍的基本信息。

字段名	类型	含义
id	bigint	书籍的唯一标识
title	varchar(100)	书籍名称
type	enum	书籍类型（如：杂志、动漫、教辅等）
stock	bigint	库存
price	decimal(15,2)	价格
published_at	datetime	出版时间

13.1.2.2 authors 表

该表用于存储作者的基本信息。

字段名	类型	含义
id	bigint	作者的唯一标识

字段名	类型	含义
name	varchar(100)	姓名
gender	tinyint	生理性别 (0: 女, 1: 男，NULL: 未知)
birth_year	smallint	生年
death_year	smallint	卒年

13.1.2.3 users 表

该表用于存储使用 Bookshop 应用程序的用户。

字段名	类型	含义
id	bigint	用户的唯一标识
balance	decimal(15,2)	余额
nickname	varchar(100)	昵称

13.1.2.4 ratings 表

该表用于存储用户对书籍的评分记录。

字段名	类型	含义
book_id	bigint	书籍的唯一标识（关联至 books）
user_id	bigint	用户的唯一标识（关联至 users）
score	tinyint	用户评分 (1-5)
rated_at	datetime	评分时间

13.1.2.5 book_authors 表

一个作者可能会编写多本书，一本书可能需要多个作者同时编写，该表用于存储书籍与作者之间的对应关系。

字段名	类型	含义
book_id	bigint	书籍的唯一标识（关联至 books）
author_id	bigint	作者的唯一标识（关联至 authors）

13.1.2.6 orders 表

该表用于存储用户购买书籍的订单信息。

字段名	类型	含义
id	bigint	订单的唯一标识
book_id	bigint	书籍的唯一标识（关联至 books）
user_id	bigint	用户唯一标识（关联至 users）
quantity	tinyint	购买数量
ordered_at	datetime	购买时间

13.1.3 数据库初始化 dbinit.sql 脚本

如果你希望手动创建 Bookshop 应用的数据库表结构，你可以运行以下 SQL 语句：

```
CREATE DATABASE IF NOT EXISTS `bookshop`;

DROP TABLE IF EXISTS `bookshop`.`books`;
CREATE TABLE `bookshop`.`books` (
  `id` bigint AUTO_RANDOM NOT NULL,
  `title` varchar(100) NOT NULL,
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humani
ties & Social Sciences', 'Science & Technology', 'Kids', 'Sports') NOT NULL,
  `published_at` datetime NOT NULL,
  `stock` int DEFAULT '0',
  `price` decimal(15,2) DEFAULT '0.0',
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`authors`;
CREATE TABLE `bookshop`.`authors` (
  `id` bigint AUTO_RANDOM NOT NULL,
  `name` varchar(100) NOT NULL,
  `gender` tinyint DEFAULT NULL,
  `birth_year` smallint DEFAULT NULL,
  `death_year` smallint DEFAULT NULL,
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

```
DROP TABLE IF EXISTS `bookshop`.`book_authors`;
CREATE TABLE `bookshop`.`book_authors` (
  `book_id` bigint NOT NULL,
  `author_id` bigint NOT NULL,
  PRIMARY KEY (`book_id`,`author_id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`ratings`;
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint NOT NULL,
  `user_id` bigint NOT NULL,
  `score` tinyint NOT NULL,
  `rated_at` datetime NOT NULL DEFAULT NOW() ON UPDATE NOW(),
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED,
  UNIQUE KEY `uniq_book_user_idx` (`book_id`,`user_id`)
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
ALTER TABLE `bookshop`.`ratings` SET TIFLASH REPLICA 1;

DROP TABLE IF EXISTS `bookshop`.`users`;
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM NOT NULL,
  `balance` decimal(15,2) DEFAULT '0.0',
  `nickname` varchar(100) UNIQUE NOT NULL,
  PRIMARY KEY (`id`)
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`orders`;
CREATE TABLE `bookshop`.`orders` (
  `id` bigint AUTO_RANDOM NOT NULL,
  `book_id` bigint NOT NULL,
  `user_id` bigint NOT NULL,
  `quality` tinyint NOT NULL,
  `ordered_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURR
ENT_TIMESTAMP,
  PRIMARY KEY (`id`) CLUSTERED,
  KEY `orders_book_id_idx` (`book_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

13.2 规范

13.2.1 对象命名规范

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

13.2.1.1 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。

13.2.1.2 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)。

13.2.1.3 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_{business code}_{creator abbreviation}_{date}）
    - 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。

13.2.1.4 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_{description}。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，请参考附录部分的官方保留字。

13.2.1.5 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。

13.2.2 SQL 开发规范

本章将介绍一些使用 SQL 的一般化开发规范。

13.2.2.1 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 SQL 命令运行异常造成的异常中断。

13.2.2.2 SELECT * 使用规范

- 基本原则：避免使用 SELECT * 进行查询。
- 详细说明：按需求选择合适的字段列，避免盲目地 SELECT * 读取全部字段，因为其会消耗网络带宽。考虑将被查询的字段也加入到索引中，以有效利用覆盖索引功能。

13.2.2.3 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避免使用任何函数,包括数据类型转换函数，以避免索引失效。或者可以考虑使用表达式索引功能。

- 详细说明：

  不推荐的写法：

  ```
  SELECT gmt_create
  FROM ...
  WHERE DATE_FORMAT(gmt_create, '%Y%m%d %H:%i:%s') = '20090101 00:00:00'
  ```

  推荐的写法：

  ```
  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')
  ```

13.2.2.4 其他规范

- WHERE 条件中不要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。
- 避免使用 %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，一次性地发给客户端执行，TiDB 只会返回第一个 SQL 的执行结果。
- 当使用表达式时，检查其是否支持计算下推到存储层的功能 (TiKV、TiFlash)，否则应有预期在 TiDB 层需要消耗更多内存、甚至 OOM。计算下推到存储层的功能列表如下：
  - TiFlash 支持的计算下推清单。
  - 下推到 TiKV 的表达式列表。
  - 谓词下推。

14 云原生开发环境

14.1 Gitpod

使用 Gitpod，只需单击一个按钮或链接即可在浏览器中获得完整的开发环境，并且可以立即编写代码。

Gitpod 是一个开源 Kubernetes 应用程序（GitHub 仓库地址)，适用于可直接编写代码的开发环境，可为云中的每个任务提供全新的自动化开发环境，非常迅速。此外，Gitpod 能够将你的开发环境描述为代码，并直接从你的浏览器或桌面 IDE 启动即时、远程和基于云的开发环境。

14.1.1 快速开始

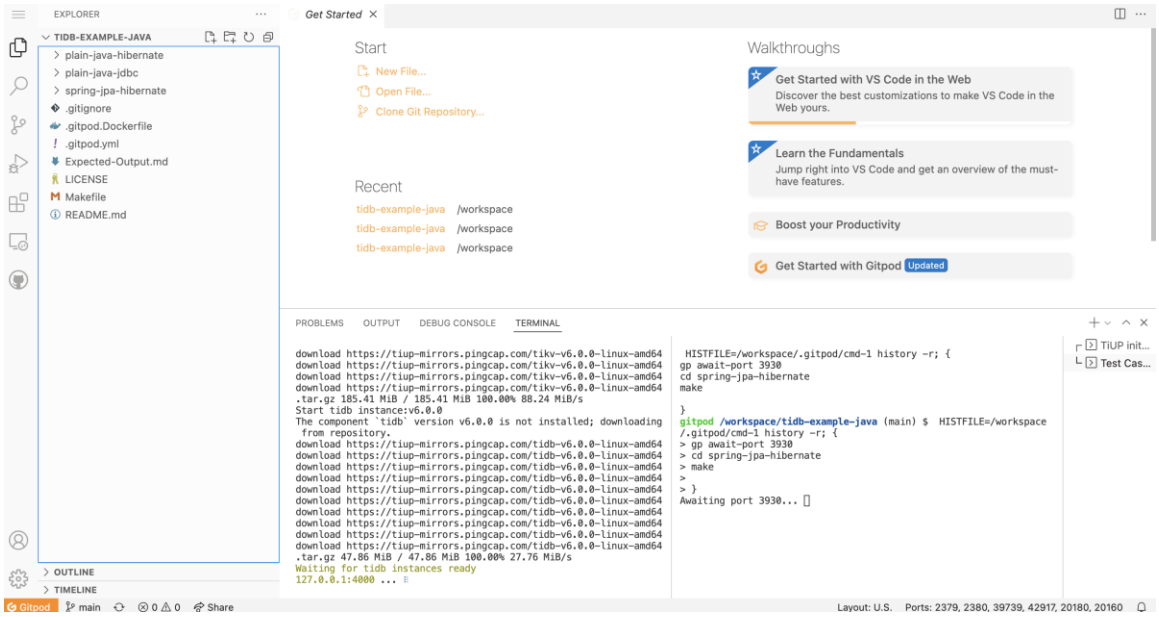
1. Fork 出 TiDB 应用开发的示例代码仓库 pingcap-inc/tidb-example-java。
2. 通过浏览器的地址栏，在示例代码仓库的 URL 前加上 https://gitpod.io/# 来启动你的 gitpod 工作区。
    - 例如，https://gitpod.io/#https://github.com/pingcap-inc/tidb-example-java。
    - 支持在 URL 中配置环境变量。例如，https://gitpod.io/#targetFile=spring-jpa-hibernate_Makefile,targetMode=spring-jpa-hibernate/https://github.com/pingcap-inc/tidb-example-java。
3. 使用列出的提供商之一登录并启动工作区，例如，Github。

14.1.2 使用默认的 Gitpod 配置和环境

完成快速开始 的步骤之后，Gitpod 会需要一段时间来设置你的工作区。

以 Spring Boot Web 应用程序为例，通过 URL https://gitpod.io/#targetFile=spring-jpa-hibernate_Makefile,targetMode=spring-jpa-hibernate/https://github.com/pingcap-inc/tidb-example-java 可以创建一个新工作区。

完成后，你将看到如下所示的页面。

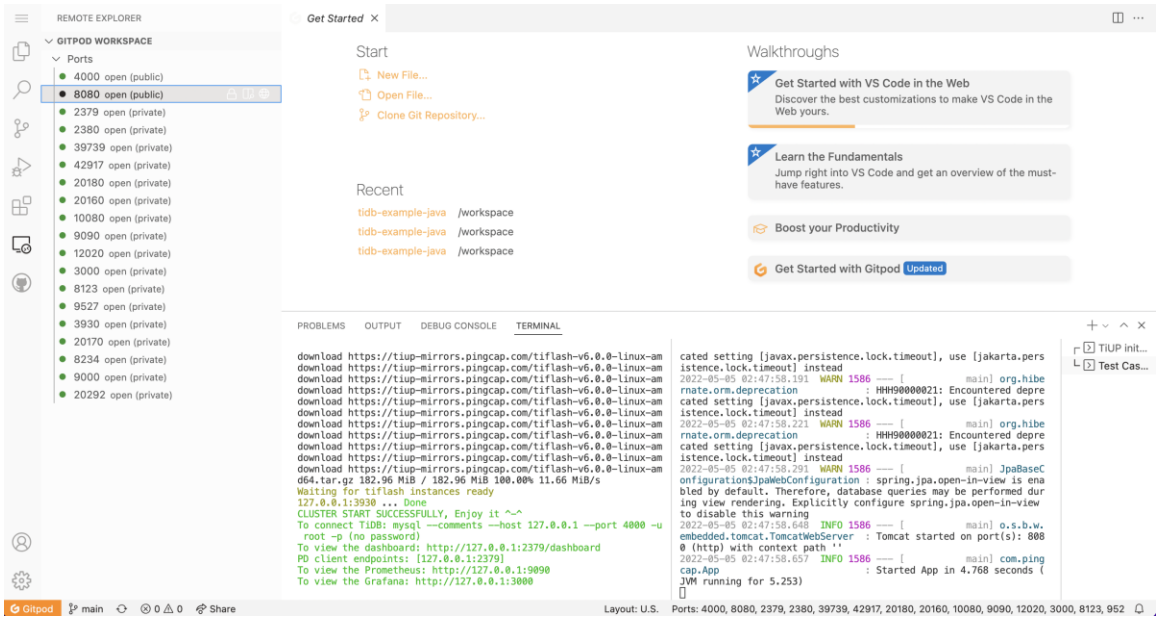


playground gitpod workspace init

页面中的这个场景使用了 TiUP 来搭建一个 TiDB Playground。你可以在终端的左侧查看进度。

一旦 TiDB Playground 准备就绪，另一个 Spring JPA Hibernate 任务将运行。 你可以在终端的右侧查看进度。

完成所有任务后，你可以看到如下所示的页面，并在左侧导航栏的 REMOTE EXPLORER 中找到你的端口 8080 URL（Gitpod 支持基于 URL 的端口转发）。



playground gitpod workspace ready

14.1.3 使用自定义的 Gitpod 配置和 Docker 镜像

14.1.3.1 自定义 Gitpod 配置

在项目的根目录中，参考示例 .gitpod.yml，创建一个 .gitpod.yml 文件用于配置 Gitpod 工作空间。

```
## This configuration file was automatically generated by Gitpod.
## Please adjust to your needs (see https://www.gitpod.io/docs/config-gitpod-file)
## and commit this file to your remote git repository to share the goodness with others.

## image:
##   file: .gitpod.Dockerfile

tasks:
  - name: Open Target File
    command: |
      if [ -n "$targetFile" ]; then code ${targetFile//[_]//};  fi
  - name: TiUP init playground
    command: |
      $HOME/.tiup/bin/tiup playground
  - name: Test Case
    openMode: split-right
    init: echo "*** Waiting for TiUP Playground Ready! ***"
    command: |
```

```
    gp await-port 3930
    if [ "$targetMode" == "plain-java-jdbc" ]
    then
      cd plain-java-jdbc
      code src/main/resources/dbinit.sql
      code src/main/java/com/pingcap/JDBCExample.java
      make mysql
    elif [ "$targetMode" == "plain-java-hibernate" ]
    then
      cd plain-java-hibernate
      make
    elif [ "$targetMode" == "spring-jpa-hibernate" ]
    then
      cd spring-jpa-hibernate
      make
    fi
ports:
  - port: 8080
    visibility: public
  - port: 4000
    visibility: public
  - port: 2379-36663
    onOpen: ignore
```

14.1.3.2 自定义 Gitpod Docker 镜像

默认情况下，Gitpod 使用名为 Workspace-Full 的标准 Docker 镜像作为工作空间的基础。 基于此默认镜像启动的工作区预装了 Docker、Go、Java、Node.js、C/C++、Python、Ruby、Rust、PHP 以及 Homebrew、Tailscale、Nginx 等工具。

你可以提供公共 Docker 镜像或 Dockerfile。 并为你的项目安装所需的任何依赖项。

这是一个 Dockerfile 示例：示例 .gitpod.Dockerfile

```
FROM gitpod/workspace-java-17

RUN sudo apt install mysql-client -y
RUN curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

然后需要更新.gitpod.yml：

```
## This configuration file was automatically generated by Gitpod.
## Please adjust to your needs (see https://www.gitpod.io/docs/config-gitpod-file)
## and commit this file to your remote git repository to share the goodness with others.

image:
  # 在这里导入你的 Dockerfile
  file: .gitpod.Dockerfile

tasks:
  - name: Open Target File
    command: |
      if [ -n "$targetFile" ]; then code ${targetFile//[_]//};  fi
  - name: TiUP init playground
    command: |
      $HOME/.tiup/bin/tiup playground
  - name: Test Case
    openMode: split-right
    init: echo "*** Waiting for TiUP Playground Ready! ***"
    command: |
      gp await-port 3930
      if [ "$targetMode" == "plain-java-jdbc" ]
      then
        cd plain-java-jdbc
        code src/main/resources/dbinit.sql
        code src/main/java/com/pingcap/JDBCExample.java
        make mysql
      elif [ "$targetMode" == "plain-java-hibernate" ]
      then
        cd plain-java-hibernate
        make
      elif [ "$targetMode" == "spring-jpa-hibernate" ]
      then
        cd spring-jpa-hibernate
        make
      fi
ports:
  - port: 8080
    visibility: public
  - port: 4000
    visibility: public
```

```
- port: 2379-36663
  onOpen: ignore
```

14.1.3.3 应用更改

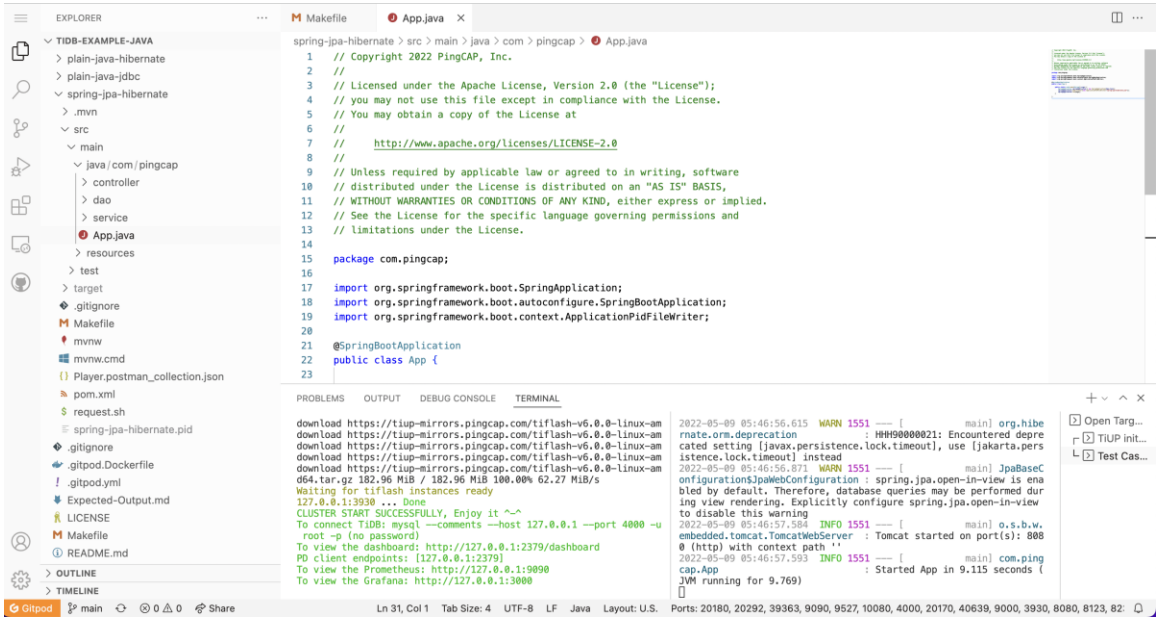
完成对 .gitpod.yml 文件配置后，请保证最新的代码已在你对应的 GitHub 代码仓库中可用。

访问 https://gitpod.io/#<YOUR_REPO_URL> 以建立新的 Gitpod 工作区，新工作区会应用最新的代码。

访问 https://gitpod.io/workspaces 以获取所有建立的工作区。

14.1.4 总结

Gitpod 提供了完整的、自动化的、预配置的云原生开发环境。无需本地配置，你可以直接在浏览器中开发、运行、测试代码。



playground gitpod summary

15 第三方工具支持

15.1 平凯数据库支持的第三方工具

注意：

本文档仅列举了常见的 TiDB 支持的第三方工具，未被列入其中的第三方工具并非代表不支持，但 PingCAP 无法了解其是否使用到 TiDB 不支持的特性，从而无法保证兼容性。

平凯数据库高度兼容 MySQL 协议，使得大部分适配 MySQL 的 Driver、ORM 及其他工具与 TiDB 兼容。本文主要介绍这些工具和它们的支持等级。

15.1.1 支持等级

PingCAP 与开源社区合作，通过三方工具提供以下支持：

- Full：表明 PingCAP 已经支持该工具的绝大多数功能兼容性，并且在新版本中对其保持兼容，将定期地对下表中记录的新版本进行兼容性测试。
- Compatible：表明由于该工具已适配 MySQL，而 TiDB 高度兼容 MySQL 协议，因此可以使用此工具的大部分功能。但 PingCAP 并未对该工具作出完整的兼容性验证，有可能出现一些意外的行为。

注意：

除非明确说明，否则对于支持的 Driver 或者 ORM 框架并不包括应用端事务重试和错误处理。

如果在使用本文列出的工具连接 TiDB 时出现问题，请反馈给我们。

15.1.2 Driver

编程语言	驱动	最新已测试版本	支持等级	TiDB 适配器	教程
Go	Go-MySQL-Driver	v1.6.0	Full	N/A	使用 Go-MySQL-

编程语言	驱动	最新已测试版本	支持等级	TiDB 适配器	教程
					Driver 连接到 TiDB
Java	JDBC	8.0	Full	4. pingcap/mysql-connector-j 5. pingcap/tidb-loadbalance	使用 JDBC 连接到 TiDB

15.1.3 ORM

编程语言	ORM 框架	最新已测试版本	支持等级	TiDB 适配器	教程
Go	gorm	v1.23.5	Full	N/A	使用 GORM 连接到 TiDB
	beego	v2.0.3	Full	N/A	N/A
	upper/db	v4.5.2	Full	N/A	N/A
	xorm	v1.3.1	Full	N/A	N/A
Java	Hibernate	6.1.0.Final	Full	N/A	使用 Hibernate 连接到 TiDB
	MyBatis	v3.5.10	Full	N/A	使用 MyBatis 连接到 TiDB
	Spring Data JPA	2.7.2	Full	N/A	使用 Spring Boot 连接到 TiDB
	jOOQ	v3.16.7 (Open Source)	Full	N/A	N/A
Ruby	Active Record	v7.0	Full	N/A	使用 Rails 框架和 ActiveRecord ORM 连接到 TiDB
JavaScript / TypeScript	Sequelize	v6.20.1	Full	N/A	N/A
	Prisma	4.16.2	Full	N/A	使用 Prisma 连接到 TiDB
	TypeORM	v0.3.17	Full	N/A	使用 TypeORM 连接到 TiDB

编程语言	ORM 框架	最新已测试版本	支持等级	TiDB 适配器	教程
Python	Django	v4.2	Full	django-tidb	使用 Django 连接到 TiDB
	SQLAlchemy	v1.4.37	Full	N/A	使用 SQLAlchemy 连接到 TiDB

15.1.4 GUI

GUI	最新已测试版本	支持等级	教程
JetBrains DataGrip	2023.2.1	Full	N/A
DBeaver	23.0.3	Full	N/A
Visual Studio Code	1.72.0	Full	N/A

15.2 已知的第三方工具兼容问题

注意：

TiDB 已列举不支持的功能特性，典型的不支持特性有：

- 存储过程与函数
- 触发器
- 事件
- 自定义函数
- 空间类型的函数、数据类型和索引
- XA 语法

这些不支持的功能不兼容将被视为预期行为，不再重复叙述。关于更多 TiDB 与 MySQL 的兼容性对比，你可以查看与 MySQL 兼容性对比。

本文列举的兼容性问题是在一些平凯数据库支持的第三方工具中发现的。

15.2.1 通用

15.2.1.1 平凯数据库中 SELECT CONNECTION_ID() 返回结果类型为 64 位整型

描述

TiDB 中 SELECT CONNECTION_ID() 的返回值为 64 位，如 2199023260887。而 MySQL 中返回值为 32 位，如 391650。

规避方法

在 TiDB 应用程序中，请注意使用各语言的 64 位整型类型（或字符串类型）存储 SELECT CONNECTION_ID() 的结果，防止溢出。如 Java 应使用 Long 或 String 进行接收，JavaScript/TypeScript 应使用 string 类型进行接收。

15.2.1.2 平凯数据库未设置 Com_* 计数器

描述

MySQL 维护了一系列 Com_ 开头的服务端变量来记录你对数据库的操作总数，如 Com_select 记录了 MySQL 数据库从上次启动开始，总共发起的 SELECT 语句数（即使语句并未成功执行）。而 TiDB 并未维护此变量。你可以使用语句 SHOW GLOBAL STATUS LIKE 'Com_%' 观察 TiDB 与 MySQL 的差异。

规避方法

请勿使用这样的变量。在 MySQL 中 Com_* 常见的使用场景之一是监控。TiDB 的可观测性较为完善，无需从服务端变量进行查询。如需定制监控工具，可阅读平凯数据库监控框架概述来获得更多信息。

15.2.1.3 平凯数据库错误日志区分 TIMESTAMP 与 DATETIME 类型

描述

TiDB 错误日志区分 TIMESTAMP 与 DATETIME，而 MySQL 不区分，全部返回为 DATETIME。即 MySQL 会将 TIMESTAMP 类型的报错信息错误地写为 DATETIME 类型。

规避方法

请勿使用错误日志进行字符串匹配，要使用错误码进行故障诊断。

15.2.1.4 平凯数据库不支持 CHECK TABLE 语句

描述

TiDB 不支持 CHECK TABLE 语句。

规避方法

在 TiDB 中使用 ADMIN CHECK [TABLE|INDEX] 语句进行表中数据和对应索引的一致性校验。

15.2.2 与 MySQL JDBC 的兼容性

测试版本为 MySQL Connector/J 8.0.29。

15.2.2.1 默认排序规则不一致

描述

MySQL Connector/J 的排序规则保存在客户端内，通过获取的服务端版本进行判别。

下表列出了已知的客户端与服务端排序规则不一致的字符集：

字符集	客户端默认排序规则	服务端默认排序规则
ascii	ascii_general_ci	ascii_bin
latin1	latin1_swedish_ci	latin1_bin
utf8mb4	utf8mb4_0900_ai_ci	utf8mb4_bin

规避方法

在 TiDB 中手动设置排序规则，不要依赖客户端默认排序规则。客户端默认排序规则由 MySQL Connector/J 配置文件保存。

15.2.2.2 参数 NO_BACKSLASH_ESCAPES 不生效

描述

TiDB 中无法使用 NO_BACKSLASH_ESCAPES 参数从而不进行 \ 字符的转义。已提 issue。

规避方法

在 TiDB 中不搭配使用 NO_BACKSLASH_ESCAPES 与 \，而是使用 \\ 编写 SQL 语句。

15.2.2.3 未设置索引使用情况参数

描述

TiDB 在通讯协议中未设置 SERVER_QUERY_NO_GOOD_INDEX_USED 与 SERVER_QUERY_NO_INDEX_USED 参数。这将导致以下参数返回与实际不一致：

- com.mysql.cj.protocol.ServerSession.noIndexUsed()
- com.mysql.cj.protocol.ServerSession.noGoodIndexUsed()

规避方法

不使用 noIndexUsed() 与 noGoodIndexUsed() 函数。

15.2.2.4 不支持 enablePacketDebug 参数

描述

TiDB 不支持 enablePacketDebug 参数，这是一个 MySQL Connector/J 用于调试的参数，将保留数据包的 Buffer。这将导致连接的**意外关闭，请勿**打开。

规避方法

不设置 enablePacketDebug 参数。

15.2.2.5 不支持 UpdatableResultSet

描述

TiDB 暂不支持 UpdatableResultSet，即请勿指定 ResultSet.CONCUR_UPDATABLE 参数，也不要在 ResultSet 内部进行数据更新。

规避方法

使用 UPDATE 语句进行数据更新，可使用事务保证数据一致性。

15.2.3 MySQL JDBC Bug

15.2.3.1 useLocalTransactionState 和 rewriteBatchedStatements 同时开启将导致事务无法提交或回滚

描述

在使用 8.0.32 或以下版本的 MySQL Connector/J 时，同时开启 useLocalTransactionState 和 rewriteBatchedStatements 参数将导致事务无法提交。你可以使用代码复现。

规避方法

> **注意：**
>
> maxPerformance 配置中包含多个参数，其中包括 useLocalSessionState 参数。要查看当前 MySQL Connector/J 中 maxPerformance 包含的具体参数，可参考 MySQL Connector/J 8.0 版本 或 5.1 版本 的配置文件。在使用 maxPerformance 时，请关闭 useLocalTransactionState，即 useConfigs=maxPerformance&useLocalTransactionState=false。

MySQL Connector/J 已在 8.0.33 版本修复此问题。请使用 8.0.33 或更高版本。基于稳定性和性能考量，并结合到 8.0.x 版本已经停止更新，强烈建议升级 MySQL Connector/J 到最新的 GA 版本。

15.2.3.2 Connector 无法兼容 5.7.5 版本以下的服务端

描述

8.0.31 及以下版本 MySQL Connector/J，在与 5.7.5 版本以下的 MySQL 服务端，或使用 5.7.5 版本以下 MySQL 服务端协议的数据库（如 TiDB 6.3.0 版本以下）同时使用时，将在某些情况下导致数据库连接的挂起。关于更多细节信息，可查看此 Bug Report。

规避方法

MySQL Connector/J 已在 8.0.32 版本修复此问题。请使用 8.0.32 或更高版本。基于稳定性和性能考量，并结合到 8.0.x 版本已经停止更新，强烈建议升级 MySQL Connector/J 到最新的 GA 版本。

TiDB 也对其进行了两个维度的修复：

- 客户端方面：**pingcap/mysql-connector-j** 中修复了该 Bug，你可以使用 pingcap/mysql-connector-j 替换官方的 MySQL Connector/J。
- 服务端方面：TiDB v6.3.0 修复了此兼容性问题，你可以升级服务端至 v6.3.0 或以上版本。

15.2.4 与 Sequelize 的兼容性

本小节描述的兼容性信息基于 Sequelize v6.32.1 测试。

根据测试结果，TiDB 支持绝大部分 Sequelize 功能（使用 MySQL 作为方言），不支持的功能有：

- 不支持 GEOMETRY 相关。
- 不支持修改整数主键。
- 不支持 PROCEDURE 相关。
- 不支持 READ-UNCOMMITTED 和 SERIALIZABLE 隔离级别。
- 默认不允许修改列的 AUTO_INCREMENT 属性。

- 不支持 FULLTEXT、HASH 和 SPATIAL 索引。
- 不支持 sequelize.queryInterface.showIndex(Model.tableName);。
- 不支持 sequelize.options.databaseVersion。
- 不支持使用 queryInterface.addColumn 添加外键引用。

15.2.4.1 不支持修改整数主键

描述

不支持修改整数类型的主键，这是由于当主键为整数类型时，TiDB 使用其作为数据组织的索引。你可以在此 Issue 或聚簇索引一节中获取更多信息。

15.2.4.2 不支持 READ-UNCOMMITTED 和 SERIALIZABLE 隔离级别

描述

TiDB 不支持 READ-UNCOMMITTED 和 SERIALIZABLE 隔离级别。设置事务隔离级别为 READ-UNCOMMITTED 或 SERIALIZABLE 时将报错。

规避方法

仅使用 TiDB 支持的 REPEATABLE-READ 或 READ-COMMITTED 隔离级别。

如果你的目的是兼容其他设置 SERIALIZABLE 隔离级别的应用，但不依赖于 SERIALIZABLE，你可以设置 tidb_skip_isolation_level_check 为 1，此后如果对 tx_isolation（transaction_isolation 别名）赋值一个 TiDB 不支持的隔离级别（READ-UNCOMMITTED 和 SERIALIZABLE），不会报错。

15.2.4.3 默认不允许修改列的 AUTO_INCREMENT 属性

描述

默认不允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来增加或移除某个列的 AUTO_INCREMENT 属性。

规避方法

参考 AUTO_INCREMENT 的使用限制。

设置 @@tidb_allow_remove_auto_inc 为 true，即可允许移除 AUTO_INCREMENT 属性。

15.2.4.4 不支持 FULLTEXT、HASH 和 SPATIAL 索引

描述

TiDB 不支持 FULLTEXT、HASH 和 SPATIAL 索引。

15.3 ProxySQL 集成指南

本文简要介绍 ProxySQL，描述如何在开发环境和生产环境中将 ProxySQL 与 TiDB 集成，并通过查询规则的场景展示集成的主要优势。

关于 TiDB 和 ProxySQL 的更多信息，请参考以下文档：

- 平凯数据库开发者指南
- ProxySQL 文档

15.3.1 什么是 ProxySQL?

ProxySQL 是一个高性能的开源 SQL 代理。它具有灵活的架构，可以通过多种方式部署，适合各类使用场景。例如，ProxySQL 可以通过缓存频繁访问的数据来提高性能。

ProxySQL 的设计目标是快速、高效且易于使用。它完全兼容 MySQL，并支持高质量 SQL 代理的所有功能。此外，ProxySQL 还提供了许多独特功能，使其成为各种应用程序的理想选择。

15.3.2 为什么集成 ProxySQL?

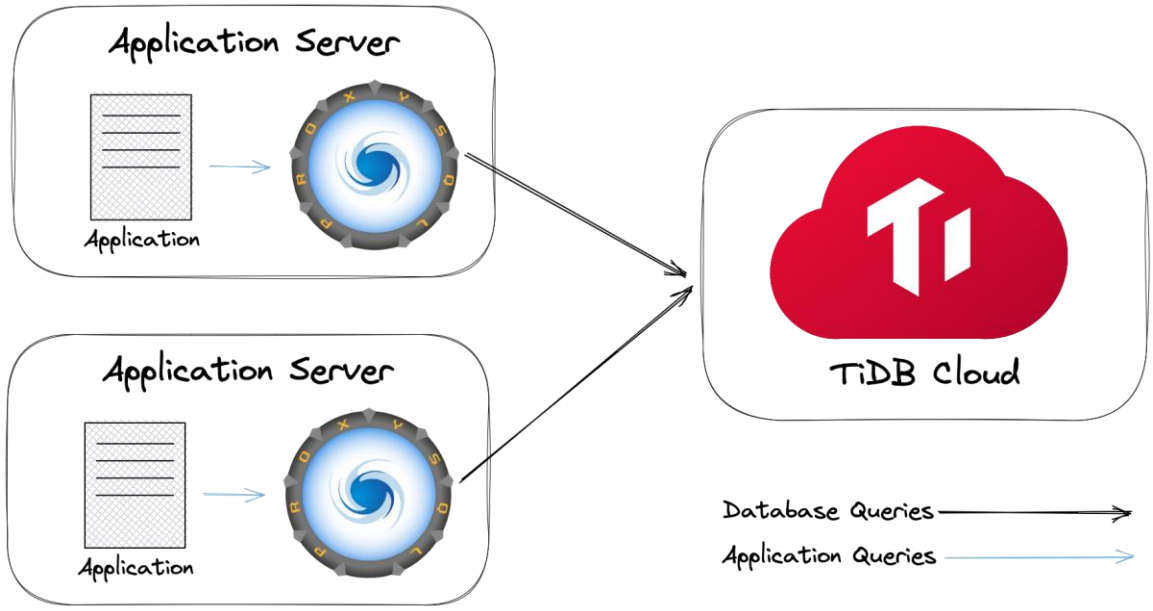
- ProxySQL 可以通过降低与 TiDB 交互的延迟来提升应用程序性能。无论你构建什么，无论是使用 Lambda 等无服务器函数的可扩展应用程序（其工

作负载不确定并且可能激增），还是构建执行大量数据查询的应用程序，都可以利用 ProxySQL 的强大功能（例如连接池和缓存常用查询）。

- ProxySQL 可以作为应用程序安全防护的附加层，使用查询规则防止 SQL 漏洞（例如 SQL 注入）。
- 由于 ProxySQL 和 TiDB 都是开源项目，你可以享受到零供应商锁定的好处。

15.3.3 部署架构

将 ProxySQL 与 TiDB 集成的最直接方式是在应用层和 TiDB 之间添加 ProxySQL 作为独立中介。但是，这种方式无法保证可扩展性和容错性，而且可能因为网络跳转而增加延迟。为避免这些问题，一种替代部署架构是将 ProxySQL 作为附属容器部署，如下图所示：



proxysql-client-side-tidb-cloud

注意：

上图仅供参考，你需要根据实际的部署架构进行调整。

15.3.4 开发环境

本节介绍如何在开发环境中将 TiDB 与 ProxySQL 集成。在满足前提条件的情况下，你可以集成本地部署的 TiDB 与 ProxySQL。

15.3.4.1 前提条件

根据选择的方案，你可能需要以下依赖：

- Git
- Docker
- Python 3
- Docker Compose
- MySQL Client

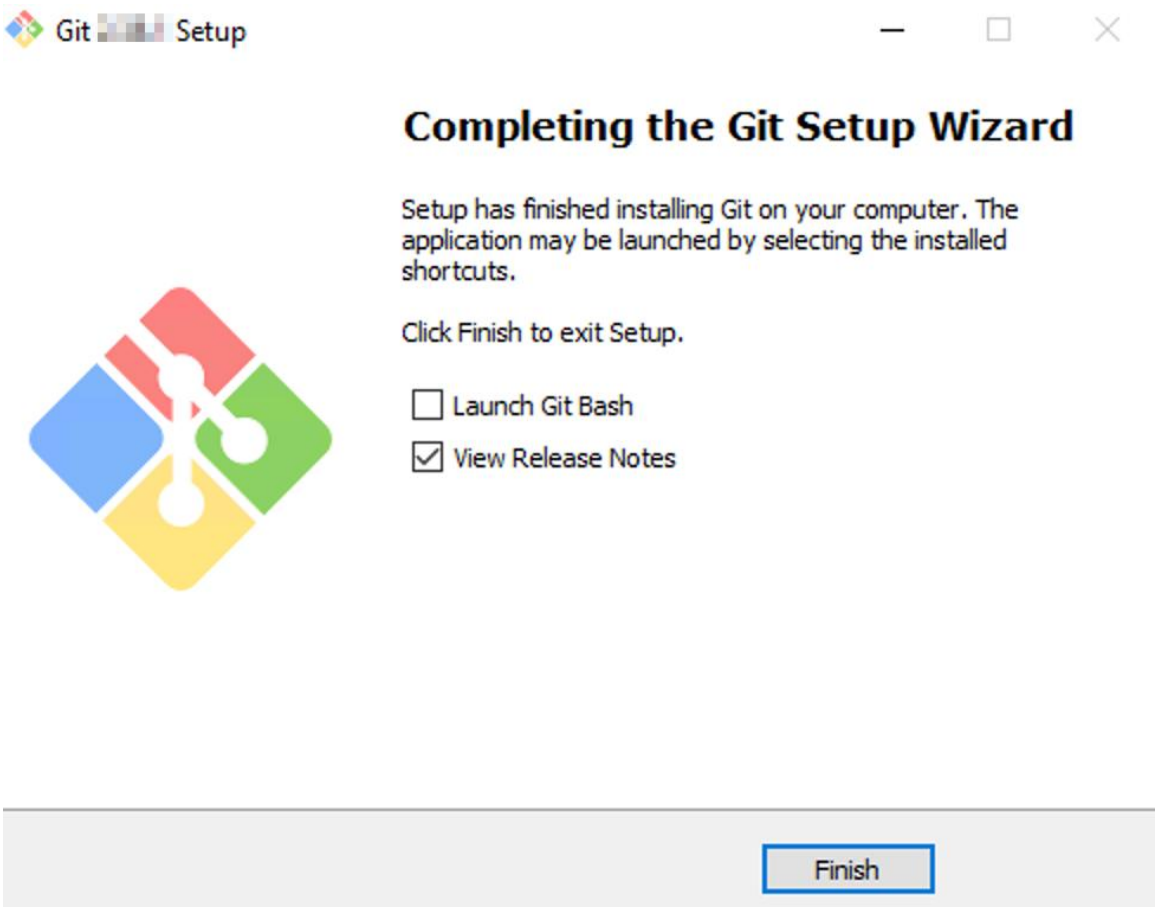
你可以按照下面的说明进行安装：

1. 下载并启动 Docker，其中 Docker Desktop 已包含 Docker Compose。
2. 运行以下命令安装 Python 和 mysql-client：

   ```
   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
   brew install python mysql-client
   ```

```
curl -fsSL https://get.docker.com | bash -s docker
yum install -y git python39 docker-ce docker-ce-cli containerd.io docker-compose-plugin mysql
systemctl start docker
```

- 下载并安装 Git。
  1. 从 Download for Windows 页面下载 **64-bit Git for Windows Setup** 安装程序。
  2. 按照安装向导提示安装 Git。你可以多次点击 **Next** 使用默认的安装设置。



proxysql-windows-git-install

- 下载并安装 MySQL Shell。

    1. 从 MySQL Community Server Download 页面下载 MySQL Installer 的 ZIP 文件。

    2. 解压文件，并在 bin 文件夹中找到 mysql.exe。你需要将该 bin 文件夹的路径添加到系统变量中，并在 Git Bash 中将其设置到 PATH 变量中。

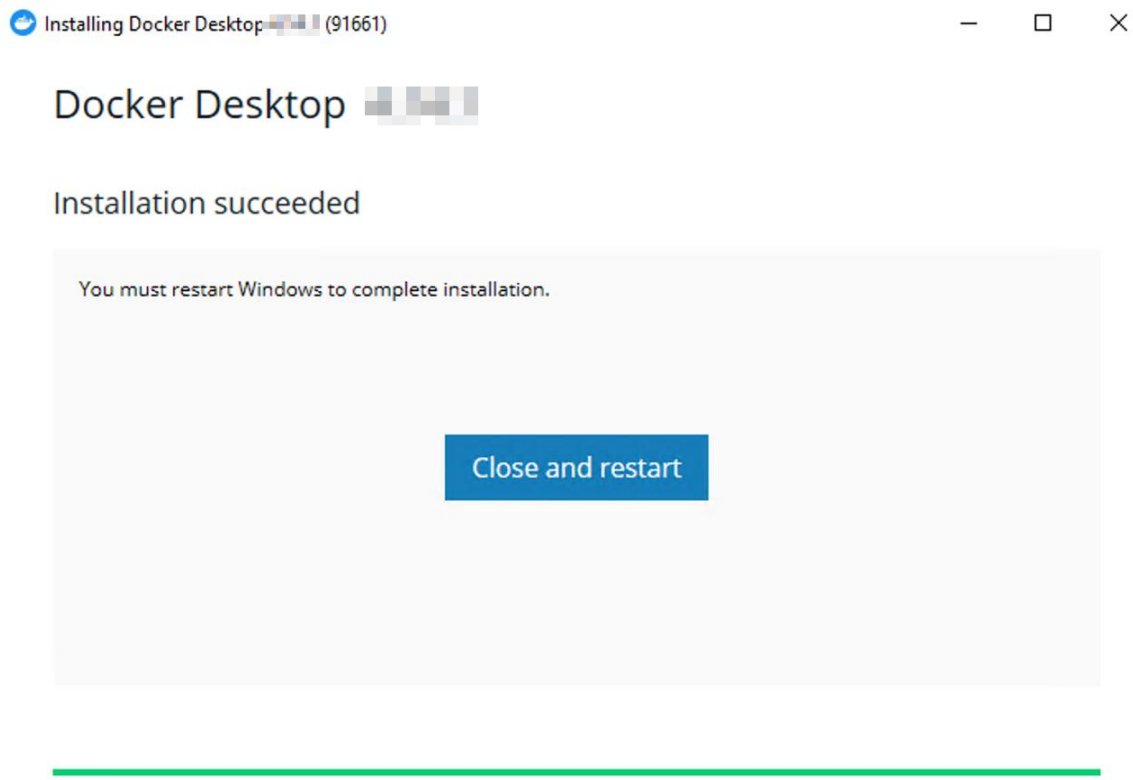
        ```
        echo 'export PATH="(your bin folder)":$PATH' >>~/.bash_profile
        source ~/.bash_profile
        ```

        例如：

```
echo 'export PATH="/c/Program Files (x86)/mysql-8.0.31-winx64/bin":$PA
TH' >>~/.bash_profile
source ~/.bash_profile
```

- 下载并安装 Docker。

    1. 从 Docker Download 页面下载 Docker Desktop 安装程序。
    2. 双击安装程序运行。安装完成后，会提示你重新启动。



proxysql-windows-docker-install

- 从 Python Download 页面下载最新版的 Python 3 安装程序并运行。

15.3.4.2 集成本地部署的平凯数据库与 ProxySQL

在这个集成中，你将使用 TiDB 和 ProxySQL 的 Docker 镜像设置环境。你也可以尝试其他方式安装 TiDB。

下面的步骤将在端口 6033 和 4000 上分别设置 ProxySQL 和 TiDB，请确保这些端口可用。

1. 启动 Docker。如果 Docker 已经启动，请跳过此步骤：

   双击已安装的 Docker 的图标来启动它。

   systemctl start docker

   双击已安装的 Docker 的图标来启动它。

2. 克隆 TiDB 和 ProxySQL 的集成示例代码仓库 pingcap-inc/tidb-proxysql-integration：

   ```
   git clone https://github.com/pingcap-inc/tidb-proxysql-integration.git
   ```

   ```
   git clone https://github.com/pingcap-inc/tidb-proxysql-integration.git
   ```
   ```
   git clone https://github.com/pingcap-inc/tidb-proxysql-integration.git
   ```


3. 拉取 ProxySQL 和 TiDB 的最新镜像：

   ```
   cd tidb-proxysql-integration && docker compose pull
   ```

   ```
   cd tidb-proxysql-integration && docker compose pull
   ```
   ```
   cd tidb-proxysql-integration && docker compose pull
   ```


4. 使用 TiDB 和 ProxySQL 容器启动一个集成环境：

```
docker compose up -d
```

```
docker compose up -d
```

```
docker compose up -d
```

你可以使用 root 用户名及空密码登录到 ProxySQL 的 6033 端口。

5. 通过 ProxySQL 连接到 TiDB：

```
mysql -u root -h 127.0.0.1 -P 6033
```

```
mysql -u root -h 127.0.0.1 -P 6033
```

```
mysql -u root -h 127.0.0.1 -P 6033
```

6. 连接 TiDB 集群后，可以使用以下 SQL 语句验证连接：

```
SELECT VERSION();
```

如果输出了 TiDB 的版本信息，则表示你已经成功通过 ProxySQL 连接到 TiDB 集群。

7. 要停止和删除容器，并返回上一个目录，运行以下命令：

```
docker compose down
cd -
```

```
docker compose down
cd -
```

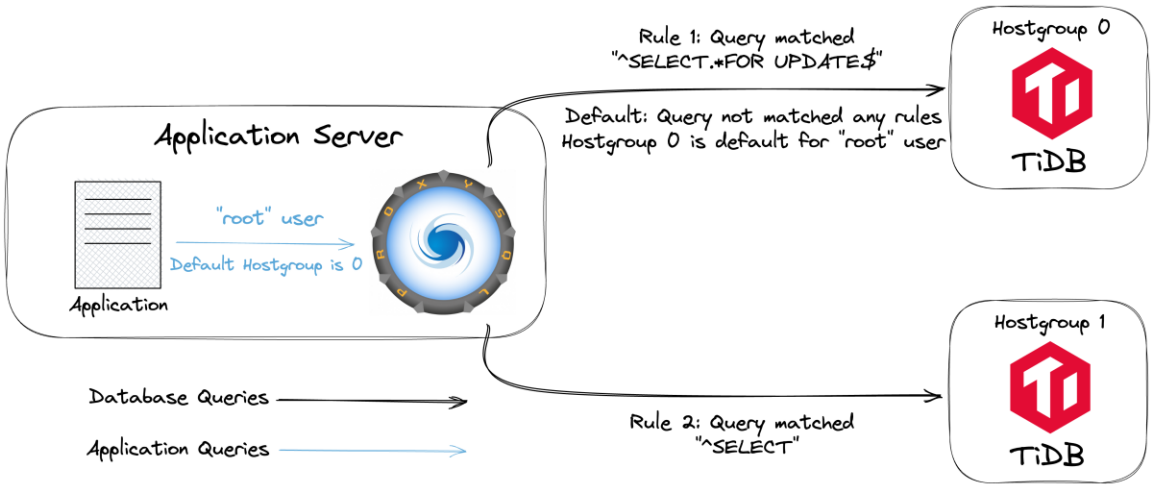
```
docker compose down
cd -
```

15.3.5 典型场景

本节以查询规则为例，介绍集成 TiDB 与 ProxySQL 能带来的一些优势。

15.3.5.1 查询规则

数据库可能会因为高流量、错误代码或恶意攻击而过载。因此，审核 SQL 是必要的。使用 ProxySQL 的查询规则，你可以有效地应对这些问题，例如通过重路由、改写 SQL 或者拒绝查询等方式。



proxysql-client-side-rules

注意：

以下步骤使用 TiDB 和 ProxySQL 的容器镜像配置查询规则。如果你还没有拉取这些镜像，请参考集成本地部署的 TiDB 与 ProxySQL 部分的详细步骤。

1. 克隆 TiDB 和 ProxySQL 的集成示例代码仓库 pingcap-inc/tidb-proxysql-integration。如果你已经在前面的步骤中克隆了它，请跳过这一步。

```
git clone https://github.com/pingcap-inc/tidb-proxysql-integration.git
git clone https://github.com/pingcap-inc/tidb-proxysql-integration.git
git clone https://github.com/pingcap-inc/tidb-proxysql-integration.git
```

2. 进入 ProxySQL 查询规则的示例目录：

```
cd tidb-proxysql-integration/example/proxy-rule-admin-interface
```

```
cd tidb-proxysql-integration/example/proxy-rule-admin-interface
```
```
cd tidb-proxysql-integration/example/proxy-rule-admin-interface
```


3. 运行下面的命令启动两个 TiDB 容器和一个 ProxySQL 容器：

```
docker compose up -d
```

```
docker compose up -d
```
```
docker compose up -d
```


如果运行成功，以下容器将被启动：

- 两个 Docker 容器的 TiDB 集群，端口分别为 4001 和 4002
- 一个 Docker 容器的 ProxySQL，端口为 6034

4. 在两个 TiDB 容器中，使用 mysql 创建一个具有相同 schema 的表，然后插入不同的数据 ('tidb-server01-port-4001', 'tidb-server02-port-4002') 以区分这两个容器。

```
mysql -u root -h 127.0.0.1 -P 4001 << EOF
DROP TABLE IF EXISTS test.tidb_server;
CREATE TABLE test.tidb_server (server_name VARCHAR(255));
INSERT INTO test.tidb_server (server_name) VALUES ('tidb-server01-port-4001');
EOF

mysql -u root -h 127.0.0.1 -P 4002 << EOF
```

```
DROP TABLE IF EXISTS test.tidb_server;
CREATE TABLE test.tidb_server (server_name VARCHAR(255));
INSERT INTO test.tidb_server (server_name) VALUES ('tidb-server02-port-4002');
EOF

mysql -u root -h 127.0.0.1 -P 4001 << EOF
DROP TABLE IF EXISTS test.tidb_server;
CREATE TABLE test.tidb_server (server_name VARCHAR(255));
INSERT INTO test.tidb_server (server_name) VALUES ('tidb-server01-port-4001');
EOF

mysql -u root -h 127.0.0.1 -P 4002 << EOF
DROP TABLE IF EXISTS test.tidb_server;
CREATE TABLE test.tidb_server (server_name VARCHAR(255));
INSERT INTO test.tidb_server (server_name) VALUES ('tidb-server02-port-4002');
EOF

mysql -u root -h 127.0.0.1 -P 4001 << EOF
DROP TABLE IF EXISTS test.tidb_server;
CREATE TABLE test.tidb_server (server_name VARCHAR(255));
INSERT INTO test.tidb_server (server_name) VALUES ('tidb-server01-port-4001');
EOF

mysql -u root -h 127.0.0.1 -P 4002 << EOF
DROP TABLE IF EXISTS test.tidb_server;
CREATE TABLE test.tidb_server (server_name VARCHAR(255));
INSERT INTO test.tidb_server (server_name) VALUES ('tidb-server02-port-4002');
EOF
```

5. 运行下面的命令配置 ProxySQL，该命令会在 ProxySQL Admin Interface 中执行 proxysql-prepare.sql，从而在 TiDB 容器和 ProxySQL 之间建立一个代理连接。

```
docker compose exec proxysql sh -c "mysql -uadmin -padmin -h127.0.0.1 -P6032
 < ./proxysql-prepare.sql"

docker compose exec proxysql sh -c "mysql -uadmin -padmin -h127.0.0.1 -P6032
 < ./proxysql-prepare.sql"
```

```
docker compose exec proxysql sh -c "mysql -uadmin -padmin -h127.0.0.1 -P6032
 < ./proxysql-prepare.sql"
```

注意：

proxysql-prepare.sql 脚本完成以下操作：

- 在 ProxySQL 中添加 TiDB 集群，hostgroup_id 分别为 0 和 1。
- 添加一个用户 root，密码为空，并设置 default_hostgroup 为 0。
- 添加规则 ^SELECT.*FOR UPDATE$，rule_id 为 1，destination_hostgroup 为 0。这代表如果一个 SQL 语句与此规则相匹配，该请求将被转发到 hostgroup 为 0 的 TiDB 集群。
- 添加规则 ^SELECT，rule_id 为 2，destination_hostgroup 为 1。这代表如果一个 SQL 语句与此规则相匹配，该请求将被转发到 hostgroup 为 1 的 TiDB 集群。

为了更好地理解此处的配置流程，强烈建议查看 proxysql-prepare.sql 文件。关于 ProxySQL 配置的更多信息，参考 ProxySQL 文档。

下面是关于 ProxySQL 匹配 SQL 查询的规则的一些补充信息：

- ProxySQL 尝试按照 rule_id 的升序逐一匹配规则。
- 规则中的 ^ 符号用于匹配 SQL 语句的开头，$ 符号用于匹配语句的结尾。

关于 ProxySQL 正则表达式和模式匹配的更多信息，参考 ProxySQL 文档 mysql-query_processor_regex。

关于完整的参数列表，参考 ProxySQL 文档 mysql_query_rules。

6. 验证配置并检查查询规则是否有效。

    1. 使用 root 用户登录 ProxySQL MySQL Interface：

```
mysql -u root -h 127.0.0.1 -P 6034
mysql -u root -h 127.0.0.1 -P 6034
mysql -u root -h 127.0.0.1 -P 6034
```

2. 执行以下 SQL 语句：

   - 执行一个 SELECT 语句：

     ```
     SELECT * FROM test.tidb_server;
     ```

     这个语句将匹配 rule_id 为 2 的规则，因此将转发语句到 hostgroup 为 1 上的 TiDB 集群中。

   - 执行一个 SELECT ... FOR UPDATE 语句：

     ```
     SELECT * FROM test.tidb_server FOR UPDATE;
     ```

     这个语句将匹配 rule_id 为 1 的规则，因此将转发语句到 hostgroup 为 0 上的 TiDB 集群中。

   - 启动一个事务：

     ```
     BEGIN;
     INSERT INTO test.tidb_server (server_name) VALUES ('insert this a
     nd rollback later');
     SELECT * FROM test.tidb_server;
     ROLLBACK;
     ```

     在这个事务中，BEGIN 语句将不会匹配任何规则。因此，它将使用默认的 hostgroup（在这个例子中为 hostgroup 0）。因为 ProxySQL 默认启用了用户 transaction_persistent，它将在同一事务中，将所有语句都转发至相同的 hostgroup，所以 INSERT 和 SELECT * FROM test.tidb_server; 语句也将被转发到 hostgroup 为 0 的 TiDB 集群。

下面是一个输出示例。如果你得到类似的输出，表示你已经成功配置了 ProxySQL 的查询规则。

```
+-----------------------+
| server_name           |
+-----------------------+
| tidb-server02-port-4002 |
+-----------------------+
+-----------------------+
| server_name           |
+-----------------------+
| tidb-server01-port-4001 |
+-----------------------+
+------------------------------+
| server_name                  |
+------------------------------+
| tidb-server01-port-4001      |
| insert this and rollback later |
+------------------------------+
```

3. 如需退出 MySQL 客户端，输入 quit 并按下 Enter 键。

7. 要停止和删除容器，并返回上一个目录，运行以下命令：

```
docker compose down
cd -

docker compose down
cd -

docker compose down
cd -
```