

# Table of Contents

Introduction	1.1
推荐序	1.2
作者名单	1.3

## 第一部分 TiDB 原理和特性

1 TiDB 整体架构	2.1
2 说存储	2.2
3 谈计算	2.3
3.1 关系模型到 Key-Value 模型的映射	2.3.1
3.2 元信息管理	2.3.2
3.3 SQL 层简介	2.3.3
4 讲调度	2.4
4.1 调度概述	2.4.1
4.2 弹性调度	2.4.2
5 TiDB 和 MySQL 的区别	2.5
6 TiDB 事务模型	2.6
6.1 乐观事务	2.6.1
6.2 悲观事务	2.6.2
6.3 4.0 的大事务支持	2.6.3
7 TiDB DDL	2.7
7.1 表结构设计最佳实践	2.7.1
7.2 如何查看 DDL 状态	2.7.2
7.3 Sequence	2.7.3
7.4 AutoRandom	2.7.4
8 Titan 简介与实战	2.8
8.1 Titan 原理介绍	2.8.1
8.2 在 TiDB 集群中开启 Titan	2.8.2
9 TiFlash 简介与 HTAP 实战	2.9
9.1 TiDB HTAP 的特点	2.9.1
9.2 TiFlash 架构与原理	2.9.2
9.3 TiFlash 的使用	2.9.3
10 TiDB 安全	2.10
10.1 权限管理	2.10.1
10.2 RBAC	2.10.2
10.3 证书管理与数据加密	2.10.3
11 TiSpark 简介与实战	2.11
11.1 TiSpark 架构与原理	2.11.1
11.2 TiSpark 的使用	2.11.2

---

11.3 TiSpark on TiFlash	2.11.3
11.4 TiSpark 结合大数据体系	2.11.4

---

## 第二部分 系统安装部署与管理

1 部署安装 & 常规运维	3.1
1.1 TiUP	3.1.1
1.1.1 TiUP 简介	3.1.1.1
1.1.2 用 TiUP 部署本地测试环境	3.1.1.2
1.1.3 TiUP cluster 简介	3.1.1.3
1.1.4 TiUP cluster 部署生产环境集群	3.1.1.4
1.2 TiDB on Kubernetes	3.1.2
1.2.1 TiDB-Operator 简介及原理	3.1.2.1
1.2.2 TiDB-Operator 部署本地测试集群（基于 Kind）	3.1.2.2
1.2.3 用 TiDB-Operator 部署生产环境集群	3.1.2.3
1.2.3.1 公有云部署	3.1.2.3.1
1.2.3.1.1 在 AWS EKS 上部署 TiDB 集群	3.1.2.3.1.1
1.2.3.1.2 在 GCP GKE 上部署 TiDB 集群	3.1.2.3.1.2
1.2.3.1.3 在阿里云上部署 TiDB 集群	3.1.2.3.1.3
1.2.3.1.4 在京东云上部署 TiDB 集群	3.1.2.3.1.4
1.2.3.2 私有云部署	3.1.2.3.2
1.2.3.2.1 集群环境、资源需求	3.1.2.3.2.1
1.2.3.2.2 PV 配置	3.1.2.3.2.2
1.2.3.2.3 部署 TiDB Operator	3.1.2.3.2.3
1.2.3.2.4 部署 TiDB 集群	3.1.2.3.2.4
1.2.3.2.5 节点维护	3.1.2.3.2.5
1.2.3.2.6 删除 TiDB 集群	3.1.2.3.2.6
1.2.4 访问 Kubernetes 上的 TiDB 集群及其监控	3.1.2.4
1.2.5 在 Kubernetes 集群上使用 BR 备份、恢复 TiDB 集群	3.1.2.5
1.2.6 在 Kubernetes 集群上使用 Lightning 导入数据	3.1.2.6
1.2.7 在 Kubernetes 集群上使用 TiDB 工具指南	3.1.2.7
1.2.8 TiDB-Operator 升级	3.1.2.8
1.3 集群扩容缩容	3.1.3
1.3.1 基于 TiUP cluster 的集群扩缩容	3.1.3.1
1.3.2 基于 TiDB-Operator 的集群扩缩容	3.1.3.2
1.4 集群版本升级	3.1.4
1.4.1 基于 TiUp cluster 的集群滚动更新	3.1.4.1
1.4.2 基于 TiDB-Operator 的集群滚动更新	3.1.4.2
1.5 如何做动态配置修改	3.1.5
2 TiDB 备份恢复和导入导出工具	3.2
2.1 4.0 增量数据订阅 CDC	3.2.1
2.1.1 CDC 解决什么问题	3.2.1.1

---

---

2.1.2 CDC 工作原理	3.2.1.2
2.1.3 CDC 实操指南	3.2.1.3
2.2 TiDB 数据导入工具 Lightning	3.2.2
2.2.1 Lightning 工作原理	3.2.2.1
2.2.2 Lightning 实操指南	3.2.2.2
2.3 4.0 分布式备份恢复工具 BR	3.2.3
2.3.1 BR 工作原理	3.2.3.1
2.3.2 BR 实操指南	3.2.3.2
2.4 4.0 分布式导出工具 Dumpling	3.2.4
2.4.1 Dumpling 工作原理	3.2.4.1
2.4.2 Dumpling 实操指南	3.2.4.2

## 第三部分 TiDB Troubleshooting 指南与工具

1 SQL 调优原理	4.1
1.1 TiDB 执行计划概览	4.1.1
1.2 优化器简介	4.1.2
1.3 SQL Plan Management	4.1.3
1.4 参数调优指南	4.1.4
1.5 限制 SQL 内存使用和执行时间	4.1.5
2 TiDB Dashboard	4.2
2.1 识别集群热点和业务模式	4.2.1
2.2 分析 SQL 执行性能	4.2.2
2.3 生成集群诊断报告	4.2.3
2.4 日志搜索和导出	4.2.4
2.5 分析组件 CPU 消耗情况	4.2.5
3 诊断系统表	4.3
3.1 集群信息表	4.3.1
3.2 监控表	4.3.2
3.3 诊断结果表	4.3.3
3.4 监控汇总表	4.3.4
3.5 SQL 慢查询内存表	4.3.5
3.6 Processlist	4.3.6
3.7 Statement Summary	4.3.7
4 TiDB 集群监控与报警	4.4
4.1 性能调优地图	4.4.1
4.2 TiDB 读写流程相关监控原理解析	4.4.2
4.4 Prometheus 使用指南	4.4.3
5 灾难快速恢复	4.5
5.1 利用 GC 快照读恢复数据	4.5.1
5.2 利用 Recover/Flashback 命令秒恢复误删表	4.5.2
5.3 多数副本丢失数据恢复指南	4.5.3

# 第四部分 TiDB 最佳实践

1 适用场景介绍	5.1
2 硬件选型规划	5.2
3 常见性能压测	5.3
3.1 Sysbench 基准性能测试	5.3.1
3.2 TPC-C 基准性能测试	5.3.2
4 跨数据中心方案	5.4
4.1 两中心异步复制方案 (binlog 复制)	5.4.1
4.2 两中心同步复制方案 (三副本 Raft)	5.4.2
4.3 两地三中心	5.4.3
4.4 AWS 跨 AZ 部署 TiDB	5.4.4
5 数据迁移方案	5.5
5.1 MySQL 到 TiDB (DM)	5.5.1
5.1.1 DM 同步单机 MySQL 到 TiDB 的实践	5.5.1.1
5.1.2 DM 同步分库分表 MySQL 到 TiDB 的实践	5.5.1.2
5.2 Oracle 到 TiDB (OGG)	5.5.2
5.3 SqlServer 到 TiDB	5.5.3
5.4 SqlServer 到 TiDB (DATAx)	5.5.4
5.5 DB2 到 TiDB (CDC)	5.5.5
5.6 TiDB 到 TiDB (DATAx)	5.5.6
5.7 Mongodob 迁移到 TiDB	5.5.7
6 业务适配最佳实践	5.6
6.1 业务开发最佳实践	5.6.1
6.1.1 乐观锁模式下的事务最佳实践	5.6.1.1
6.1.2 TiDB 中事务限制及应对方案	5.6.1.2
6.1.3 高并发的唯一序列号生成方案	5.6.1.3
6.1.4 一种高效分页批处理方案	5.6.1.4
6.1.5 通过 hint 调整执行计划	5.6.1.5
6.2 SQL 调优案例	5.6.2
6.3 TiDB + TiSpark 跑批最佳实践	5.6.3
6.4 分区表实践及问题处理	5.6.4
6.4.1 TiDB分区表简介	5.6.4.1
6.4.2 TiDB分区表使用场景	5.6.4.2
6.4.3 TiDB分区表最佳实践	5.6.4.3
6.4.4 TiDB分区表问题处理	5.6.4.4
6.5 TiDB 在企业数据分级存储中的应用实践	5.6.5
6.6 TiDB 与 HBase、ES、Druid 的数据交互实战	5.6.6
6.7 TiDB 与可视化展现 Saiku、Grafana 的集成应用	5.6.7
7 常见问题处理思路	5.7

---

7.1 Oncall 地图	5.7.1
7.2 热点问题处理思路	5.7.2
7.3 TiKV is busy 处理思路	5.7.3
7.4 TiDB OOM 的常见原因	5.7.4
7.5 TiKV 磁盘空间占用与回收常见问题	5.7.5
<b>8 TiDB 调优指南</b>	<b>5.8</b>
8.1 TiDB 常见配置优化	5.8.1
8.2 TiKV 常见配置优化	5.8.2
8.2.1 TiKV 线程池优化	5.8.2.1
8.2.2 海量 Region 集群调优	5.8.2.2
8.2.3 其他常见优化设置	5.8.2.3
8.3 添加索引调优加速	5.8.3
8.3.1 TiDB 增加索引原理	5.8.3.1
8.3.2 动态调整新增索引速度	5.8.3.2

## 第五部分 如何参与 TiDB 社区及周边生态

<b>1 TiDB 开源社区历史及其现状</b>	<b>6.1</b>
1.1 TiDB 开源社区现状及发展简史	6.1.1
1.2 TiDB 开源生态介绍	6.1.2
1.3 TiDB 开源社区治理	6.1.3
1.4 TiDB 开源社区重要合作开发	6.1.4
<b>2 TiDB 开源生态</b>	<b>6.2</b>
2.1 社区重要活动介绍	6.2.1
2.1.1 TiDB Devcon	6.2.1.1
2.1.2 TiDB TechDay	6.2.1.2
2.1.3 Infra Meetup	6.2.1.3
2.1.4 TiDB Hackathon	6.2.1.4
2.1.5 TUG 企业行	6.2.1.5
2.2 TUG (TiDB User Group)	6.2.2
2.3 Talent Plan	6.2.3
2.4 Challenge Program	6.2.4
2.5 PingCAP Incubator	6.2.5
2.6 PingCAP University	6.2.6
2.7 AskTUG	6.2.7
2.8 Contributor Map	6.2.8

## 附录

<b>专用术语解释</b>	<b>7.1</b>
---------------	------------



# Gitbook

Read it: [TiDB In Action: based on 4.0](#)

Download PDF: [TiDB In Action: based on 4.0](#)

## 如何贡献

贡献指引：

- 从 [Project](#) 找到感兴趣的模块
- 在具体模块的 TODO 列表中选择一个感兴趣的任务。
- 阅读并更新内容
- 将内容提交 Pull Request

选题参考目录：[目录](#)

- 文章内容格式采用 [markdown](#).
- [Github 简易入门指南](#)

## 图片目录

图片存放目录与文章存放一一对应，图片存放目录：`res/{doc-path}/`

- 其中 `{doc-path}` 为对应文章路径。

如例：

- 文章存放路径：`session1/chapter1/tidb-intro.md`
- 对应图片存放目录为：`/res/session1/chapter1/tidb-intro/`
- 图片路径对应到 markdown 里为：`![1.png] (/res/session1/chapter1/tidb-intro/1.png)`

## TiDB in Action 写作规则

《TiDB in Action》是一本重视实操的书

- 工具的介绍和使用部分会深入浅出，浅显易懂，这一部分可以当作工具书来查阅。
- 原理与实现部分会相对言简意赅，意在帮助读者能够理解原理，从而更好地使用 TiDB。但无需陷入具体实现细节中，这部分读者浅尝辄止即可。

## 好的例子

### 关于特性和产品介绍

- 介绍工具的简单原理
- 介绍工具如何使用（常见的参数，常见的应用场景）
- 给出一两个简单的例子

### 关于最佳实践

- 力求用文字或图清晰按照时间轴描述整个过程
  - 事前准备

- 事中操作流程
- 事后效果检验
- 轻原理，重实操

## 不好的例子

- 贴原理实现的代码
- 直接复制粘贴案例，但是无操作流程
- 对工具只介绍，不给例子
- 选择的例子是个特例（不可复现）

## 写作规范

一千个作者一千个写作习惯，本书作者百家，文风也是百家争鸣，百花齐放，文笔各有千秋。因此我们对文章内容只有一个标准 -- 清晰易懂

建议读者在阅读过程中，修正后的內容需要：

- 标题能够突出本章重点，言简意赅，忌：词不达意，
- 内容上下文衔接顺畅，逻辑通顺，忌：
  - 技术点错误
  - 内容描述不清晰的，语句不通
  - 内容缺失，缺乏过渡
  - 章节缺少开头、总结

此外，如果您有地方读不懂，想要修改又无从下手，请在內容附近使用 markdown 惯用的隐藏注释提问：

```
<!-- TODO: 这段看不懂/这里为什么是这样。-->
```

## License

Shield:  CC BY-SA 4.0

This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#).





# 作者名单

TiDB 4.0 发布在即，在 Master 分支上可以看到社区也是一片热火朝天的景象。而 TiDB 4.0 版本也是一个具有里程碑意义的版本。社区一直以来就有一个遗憾，这两年 TiDB 的社区日益壮大，已经有无数的用户将 TiDB 使用了起来，但是市面上一直没有一本关于 TiDB 的技术书籍。这背后其实也有很多原因，例如产品的迭代速度比较快，另外很多时候通过在线文档和社区答疑，问题也就解决了。所以 TiDB 的第一本书搁浅至今。不过，在 3 月的一个周末，这个心愿终于实现了，而且是用一种特殊的分布式方式。PingCAP 召集了 102 位 社区深度用户，48 小时之内写完这本关于 TiDB 的技术书籍。这本书对于一个社区想学习和使用 TiDB 的新人来说，算是一本入门书籍，系统完整且简明扼要地介绍一下整个 TiDB 的使用和周边工具生态。内容丰富并且实用，是聚集了 102 位用户的心血与精华。同时，为了保证书籍的质量，PingCAP 也联合了内外部 9 名专家，耗时近 1 个月，对书籍的内容进行了审校。下面是作者以及审校专家的列表。

## 社区作者列表

### 第一部分作者

白雪、陈书宁、董红亮、杜振强、高海涛、黄东旭、季朋、李宋高、李振环、刘筠松、罗瑞星、潘迪、彭鑫、宋翎宇、孙晓光、徐嘉靖、童智高、王传义、王军、王文安、谢腾进、王聪、薛超、羊欢、张雯、赵磊、郑智辉、Qiannan、zhiqiangxu、李霞

### 第二部分作者

曹贺、程威、洪超、黄靓、李红、李银龙、刘春辉、刘春雷、刘伟、刘宇、栾成、罗瑞星、邱文辉、宋歌、我不叫大脸猫、杨非、杨文、张广超、张海龙、周帅、高恺迪、郭大瑞、沈泰宁、尹亮

### 第三部分作者

陈付、陈霜、陈子军、郭倪、黄潇、季朋、李宋高、李迅、刘玮、龙恒、罗霞、倪健、石壹笑、谭清如、唐明华、王志广、薛超、张明、朱博帅、max、Qiannan

### 第四部分作者

白鳝、北丐、沈均、代晓磊、杜蓉、樊一蒙、高林、胡盼盼、黄蔚、黄潇、冀浩东、贾世闻、李坤、李宋高、李仲舒、谭仁刚、刘浩然、路思勇、吕磊、聂泽峯、潘博存、秦天爽、于沛涛、唐希元、王君轶、王轲、王伦伟、王新宇、王英杰、魏巍、张帆、张雯、郑俊博、郑智辉、朱博帅、申海龙、GeorgeLi、吴剑锋、Shinno、郭大瑞、庄培培、陶政、高振娇、戚铮

### 第五部分作者

崔秋、房晓乐、李莎莎、彭琴、荣毅龙、唐刘、唐小丽、童牧、王琳琳、杨可奥、姚维、叶奔、殷成文、余梦杰、张婵、张建、张金鹏

## 审校专家列表

- 王兵，寰信通科技有限公司技术部副总监，有十余年技术服务经验，70 后技术老兵，PCTP 首批官方认证专家。
- 刘春辉，Shopee 数据库运维团队负责人，TiDB 的早期深度用户。
- 覃左言，小米云平台部门存储平台总监，长期专注于分布式存储和数据库领域，对开源有很高的热情。
- 吴林宁，上海某银行资深架构师，多年银行核心业务场景架构经验，尤其擅长数据库及数据交互相关模块的架构设计。
- 神秘商业银行资深技术专家
- 张雯，B 站 DBA，TiDB 早期深度用户，主导开发 B 站数据库管理平台 Akso，支持 MariaDB & TiDB 方案及落地。
- 李凯，美团点评数据库团队研究员，目前从事数据库运维管理和分布式数据库研发工作
- 吴镝，字节跳动分布式 KV，表格存储负责人，长期专注于分布式数据库，NoSQL 领域

- 杜川，腾讯云高级工程师，TiDB Committer。对分布式系统与数据库领域有浓厚的兴趣，目前主要从事数据库云平台相关发开工作。

## PingCAP 审校人员

唐刘 唐小丽 崔秋 刘博 戚铮 张婵 张建 张明 张海龙 张金鹏 栾成 房晓乐 聂泽峯 陈书宁 李仲舒 李坤 李宋高 李莎莎 李霞 余梦杰 博康 荣毅龙 汤博文 杨可奥 马晓宇 彭琴 赵磊 陶政 庄培培 沈泰宁 王琳琳 王硕 宋翔宇 王聪 王轲 龙恒 殷成文 金文涛 金灵 高振娇 付磊 姚维 童智高 童牧 郭大瑞 叶奔 齐智 周跃跃 王贤净 王军 崔国文 孙浩

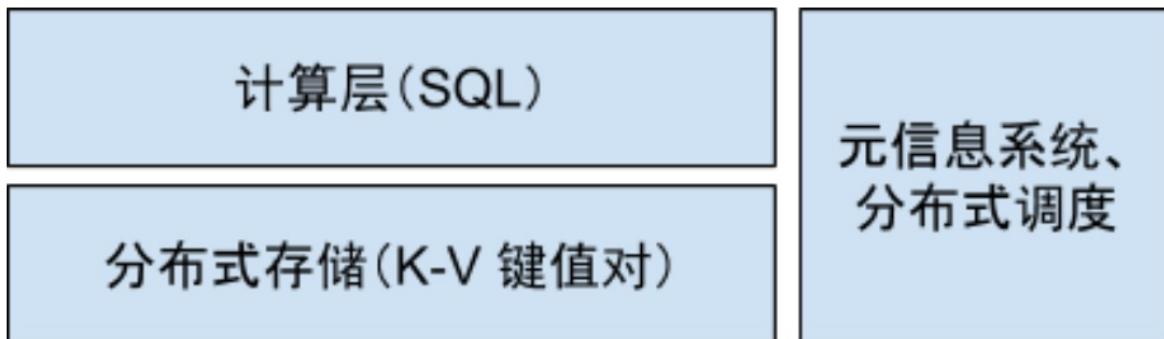
# 第1章 TiDB 整体架构

近年来，随着移动互联网、云计算、大数据和人工智能等技术的飞速发展，给各行业带来了深刻的影响和变革，使得企业的数据量越来越庞大，应用的规模也越来越复杂。在这个背景之下，传统的单机数据库已经在很多场景下表现的力不从心，为了解决海量数据平台的扩展性的问题，TiDB 分布式数据库应运而生。 TiDB 是当今开源 NewSQL 数据库领域的代表产品之一，相比传统的单机数据库，TiDB 有以下的一些优势：

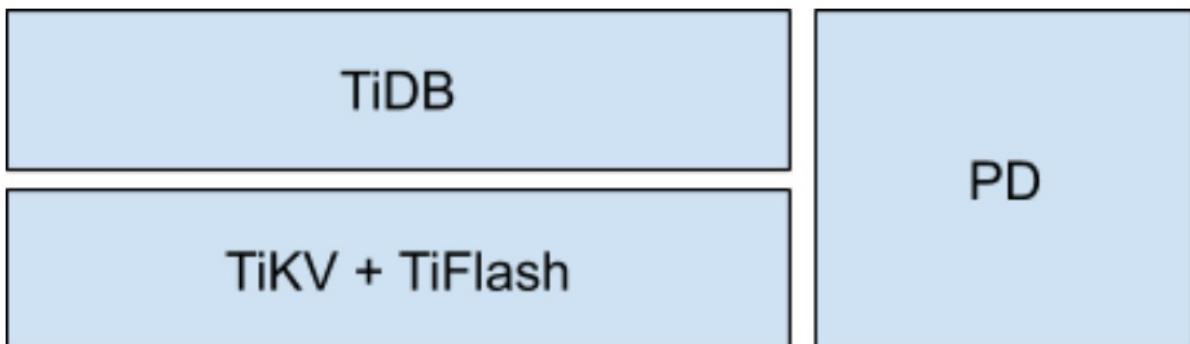
- 纯分布式架构，拥有良好的扩展性，支持弹性的扩缩容
- 支持 SQL，对外暴露 MySQL 的网络协议，并兼容大多数 MySQL 的语法，在大多数场景下可以直接替换 MySQL
- 默认支持高可用，在少数副本失效的情况下，数据库本身能够自动进行数据修复和故障转移，对业务透明
- 支持 ACID 事务，对于一些有强一致需求的场景友好，例如：银行转账
- 具有丰富的工具链生态，覆盖数据迁移、同步、备份等多种场景

本书会专注于 TiDB 4.0 的实操与最佳实践，详细介绍 TiDB 的使用和一些相关的原理。

TiDB 分布式数据库最初的设计受到 Google 内部开发的知名分布式数据库 Spanner 和 F1 的启发，在内核设计上将整体的架构拆分成多个大的模块，大的模块之间互相通信，组成完整的 TiDB 系统。大的架构如下：



这三个大模块相互通信，每个模块都是分布式的架构，在 TiDB 中，对应的这几个模块叫做：



1. TiDB (tidb-server, <https://github.com/pingcap/tidb>)：SQL 层，对外暴露 MySQL 协议的连接 endpoint，负责接受客户端的连接，执行 SQL 解析和优化，最终生成分布式执行计划。TiDB 层本身是无状态的，实践中可以启动多个 TiDB 实例，客户端的连接可以均匀的分摊在多个 TiDB 实例上以达到负载均衡的效果。tidb-server 本身并不存储数据，只是解析 SQL，将实际的数据读取请求转发给底层的存储层 TiKV。
2. TiKV (tikv-server, <https://github.com/pingcap/tikv>)：分布式 KV 存储，类似 NoSQL 数据库，作为 TiDB 的默认分布式存储引擎，支持完全弹性的扩容和缩容，数据分布在多个 TiKV 存储节点中，系统会动态且自动地进行均衡，绝大多数情况下不需要人工介入。与普通的 NoSQL 系统不一样的是，TiKV 的 API 能够在 KV 键值对层面提供对分布式事务的原生支持，默认提供了 SI (Snapshot Isolation) 的隔离级别，这也是 TiDB 在 SQL 层面支持分布式事务的核心，上面提到的 TiDB SQL 层做完 SQL 解析后，会将 SQL 的执行计划转换为实际对 TiKV API 的调用。所以实际上数据都是存储在 TiKV

中。另外，TiKV 中的数据都会自动维护多副本（默认为 3）, 天然支持高可用和自动故障转移。TiFlash 是一类特殊的存储节点，和普通 TiKV 节点不一样的是，在 TiFlash 内部，数据是以列式的形式进行存储，主要的功能是为分析型的场景加速。后面的章节会详细介绍。

3. Placement Driver (pd-server, 简称 PD, <https://github.com/pingcap/pd>): 整个 TiDB 集群的元信息管理模块，负责存储每个 TiKV 节点实时的数据分布情况和集群的整体拓扑结构，提供 Dashboard 管控界面，并为分布式事务分配事务 ID。PD 不仅仅是单纯的元信息存储，同时 PD 会根据 TiKV 节点实时上报的数据分布状态，下发数据调度命令给具体的 TiKV 节点，可以说是整个集群的「大脑」，另外 PD 本身也是由至少 3 个对等节点构成，拥有高可用的能力。

## 第2章 说存储

在前一节中，我们介绍了 TiDB 项目的几个主要的组成部分，本节向大家介绍一下 TiKV 的一些设计思想和关键概念。

### 2.1 Key-Value Pairs (键值对)

作为保存数据的系统，首先要决定的是数据的存储模型，也就是数据以什么样的形式保存下来。TiKV 的选择是 Key-Value 模型，并且提供有序遍历方法。TiKV 数据存储的两个关键点：

1. 这是一个巨大的 Map（可以类比一下 C++ 的 std::map），也就是存储的是 Key-Value Pairs（键值对）
2. 这个 Map 中的 Key-Value pair 按照 Key 的二进制顺序有序，也就是可以 Seek 到某一个 Key 的位置，然后不断地调用 Next 方法以递增的顺序获取比这个 Key 大的 Key-Value。

有人可能会问，这里讲的存储模型和 SQL 中表是什么关系？在这里有一件重要的事情需要强调：

TiKV 的 KV 存储模型和 SQL 中的 Table 无关！

现在让我们忘记 SQL 中的任何概念，专注于讨论如何实现 TiKV 这样一个高性能、高可靠性、分布式的 Key-Value 存储。

### 2.2 本地存储 (RocksDB)

任何持久化的存储引擎，数据终归要保存在磁盘上，TiKV 也不例外。但是 TiKV 没有选择直接向磁盘上写数据，而是把数据保存在 RocksDB 中，具体的数据落地由 RocksDB 负责。这个选择的原因是开发一个单机存储引擎工作量很大，特别是要做一个高性能的单机引擎，需要做各种细致的优化，而 RocksDB 是由 Facebook 开源的一个非常优秀的单机 KV 存储引擎，可以满足 TiKV 对单机引擎的各种要求。这里可以简单的认为 RocksDB 是一个单机的持久化 Key-Value Map。

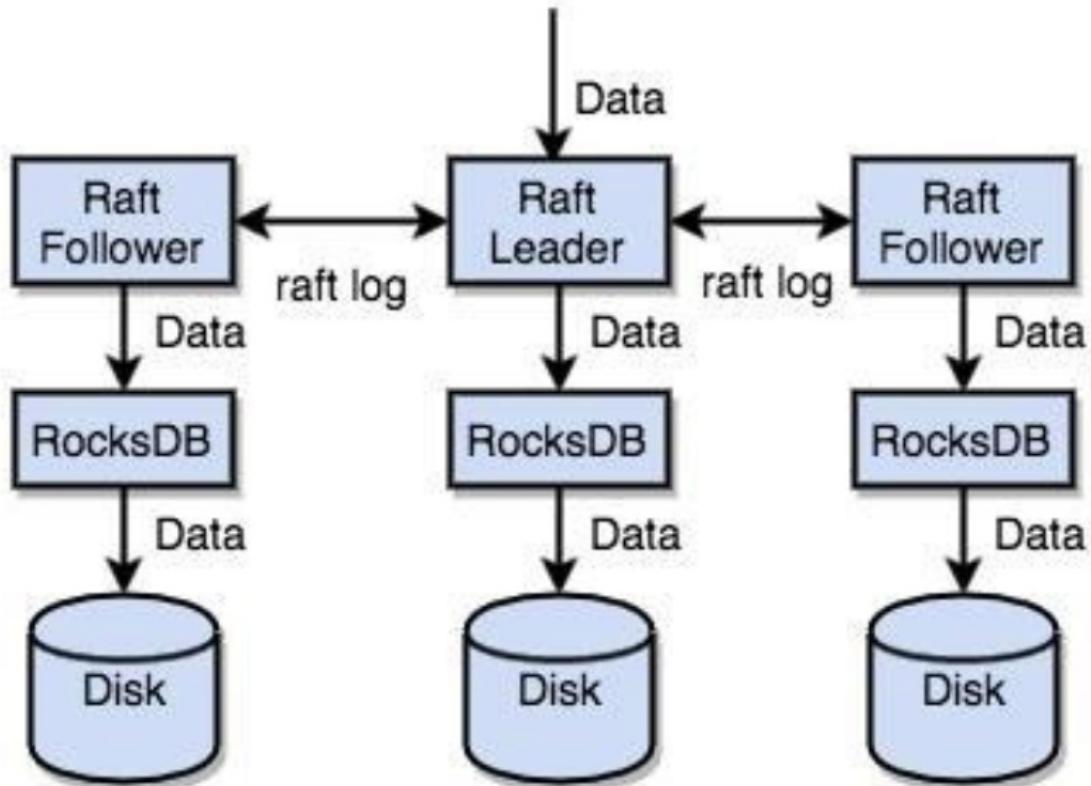
### 2.3 Raft 协议

接下来 TiKV 的实现面临一件更难的事情：如何保证单机失效的情况下，数据不丢失，不出错？

简单来说，需要想办法把数据复制到多台机器上，这样一台机器挂了，其他的机器上的副本还能提供服务；复杂来说，还需要这个数据复制方案是可靠和高效的，并且能处理副本失效的情况。TiKV 选择了 Raft 算法。Raft 是一个一致性协议，它和 Multi Paxos 实现一样的功能，但是更加易于理解。[这里](#) 是 Raft 的论文，感兴趣的可以看一下。本书只会对 Raft 做一个简要的介绍，细节问题可以参考论文。Raft 提供几个重要的功能：

1. Leader (主副本) 选举
2. 成员变更 (如添加副本、删除副本、转移 Leader 等操作)
3. 日志复制

TiKV 利用 Raft 来做数据复制，每个数据变更都会落地为一条 Raft 日志，通过 Raft 的日志复制功能，将数据安全可靠地同步到复制组的每一个节点中。不过在实际写入中，根据 Raft 的协议，只需要同步复制到多数节点，即可安全地认为数据写入成功。



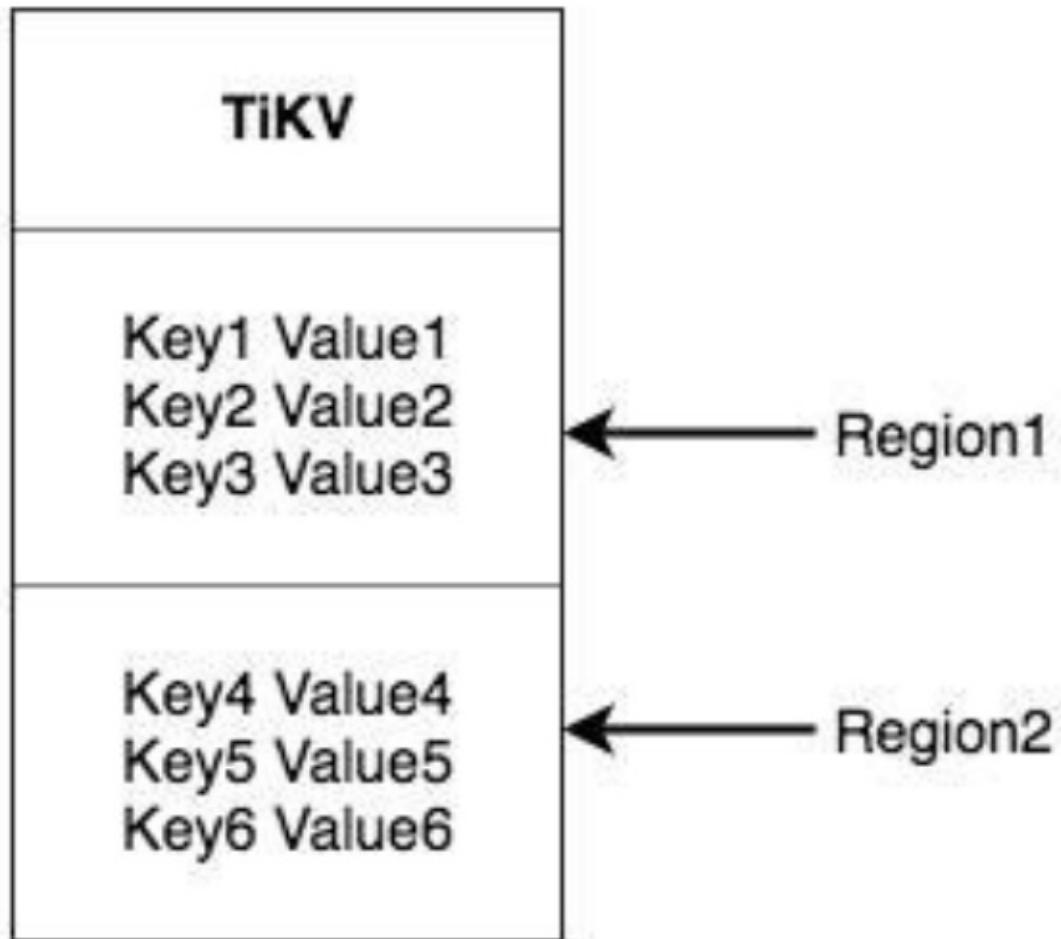
总结一下，通过单机的 RocksDB，TiKV 可以将数据快速地存储在磁盘上；通过 Raft，将数据复制到多台机器上，以防单机失效。数据的写入是通过 Raft 这一层的接口写入，而不是直接写 RocksDB。通过实现 Raft，TiKV 变成了一个分布式的 Key-Value 存储，少数几台机器宕机也能通过原生的 Raft 协议自动把副本补全，可以做到对业务无感知。

## 2.4 Region

讲到这里，我们需要提到一个非常重要的概念：Region。这个概念是理解后续一系列机制的基础，请仔细阅读这一小节。前面提到，我们将 TiKV 看做一个巨大的有序的 KV Map，那么为了实现存储的水平扩展，我们需要将数据分散在多台机器上。这里提到的数据分散在多台机器上和 Raft 的数据复制不是一个概念，在这一节我们先忘记 Raft，假设所有的数据都只有一个副本，这样更容易理解。对于一个 KV 系统，将数据分散在多台机器上有两种比较典型的方案：

- Hash：按照 Key 做 Hash，根据 Hash 值选择对应的存储节点
- Range：按照 Key 分 Range，某一段连续的 Key 都保存在一个存储节点上

TiKV 选择了第二种方式，将整个 Key-Value 空间分成很多段，每一段是一系列连续的 Key，将每一段叫做一个 Region，并且会尽量保持每个 Region 中保存的数据不超过一定的大小，目前在 TiKV 中默认是 96MB。每一个 Region 都可以用 [StartKey, EndKey) 这样一个左闭右开区间来描述。

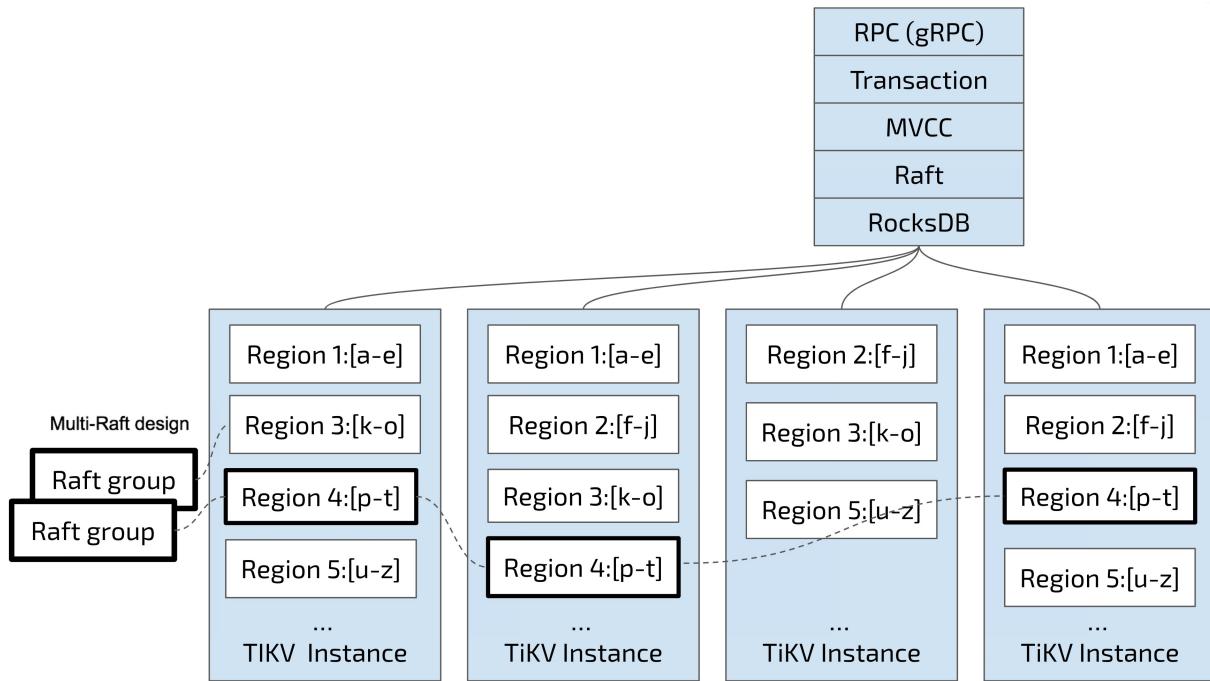


注意，这里的 Region 还是和 SQL 中的表没什么关系！请各位继续忘记 SQL，只谈 KV。将数据划分成 Region 后，TiKV 将会做两件重要的事情：

- 以 Region 为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的 Region 数量差不多
- 以 Region 为单位做 Raft 的复制和成员管理

这两点非常重要，我们一点一点来说。先看第一点，数据按照 Key 切分成很多 Region，每个 Region 的数据只会保存在一个节点上面（暂不考虑多副本）。TiDB 系统会有一个组件（PD）来负责将 Region 尽可能均匀的散布在集群中所有的节点上，这样一方面实现了存储容量的水平扩展（增加新的节点后，会自动将其他节点上的 Region 调度过来），另一方面也实现了负载均衡（不会出现某个节点有很多数据，其他节点上没什么数据的情况）。同时为了保证上层客户端能够访问所需要的数据，系统中也会有一个组件（PD）记录 Region 在节点上面的分布情况，也就是通过任意一个 Key 就能查询到这个 Key 在哪个 Region 中，以及这个 Region 目前在哪个节点上（即 Key 的位置路由信息）。至于负责这两项重要工作的组件（PD），会在后续介绍。

对于第二点，TiKV 是以 Region 为单位做数据的复制，也就是一个 Region 的数据会保存多个副本，TiKV 将每一个副本叫做一个 Replica。Replica 之间是通过 Raft 来保持数据的一致，一个 Region 的多个 Replica 会保存在不同的节点上，构成一个 Raft Group。其中一个 Replica 会作为这个 Group 的 Leader，其他的 Replica 作为 Follower。所有的读和写都是通过 Leader 进行，读操作在 Leader 上即可完成，而写操作再由 Leader 复制给 Follower。大家理解了 Region 之后，应该可以理解下面这张图：



以 Region 为单位做数据的分散和复制，就有了一个分布式的具备一定容灾能力的 KeyValue 系统，不用再担心数据存不下，或者是磁盘故障丢失数据的问题。

## 2.5 MVCC

很多数据库都会实现多版本并发控制（MVCC），TiKV 也不例外。设想这样的场景，两个客户端同时去修改一个 Key 的 Value，如果没有数据的多版本控制，就需要对数据上锁，在分布式场景下，可能会带来性能以及死锁问题。TiKV 的 MVCC 实现是通过在 Key 后面添加版本号来实现，简单来说，没有 MVCC 之前，可以把 TiKV 看做这样的：

```
Key1 -> Value
Key2 -> Value
...
KeyN -> Value
```

有了 MVCC 之后，TiKV 的 Key 排列是这样的：

```
Key1_Version3 -> Value
Key1_Version2 -> Value
Key1_Version1 -> Value
...
Key2_Version4 -> Value
Key2_Version3 -> Value
Key2_Version2 -> Value
Key2_Version1 -> Value
...
KeyN_Version2 -> Value
KeyN_Version1 -> Value
....
```

注意，对于同一个 Key 的多个版本，我们把版本号较大的放在前面，版本号小的放在后面（回忆一下 Key-Value 一节我们介绍过的 Key 是有序的排列），这样当用户通过一个 Key + Version 来获取 Value 的时候，可以通过 Key 和 Version 构造出 MVCC 的 Key，也就是 Key\_Version。然后可以直接通过 RocksDB 的 SeekPrefix(Key\_Version) API，定位到第一个大于等于这个 Key\_Version 的位置。

## 2.6 分布式 ACID 事务

TiKV 的事务采用的是 Google 在 BigTable 中使用的事务模型：[Percolator](#)，TiKV 根据这篇论文实现，并做了大量的优化。这个在后续的章节中会有详细的介绍。

在 TiKV 层的事务 API 的语义类似下面的伪代码：

```
tx = tikv.Begin()
tx.Set(Key1, Value1)
tx.Set(Key2, Value2)
tx.Set(Key3, Value3)
tx.Commit()
```

这个事务中包含3条 Set 操作，TiKV 能保证这些操作要么全部成功，要么全部失败，不会出现中间状态或脏数据。就如前面提到的，TiDB 的 SQL 层会将 SQL 的执行计划转换成多个 KV 操作，对于上层的同一个业务层的 SQL 事务，在底层也是对应一个 KV 层的事务，这是 TiDB 实现 MySQL 的事务语义的关键。

## 第3章 谈计算

作为一个优秀的 NewSQL 数据库，TiDB 在 TiKV 提供的分布式存储能力基础上，构建了兼具优异的交易处理能力与良好的数据分析能力的计算引擎。本章首先从数据映射算法入手揭秘 TiDB 如何将库表中的数据映射到 TiKV 中的 (Key, Value) 键值对，然后描述了 TiDB 元信息管理方式。在此基础上，本章最后一节介绍了 TiDB SQL 层的主要架构。需要注意的是，对于计算层依赖的存储方案，本章只介绍了基于 TiKV 的行存储结构。针对分析型业务的特点，TiDB 推出了作为 TiKV 扩展的列存储方案 TiFlash。关于 TiFlash 的特点和设计细节，本书[TiFlash 简介与 HTAP 实战](#)一章有详细介绍，这里就不再赘述了。

## 3.1 表数据与 Key-Value 的映射关系

本小节为大家介绍 TiDB 中数据到 (Key, Value) 键值对的映射方案。这里的数据主要包括两个方面：

1. 表中每一行的数据，以下简称表数据
2. 表中所有索引的数据，以下简称索引数据

下面分别对这两个方面进行介绍。

### 3.1.1 表数据与 Key-Value 的映射关系

在关系型数据库中，一个表可能有很多列。要将一行中各列数据映射成一个 (Key, Value) 键值对，需要考虑如何构造 Key。首先，OLTP 场景下有大量针对单行或者多行的增、删、改、查等操作，要求数据库具备快速读取一行数据的能力。因此，对应的 Key 最好有一个唯一 ID（显示或隐式的 ID），以方便快速定位。其次，很多 OLAP 型查询需要进行全表扫描。如果能够将一个表中所有行的 Key 编码到一个区间内，就可以通过范围查询高效完成全表扫描的任务。基于上述考虑：

1. 为了保证同一个表的数据放在一起，方便查找，TiDB 会为每个表分配一个表 ID，用 `TableID` 表示。表 ID 是一个整数，在整个集群内唯一。
2. TiDB 会为表中每行数据分配一个行 ID，用 `RowID` 表示。行 ID 也是一个整数，在表内唯一。对于行 ID，TiDB 做了一个小优化，如果某个表有整型的主键，TiDB 会使用主键的值当做这一行数据的行 ID。

每行数据按照如下规则编码成 (Key, Value) 键值对：

```
Key:  tablePrefix{TableID}_recordPrefixSep{RowID}
Value: [col1, col2, col3, col4]
```

其中 `tablePrefix` 和 `recordPrefixSep` 都是特定的字符串常量，用于在 Key 空间内区分其他数据。其具体值在后面的小结中给出。

### 3.1.2 索引数据和 Key-Value 的映射关系

TiDB 同时支持主键和二级索引（包括唯一索引和非唯一索引）。与表数据映射方案类似，TiDB 为表中每个索引分配了一个索引 ID，用 `IndexID` 表示。

对于主键和唯一索引，我们需要根据键值快速定位到对应的 RowID，因此，按照如下规则编码成 (Key, Value) 键值对：

```
Key:  tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue
Value: RowID
```

对于不需要满足唯一性约束的普通二级索引，一个键值可能对应多行，我们需要根据键值范围查询对应的 RowID。因此，按照如下规则编码成 (Key, Value) 键值对：

```
Key:  tablePrefix{TableID}_indexPrefixSep{IndexID}_indexedColumnsValue_{RowID}
Value: null
```

### 3.1.3 映射关系小结

最后，上述所有编码规则中的 `tablePrefix`，`recordPrefixSep` 和 `indexPrefixSep` 都是字符串常量，用于在 Key 空间内区分其他数据，定义如下：

```
tablePrefix      = []byte{'t'}
recordPrefixSep = []byte{'r'}
indexPrefixSep  = []byte{'i'}
```

另外请注意，上述方案中，无论是表数据还是索引数据的 Key 编码方案，一个表内所有的行都有相同的 Key 前缀，一个索引的所有数据也都有相同的前缀。这样具有相同的前缀的数据，在 TiKV 的 Key 空间内，是排列在一起的。因此只要小心地设计后缀部分的编码方案，保证编码前和编码后的比较关系不变，就可以将表数据或者索引数据有序地保存在 TiKV 中。采用这种编码后，一个表的所有行数据会按照 `RowID` 顺序地排列在 TiKV 的 Key 空间中，某一个索引的数据也会按照索引数据的具体的值（编码方案中的 `indexedColumnsValue`）顺序地排列在 Key 空间内。

### 3.1.4 Key-Value 映射关系的一个例子

最后通过一个简单的例子，来理解 TiDB 的 Key-Value 映射关系。假设 TiDB 中有如下这个表：

```
CREATE TABLE User {
    ID int,
    Name varchar(20),
    Role varchar(20),
    Age int,
    PRIMARY KEY (ID),
    KEY idxAge (Age)
};
```

假设该表中有 3 行数据：

```
1, "TiDB", "SQL Layer", 10
2, "TiKV", "KV Engine", 20
3, "PD", "Manager", 30
```

首先每行数据都会映射为一个 (Key, Value) 键值对，同时该表有一个 `int` 类型的主键，所以 `RowID` 的值即为该主键的值。假设该表的 `TableID` 为 10，则其存储在 TiKV 上的表数据为：

```
t10_r1 --> ["TiDB", "SQL Layer", 10]
t10_r2 --> ["TiKV", "KV Engine", 20]
t10_r3 --> ["PD", "Manager", 30]
```

除了主键外，该表还有一个非唯一的普通二级索引 `idxAge`，假设这个索引的 `IndexID` 为 1，则其存储在 TiKV 上的索引数据为：

```
t10_i1_10_1 --> null
t10_i1_20_2 --> null
t10_i1_30_3 --> null
```

希望通过上面的例子，读者可以更好的理解 TiDB 中关系模型到 Key-Value 模型的映射规则以及选择该方案背后的考量。

## 3.2 元信息管理

上节介绍了表中的数据和索引如何映射为 (Key, Value) 键值对，本节介绍一下元信息的存储。TiDB 中每个 `Database` 和 `Table` 都有元信息，也就是其定义以及各项属性。这些信息也需要持久化，TiDB 将这些信息也存储在了 TiKV 中。

每个 `Database` / `Table` 都被分配了一个唯一的 ID，这个 ID 作为唯一标识，并且在编码为 Key-Value 时，这个 ID 都会编码到 Key 中，再加上 `_m_` 前缀。这样可以构造出一个 Key, Value 中存储的是序列化后的元信息。

除此之外，TiDB 还用一个专门的 (Key, Value) 键值对存储当前所有表结构信息的最新版本号。这个键值对是全局的，每次 DDL 操作的状态改变时其版本号都会加1。目前，TiDB 把这个键值对存放在 pd-server 内置的 etcd 中，其 Key 为`"/tidb/ddl/global_schema_version"`，Value 是类型为 int64 的版本号值。TiDB 使用 Google F1 的 Online Schema 变更算法，有一个后台线程在不断的检查 etcd 中存储的表结构信息的版本号是否发生变化，并且保证在一定时间内一定能够获取版本的变化。

## 3.3 SQL 层简介

TiDB 的 SQL 层，即 `tidb-server`，跟 Google 的 F1 比较类似，负责将 SQL 翻译成 Key-Value 操作，将其转发给共用的分布式 Key-Value 存储层 TiKV，然后组装 TiKV 返回的结果，最终将查询结果返回给客户端。

这一层的节点都是无状态的，节点本身并不存储数据，节点之间完全对等。

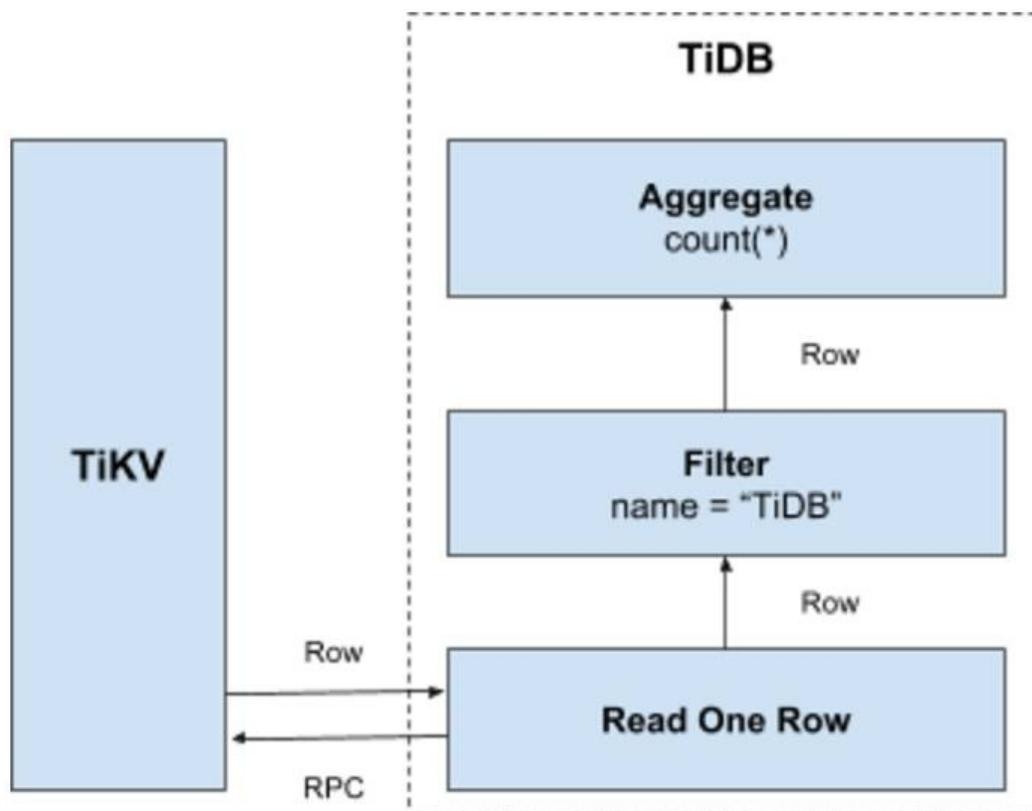
### 3.3.1 SQL 运算

能想到的最简单的方案就是通过上一节所述的 [表中所有数据和 Key-Value 的映射关系](#) 映射方案，将 SQL 查询映射为对 KV 的查询，再通过 KV 接口获取对应的数据，最后执行各种计算。

比如 `select count(*) from user where name = "TiDB"` 这样一个语句，我们需要读取表中所有的数据，然后检查 `name` 字段是否是 `TiDB`，如果是的话，则返回这一行。具体流程是：

1. 构造出 Key Range：一个表中所有的 `RowID` 都在 `[0, MaxInt64)` 这个范围内，那么我们用 `0` 和 `MaxInt64` 根据行数据的 `key` 编码规则，就能构造出一个 `[StartKey, EndKey)` 的左闭右开区间
2. 扫描 Key Range：根据上面构造出的 Key Range，读取 TiKV 中的数据
3. 过滤数据：对于读到的每一行数据，计算 `name = "TiDB"` 这个表达式，如果为真，则向上返回这一行，否则丢弃这一行数据
4. 计算 `Count(*)`：对符合要求的每一行，累计到 `count(*)` 的结果上面

整个流程示意图如下：



这个方案肯定是可以 Work 的，但是并不能 Work 的很好，原因是显而易见的：

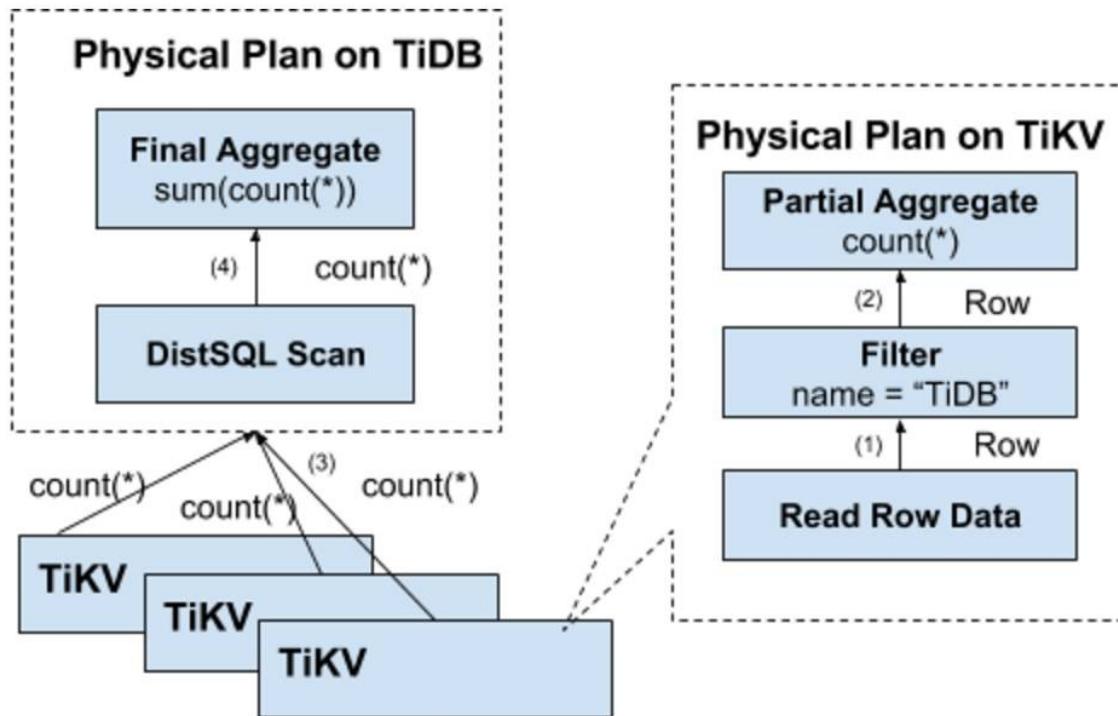
1. 在扫描数据的时候，每一行都要通过 KV 操作从 TiKV 中读取出来，至少有一次 RPC 开销，如果需要扫描的数据很多，那么这个开销会非常大
2. 并不是所有的行都有用，如果不满足条件，其实可以不读取出来

3. 符合要求的行的值并没有什么意义，实际上这里只需要有几行数据这个信息就行

### 3.3.2 分布式 SQL 运算

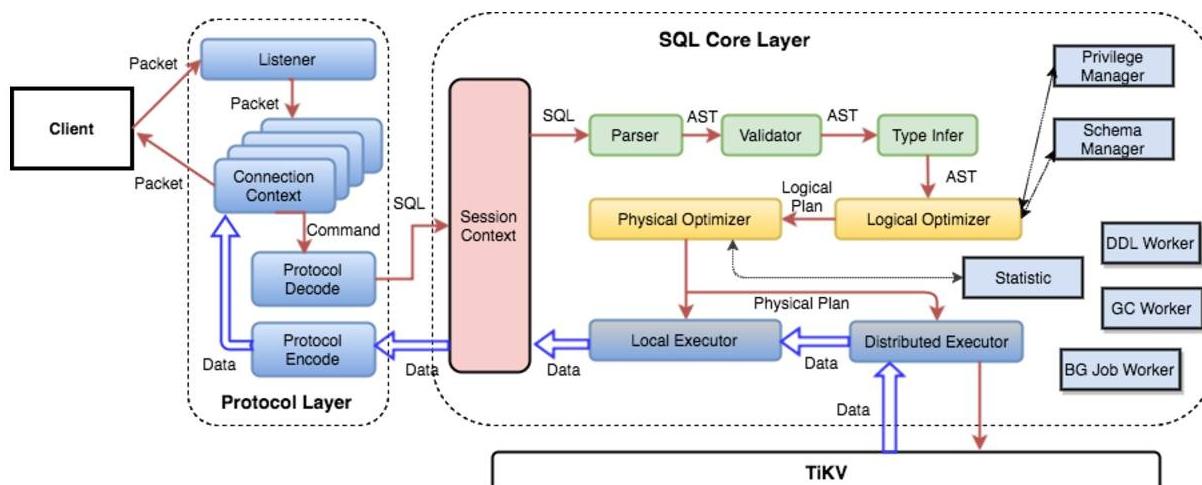
如何避免上述缺陷也是显而易见的，我们需要将计算尽量靠近存储节点，以避免大量的 RPC 调用。首先，我们需要将 SQL 中的谓词条件下推到存储节点进行计算，这样只需要返回有效的行，避免无意义的网络传输。然后，我们还可以将聚合函数 `Count(*)` 也下推到存储节点，进行预聚合，每个节点只需要返回一个 `Count(*)` 的结果即可，再由 SQL 层将各个节点返回的 `Count(*)` 的结果累加求和。

这里有一个数据逐层返回的示意图：



### 3.3.3 SQL 层架构

通过上面的例子，希望大家对 SQL 语句的处理有一个基本的了解。实际上 TiDB 的 SQL 层要复杂的多，模块以及层次非常多，下面这个图列出了重要的模块以及调用关系：



用户的 SQL 请求会直接或者通过 Load Balancer 发送到 tidb-server，tidb-server 会解析 MySQL Protocol Packet，获取请求内容，对 SQL 进行语法解析和语义分析，制定和优化查询计划，执行查询计划并获取和处理数据。数据全部存储在 TiKV 集群中，所以在这个过程中 tidb-server 需要和 TiKV 交互，获取数据。最后 tidb-server 需要将查询结果返回给用户。



## 第4章 讲调度

任何一个复杂的系统，用户感知到的都只是冰山一角，数据库也不例外。前面两个章节介绍了 TiKV、TiDB 的基本概念以及一些核心功能的实现原理，这两个组件一个负责 KV 存储，一个负责 SQL 引擎，都是大家看得见的东西。在这两个组件的后面，还有一个叫做 PD (Placement Driver) 的组件，它虽然不直接和业务接触，但是这个组件是整个集群的核心，负责全局元信息的存储以及 TiKV 集群负载均衡调度，本章第一节将为大家介绍 PD 的运行调度机制。同时伴随着云环境的兴起，自适应的弹性调度能力也越来越重要，TiDB 4.0 作为一个全面拥抱云的版本，能够在云上对业务进行自适应调节，本章第二节将为大家介绍 TiDB 4.0 在弹性调度方面的一些新的特性。

通过本章的描述，能够让大家对于 TiDB 的调度机制有更清晰的了解。

## 4.1 调度概述

本章介绍一下 PD 这个神秘的模块。这部分比较复杂，很多东西大家平时不会想到，也很少在其他文章中见到类似东西的描述。我们还是按照前两篇的思路，先讲我们需要什么样的功能，再讲我们如何实现，大家带着需求去看实现，会更容易的理解我们做这些设计时背后的考量。

### 4.1.1 为什么要进行调度

TiKV 集群是 TiDB 数据库的分布式 KV 存储引擎，数据以 Region 为单位进行复制和管理，每个 Region 会有多个 Replica（副本），这些 Replica 会分布在不同的 TiKV 节点上，其中 Leader 负责读/写，Follower 负责同步 Leader 发来的 raft log。了解了这些信息后，请思考下面这些问题：

- 如何保证同一个 Region 的多个 Replica 分布在不同的节点上？更进一步，如果在一台机器上启动多个 TiKV 实例，会有什么问题？
- TiKV 集群进行跨机房部署的时候，如何保证一个机房掉线，不会丢失 Raft Group 的多个 Replica？
- 添加一个节点进入 TiKV 集群之后，如何将集群中其他节点上的数据搬过来？
- 当一个节点掉线时，会出现什么问题？整个集群需要做什么事情？
  - 从节点的恢复时间来看
    - 如果节点只是短暂掉线（重启服务），如何处理？
    - 如果节点是长时间掉线（磁盘故障，数据全部丢失），如何处理？
  - 假设集群需要每个 Raft Group 有 N 个副本，从单个 Raft Group 的 Replica 个数来看
    - Replica 数量不够（例如节点掉线，失去副本），如何处理？
    - Replica 数量过多（例如掉线的节点又恢复正常，自动加入集群），如何处理？
- 读/写都是通过 Leader 进行，如果 Leader 只集中在少量节点上，会对集群有什么影响？
- 并不是所有的 Region 都被频繁的访问，可能访问热点只在少数几个 Region，这个时候我们需要做什么？
- 集群在做负载均衡的时候，往往需要搬迁数据，这种数据的迁移会不会占用大量的网络带宽、磁盘 IO 以及 CPU，进而影响在线服务？

这些问题单独拿出可能都能找到简单的解决方案，但是混杂在一起，就不太好解决。有的问题貌似只需要考虑单个 Raft Group 内部的情况，比如根据副本数量是否充足来决定是否需要添加副本，但是实际上这个副本添加在哪里，是需要考虑全局信息的。同时整个系统也是在动态变化，Region 分裂、节点加入、节点失效、访问热点变化等情况会不断发生，整个调度系统也需要在动态中不断向最优状态前进，如果没有一个掌握全局信息，可以对全局进行调度，并且可以配置的组件，就很难满足这些需求。因此我们需要一个中心节点，来对系统的整体状况进行把控和调整，所以有了 PD 这个模块。

### 4.1.2 调度的需求

上面罗列了一大堆问题，我们先进行分类和整理。总体来看，问题有两大类：

作为一个分布式高可用存储系统，必须满足的需求，包括四种：

- 副本数量不能多也不能少
- 副本需要分布在不同的机器上
- 新加节点后，可以将其他节点上的副本迁移过来
- 自动下线失效节点，同时将该节点的数据迁移走

作为一个良好的分布式系统，需要优化的地方，包括：

- 维持整个集群的 Leader 分布均匀
- 维持每个节点的储存容量均匀
- 维持访问热点分布均匀
- 控制负载均衡的速度，避免影响在线服务
- 管理节点状态，包括手动上线/下线节点

满足第一类需求后，整个系统将具备强大的容灾功能。满足第二类需求后，可以使得系统整体的负载更加均匀，管理更加容易方便。

为了满足这些需求，首先我们需要收集足够的信息，比如每个节点的状态、每个 Raft Group 的信息、业务访问操作的统计等；其次需要设置一些策略，PD 根据这些信息以及调度的策略，制定出尽量满足前面所述需求的调度计划；最后需要一些基本的操作，来完成调度计划。

### 4.1.3 调度的基本操作

调度的基本操作指的是为了满足调度的策略，我们有哪些功能可以用。这是整个调度的基础，了解了手里有什么样的锤子，才知道用什么样的姿势去砸钉子。

上述调度需求看似复杂，但是整理下来无非是下面三个操作：

- 增加一个 Replica
- 删除一个 Replica
- 将 Leader 角色在一个 Raft Group 的不同 Replica 之间 transfer（迁移）。

刚好 Raft 协议通过 AddReplica、RemoveReplica、TransferLeader 这三个命令，可以支撑上述三种基本操作。

### 4.1.4 信息收集

调度依赖于整个集群信息的收集，简单来说，我们需要知道每个 TiKV 节点的状态以及每个 Region 的状态。TiKV 集群会向 PD 汇报两类消息，TiKV 节点信息和 Region 信息：

每个 TiKV 节点会定期向 PD 汇报节点的状态信息

TiKV 节点（Store）与 PD 之间存在心跳包，一方面 PD 通过心跳包检测每个 Store 是否存活，以及是否有新加入的 Store；另一方面，心跳包中也会携带这个 [Store 的状态信息](#)，主要包括：

- 总磁盘容量
- 可用磁盘容量
- 承载的 Region 数量
- 数据写入/读取速度
- 发送/接受的 Snapshot 数量（Replica 之间可能会通过 Snapshot 同步数据）
- 是否过载
- labels 标签信息（标签是具备层级关系的一系列 Tag）

每个 Raft Group 的 Leader 会定期向 PD 汇报 Region 的状态信息

每个 Raft Group 的 Leader 和 PD 之间存在心跳包，用于汇报这个 [Region 的状态](#)，主要包括下面几点信息：

- Leader 的位置
- Followers 的位置
- 掉线 Replica 的个数
- 数据写入/读取的速度

PD 不断的通过这两类心跳消息收集整个集群的信息，再以这些信息作为决策的依据。除此之外，PD 还可以通过管理接口接受额外的信息，用来做更准确的决策。比如当某个 Store 的心跳包中断的时候，PD 并不能判断这个节点是临时失效还是永久失效，只能经过一段时间的等待（默认是 30 分钟），如果一直没有心跳包，就认为该 Store 已经下线，再决定需要将这个 Store 上面的 Region 都调度走。但是有的时候，是运维人员主动将某台机器下线，这个时候，可以通过 PD 的管理接口通知 PD 该 Store 不可用，PD 就可以马上判断需要将这个 Store 上面的 Region 都调度走。

### 4.1.5 调度的策略

PD 收集了这些信息后，还需要一些策略来制定具体的调度计划。

一个 Region 的 Replica 数量正确

当 PD 通过某个 Region Leader 的心跳包发现这个 Region 的 Replica 数量不满足要求时，需要通过 Add/Remove Replica 操作调整 Replica 数量。出现这种情况的可能原因是：

- 某个节点掉线，上面的数据全部丢失，导致一些 Region 的 Replica 数量不足

- 某个掉线节点又恢复服务，自动接入集群，这样之前已经补足了 Replica 的 Region 的 Replica 数量多过，需要删除某个 Replica
- 管理员调整了副本策略，修改了 `max-replicas` 的配置

#### 一个 Raft Group 中的多个 Replica 不在同一个位置

注意这里用的是『同一个位置』而不是『同一个节点』。在一般情况下，PD 只会保证多个 Replica 不落在一个节点上，以避免单个节点失效导致多个 Replica 丢失。在实际部署中，还可能出现下面这些需求：

- 多个节点部署在同一台物理机器上
- TiKV 节点分布在多个机架上，希望单个机架掉电时，也能保证系统可用性
- TiKV 节点分布在多个 IDC 中，希望单个机房掉电时，也能保证系统可用性

这些需求本质上都是某一个节点具备共同的位置属性，构成一个最小的『容错单元』，我们希望这个单元内部不会存在一个 Region 的多个 Replica。这个时候，可以给节点配置 `labels` 并且通过在 PD 上配置 `location-labels` 来指名哪些 label 是位置标识，需要在 Replica 分配的时候尽量保证一个 Region 的多个 Replica 不会分布在具有相同的位置标识的节点上。

#### 副本在 Store 之间的分布均匀分配

由于每个 region 副本中存储的数据容量上限是固定的，所以我们通过维持每个节点上面副本数量的均衡，使得各节点间承载的数据更均衡。

#### Leader 数量在 Store 之间均匀分配

Raft 协议要求读取和写入都通过 Leader 进行，所以计算的负载主要在 Leader 上面，PD 会尽可能将 Leader 在节点间分散开。

#### 访问热点数量在 Store 之间均匀分配

每个 Store 以及 Region Leader 在上报信息时携带了当前访问负载的信息，比如 Key 的读取/写入速度。PD 会检测出访问热点，且将其在节点之间分散开。

#### 各个 Store 的存储空间占用大致相等

每个 Store 启动的时候都会指定一个 Capacity 参数，表明这个 Store 的存储空间上限，PD 在做调度的时候，会考虑节点的存储空间剩余量。

#### 控制调度速度，避免影响在线服务

调度操作需要耗费 CPU、内存、磁盘 IO 以及网络带宽，我们需要避免对线上服务造成太大影响。PD 会对当前正在进行的操作数量进行控制，默认的速度控制是比较保守的，如果希望加快调度（比如停服务升级或者增加新节点，希望尽快调度），那么可以通过调节 PD 参数来加快调度速度。

### 4.1.6 调度的实现

了解了上面这些信息后，接下来我们看一下整个调度的流程。

PD 不断的通过 Store 或者 Leader 的心跳包收集整个集群信息，并且根据这些信息以及调度策略生成调度操作序列。每次收到 Region Leader 发来的心跳包时，PD 都会检查这个 Region 是否有待进行的操作，然后通过心跳包的回复消息，将需要进行的操作返回给 Region Leader，并在后面的心跳包中监测执行结果。注意这里的操作只是给 Region Leader 的建议，并不保证一定能得到执行，具体是否会执行以及什么时候执行，由 Region Leader 根据当前自身状态来定。

### 4.1.7 总结

本篇文章讲的东西，大家可能平时很少会在其他文章中看到，每一个设计都有背后的考量，希望大家能了解到一个分布式存储系统在做调度的时候，需要考虑哪些东西，如何将策略、实现进行解耦，更灵活的支持策略的扩展。

## 4.2 弹性调度

弹性调度 (*Elastic Schedule*) 是 TiDB 在 4.0 的新特性，通过与云环境结合后提供的一系列调度策略，可以让 TiDB 具备自适应能力 (*Adaptive Capacity*)，即 TiDB 能根据用户的 workload 模式自动调节形态以达到资源的最大利用率。自适应能力是 TiDB 能够提供 DBaaS 服务的一项关键能力。

### 4.2.1 需求背景

传统上，我们一般将 TiDB 集群部署在 IDC 环境中，在这种情况下，用户通常希望各台机器的资源利用率比较平均，并且各台机器需要预留足够的资源以应对高峰期，但大部分时间业务流量比较低且平均，机器的利用率相对于高峰期处在一个比较低的水平，造成了机器资源的浪费。而在云环境下，机器资源可以按需分配，并且云厂商能够支持秒级或分钟级交付，那么在平常的大部分时间里，就不需要让每台机器预留资源，而是应该尽可能地利用每台机器资源。当遇到资源利用高峰期时，可以临时扩容机器并且将一部分负载调度到新机器上，进而分散集群压力，保证性能稳定。

如何在云上来实现弹性调度，这不仅需要让 TiDB 内核具备更灵活的处理方式，还需要结合 TiDB Operator 来让它在云上对业务进行自适应调节。目前 4.0 已经初步具备以下两个方面的功能：

- 自动伸缩
- 动态调度

### 4.2.2 自动伸缩

自动伸缩 (Auto-Scale) 包含两方面的内容，一是弹性扩缩容节点，二是在扩缩容节点后自动均衡集群负载。

和 [Aurora](#) 做法类似，弹性伸缩节点可通过对一些系统指标设置一个阈值，比如 CPU 利用率 (TiDB Server 或 TiKV Server)、QPS (TiKV Server) 等，当集群在平衡状态下目标指标等于或者超过阈值一段时间以后，就会自动触发水平的弹性伸缩。

TiDB 借助 TiDB Operator 和 PD 来实现 Auto-Scale：

- TiDB Operator 通过 API 的方式暴露出期望的 TiDB / TiKV 节点数量
- TiDB Operator 定期获取 TiDB / TiKV 的 metrics 信息和 PD 上的集群状态信息
- TiDB Operator 通过内部的 Auto-Scaling 算法对 `TiDBCluster.Spec.Replicas` 进行调整，从而实现 Auto-Scaling。

在 TiDB Operator 中，新增了 AutoScaler API 和 AutoScaler Controller，下面是一个 AutoScaler API 的例子：

```

apiVersion: pingcap.com/v1alpha1
kind: TidbClusterAutoScaler
metadata:
  name: autoscaler
  namespace: ela-demo
spec:
  cluster:
    name: ela-scheduling
    namespace: ela-demo
  metricsUrl: http://monitor-prometheus.elo-demo.svc:9090
  tidb:
    minReplicas: 8
    maxReplicas: 8
    scaleOutIntervalSeconds: 100
    scaleInIntervalSeconds: 100
    metricsTimeDuration: "1m"
    metrics:
      - type: "Resource"
        resource:
          name: "cpu"
          target:
            type: "Utilization"
            averageUtilization: 90
  tikv:
    minReplicas: 3
    maxReplicas: 5
    scaleOutIntervalSeconds: 100
    scaleInIntervalSeconds: 100
    metricsTimeDuration: "1m"
    metrics:
      - type: "Resource"
        resource:
          name: "cpu"
          target:
            type: "Utilization"
            averageUtilization: 70

```

其中：

- minReplicas：最小实例数
- maxReplicas：最大实例数
- scaleOutIntervalSeconds：每次触发 scale-out 的间隔时间
- scaleInIntervalSeconds：每次触发 scale-in 的间隔时间

当集群扩缩容节点后，还需要进行快速的负载均衡。对于 TiDB 的负载均衡，需要客户端具备自动重新调整长连接的能力，使建立到 TiDB 上的连接能够重新均衡。而对于 TiKV，主要是通过 PD 发起对热点 Region 的动态调度，以达到快速分摊压力的目的，同时也能以最小的调度代价来提高弹性伸缩的速度。

### 4.2.3 动态调度

在上面提到了通过 TiDB Operator 扩缩容 TiKV 节点后，需要由 PD 来发起 Region 的热点调度，一般来说分为以下几种情况：

1. 请求分布相对平均，区域广
2. 请求分布相对平均，区域小
3. 请求分布不平均，集中在多个点
4. 请求分布不平均，集中在单个点

对于第一种情况，访问平均分布在集群的大部分 Region 中，目前调度不会对其做相关的特殊处理。对于第三种情况，现有的热点调度器已经能够识别并且对其进行调度。下面来介绍下对于第 2 种和第 4 种情况如何去做动态调整：

1. 根据负载动态分裂 (Load Base Splitting)

对于上述第二种情况，会出现小区域的热点问题。特别是在 TiDB 实践中经常遇到的热点小表问题，热点数据集中在几个 Region 中，造成无法利用多台机器资源的情况。TiDB 4.0 中引入了根据负载动态分裂特性，即根据负载自动拆分 Region。其主要的思路借鉴了 CRDB 的[实现](#)，会根据设定的 QPS 阈值来进行自动的分裂。其主要原理是，若对该 Region 的请求 QPS 超

过阈值则进行采样，对采样的请求分布进行判断。采样的方法是通过蓄水池采样出请求中的 20 个 key，然后统计请求在这些 key 的左右区域的分布来进行判断，如果分布比较平均并能找到合适的 key 进行分裂，则自动地对该 Region 进行分裂。

### 1. 热点隔离 (Isolate Frequently Access Region)

由于 TiKV 的分区是按 Range 切分的，在 TiDB 的实践中自增主建、递增的索引的写入等都会造成单一热点的情况，另外如果用户没有对 workload 进行分区，且访问是 non-uniform 的，也会造成单一热点问题。这些都是上述的第四种情况。根据过去的最佳实践经验，往往需要用户调整表结构，采用分区表，使用 shard\_bits 等方式来使得单一分区变成多分区，才能进行负载均衡。而在云环境中，在用户不用调整 workload 或者表结构的情况下，TiDB 可以通过在云上弹性一个高性能的机器，并由 PD 通过识别自动将单一热点调度到该机器上，达到热点隔离的目的。该方法也特别适用于时事、新闻等突然出现爆发式业务热点的情况。

## 4.2.4 总结

TiDB 4.0 是一个更加成熟，易用的版本，并且随着 TiDB Operator 的成熟以及 DBaaS 的推出，TiDB 4.0 开始成为一个拥抱云的版本。在云上，调度关注的视角也发生了改变，这使得让 TiDB 自适应 workload 去调整数据库形态变成了可能。后续弹性调度这一块，TiDB 还将有更多的玩法，比如 Follower Read 与多数据中心场景的结合，以及 TiFlash 大家族的加入。未来的 TiDB，除了是一个 HTAP 的数据库，也会变成一个“智能”的数据库。

## 第5章 TiDB 和 MySQL 的区别

TiDB 作为开源 NewSQL 数据库的典型代表之一，同样支持 SQL，支持事务 ACID 特性。在通讯协议上，TiDB 选择与 MySQL 完全兼容，并尽可能兼容 MySQL 的语法。因此，基于 MySQL 数据库开发的系统，大多数可以平滑迁移至 TiDB，而几乎不用修改代码。对用户来说，迁移成本极低，过渡自然。

然而，仍有一些 MySQL 的特性和行为，TiDB 目前暂时不支持或表现与 MySQL 有差异。除此之外，TiDB 提供了一些扩展语法和功能，为用户提供更多的便利。

TiDB 仍处在快速发展的道路上，对 MySQL 功能和行为的支持方面，正按 [路线图](#) 的规划在前行。

### 5.1 兼容策略

先从总体上概括 TiDB 和 MySQL 兼容策略，如下表：

通讯协议	SQL语法	功能和行为
完全兼容	兼容绝大多数	兼容大多数

截至 4.0 版本，TiDB 与 MySQL 的区别总结如下表：

	MySQL	TiDB
隔离级别	支持读未提交、读已提交、可重复读、串行化，默认为可重复读	乐观事务支持快照隔离，悲观事务支持快照隔离和读已提交
锁机制	悲观锁	乐观锁、悲观锁
存储过程	支持	不支持
触发器	支持	不支持
事件	支持	不支持
自定义函数	支持	不支持
窗口函数	支持	部分支持
JSON	支持	不支持部分 MySQL 8.0 新增的函数
外键约束	支持	忽略外键约束
字符集		只支持 ascii、latin1、binary、utf8、utf8mb4
增加/删除主键	支持	通过 <a href="#">alter-primary-key</a> 配置开关提供
CREATE TABLE tblName AS SELECT stmt	支持	不支持
CREATE TEMPORARY TABLE	支持	TiDB 忽略 TEMPORARY 关键字，按照普通表创建
DML affected rows	支持	不支持
AutoRandom 列属性	不支持	支持
Sequence 序列生成器	不支持	支持

### 5.2 区别点详述及应对方案

#### (1) 字符集支持

TiDB 目前支持以下字符集：

```
tidb> SHOW CHARACTER SET;
+-----+-----+-----+-----+
| Charset | Description | Default collation | Maxlen |
+-----+-----+-----+-----+
| utf8 | UTF-8 Unicode | utf8_bin | 3 |
| utf8mb4 | UTF-8 Unicode | utf8mb4_bin | 4 |
| ascii | US ASCII | ascii_bin | 1 |
| latin1 | Latin1 | latin1_bin | 1 |
| binary | binary | binary | 1 |
+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

注意：TiDB 的默认字符集为 `utf8mb4`，MySQL 5.7 中为 `latin1`，MySQL 8.0 中修改为 `utf8mb4`。当指定的字符集为 `utf8` 或 `utf8mb4` 时，TiDB 仅支持合法的 UTF8 字符。对于不合法的字符，会报错：`incorrect utf8 value`，该字符合法性检查与 MySQL 8.0 一致。对于 MySQL 5.7 及以下版本，会存在允许插入非法 UTF8 字符，但同步到 TiDB 报错的情况。此时，可以通过 TiDB 配置 "`tidb_skip_utf8_check`" 跳过 UTF8 字符合法性检查强制写入 TiDB。

每一个字符集，都有一个默认的 Collation，例如 `utf8` 的默认 Collation 为 `utf8_bin`，TiDB 中字符集的默认 Collation 与 MySQL 不一致，具体如下：

字符集	TiDB 默认 Collation	MySQL 5.7 默认 Collation	MySQL 8.0 默认 Collation
utf8	utf8_bin	utf8_general_ci	utf8_general_ci
utf8mb4	utf8mb4_bin	utf8mb4_general_ci	utf8mb4_0900_ai_ci
ascii	ascii_bin	ascii_general_ci	ascii_general_ci
latin1	latin1_bin	latin1_swedish_ci	latin1_swedish_ci
binary	binary	binary	binary

在 4.0 版本之前，TiDB 中可以任意指定字符集对应的所有 Collation，并把它们按照默认 Collation 处理，即以编码字节序为字符串序。同时，并未像 MySQL 一样，在比较前按照 Collation 的 `PADDING` 属性将字符补齐空格。因此，会造成以下的行为区别：

```
tidb> create table t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci primary key);
Query OK, 0 rows affected
tidb> insert into t values ('A');
Query OK, 1 row affected
tidb> insert into t values ('a');
Query OK, 1 row affected // MySQL 中, 由于 utf8mb4_general_ci 大小写不敏感, 报错 Duplicate entry 'a'.
tidb> insert into t1 values ('a ');
Query OK, 1 row affected // MySQL 中, 由于补齐空格比较, 报错 Duplicate entry 'a '
```

TiDB 4.0 新增了完整的 Collation 支持框架，允许实现所有 MySQL 中的 Collation，并新增了配置开关 `new_collation_enabled_on_first_bootstrap`，在集群初次初始化时决定是否启用新 Collation 框架。在该配置开关打开之后初始化集群，可以通过 `mysql.tidb` 表中的 `new_collation_enabled` 变量确认新 Collation 是否启用：

```
tidb> select VARIABLE_VALUE from mysql.tidb where VARIABLE_NAME='new_collation_enabled';
+-----+
| VARIABLE_VALUE |
+-----+
| True          |
+-----+
1 row in set (0.00 sec)
```

在新 Collation 启用后，TiDB 修正了 `utf8mb4_general_bin` 和 `utf8_general_bin` 的 `PADDING` 行为，会将字符串补齐空格后比较；同时支持了 `utf8mb4_general_ci` 和 `utf8_general_ci`，这两个 Collation 与 MySQL 保持兼容。

## (2) 系统时区

在 MySQL 中，系统时区 `system_time_zone` 在 MySQL 服务启动时通过 环境变量 `tz` 或命令行参数 `--timezone` 指定。

对于 TiDB 而言，作为一个分布式数据库，TiDB 需要保证整个集群的系统时区始终一致。因此 TiDB 的系统时区在集群初始化时，由负责初始化的 TiDB 节点环境变量 `tz` 决定。集群初始化后，固定在集群状态表 `mysql.tidb` 中：

```
tidb> select VARIABLE_VALUE from mysql.tidb where VARIABLE_NAME='system_tz';
+-----+
| VARIABLE_VALUE |
+-----+
| Asia/Shanghai  |
+-----+
1 row in set (0.00 sec)
```

通过查看 `system_time_zone` 变量，可以看到该值与状态表中的 `system_tz` 保持一致：

```
tidb> select @@system_time_zone;
+-----+
| @@system_time_zone |
+-----+
| Asia/Shanghai      |
+-----+
1 row in set (0.00 sec)
```

请注意，这意味着 TiDB 的系统时区在初始化后不再更改。若需要改变集群的时区，可以显式指定 `time_zone` 系统变量，例如：

```
tidb> set @@global.time_zone='UTC';
Query OK, 0 rows affected (0.00 sec)
```

## 第六章 TiDB 事务模型

作为分布式数据库，分布式事务是既是重要特性之一，也是难点之一。TiDB 实现了快照隔离级别的分布式事务，支持悲观锁、乐观锁，同时也解决了业界的难点之一：大事务。本章我们将深入浅出 TiDB 事务的使用与原理，由浅入深介绍以下内容：

- 乐观事务
- 悲观事务
- 大事务

## 6.1 乐观事务

事务是数据库的基础，提供高效的、支持完整 ACID 的分布式事务更是分布式数据库的立足之本。本章节会首先介绍事务的基本概念，然后介绍 TiDB 基于 Percolator 实现的乐观事务以及在使用上的最佳实践。

### 6.1.1 事务

事务是数据库执行的最小单元，允许用户将多个读写操作组合为一个逻辑单元。事务需要满足原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），也就是 ACID。

#### 6.1.1.1 隔离级别

对用户来说，最友好的并发事务执行顺序为每个事务独占整个数据库，并发事务执行的结果与一个个串行执行相同，也就是串行化，能够避免所有的异常情况。但在这种隔离级别下，并发执行的事务性能较差，提供更弱保证的隔离级别能够显著提升系统的性能。根据允许出现的异常，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

#### 6.1.1.2 并发控制

数据库有多种并发控制方法，这里只介绍以下两种：

- 乐观并发控制 (OCC)：在事务提交阶段检测冲突
- 悲观并发控制 (PCC)：在事务执行阶段检测冲突

乐观并发控制期望事务间数据冲突不多，只在提交阶段检测冲突能够获取更高的性能。悲观并发控制更适合数据冲突较多的场景，能够避免乐观事务在这类场景下事务因冲突而回滚的问题，但相比乐观并发控制，在没有数据冲突的场景下，性能相对要差。

### 6.1.2 TiDB 乐观事务实现

TiDB 基于 Google Percolator 实现了支持完整 ACID、基于快照隔离级别 (Snapshot Isolation) 的分布式乐观事务。TiDB 乐观事务需要将事务的所有修改都保存在内存中，直到提交时才会写入 TiKV 并检测冲突。

#### 6.1.2.1 Snapshot Isolation

Percolator 使用多版本并发控制 (MVCC) 来实现快照隔离级别，与可重复读的区别在于整个事务是在一个一致的快照上执行。TiDB 使用 PD 作为全局授时服务 (TSO) 来提供单调递增的版本号：

- 事务开始时获取 start timestamp，也是快照的版本号；事务提交时获取 commit timestamp，同时也是数据的版本号
- 事务只能读到在事务 start timestamp 之前最新已提交的数据
- 事务在提交时会根据 timestamp 来检测数据冲突

#### 6.1.2.2 两阶段提交 (2PC)

TiDB 使用两阶段提交(Two-Phase Commit) 来保证分布式事务的原子性，分为 Prewrite 和 Commit 两个阶段：

- Prewrite：对事务修改的每个 Key 检测冲突并写入 lock 防止其他事务修改。对于每个事务，TiDB 会从涉及到改动的所有

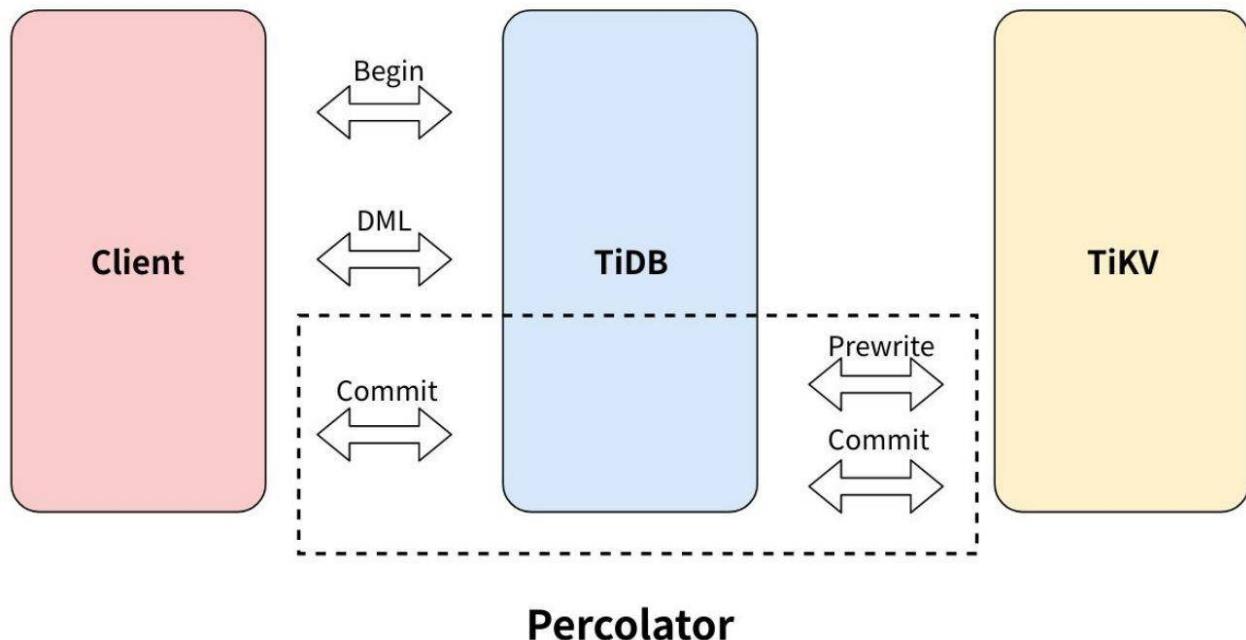
Key 中选中一个作为当前事务的 Primary Key，事务提交或回滚都需要先修改 Primary Key，以它的提交与否作为整个事务执行结果的标识。

- Commit : Prewrite 全部成功后，先同步提交 Primary Key，成功后事务提交成功，其他 Secondary Keys 会异步提交。

Percolator 将事务的所有状态都保存在底层支持高可用、强一致性的存储系统中，从而弱化了传统两阶段提交中协调者（Coordinator）的作用，所有的客户端都可以根据存储系统中的事务状态对事务进行提交或回滚。

### 6.1.2.3 两阶段提交过程

事务的两阶段提交过程如下：



### Percolator

1. 客户端开始一个事务。
2. TiDB 向 PD 获取 tso 作为当前事务的 start timestamp。
3. 客户端发起读或写请求。
4. 客户端发起 Commit。
5. TiDB 开始两阶段提交，保证分布式事务的原子性，让数据真正落盘。
  - i. TiDB 从当前要写入的数据中选择一个 Key 作为当前事务的 Primary Key。
  - ii. TiDB 并发地向所有涉及的 TiKV 发起 Prewrite 请求。TiKV 收到 Prewrite 请求后，检查数据版本信息是否存在冲突，符合条件的数据会被加锁。
  - iii. TiDB 收到所有 Prewrite 响应且所有 Prewrite 都成功。
  - iv. TiDB 向 PD 获取第二个全局唯一递增版本号，定义为本次事务的 commit timestamp。
  - v. TiDB 向 Primary Key 所在 TiKV 发起第二阶段提交。TiKV 收到 Commit 操作后，检查锁是否存在并清理 Prewrite 阶段留下的锁。
6. TiDB 向客户端返回事务提交成功的信息。
7. TiDB 异步清理本次事务遗留的锁信息。

### 6.1.3 最佳实践

#### 6.1.3.1 小事务

从上面得知，每个事务提交需要经过 4 轮 RTT (Round trip time) :

- 从 PD 获取 2 次 Timestamp；
- 提交时的 Prewrite 和 Commit。

为了降低网络交互对于小事务的影响，建议将小事务打包来做。以如下 query 为例，当 `autocommit = 1` 时，下面三条语句各为一个事务：

```
UPDATE my_table SET a='new_value' WHERE id = 1;
UPDATE my_table SET a='newer_value' WHERE id = 2;
UPDATE my_table SET a='newest_value' WHERE id = 3;
```

此时每一条语句都需要经过两阶段提交，频繁的网络交互致使延迟率高。为提升事务执行效率，可以选择使用显式事务，即在一个事务内执行三条语句。优化后版本：

```
START TRANSACTION;
UPDATE my_table SET a='new_value' WHERE id = 1;
UPDATE my_table SET a='newer_value' WHERE id = 2;
UPDATE my_table SET a='newest_value' WHERE id = 3;
COMMIT;
```

同理，执行 `INSERT` 语句时，也建议使用显式事务。

### 6.1.3.2 大事务

既然小事务有问题，那事务是不是越大越好呢？由于 TiDB 两阶段提交的要求，修改数据的单个事务过大时会存在以下问题：

- 客户端在提交之前，数据都写在内存中，而数据量过多时易导致 OOM (Out of Memory) 错误。
- 在第一阶段写入数据耗时增加，与其他事务出现读写冲突的概率会指数级增长，容易导致事务提交失败。
- 最终导致事务完成提交的耗时增加。

因此，对于 TiDB 乐观事务而言，事务太大或者太小，都会出现性能上的问题。为了使性能达到最优，建议每 100~500 行写入一个事务。

### 6.1.3.3 事务冲突

(1) 冲突检测 乐观事务下，检测底层数据是否存在写写冲突是一个很重的操作。具体而言，TiKV 在 Prewrite 阶段就需要读取数据进行检测。为了优化这一块性能，TiDB 集群会在内存里面进行一次冲突预检测。作为一个分布式系统，TiDB 在内存中的冲突检测主要在两个模块进行：

- TiDB 层，如果在 TiDB 实例本身发现存在写写冲突，那么第一个写入发出去后，后面的写入就已经能清楚地知道自己冲突了，没必要再往下层 TiKV 发送请求去检测冲突。
- TiKV 层，主要发生在 Prewrite 阶段。因为 TiDB 集群是一个分布式系统，TiDB 实例本身无状态，实例之间无法感知到彼此的存在，也就无法确认自己的写入与别的 TiDB 实例是否存在冲突，所以会在 TiKV 这一层检测具体的数据是否有冲突。

其中 TiDB 层的冲突检测可以关闭，配置项可以启用：

`txn-local-latches`：事务内存锁相关配置，当本地事务冲突比较多时建议开启。可以在默认值配置文件中对默认值进行修改。

- `enable`
  - 开启
  - 默认值：`false`
- `capacity`
  - Hash 对应的 slot 数，会自动向上调整为 2 的指数倍。每个 slot 占 32 Bytes 内存。当写入数据的范围比较广时（如导数据），设置过小会导致变慢，性能下降。
  - 默认值：`1024000`

配置项 **capacity** 主要影响到冲突判断的正确性。在实现冲突检测时，不可能把所有的 Key 都存到内存里，所以真正存下来的是每个 Key 的 Hash 值。有 Hash 算法就有碰撞也就是误判的概率，这里可以通过配置 **capacity** 来控制 Hash 取模的值：

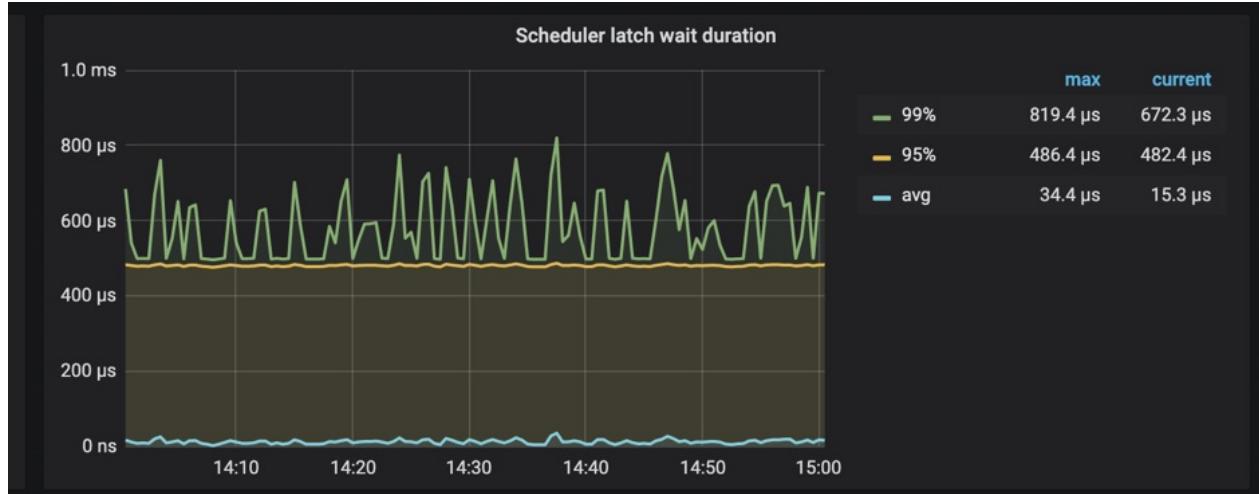
- **capacity** 值越小，占用内存小，误判概率越大。
- **capacity** 值越大，占用内存大，误判概率越小。

在实际应用时，如果业务场景能够预判断写入不存在冲突（如导入数据操作），建议关闭冲突检测。

相应地，TiKV 内存中的冲突检测也有一套类似的东西。不同的是，TiKV 的检测会更严格，不允许关闭，只提供了一个 hash 取模值的配置项，可以在默认值配置文件中对默认值进行修改：

- **scheduler-concurrency**
  - **scheduler** 内置一个内存锁机制，防止同时对一个 Key 进行操作。每个 Key hash 到不同的槽。
  - 默认值：2048000

此外，TiKV 提供了监控查看具体消耗在 latch 等待的时间：



当 Scheduler latch wait duration 的值特别高时，说明大量时间消耗在等待锁的请求上。如果不存在底层写入慢的问题，基本上可以判断该段时间内冲突比较多。

(2) 事务的重试 使用乐观事务模型时，在高冲突率的场景中，事务很容易提交失败。比如 2 个事务同时修改相同行，提交时会有一个事务报错：

```
ERROR 8005 (HY000) : Write Conflict, txnStartTS is stale
```

而 MySQL 内部使用的是悲观事务模型，在执行 SQL 语句的过程中进行冲突检测，所以提交时很难出现异常。为了兼容 MySQL 的悲观事务行为，降低用户开发和迁移的成本，TiDB 乐观事务提供了重试机制。当事务提交后，如果发现冲突，TiDB 内部重新执行包含写操作的 SQL 语句。可以通过设置 `tidb_disable_txn_auto_retry = off` 开启自动重试，并通过 `tidb_retry_limit` 设置重试次数：

- `tidb_disable_txn_auto_retry`：这个参数控制是否自动重试，默认为 1，即不重试。
- `tidb_retry_limit`：用来控制重试次数，注意只有第一个参数启用时该参数才会生效。

如何设置以上参数呢？可以通过 set 语句对系统变量进行设置，推荐两种方式设置：

- session 级别设置：

```
set @@tidb_disable_txn_auto_retry = 0;
set @@tidb_retry_limit = 10;
```

- 全局设置：

```
set @@global.tidb_disable_txn_auto_retry = 0;
set @@global.tidb_retry_limit = 10;
```

注意：

`tidb_retry_limit` 变量决定了事务重试的最大次数。当它被设置为 0 时，所有事务都不会自动重试，包括自动提交的单语句隐式事务。这是彻底禁用 TiDB 中自动重试机制的方法。禁用自动重试后，所有冲突的事务都会以最快的方式上报失败信息（`try again later`）给应用层。

(3) 重试的局限性 TiDB 默认不进行事务重试，因为重试事务可能会导致更新丢失，从而破坏可重复读的隔离级别。事务重试的局限性与其原理有关。事务重试可概括为以下三个步骤：

1. 重新获取 start timestamp。
2. 重新执行包含写操作的 SQL 语句。
3. 再次进行两阶段提交。

第二步中，重试时仅重新执行包含写操作的 SQL 语句，并不涉及读操作的 SQL 语句。但是当前事务中读到数据的时间与事务真正开始的时间发生了变化，写入的版本变成了重试时获取的 start timestamp 而非事务一开始时获取的 start timestamp。

因此，当事务中存在依赖查询结果来更新的语句时，重试将无法保证事务原本可重复读的隔离级别，最终可能导致结果与预期出现不一致。在这种场景下可以使用 `SELECT FOR UPDATE` 来保证事务提交成功时原先查询的结果没有被修改，但包含 `SELECT FOR UPDATE` 的事务无法自动重试。

如果业务可以容忍事务重试导致的异常，或并不关注事务是否以可重复读的隔离级别来执行，则可以开启自动重试。但更建议的是在冲突严重的场景下，使用 TiDB 的悲观事务。

#### 6.1.3.4 垃圾回收 (GC)

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。数据版本过多会占用大量空间，同时影响数据库的查询性能，GC 的任务便是清理不再需要的旧数据。

GC 会被定期触发，默认情况下每 10 分钟一次。每次 GC 时，首先，TiDB 会计算一个称为 `safe point` 的时间戳（默认为当前时间减去 10 分钟），接下来 TiDB 会在保证 `safe point` 之后的快照全部拥有正确数据的前提下，删除更早的过期数据。

TiDB 的 GC 相关的配置存储于 `mysql.tidb` 系统表中，可以通过 SQL 语句对这些参数进行查询和更改：

```
select VARIABLE_NAME, VARIABLE_VALUE from mysql.tidb;
```

例如，如果需要将 GC 调整为保留最近一天以内的数据，只需执行下列语句即可：

```
update mysql.tidb set VARIABLE_VALUE="24h" where VARIABLE_NAME="tikv_gc_life_time";
```

GC 在执行过程中会删除大量数据，可能会对线上业务造成影响。可通过修改 TiKV 配置中的 `gc.max-write-bytes-per-sec` 限制 GC worker 每秒数据写入量，降低对正常请求的影响，0 为关闭该功能。

#### 6.1.4 乐观锁初体验

测试数据准备：

```
MySQL [test]> insert into tran select id,star from eyesight.t_request;
Query OK, 3327 rows affected (0.17 sec)
Records: 3327  Duplicates: 0  Warnings: 0
MySQL [test]>
```

左边事务先进行 `BEGIN`，两个窗口同时按下 `COMMIT`：

<pre>Database changed MySQL [test]&gt; set autocommit=0; Query OK, 0 rows affected (0.00 sec)  MySQL [test]&gt; begin; Query OK, 0 rows affected (0.00 sec)  MySQL [test]&gt; update tran set b=8888; Query OK, 3327 rows affected (0.12 sec) Rows matched: 3327  Changed: 3327  Warnings: 0  MySQL [test]&gt; commit; Query OK, 0 rows affected (2.31 sec)  MySQL [test]&gt; []</pre>	<pre>Database changed MySQL [test]&gt; set autocommit=0; Query OK, 0 rows affected (0.00 sec)  MySQL [test]&gt; begin; Query OK, 0 rows affected (0.00 sec)  MySQL [test]&gt; update tran set b=9999; Query OK, 3327 rows affected (0.04 sec) Rows matched: 3327  Changed: 3327  Warnings: 0  MySQL [test]&gt; commit; Query OK, 0 rows affected (1.17 sec)  MySQL [test]&gt; []</pre>
--	--

两个事务可能发生冲突但客户端并没有报错，所以要么就是串行执行没有冲突，要么就是冲突了内部 TiDB 进行重试。通过查看 `tidb.log` 日志发现 `startTs(415144181973647361)` 和 `startTs(415144181960540213)` 事务冲突，进行重试的是 `415144181973647361` (图片终端左边的事务)

```
[2020/03/08 14:36:17.121 +08:00] [WARN] [session.go:419] [sql] [label=internal] [error="[kv:9007]Write conflict, txnStartTS=415144181973647361, conflictStartTS=415144181960540213, conflictCommitTS=415144182052290562, key={tableID=19, indexID=1, indexValues={415144166153256976, 114421, }} primary={tableID=19, indexID=1, indexValues={415144166153256976, 114421, }} [try again later]" [txn="Txn{state=invalid}"]

[2020/03/08 14:36:17.121 +08:00] [WARN] [session.go:611] [retrying] [schemaVersion=1813] [retryCnt=0] [queryNum=1] [sql="update mysql.stats_meta set version = 415144181973647361, count = count + 52587, modify_count = modify_count + 52587 where table_id = 1807"]

[2020/03/08 14:36:17.125 +08:00] [WARN] [session.go:632] ["transaction association"] ["retrying txnStartTS=415144182301327413] ["original txnStartTS=415144181973647361] 重新获取startTs进行事务重新提交

[2020/03/08 14:36:17.279 +08:00] [INFO] [2pc.go:1039] ["2PC clean up done"] [txnStartTS=415144181973647361]
```

从图片的两个事务的 `commit` 时间(2.31和1.17)，就能猜测到时右边事务提交在前，左边事务提交在后，通过 `SELECT` 数据也是验证了如上猜测。

```
MySQL [test]> select * from tran limit 2;
+-----+-----+
| a    | b    |
+-----+-----+
| 100007 | 8888 |
| 100036 | 8888 |
+-----+-----+
2 rows in set (0.00 sec)

MySQL [test]>
```

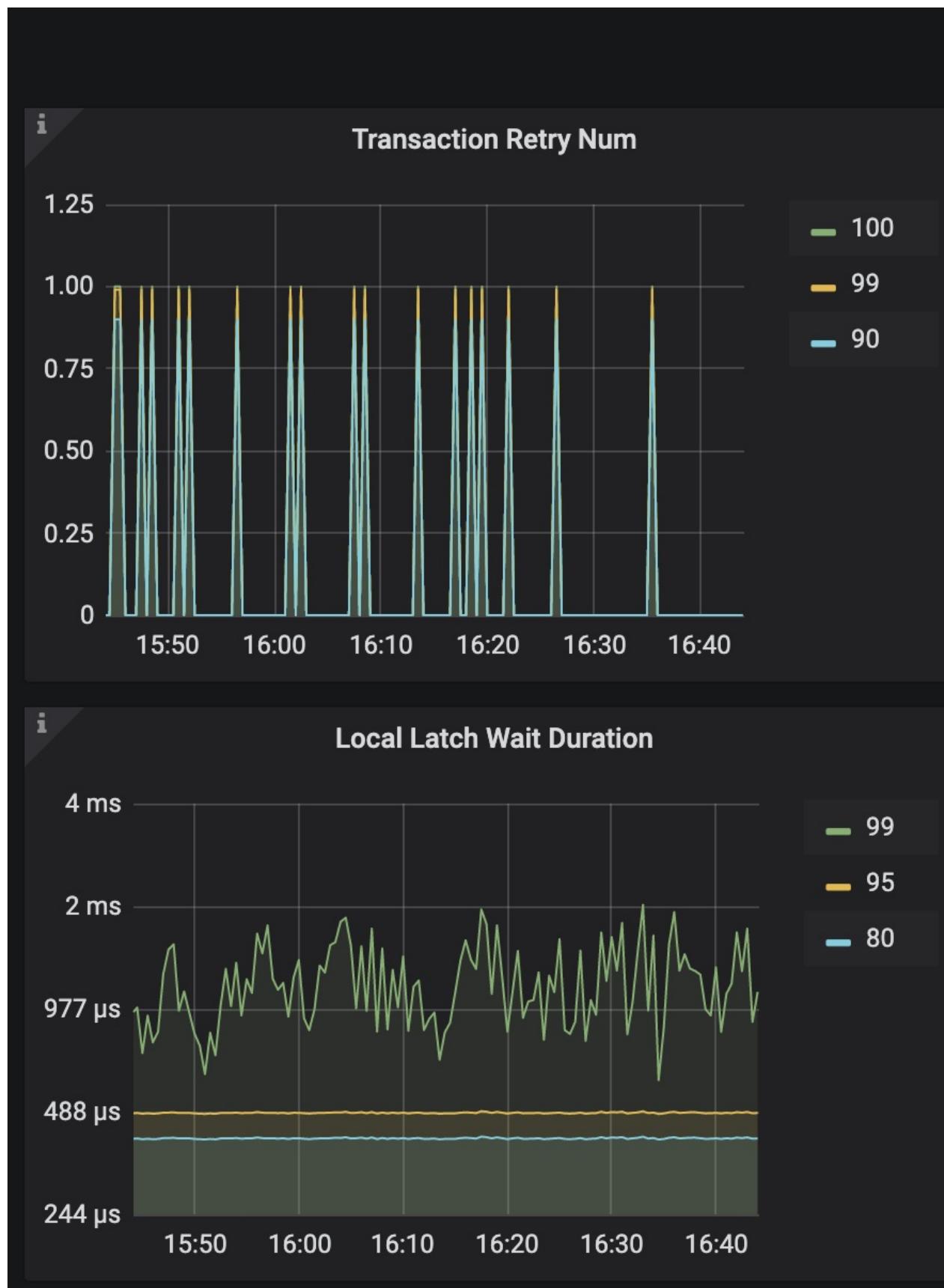
## 6.1.5 如何判断集群的事务健康

### 6.1.5.1 tikv监控--server板块

lock 的大小应该是比较小。



### 6.1.5.2 tidb监控-transaction板块



## 6.2 悲观事务

乐观事务模型在分布式系统中有着极大的性能优势，但为了让 TiDB 的使用方式更加贴近传统单机数据库，更好的适配用户场景，TiDB v3.0 及之后版本在乐观事务模型的基础上实现了悲观事务模型。本文将介绍 TiDB 悲观事务模型特点。

### 6.2.1 悲观锁解决的问题

通过支持悲观事务，降低用户修改代码的难度甚至不用修改代码：

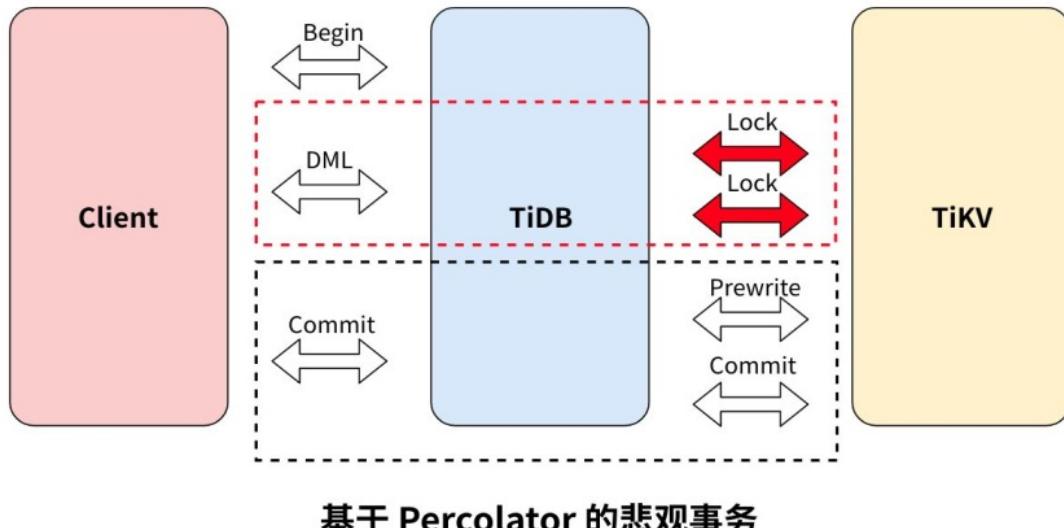
- 在 v3.0.8 之前，TiDB 默认使用的乐观事务模式会导致事务提交时因为冲突而失败。为了保证事务的成功率，需要修改应用程序，加上重试的逻辑。
- 乐观事务模型在冲突严重的场景和重试代价大的场景无法满足用户需求，支持悲观事务可以弥补这方面的缺陷，拓展 TiDB 的应用场景。

以发工资场景为例：对于一个用人单位来说，发工资的过程其实是从企业账户给多个员工的个人账户转账的过程，一般来说都是批量操作，在一个大的转账事务中可能涉及到成千上万的更新，想象一下如果这个大事务执行的这段时间内，某个个人账户发生了消费（变更），如果这个大事务是乐观事务模型，提交的时候肯定要回滚，涉及上万个个人账户发生消费是大概率事件，如果不做任何处理，最坏的情况是这个大事务永远没办法执行，一直在重试和回滚（饥饿）。

### 6.2.2 基于 Percolator 的悲观事务

悲观事务在 Percolator 乐观事务基础上实现，在 Prewrite 之前增加了 Acquire Pessimistic Lock 阶段用于避免 Prewrite 时发生冲突：

- 每个 DML 都会加悲观锁，锁写到 TiKV 里，同样会通过 raft 同步。
- 悲观事务在加悲观锁时检查各种约束，如 Write Conflict、key 唯一性约束等。
- 悲观锁不包含数据，只有锁，只用于防止其他事务修改相同的 Key，不会阻塞读，但 Prewrite 后会阻塞读（和 Percolator 相同，但有了大事务支持后将不会阻塞读）。
- 提交时同 Percolator，悲观锁的存在保证了 Prewrite 不会发生 Write Conflict，保证了提交一定成功。



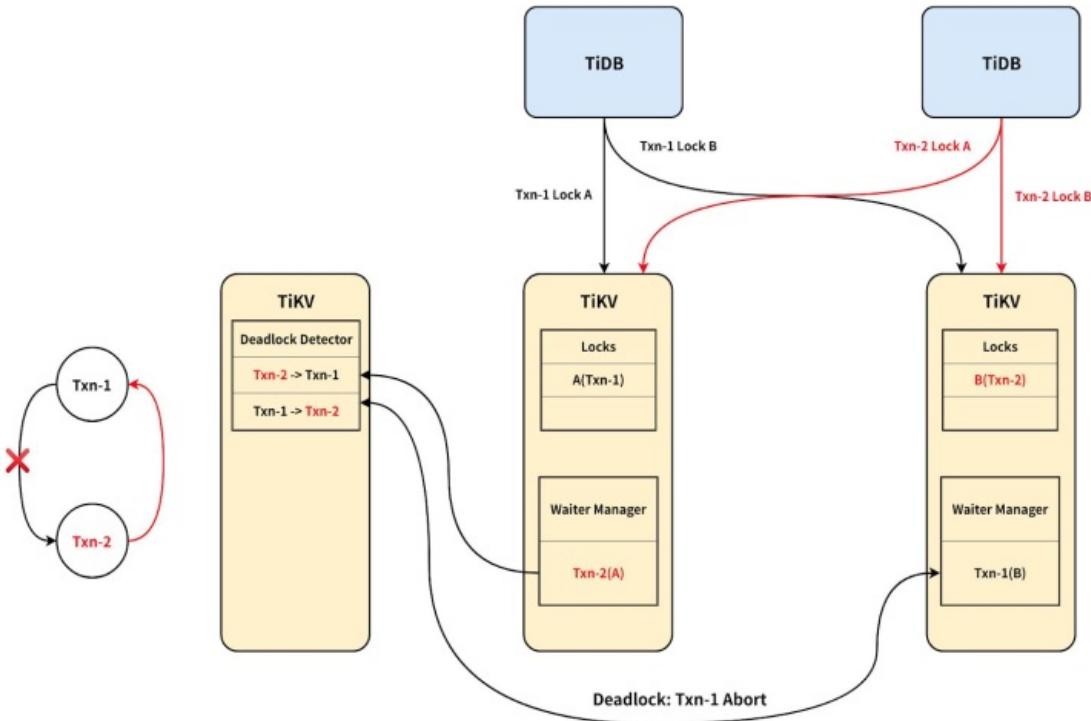
#### 6.2.2.1 等锁顺序

TiKV 中实现了 Waiter Manager 用于管理等锁的事务，当悲观事务加锁遇到其他事务的锁时，将会进入 Waiter Manager 中等待锁被释放，TiKV 会尽可能按照事务 start timestamp 的顺序来依次获取锁，从而避免事务间无用的竞争。

### 6.2.2.2 分布式死锁检测

在 Waiter Manager 中等待锁的事务间可能发生死锁，而且可能发生在不同的机器上，TiDB 采用分布式死锁检测来解决死锁问题：

- 在整个 TiKV 集群中，有一个死锁检测器 leader。
- 当要等锁时，其他节点会发送检测死锁的请求给 leader。



死锁检测器基于 Raft 实现了高可用，等锁事务也会定期发送死锁检测请求给死锁检测器的 leader，从而保证了即使之前 leader 宕机的情况下也能检测到死锁。

### 6.2.3 最佳实践

#### 6.2.3.1 事务模型的选择

TiDB 支持乐观事务和悲观事务，并且允许在同一个集群中混合使用事务模式。由于悲观事务和乐观事务的差异，用户可以根据使用场景灵活的选择适合自己的事务模式：

- 乐观事务：事务间没有冲突或允许事务因数据冲突而失败；追求极致的性能。
- 悲观事务：事务间有冲突且对事务提交成功率有要求；因为加锁操作的存在，性能会比乐观事务差。

#### 6.2.3.2 使用方法

v3.0.8 及之后版本新建的 TiDB 集群将默认使用悲观事务模式，从乐观事务模式升级的集群仍将使用乐观事务模式。进入悲观事务模式有以下三种方式：

- 执行 `BEGIN PESSIMISTIC`；语句开启的事务，会进入悲观事务模式。

可以通过写成注释的形式 `BEGIN /*!90000 PESSIMISTIC */;` 来兼容 MySQL 语法。

- 执行 `set @@tidb_txn_mode = 'pessimistic'`，使这个 session 执行的所有显式事务（即非 `autocommit` 的事务）都会进入悲观事务模式。
- 执行 `set @@global.tidb_txn_mode = 'pessimistic'`，使之后整个集群所有新创建 session 执行的所有显示事务（即非 `autocommit` 的事务）都会进入悲观事务模式。

可通过执行 `set @@global.tidb_txn_mode = '';` 还原回乐观事务模式。

### 6.2.3.3 Batch DML

从上面可以看到，悲观事务在执行每个 DML 时都需要向 TiKV 发送加锁请求，如果事务内 DML 数量很多但 DML 操作很小时，加锁操作会显著增加事务的延迟，所以建议使用悲观事务时尽可能用一条 DML 操作更多的数据。

例如：以下每条 INSERT 都需要向 TiKV 中写入悲观锁，带来了极大的延迟：

```
BEGIN;
INSERT INTO my_table VALUES (1);
INSERT INTO my_table VALUES (2);
INSERT INTO my_table VALUES (3);
COMMIT;
```

如果修改为 INSERT 多行，性能将会成倍的提升：

```
BEGIN;
INSERT INTO my_table VALUES (1), (2), (3);
COMMIT;
```

### 6.2.3.4 隔离级别的选择

TiDB 在悲观事务模式下支持了 2 种隔离级别。

一、默认的与 MySQL 行为基本相同的可重复读隔离级别（Repeatable Read）隔离级别。

但因架构和实现细节的不同，TiDB 和 MySQL InnoDB 的行为在细节上有一些不同：

1. TiDB 使用 range 作为 WHERE 条件，执行 DML 和 `SELECT FOR UPDATE` 语句时不会阻塞范围内并发的 `INSERT` 语句的执行。

InnoDB 通过实现 gap lock，支持阻塞 range 内并发的 `INSERT` 语句的执行，其主要目的是为了支持 statement based binlog，因此有些业务会通过将隔离级别降低至 READ COMMITTED 来避免 gap lock 导致的并发性能问题。TiDB 不支持 gap lock，也就不需要付出相应的并发性能的代价。

2. TiDB 不支持 `SELECT LOCK IN SHARE MODE`。

使用这个语句执行的时候，效果和没有加锁是一样的，不会阻塞其他事务的读写。

3. DDL 可能会导致悲观事务提交失败。

MySQL 在执行 DDL 时会被正在执行的事务阻塞住，而在 TiDB 中 DDL 操作会成功，造成悲观事务提交失败：`ERROR 1105 (HY000): Information schema is changed. [try again later]`。

4. `START TRANSACTION WITH CONSISTENT SNAPSHOT` 之后，MySQL 仍然可以读取到之后在其他事务创建的表，而 TiDB 不能。

5. autocommit 事务不支持悲观锁

所有自动提交的语句都不会加悲观锁，该类语句在用户侧感知不到区别，因为悲观事务的本质是把整个事务的重试变成了单个 DML 的重试，autocommit 事务即使在 TiDB 关闭重试时也会自动重试，效果和悲观事务相同。

自动提交的 select for update 语句也不会等锁。

二、可设置 `SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;` 使用与 Oracle 行为相同的读已提交隔离级别（Read Committed）。

由于历史原因，当前主流数据库的读已提交隔离级别本质上都是 Oracle 定义的一致性读隔离级别。TiDB 为了适应这一历史原因，悲观事务中的读已提交隔离级别的实质行为也是一致性读。用户可以自由选择适合业务场景的隔离级别。



## 6.3 4.0 的大事务支持

### 6.3.1 背景

如果做一个调研，在开发或 DBA 日常使用 TiDB 的过程中，最经常遇到的问题或报错是啥，我相信有一个肯定会让大家咬牙切齿，那就是 `transaction too large`。

我们来检索一下 PingCAP 官网文档，以下引用官网 FAQ：

4.3.3 `transaction too large` 是什么原因，怎么解决？

由于分布式事务要做两阶段提交，并且底层还需要做 Raft 复制，如果一个事务非常大，会使得提交过程非常慢，并且会卡住下面的 Raft 复制流程。为了避免系统出现被卡住的情况，我们对事务的大小做了限制：

- 单个事务包含的 SQL 语句不超过 5000 条（默认）
- 单条 KV entry 不超过 6MB
- KV entry 的总条数不超过 30w
- KV entry 的总大小不超过 100MB

在 Google 的 Cloud Spanner 上面，也有类似的限制。不过对于初次接触 TiDB 的同学来说，经常是丈二和尚摸不着头脑，会有如下来自灵魂的拷问：

- 为什么在 MySQL/Oracle 中运行的好好的跑批程序，迁移到 TiDB 中就报错 `statement count 5001 exceeds the transaction limitation ?`
- 为什么说 kv 总条数不超过 30W，但是我一次更新 10W 条数据就报错 `ERROR 8004 (HY000): transaction too large, len:300001 ?`

当然更具体的原因和解决办法在 asktug、简书等上面大家可以自行搜索，这里不作赘述。不过好消息是，4.0 版本重大改进，TiDB 终于支持大事务了，下面就带大家一起来探索和体验一下。

### 6.3.2 大事务实现原理

在 4.0 版本之前对于事务的严格限制的原因有很多，但影响最大的是这两点：

- Prewrite 写下的锁会阻塞其他事务的读，大事务的 Prewrite 时间长，阻塞的时间也就长。
- 大事务 Prewrite 时间长，可能会被其他事务终止导致提交失败。

4.0 大事务实际上是对事务机制的优化，适用于所有事务模型。

#### 6.3.2.1 Min Commit Timestamp

以乐观事务为例，TiDB 支持的是 Snapshot Isolation，每个事务只能读到在事务 `start timestamp` 之前最新已提交的数据。在这种隔离级别下如果一个事务读到了锁需要等到锁被释放才能读到值，原因是有可能这个锁所属的事务已经获取了 `commit timestamp` 且比读到锁的事务 `start timestamp` 小，读事务应该读到写事务提交的新值。

为了实现写不阻塞读，TiDB 在事务的 Primary Lock 里保存了 `minCommitTs`，即事务提交时满足隔离级别的最小的 `commit timestamp`。读事务读到锁时会使用自己的 `start timestamp` 来更新锁对应事务的 Primary Lock 里的该字段，从而将读写事务进行了强制排序，保证了读事务读不到写事务提交的值，从而实现了写不阻塞读。

#### 6.3.2.2 Time to live(TTL)

从前面乐观事务部分得知，Percolator 将事务的所有状态都保存在底层存储系统中，Prewrite 也会写下锁用于避免写写冲突，但如果事务在提交过程中 TiDB 挂掉会导致事务遗留下大量的锁阻塞其他事务的执行。TiDB 使用 TTL 来限制锁的存在时间，当锁超时时就会终止对应的事务并清理掉锁，从而当前事务可以继续执行。

在 v4.0 之前，锁的 TTL 是根据事务大小计算得来的，无法反应事务真实的运行情况，有可能运行中事务的锁超时并被其他事务清理掉，最终导致事务提交失败。在 v4.0 中将会使用 `TTL Manager` 实时更新事务 Primary Lock 中的 TTL，从而保证运行中的事务不会被其他事务终止掉。

### 6.3.3 实践

大事务这个功能该如何使用呢？其实大事务是对事务机制的优化，唯一需要修改的是 TiDB 的配置文件，找到这一处配置：

```
[performance]
txn-total-size-limit = 104857600
```

然后把数字调大就可以了，然后就可以愉快地继续使用了，比如说往后面多加两个零，只要在 10737418240(10G) 以内就行。来直观感受下在 v3.x 版本和 v4.0 版本执行一个插入几十万条数据语句的情况。

(1) 3.0.5 版本：

```
mysql> insert into t1 (name, age) select name, age from t1;
Query OK, 131072 rows affected (1.86 sec)
Records: 131072 Duplicates: 0 Warnings: 0

mysql> select count(1) from t1;
+-----+
| count(1) |
+-----+
| 262144 |
+-----+
1 row in set (0.14 sec)

mysql> insert into t1 (name, age) select name, age from t1;
ERROR 8004 (HY000): transaction too large, len:300001
```

(2) 4.0 版本：

```
MySQL [test]> select count(1) from t1;
+-----+
| count(1) |
+-----+
| 262144 |
+-----+
1 row in set (0.20 sec)

MySQL [test]> insert into t1 (name, age) select name, age from t1;
Query OK, 262144 rows affected (9.20 sec)
Records: 262144 Duplicates: 0 Warnings: 0

MySQL [test]> select count(1) from t1;
+-----+
| count(1) |
+-----+
| 524288 |
+-----+
1 row in set (0.52 sec)

MySQL [test]> create table t2 like t1;
Query OK, 0 rows affected (0.11 sec)

MySQL [test]> insert into t2 select * from t1;
Query OK, 524288 rows affected (17.61 sec)
Records: 524288 Duplicates: 0 Warnings: 0
```

在 4.0 之前版本，如果执行几十万条数据的插入或复制操作，需要用特殊方法来才绕过事务限制，比如开启 `tidb_batch_insert` 等，但是这些操作是受限且不安全的，未来版本中会被逐渐废弃掉。而在新版本中，可以直接通过大事务支持的特性，来解决老版本中事务限制的问题。

### 6.3.4 限制和改进

TiDB 内部的 GC 执行策略默认是 10min 执行一次，如果事务执行时间太长会因超出 `gc_life_time` 而报错，而大事务执行时间一般都较长。在 v4.0 中，`safepoint` 会根据运行中事务的 `start timestamp` 计算得出，从而不影响大事务的提交。当然在新版本中这个功能依然会有一些受限，具体如下：

- 单个 kv 大小限制 6MB 的限制还在，这是存储引擎层的限制，也就是依然不建议类似 `Blob` 等超长字段存放在 TiDB 中。
- 目前单个事务大小限制在 10GB，超过 10GB 的事务依然会报错，不过 10GB 的事务已经能够覆盖大多数场景了。
- 事务对内存的占用可能会有 3~4 倍的放大，10GB 大的事务可能会占用 30~40GB 的内存。如果需要执行特别大的事务，需要提前做好内存的规划，避免对业务产生影响。

## 第 7 章 TiDB DDL

目前很多数据库在执行 DDL 操作时会锁表。那么在这段时间内，很多涉及此表的业务都处于阻塞状态（有些数据库支持读操作，但是也以消耗大量内存为代价），且表越大，影响时间越久。TiDB 是根据 Google F1 的在线异步 schema 变更算法实现，并做了一些优化。

本章节主要介绍的是 TiDB DDL 在实际中的应用和管理，以及 4.0 版本的两个新特性。此外，也会简单讲解其中涉及到的理论知识。本章主要分为四个章节，分别如下：

1. 表结构设计最佳实践。此章节主要围绕 3 个场景讲述如何在实际场景中构建最佳表结构，以及一些注意事项。
2. 如何查看 DDL 状态。此章节先描述对 DDL 任务的管理和相关参数的控制，之后讲解了 DDL 操作流程和原理。
3. Sequence。此章节介绍了其语法和实际的 3 种应用场景。
4. AutoRandom。此章节主要描述了其功能特性和使用示例，以及与其他方案的比较。

# 7.1 表结构设计最佳实践

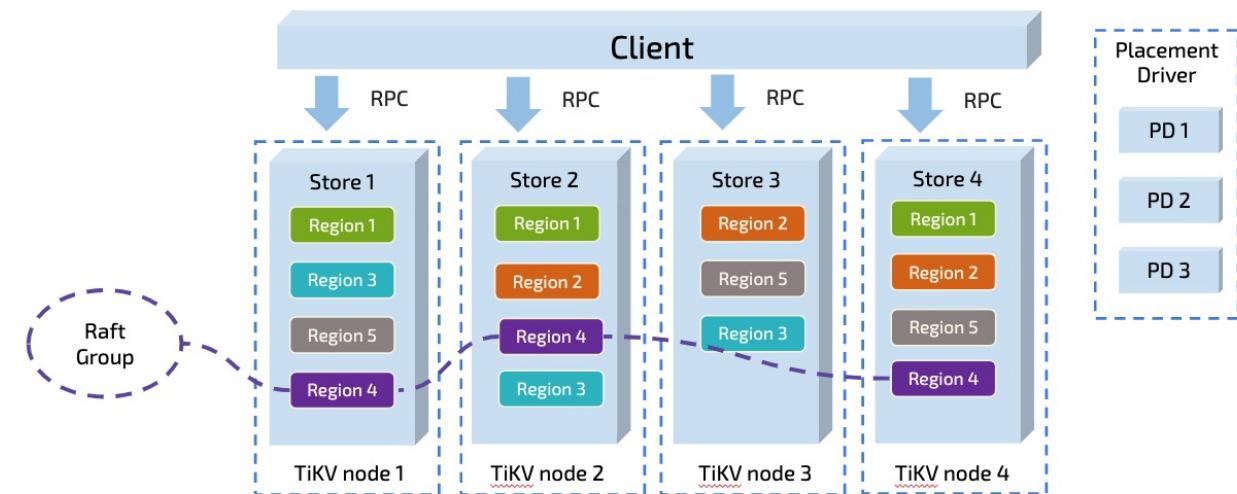
## 7.1.1 概述

在关系型数据库中，合理的表结构 (schema) 设计可以带来稳定和高效的性能，但是反过来说，如果表结构设计不合理，就可能会引起比较大的性能问题或者额外的工作成本，所以一般公司都会对业务提交的表结构进行自动的（通过审核工具）或者人工的审核，在业务上线前发现和解决这类问题。本文从表设计相关的一些 case 来总结和分享经验，给大家提供参考，避免线上踩坑。

## 7.1.2 背景知识

表结构相关的问题许多都是和数据分布有关的，所以这里简单介绍下 TiDB 数据分布相关的知识。

TiDB 以 Region 为单位对数据进行切分，每个 Region 有大小限制（默认为 96MB）。Region 的切分方式是范围切分。每个 Region 会有多个副本，每一组副本，称为一个 Raft Group。每个 Raft Group 中由 Leader 负责执行这块数据的读和写（TiDB 即将支持 [Follower-Read](#)）。Leader 会自动地被 PD 组件均匀调度在不同的物理节点上，以均分读写压力。



每个表对应了多个 Region，一个 Region 只会对应一个表，每一个 Region 里是一组有序的数据记录。这一块不太了解的请阅读下 TiDB 经典文章：[谈存储、谈调度](#)，这里篇幅原因不再进行详细说明。

下面来进行一些场景的分享。

## 7.1.3 典型场景

### 1. 大表高并发写入的性能瓶颈

#### 场景描述

在 MySQL 上，一般使用 InnoDB 存储引擎，这时候 DBA 都会建议使用自增 ID 作为主键，这样可以提升写入性能（随机写变顺序写）和降低数据页的碎片率。

在 TiDB 中，一个表被逻辑的按照主键顺序切分成了多个 Region，所以如果 TiDB 中也是用递增主键的话，写入的数据就会主要写入到最后一个 Region 里，Region 又是 PD 调度的最小单位，所以这个 Region 在业务写入量较大的情况下，就会是一个热点 Region，该 Region 所在的 TiKV 的能力决定了这个表甚至集群的写入能力。如果是多个 Region 出现热点读写，TiDB 的热点调度程序是可以将多个热点 Region 调度到不同的 TiKV 实例上来分散压力的。但是这个场景下只有一个热点 Region，因此无法通过调度来解决。

#### 如何定位

怎么定位到这类问题呢？以下以 TiDB 3.0 版本进行说明。

首先查看 Grafana 监控：*Grafana > TiKV-Trouble-Shooting > Hot write*

在这种场景下，以上监控项可以看到有某个 store 的 QPS 以及 CPU 明显高于其他节点，可以通过分析慢查询找到写入慢的表，或者通过 pd-ctl 找到写入量最大的 Region 的 region\_id：

```
pd-ctl -u "http://{{pd-instance}}:2379" -d region topwrite
```

然后通过 region\_id 找到对应的表或者索引：

```
curl http://{{tidb-instance}}:10080/regions/{{region_id}}
```

### 如何解决

既然 Region 是调度的最小单位，那就要想办法把写入的数据尽量打散到不同的 Region，以通过 Region 的调度来解决此类写入瓶颈问题。TiDB 的隐藏主键可以实现此类打散功能。

TiDB 对于 PK 非整数或没有 PK 的表，会使用一个隐式的自增主键 rowID：`_tidb_rowid`，这个隐藏的自增主键可以通过设置 `SHARD_ROW_ID_BITS` 来把 rowID 打散写入多个不同的 Region 中，缓解写入热点问题。但是设置的过大也会造成 RPC 请求数放大，增加 CPU 和网络开销。

- `SHARD_ROW_ID_BITS = 4` 表示  $2^4$  (16) 个分片
- `SHARD_ROW_ID_BITS = 6` 表示  $2^6$  (64) 个分片
- `SHARD_ROW_ID_BITS = 0` 则表示默认值 1 个分片

使用示例：

```
CREATE TABLE t (c int) SHARD_ROW_ID_BITS = 4;
ALTER TABLE t SHARD_ROW_ID_BITS = 4;
```

`SHARD_ROW_ID_BITS` 的值可以动态修改，每次修改之后，只对新写入的数据生效。可以根据业务并发度来设置合适的值来尽量解决此类热点 Region 无法打散的问题。

另外在 TiDB 3.1.0 版本中还引入了一个新的关键字 `AUTO_RANDOM`（实验功能），这个关键字可以声明在表的整数类型主键上，替代 `AUTO_INCREMENT`，让 TiDB 在插入数据时自动为整型主键列分配一个值，消除行 ID 的连续性，从而达到打散热点的目的，更详细的信息可以参考 [AUTO\\_RANDOM 详细说明](#)。

## 2. 新表高并发读写的瓶颈问题

### 场景描述

这个场景和 [TiDB 高并发写入常见热点问题及规避方法](#) 中的 case 类似。

有一张简单的表：

```
CREATE TABLE IF NOT EXISTS TEST_HOTSPOT(
    id      BIGINT PRIMARY KEY,
    age     INT,
    user_name VARCHAR(32),
    email   VARCHAR(128)
)
```

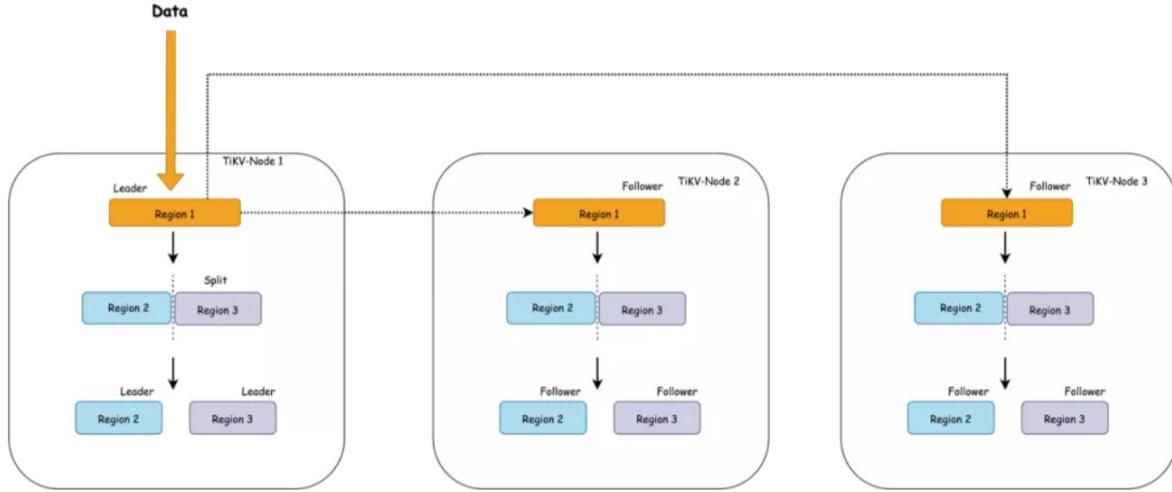
这个表结构非常简单，除了 `id` 为主键以外，没有额外的二级索引。写入的语句如下，`id` 通过随机数离散生成。短时间内密集地写入数据。乍一看这个场景是随机主键写入，应该不会有性能问题，业务上没有热点产生，只要有足够的机器，就可以充分利用 TiDB 的分布式能力了。

但是在在线上却在这个场景下遇到了热点问题，为什么呢？

造成这个现象的原因是：刚创建表的时候，这个表在 TiKV 只会对应为一个 Region，范围是：

```
[CommonPrefix + TableID, CommonPrefix + TableID + 1)
```

对于在短时间内的大量写入，它会持续写入到同一个 Region。



上图简单描述了这个过程，持续写入，TiKV 会将 Region 切分。但是由于是由原 Leader 所在的 Store 首先发起选举，所以大概率下旧的 Store 会成为新切分好的两个 Region 的 Leader。对于新切分好的 Region 2, 3。也会重复之前发生在 Region 1 上的事情。也就是压力会密集地集中在 TiKV-Node 1 中。

在持续写入的过程中，PD 能发现 Node 1 中产生了热点，它就会将 Leader 均分到其他的 Node 上。如果 TiKV 的节点数能多于副本数的话，还会发生 Region 的迁移，尽量往空闲的 Node 上迁移，在持续写入一段时间以后，整个集群会被 PD 自动地调度成一个压力均匀的状态，到那个时候才会真正利用整个集群的能力。对于大多数情况来说，这个是没有问题的，这个阶段属于表 Region 的预热阶段。

但是对于高并发批量密集写入场景来说，这个却是应该避免的。

那么我们能否跳过这个预热的过程，直接将 Region 切分为预期的数量，提前调度到集群的各个节点中呢？

### 解决方法

#### 一、建表时将该表配置 Region 打散

使用带有 shard\_row\_id\_bits 的表时，如果希望建表时就均匀切分 Region，可以考虑配合 pre\_split\_regions 一起使用（pre\_split\_regions 必须小于等于 shard\_row\_id\_bits），用来在建表成功后就开始预均匀切分  $2^{(pre\_split\_regions)}$  个 Region。语法如下：

```
CREATE TABLE t (a INT, b INT, INDEX idx1(a)) SHARD_ROW_ID_BITS = 4 PRE_SPLIT_REGIONS = 2;
```

#### 二、如果表已存在则可以用 Split Region 命令打散 Region

Split Region 是 TiDB 的预切分 Region 的功能，可以根据指定的参数，预先为某个表切分出多个 Region，并打散到各个 TiKV 上去。语法有两种：

- 均匀切分：BETWEEN lower\_value AND upper\_value REGIONS region\_num 语法是通过指定上、下边界和 Region 数量，然后在上、下边界之间均匀切分出 region\_num 个 Region

```
SPLIT TABLE table_name [INDEX index_name] BETWEEN (lower_value) AND (upper_value) REGIONS region_num
```

- 不均匀切分：BY value\_list... 语法将手动指定一系列的点，然后根据这些指定的点切分 Region，适用于数据不均匀分布的场景

```
SPLIT TABLE table_name [INDEX index_name] BY (value_list) [, (value_list)] ...
```

不过需要注意的是，如果线上打开了 Region Merge 功能，通过以上两种方式 split 的 Region，在 split 超过 split-merge-interval 时间（默认一个小时）后，如果 Region 数据量还是比较小，满足 Region Merge 的条件，就会触发 Region Merge，再次导致热点问题。

所以想彻底解决这个问题，TiDB 4.0 以下版本可以通过分区表的方式，将热点数据根据分区键打散到各个分区来解决。在未来的 TiDB 4.0 中，PD 会提供 Load Based Splitting 策略，除了根据 Region 的大小进行分裂之外，还会根据访问 QPS 负载自动分裂频繁访问的小表的 Region。

### 3. 普通表清理大量数据相关问题

#### 场景描述

在互联网场景下，线上数据库中的数据一般都有默认的保留时间，比如保留最近三个月、一年和三年等。这样就会有大量的线上过期数据 delete 操作，因为 TiDB 的 KV 层存储用的是 RocksDB，所以 delete 操作过程可以简单描述为：

1. TiDB 从 TiKV 读取符合条件的数据
2. 执行 delete 操作
3. 在 TiKV 写入这条记录的删除记录
4. 等待 RocksDB Compaction 之后，这条记录才被真正的删除

从以上过程可以看到删除的工作很重，删除的速度也就会比较慢，效率非常低下，线上可能会出现删除速度跟不上写入速度情况出现，这样既无法满足业务清理数据需求，又增加了存储和计算层的压力，进而导致删除数据期间性能抖动比较明显，影响线上正常业务。

#### 如何解决

针对这个场景，TiDB 的分区表可以比较好的解决，分区表的每一个分区都可以看做是一个独立的表，这样如果业务要按照日期清理数据，只需要按照日期建立分区，然后定期去清理指定的日期之前的分区即可。清理分区走的是 Delete Ranges 逻辑，简单过程是：

1. 将要清理的分区写入 TiDB 的 gc\_delete\_range 表
2. Delete Ranges 会 gc\_delete\_range 表中时间戳在 safe point 之前的区间进行快速的物理删除

这样就减少了 TiDB 和 TiKV 的数据交互，既避免了往 TiKV 写入大量的 delete 记录，又避免了 TiKV 的 RocksDB 的 compaction 引起的性能抖动问题，从而彻底的解决了清理数据慢影响大的问题。

分区表的使用和限制见 [官方文档](#)。

## 4. 其它

受限于篇幅问题，以下的一些 case 就不做详细介绍，简单总结下：

- 避免高并发写入的表上存在顺序写入的二级索引，因为这样会造成写入热点，并且暂时没有较好的解决方案
- 提前确认 [TiDB 和 MySQL 相比不同或不支持的特性](#)，比如不允许降低字段长度、不允许修改 DECIMAL 的精度等，如果上线后发现必需要进行相关操作，那只能通过重建表迁移数据的方式，对业务影响很大，风险也很大
- 不要在线上搞过大的宽表以及大量索引
- 线上遇到一个 case 是，一个类似数据中台的服务，把从其它存储获取的数据写入到 TiDB 的一张宽表（70 个字段）上，迁移过来的时候基本每个字段都带有索引，业务解释多个索引是为了兼顾业务多样性的需求，最后业务上线后 TP99 线直接飙升到 130ms，业务无法接受延迟，最后在推进优化为 7 个组合索引后，TP99 恢复到 60ms，达到业务预期。

### 7.1.4 总结

以上 case 是在线上遇到的一些常见的表设计相关的问题，主要是热点、GC 或者功能相关的问题。这些问题可能会对线上稳定性造成比较严重的冲击，希望大家在线上可以根据不同场景采用不同的解决方案，规避掉这些风险，保障线上数据库服务的正常运行。



## 7.2 如何查看 DDL 状态

### 7.2.1 TiDB DDL 特点

大多数数据库执行 DDL 操作时，或多或少会对正在访问该数据库的 SQL 产生影响。例如，在执行 DDL 期间可能会有锁表操作，此时访问该表的 SQL 会被阻塞。因此，一般在表结构设计阶段都会尽量避免后续产生 DDL 操作。如果必须执行 DDL 操作，也只能选择在业务低高峰期操作，尽量减少对线上业务的影响。

TiDB 上的 DDL 操作，不会阻塞任何该数据库上正在执行的 SQL，对业务的 SQL 访问和对 DBA 运维都极为友好，这也是 TiDB 相较于其他数据库产品的一大优势所在。

### 7.2.2 对 DDL 进行管理

TiDB 对 MySQL 语法进行了扩展，通过 ADMIN 语句对 DDL 操作进行管理。下面罗列了 DDL 管理的基本命令，各命令返回结果中各字段的详细含义，请参考[官方 admin 相关文档](#)。

- 查看当前 schema version，owner 信息以及正在执行的 DDL 任务。

```
ADMIN SHOW DDL;
```

- 查看当前未执行完成的 DDL 任务(包括正在运行的 DDL 任务和等待运行的任务)以及最近 NUM 条(默认 10 )已经执行完成的 DDL 任务。  
ADMIN SHOW DDL JOBS [NUM] [WHERE where\_condition];
  - 例如，显示当前未完成的 DDL 任务，以及最近 5 条已经执行完成的 DDL 任务。

```
ADMIN SHOW DDL JOBS 5;
```

- 例如，显示 test 数据库中未执行完成的 DDL 任务，以及最近 5 条已经执行完成但执行失败的 DDL 任务。

```
ADMIN SHOW DDL JOBS 5 WHERE state != 'synced' AND db_name = 'test';
```

- 根据 JOB\_ID 查询具体的 DDL 语句。

```
ADMIN SHOW DDL JOB QUERIES job_id [, job_id] ...;
```

- 取消正在执行中的 DDL 任务。

```
ADMIN CANCEL DDL JOBS job_id [, job_id] ...;
```

- 通过 JOB\_ID 恢复表，等价于：RECOVER TABLE table\_name。

```
RECOVER TABLE BY JOB ddl_job_id;
```

特别注意 在执行一些 DDL 操作时（如 `ADD INDEX`），由于执行时间较长，不会立即返回执行结果，mysql-client 会处于卡死的状态。此时可以放心的 `ctrl+c` 来终止该连接，不会影响 DDL 的实际执行。DDL 正常耗时可以参考[相关官方文档](#)。

```
> admin@sbtest02:04:32>alter table sbtest1 add key idx_c_pad(c, pad);
^CCtrl-C -- sending "KILL QUERY 15768094" to server ...
Ctrl-C -- query aborted.
^CCtrl-C -- exit!
Aborted
```

通过 `ADMIN SHOW DDL` 查询确认，被 `ctrl+c` 之后，该 DDL 依旧正常运行。

```
> admin@(none)02:04:56>ADMIN SHOW DDL\G
***** 1. row *****
SCHEMA_VER: 6765
OWNER_ID: 828a4567-91a5-4070-b55f-f90702e80e7a
OWNER_ADDRESS: 10.40.216.9:4000
RUNNING_JOBS: ID:8406, Type:add index, State:running, SchemaState:write reorganization, SchemaID:7299, TableID:7380, RowCount:196608, ArgLen:0, start time: 2020-03-07 14:04:53.321 +0800 CST, Err:<nil>, ErrCount:0, SnapshotVersion:415121039339290634
SELF_ID: 828a4567-91a5-4070-b55f-f90702e80e7a
QUERY: alter table sbtest1 add key idx_c_pad(c, pad)
1 row in set (0.02 sec)
```

除此之外，也可以通过访问 TiDB 提供的 HTTP 接口查看当前 owner 所在 TiDB，以及各个 TiDB 节点 `ddl_id`、`lease` 等信息，用法如下：

```
# 用法
curl http://{TiDBIP}:10080/info/all
# 例如
$curl http://127.0.0.1:10080/info/all
{
  "servers_num": 2,
  "owner_id": "29a65ec0-d931-4f9e-a212-338eaeffab96",
  "is_all_server_version_consistent": true,
  "all_servers_info": [
    {
      "29a65ec0-d931-4f9e-a212-338eaeffab96": {
        "version": "5.7.25-TiDB-v4.0.0-alpha-669-g8f2a09a52-dirty",
        "git_hash": "8f2a09a52fdcaf9d9bfd775d2c6023f363dc121e",
        "ddl_id": "29a65ec0-d931-4f9e-a212-338eaeffab96",
        "ip": "",
        "listening_port": 4000,
        "status_port": 10080,
        "lease": "45s",
        "binlog_status": "Off"
      },
      "cd13c9eb-c3ee-4887-af9b-e64f3162d92c": {
        "version": "5.7.25-TiDB-v4.0.0-alpha-669-g8f2a09a52-dirty",
        "git_hash": "8f2a09a52fdcaf9d9bfd775d2c6023f363dc121e",
        "ddl_id": "cd13c9eb-c3ee-4887-af9b-e64f3162d92c",
        "ip": "",
        "listening_port": 4001,
        "status_port": 10081,
        "lease": "45s",
        "binlog_status": "Off"
      }
    }
  ]
}
```

### 7.2.3 DDL 相关参数

#### 参数

- `tidb_ddl_reorg_worker_cnt`

属性	值
作用域	GLOBAL
默认值	4
作用	控制数据回填 (re-organize) 阶段的并发度

- `tidb_ddl_reorg_batch_size`

属性	值
作用域	GLOBAL
默认值	256
最小值	32
最大值	10240
作用	控制数据回填 (re-organize) 阶段一次回填的数据量

- `tidb_ddl_reorg_priority`

属性	值
作用域	GLOBAL   SESSION
默认值	PRIORITY_LOW
作用	控制数据回填 (re-organize) 阶段执行的优先级

- `tidb_ddl_error_count_limit`

属性	值
作用域	GLOBAL
默认值	512
作用	控制 DDL 操作失败重试的次数，重试次数超过该值，则取消 DDL 操作

## 使用场景

TiDB 集群中，用户执行的 DDL 操作分两类：普通 DDL 操作和加索引操作。普通 DDL 操作执行时间短，一般秒级就可以执行完成；而加索引操作由于需要回填数据，因此执行时间略长。而在回填数据期间，需要将回填的数据写入 TiKV，对 TiKV 会产生额外的写入压力，从而造成一些性能影响。相关的测试可以参考：[线上负载与 ADD INDEX 相互影响的测试](#)。

TiDB 提供了参数 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 用来控制回填数据的速度。通过调整参数，可以在业务访问高峰到来时降低 DDL 速度，保证对业务的正常访问无影响；而在业务访问低峰增加 DDL 速度，从而更快的完成 DDL 任务。

### 注意

- 参数 `tidb_ddl_reorg_priority` 调整优先级，可能会对正常的 SQL 请求有一定影响，一般默认值即可。
- 参数 `tidb_ddl_error_count_limit` 则用来控制重试次数，当发生异常（诸如由于超时、TiKV 无法连接等）时，可进行重试；超过重试次数则终止当前 DDL，一般默认值即可。

## 7.2.4 DDL 处理流程

TiDB-Server 作为 SQL 的统一入口，DDL 操作也首先经由 TiDB-Server 处理，TiDB 可以多点写入，也就是不同的 TiDB-Server 可以同时接受 DDL 操作请求。

为了保证 TiDB-Server 异常重启而丢失 DDL 信息，首先 TiDB-Server 会将 DDL 操作封装成一个拥有唯一标识的 DDL Job，存储到 TiKV 上的任务队列中，持久化保存。

每个 TiDB-Server 上都拥有执行 DDL 任务的 worker。但是，各个 TiDB-Server 会竞选出唯一一个 Owner 节点来执行实际 DDL 任务，其他竞选失败的 TiDB-Server 节点虽然可以接受 DDL 请求，但是不负责执行 DDL 任务。Owner 会定期对自己的 Owner 身份续租。如果当前 Owner 出现异常，剩余的节点会再次竞选 Owner。

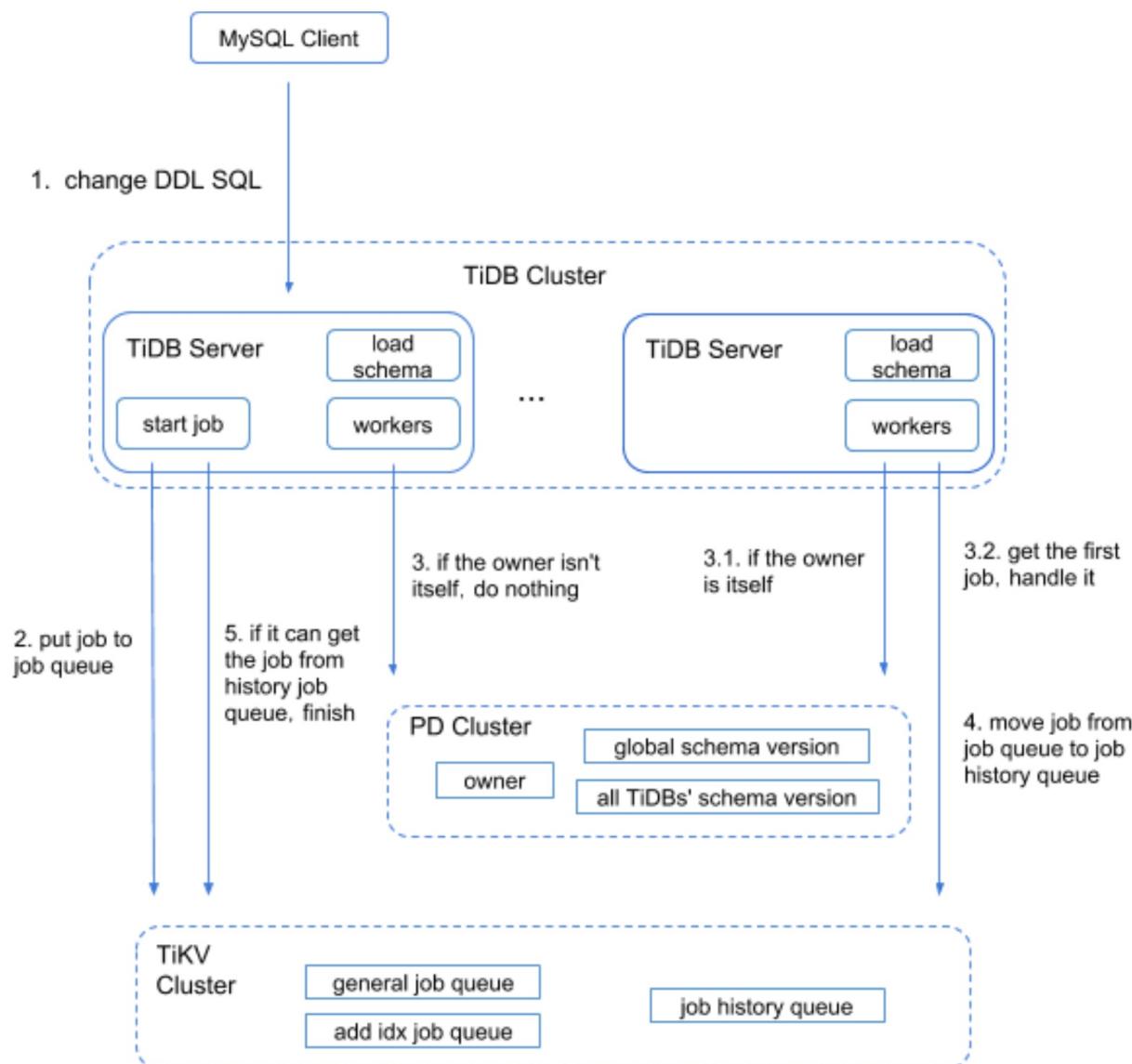
Owner 节点上的 worker 会从任务队列中依次取出 DDL 任务执行。然而由于 `ADD INDEX` 类型的 DDL 任务在数据量很大情况下执行时间特别长（需要回填数据），从而导致其余的 DDL 操作会被阻塞，TiDB 对该处进行了优化。将任务队列一分为二，一个队列用来存储非 `ADD INDEX` 类型的 DDL 任务，一个队列用来存储 `ADD INDEX` 类型的 DDL 任务。Owner 上的不同

的 worker 也会从不同的队列中获取对应的 DDL 任务进行执行。当然，该优化也会引入任务依赖问题。任务依赖问题指的是，同一张表的 DDL 任务，任务编号小的需要先执行。因此在执行 DDL 之前需要对依赖进行检查。

Owner 上的 worker 在处理 `ADD INDEX` 类型 DDL 任务时，涉及到回填数据的过程。该过程会启动 `tidb_ddl_reorg_worker_cnt` 个线程，每次每个线程处理 `tidb_ddl_reorg_batch_size` 大小的数据。因此通过调整这两个参数，可以动态控制 DDL 执行速度。

当 DDL 任务执行完成之后，会将执行完成的 DDL 任务移动至历史任务队列（job history queue）中，方便后续通过命令进行 DDL 任务的历史查询。

完整的 DDL 执行流程如下图所示，更详细的内容，可以参考 [DDL 源码解析](#)。



## 7.2.5 DDL 变更原理

TiDB 在线表变更的原理借鉴自论文《Online, Asynchronous Schema Change in F1》，通过在表结构变更过程中引入额外状态从而实现了一套在线表变更协议，使得集群存在相邻两个版本的 schema 时候，不会破坏数据完整性或发生数据异常。因此，在实现上，要求集群中在同一个表上所有 schema 版本最多存在两个相邻的版本。

例如，以 `ADD INDEX` 类型的 DDL 操作为例，其状态变化如下表所示：

schema 版本	状态	说明
schema version 1	absent	添加索引之前。
schema version 2	delete only	schema 元信息已经修改，但此时对外不可见。内部索引数据不可添加/更新，仅可删除。
schema version 3	write only	此时对外不可见，内部索引数据可任意修改。
schema version 4	write reorganization	内部索引数据可任意修改，并且进行索引的回填。
schema version 5	public	此时对外可见，索引添加完成。

TiDB 在该论文的基础上又进行了一些优化。例如，在执行 `ADD COLUMN` 类型 DDL 时，TiDB 并没有对数据进行回填，而是将最新添加的列的 default 值保存到 schema 的“原始默认值”字段中，在读取阶段如果 TiKV 发现该列值为 `null` 并且“原始默认字段”不为 `null`，则会使用“原始默认字段”对该 `null` 列进行填充，并将填充后的结果从 TiKV 返回。通过这种优化，该 DDL 操作不需要关心表中实际行数，可以更快的完成 DDL 变更。

再例如，一些涉及删除数据的 DDL 操作，诸如：`DROP INDEX`，`DROP TABLE`，`DROP DATABASE`，`TRUNCATE TABLE` 等，在实现上，除了要完成和普通 DDL 变更一样的逻辑外，还需要对待删除的数据进行处理。TiDB 做法是，将这些需要删除的数据记录到 `gc_delete_range` 表中，通过 GC 机制，将对应的数据再进行删除。当然如果是删除某列的 DDL 操作，由于目前是行存储模式，删除列的代价会比较大，所以暂时只是在 schema 上进行删除列的标记，并不会实际删除该列的数据。

更详细的 DDL 实现原理及优化细节可以参考 [TiDB DDL architecture](#)。

## 7.3 Sequence

Sequence 是数据库系统按照一定规则自增的数字序列，具有唯一和单调递增的特性。在官方 SQL 2003 标准中，其被定义为“生成连续数值的一种机制，Sequence 既可以是内部对象，也可以是外部对象”。因原生 MySQL 中并未支持 Sequence，所以 TiDB Sequence 的语法参考了 MariaDB、Oracle 和 IBM Db2。

- Create Sequence 语法

```
CREATE [TEMPORARY] SEQUENCE [IF NOT EXISTS] sequence_name
[ INCREMENT [=] BY | = ] INCREMENT ]
[ MINVALUE [=] minvalue | NO MINVALUE | NOMINVALUE ]
[ MAXVALUE [=] maxvalue | NO MAXVALUE | NOMAXVALUE ]
[ START [=] start ]
[ CACHE [=] cache | NOCACHE | NO CACHE]
[ CYCLE | NOCYCLE | NO CYCLE]
[ ORDER | NOORDER | NO ORDER]
[table_options]
```

- Show Create Sequence 语法

```
SHOW CREATE SEQUENCE sequence_name
```

- Drop Sequence

```
DROP [TEMPORARY] SEQUENCE [IF NOT EXISTS] sequence_name
```

- 获取下一个值

```
SELECT NEXT VALUE FOR sequence_name;
SELECT NEXTVAL(sequence_name);
```

- 获取上一个/当前值

```
SELECT PREVIOUS VALUE FOR sequence_name;
SELECT LASTVAL(sequence_name);
```

- 修改当前值

```
SELECT SETVAL(sequence_name, 100);
```

### 7.3.1 用例介绍

本部分将会通过一些案例介绍 TiDB Sequence 的使用方法：

- 并发应用需要获取单调递增的序列号

在使用分布式数据库的场景里，通常应用也是分布式架构，这样多个应用节点之间如何获取唯一且递增的序列号就成为一个难题。在分布式数据库没有 Sequence 的时候，应用基本通过 雪花算法 、 数据库主键自增 等方法实现，业界也有一些较为成熟的方案，比如 Leaf - 美团点评分布式 ID、百度的 uid-generator 等，上述方案中为了解决该问题引入一个新的系统或模块，极大的增加了应用系统的复杂度。接下来我们看看 TiDB 如何通过 Sequence 解决上述问题。

- (1) 首先新建一个 Sequence

```
CREATE SEQUENCE seq_for_unique START WITH 1 INCREMENT BY 1 CACHE 1000 NOCYCLE;
```

## (2) 从不同的 TiDB 节点获取到的 Sequence 值顺序有所不同

如果两个应用节点同时连接至同一个 TiDB 节点，两个节点取到的则为连续递增的值

```
节点 A : tidb[test]> SELECT NEXT VALUE FOR seq_for_unique;
+-----+
| NEXT VALUE FOR seq_for_unique |
+-----+
|          1 |
+-----+
1 row in set (0.00 sec)

节点 B : tidb[test]> SELECT NEXT VALUE FOR seq_for_unique;
+-----+
| NEXT VALUE FOR seq_for_unique |
+-----+
|          2 |
+-----+
1 row in set (0.00 sec)
```

## (3) 如果两个应用节点分别连接至不同 TiDB 节点，两个节点取到的则为区间递增（每个 TiDB 节点上为连续递增）但不连续的值

```
节点 A : tidb[test]> SELECT NEXT VALUE FOR seq_for_unique;
+-----+
| NEXT VALUE FOR seq_for_unique |
+-----+
|          1 |
+-----+
1 row in set (0.00 sec)

节点 B : tidb[test]> SELECT NEXT VALUE FOR seq_for_unique;
+-----+
| NEXT VALUE FOR seq_for_unique |
+-----+
|          1001 |
+-----+
1 row in set (0.00 sec)
```

- 在一张表里面需要有多个自增字段

MySQL 语法中每张表仅能新建一个 `auto_increment` 字段，且该字段必须定义在主键或是索引列上。在 TiDB 中通过 `Sequence` 和生成列，我们可以实现多自增字段需求。

## (1) 首先新建如下两个 Sequence

```
CREATE SEQUENCE seq_for_autooid START WITH 1 INCREMENT BY 2 CACHE 1000 NOCYCLE;
CREATE SEQUENCE seq_for_logid START WITH 100 INCREMENT BY 1 CACHE 1000 NOCYCLE;
```

(2) 在新建表的时候通过 `default nextval(seq_name)` 设置列的默认值

```
CREATE TABLE `user` (
  `userid` varchar(32) NOT NULL,
  `autooid` int(11) DEFAULT 'nextval(`test`.`seq_for_autooid`)',
  `logid` int(11) DEFAULT 'nextval(`test`.`seq_for_logid`)',
  PRIMARY KEY (`userid`)
)
```

## (3) 接下来我们插入几个用户信息进行测试：

```
INSERT INTO user (userid) VALUES ('usera');
INSERT INTO user (userid) VALUES ('userb');
INSERT INTO user (userid) VALUES ('userc');
```

(4) 查询 user 表，可以发现 autoid 和 logid 字段的值按照不同的步长进行自增，且主键仍然在列 userid 上：

```
tidb[test]> select * from user;
+-----+-----+-----+
| userid | autoid | logid |
+-----+-----+-----+
| usera  |      1 |    100 |
| userb  |      3 |    101 |
| userc  |      5 |    102 |
+-----+-----+-----+
3 rows in set (0.01 sec)
```

- 更新数据表中一列值为连续自增的值

假设我们有一张数据表，表中有 20 万行数据，我们需要更新其中一列的值为连续自增且唯一的整数，如果没有 Sequence，我们只能通过应用一条条读取记录，并用 update 更新值，同时还需要分批次提交，但现在我们有了 Sequence，一切都会变得特别简单。

(1) 新建一张测试表

```
tidb[test]> CREATE TABLE t( a int, name varchar(32));
Query OK, 0 rows affected (0.01 sec)
```

(2) 新建一个 Sequence

```
tidb[test]> CREATE SEQUENCE test;
Query OK, 0 rows affected (0.00 sec)
```

(3) 插入 1 万条记录

```
for i in $(seq 1 10000)
do
  echo "insert into t values($((RANDOM%1000)), 'user${i}');" >> user.sql
done
```

```
tidb[test]> select count(*) from t;
+-----+
| count(*) |
+-----+
| 10000 |
+-----+
1 row in set (0.05 sec)
tidb[test]> select * from t;
+-----+-----+
| a     | name   |
+-----+-----+
| 355  | user1  |
| 729  | user2  |
| 684  | user3  |
| 815  | user4  |
| 39   | user5  |
| 294  | user6  |
| 407  | user7  |
| 767  | user8  |
| 246  | user9  |
| 755  | user10 |
| 496  | user11 |
...
...
```

(4) 更新为连续的值

```
tidb[test]> update t set a=nextval(test);
Query OK, 10000 rows affected (0.20 sec)
Rows matched: 10000  Changed: 10000  Warnings: 0
```

(5) 查询结果集,可以看到字段 `a` 的值已经连续自增且唯一

```
tidb[test]> select * from t;
+-----+-----+
| a    | name   |
+-----+-----+
| 1    | user1  |
| 2    | user2  |
| 3    | user3  |
| 4    | user4  |
| 5    | user5  |
| 6    | user6  |
| 7    | user7  |
| 8    | user8  |
| 9    | user9  |
| 10   | user10 |
| 11   | user11 |
| 12   | user12 |
| 13   | user13 |
| 14   | user14 |
| 15   | user15 |
| 16   | user16 |
| 17   | user17 |
| 18   | user18 |
| 19   | user19 |
| 20   | user20 |
...
...
```

### 7.3.2 注意事项

在分布式架构的数据库中实现单调递增序列是比较有难度的，而 `Sequence` 把 `严格递增` 和 `性能` 两方面交给了使用者，在新建 `Sequence` 的时候，可以通过组合 `Order/No Order`（目前尚未实现）和 `Cache/No Cache` 来选择新建高性能的 `Sequence`，亦或是性能较差但递增较为严格的 `Sequence`。

## 7.4 AutoRandom

AutoRandom 是 TiDB 4.0 提供的一种扩展语法，用于解决整数类型主键通过 AutoIncrement 属性隐式分配 ID 时的写热点问题。

### 7.4.1 AutoRandom 功能介绍

在前面的章节提到过，TiDB 的每一行数据都包含一个隐式的 `_tidb_rowid`。`_tidb_rowid` 会被编码到存储引擎的 Key 上，在 TiKV 中，这决定了数据在 TiKV 中的 Region 位置。

如果表的主键为整数类型，则 TiDB 会把表的主键映射为 `_tidb_rowid`，即使用“主键聚簇索引”。在这种情况下，如果表使用了 `AUTO_INCREMENT` 就会造成主键的热点问题，并无法使用 `SHARD_ROW_ID_BITS` 来打散热点。

针对上述热点问题，如果使用 `AUTO_INCREMENT` 仅仅只是用来保证主键唯一性（不需要连续或递增），那么我们可以将 `AUTO_INCREMENT` 改为 `AUTO_RANDOM`，插入数据时让 TiDB 自动为整型主键列分配一个值，消除行 ID 的连续性，从而达到打散热点的目的。

AutoRandom 提供以下的功能：

- 唯一性：TiDB 始终保持填充数据在表范围内的唯一性。
- 高性能：TiDB 能够以较高的吞吐分配数据，并保证数据的随机分布以配合 `PRE_SPLIT_REGION` 语法共同使用，避免大量写入时的写热点问题。
- 支持隐式分配和显式写入：类似列的 AutoIncrement 属性，列的值既可以由 TiDB Server 自动分配，也可以由客户端直接指定写入。该需求来自于使用 Binlog 进行集群间同步时，保证上下游数据始终一致。

### 7.4.2 AutoRandom 语法介绍

在建表时，`AUTO_RANDOM` 关键字可以作为列属性，指定在 TiDB 主键列上。TiDB 4.0 中，列属性语法定义被更新为：

```
column_definition:
  data_type [NOT NULL | NULL] [DEFAULT default_value]
  [AUTO_INCREMENT | AUTO_RANDOM [(length)]]
  [UNIQUE [KEY] | [PRIMARY] KEY]
  [COMMENT 'string']
  [reference_definition]
```

注意，AutoRandom 仅支持主键列，唯一索引列尚不支持，目前也没有支持计划。`AUTO_RANDOM` 关键字后可以指定 Shard Bits 数量，默认为 5。

### 7.4.3 AutoRandom 使用示例

使用 AUTO\_RANDOM 功能前，须在 TiDB 配置文件 `experimental` 部分设置 `allow-auto-random = true`。该参数详情可参见 [allow-auto-random](#)。

以下面语句建立的表为例：

```
tidb> create table t (a int primary key auto_random);
```

此时再执行形如 `INSERT INTO t(b) values...` 的 INSERT 语句，示例如下：

```

tidb> insert into t values (), ();
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

tidb> select * from t;
+-----+
| a    |
+-----+
| 201326593 |
| 201326594 |
+-----+
2 rows in set (0.00 sec)

tidb> insert into t values (), ();
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0

tidb> select * from t;
+-----+
| a    |
+-----+
| 201326593 |
| 201326594 |
| 2080374787 |
| 2080374788 |
+-----+
4 rows in set (0.00 sec)

tidb> select last_insert_id();
+-----+
| last_insert_id() |
+-----+
|      2080374787 |
+-----+
1 row in set (0.00 sec)

```

注意：

- 如果该 INSERT 语句没有指定整型主键列（a 列）的值，TiDB 会为该列自动分配值。该值不保证自增，不保证连续，只保证唯一，避免了连续的行 ID 带来的热点问题。
- 如果该 INSERT 语句显式指定了整型主键列的值，和 AutoIncrement 属性类似，TiDB 会保存该值。
- 若在单条 INSERT 语句中写入多个值，AutoRandom 属性会保证分配 ID 的连续性，同时 `LAST_INSERT_ID()` 返回第一个分配的值，这使得可以通过 `LAST_INSERT_ID()` 结果推断出所有被分配的 ID。

#### 7.4.5 AutoRandom 与其它方案的比较

与 AutoRandom 相比，TiDB 还可以通过其他的方式避免主键自动分配时的写热点问题：

- 使用 [alter-primary-key 配置选项](#)关闭主键聚簇索引，使用 AutoIncrement + `SHARD_ROW_ID_BITS`。

在这种方式下，主键索引被当做普通的唯一索引处理，使得数据的写入可以由 `SHARD_ROW_ID_BITS` 语法打散避免热点，但缺点在于，主键仍然存在索引写入的热点，同时在查询时，由于关闭了聚簇索引，针对主键的查询需要进行一次额外的回表。

- 在主键指定 `UUID()` 函数。

这种做法同样可以为主键自动分配随机的默认值，保证数据和主键不存在写热点问题。但缺点在于 `UUID()` 分配的是字符串类型。TiDB 不支持字符串类型主键的聚簇索引，同样带来主键查询的额外回表。

参考文献：

参阅 [AUTO\\_RANDOM 的详细说明](#)





## 第8章 Titan 简介与实战

在读过分布式存储层 TiKV 的工作原理的相关介绍后，大家对 TiKV 如何将数据可靠的存储在大量服务器组成的集群中有所了解。我们还了解到 RocksDB 是运行在每一个 TiKV 实例存储数据的关键组件。RocksDB 作为一款非常优秀的单机存储引擎，在诸多方面都有着不俗的表现。但其 LSM-tree 的实现原理决定了，RocksDB 存在着数十倍的写入放大效应。在写入量大的应用场景中，这种放大效应可能会触发 IO 带宽和 CPU 计算能力的瓶颈影响在线写入性能。除了写入性能之外，写入放大还会放大 SSD 盘的写入磨损，影响 SSD 盘的寿命。

Titan 是 PingCAP 研发的基于 RocksDB 的高性能单机 key-value 存储引擎，通过把大 value 同 key 分离存储的方式减少写入 LSM-tree 中的数据量级。在 value 较大的场景下显著缓解写入放大效应，降低 RocksDB 后台 compaction 对 IO 带宽和 CPU 的占用，同时提高 SSD 寿命的效果。

### 8.1 Titan 原理介绍

Titan 存储引擎的主要设计灵感来源于 USENIX FAST 2016 上发表的一篇论文 [WiscKey](#)。WiscKey 提出了一种高度基于 SSD 优化的设计，利用 SSD 高效的随机读写性能，通过将 value 分离出 LSM-tree 的方法来达到降低写放大的目的。

在存在大 value 的典型应用场景中，Titan 在写、更新和点读等场景下性能都优于 RocksDB。同原生 RocksDB 相比，Titan 通过一定程度上牺牲硬盘空间和范围查询的性能为代价，来取得更高的写入性能。随着 SSD 单位存储空间价格的降低，和 SSD 设备随机 IO 能力的提升，Titan 的这种设计取舍在未来会产生更加积极的作用。

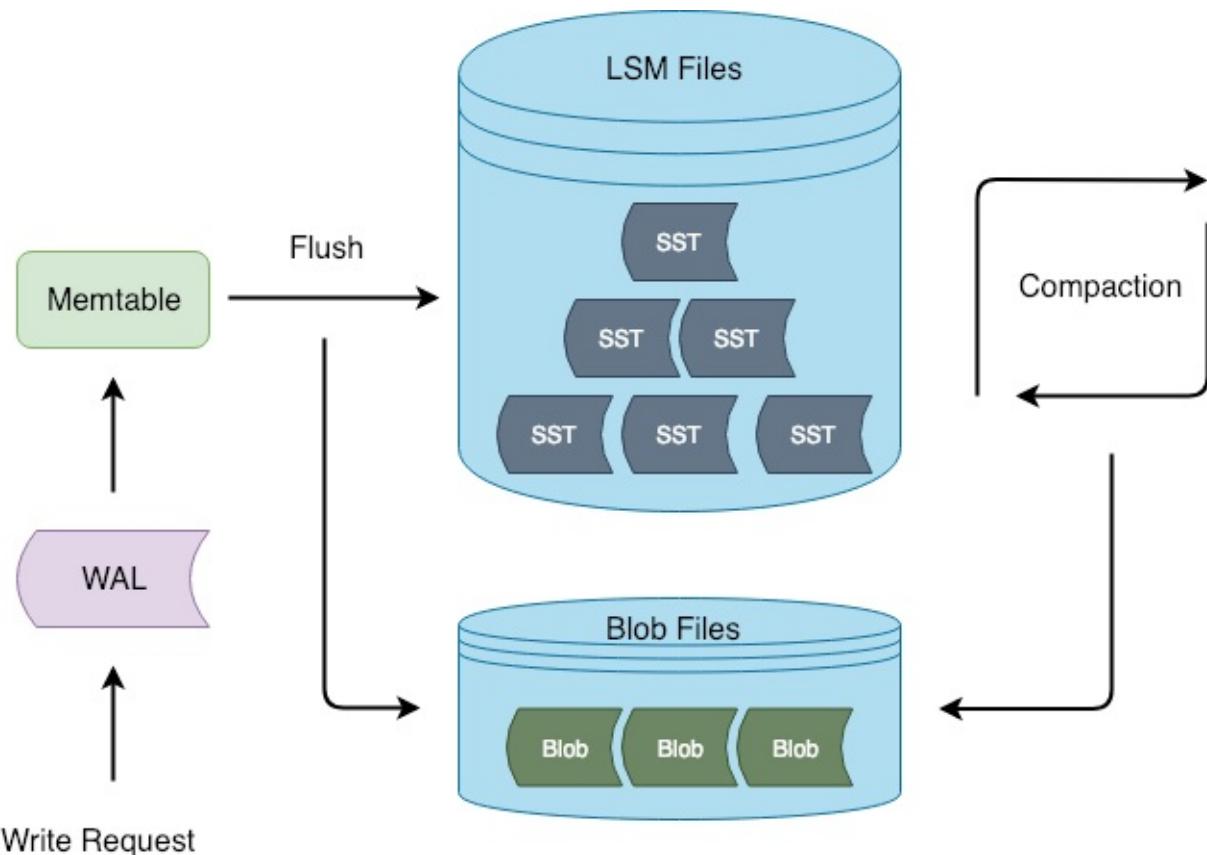
#### 8.1.1 设计目标

作为一个成熟且拥有庞大用户群体的项目，TiKV 在方方面面都需要考虑用户现有系统的平滑过渡。TiKV 使用 RocksDB 作为其底层的存储引擎，因此作为 TiKV 的子项目，Titan 首要的设计目标便是兼容 RocksDB。为用户提供现有基于 RocksDB 的 TiKV 平滑地升级到基于 Titan 的 TiKV。基于这些考虑，我们为 Titan 设定了下面的四个目标：

- 将大 value 从 LSM-tree 中分离出来单独存储，降低 value 部分的写放大。
- 现有 RocksDB 实例无需长期停机转换数据，可在线升级 Titan
- 100% 兼容 TiKV 使用的全部 RocksDB 特性
- 尽量减少对 RocksDB 的侵入性改动，提升 Titan 同未来新版本 RocksDB 的兼容性。

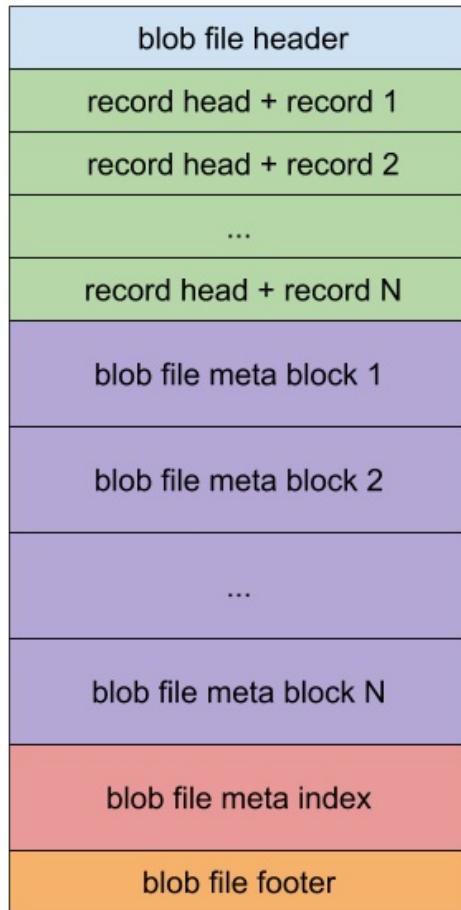
#### 8.1.2 架构与实现

Titan 维持 RocksDB 的写入流程不变，在 Flush 和 Compaction 时刻将大 value 从 LSM-tree 中进行分离并存储到 BlobFile 中。同 RocksDB 相比，Titan 增加了 BlobFile，TitanTableBuilder 和 Garbage Collection (GC) 等组件，下面我们将会对这些组件逐一介绍。



### 8.1.2.1 BlobFile

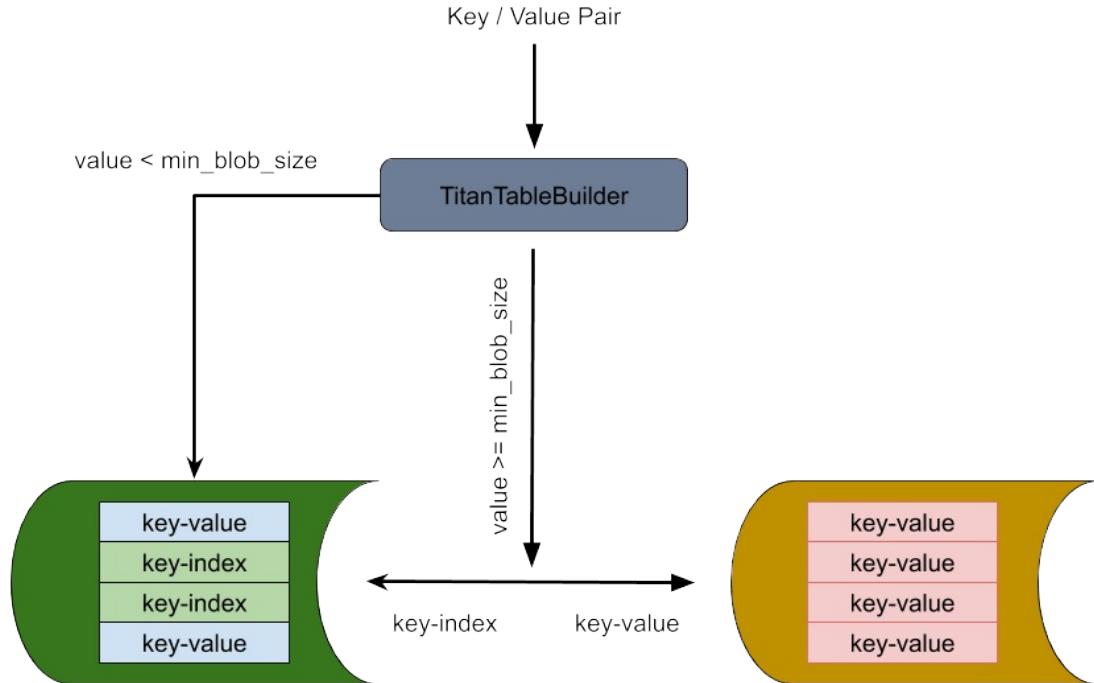
BlobFile 是存放 LSM-tree 中分离得到的 KV 记录的文件，它由 header、record、meta block、meta index 和 footer 组成。其中每个 record 用于存放一个 key-value 对；meta block 用于在未来保存用户自定义数据；而 meta index 则用于检索 meta block。



为了充分利用 prefetch 机制提高顺序扫描数据时的性能，BlobFile 中的 key-value 是按照 key 的顺序有序存放的。除了从 LSM-tree 中分离出的 value 之外，blob record 中还保存了一份 key 的数据。在这份额外存储的 key 的帮助下，Titan 用较小的写放大收获了 GC 时快速查询 key 最新状态的能力。GC 则会利用 key 的更新信息来确定 value 是否已经过期可以被回收。考虑到 Titan 中存储的 value 大小偏大，将其压缩则可以获得较为显著的空间收益。BlobFile 可以选择 Snappy、LZ4 或 Zstd 在单个记录级别对数据进行压缩，目前 Titan 默认使用的压缩算法是 LZ4。

### 8.1.2.2 TitanTableBuilder

RocksDB 提供了 TableBuilder 机制供用户自定义的 table 实现。Titan 则利用了这个能力实现了 TitanTableBuilder，在不对 RocksDB 构建 table 流程做侵入型改动的前提下，实现了将大 value 从 LSM-tree 中分离的功能。



在 RocksDB 将数据写入 SST 时，TitanTableBuilder 根据 value 大小决定是否需要将 value 分离到外部 BlobFile 中。如果 value 大大小于 Titan 设定的大 value 阈值，数据会直接写入到 RocksDB 的 SST 中；反之，value 则会持久化到 BlobFile 中，相应的位置检索信息将会替代 value 被写入 RocksDB 的 SST 文件中用于在读取时定位 value 的实际位置。同样利用 RocksDB 的 TableBuilder 机制，我们可以在 RocksDB 做 Compaction 的时候将分离到 BlobFile 中的 value 重新写入到 SST 文件中完成从 Titan 到 RocksDB 的降级。

\*熟悉另一个 KV 分离存储的 LSM-tree 实现 Badger 的读者可能想问为什么 Titan 没有选择选择将直接用 VLog 的方式保存在 WAL 中，从而避免一次额外的写入放大开销。假设我们将 LSM-tree 的 max level 和放大因子分别设定为 5 和 10，则 LSM-tree 的总写入放大约为  $1 + 1 + 10 + 10 + 10 + 10 = 42$ 。其中由 BlobFile 引入的写入放大同 LSM-tree 的整体写入放大相比仅为 1 : 42，可以忽略不计。并且维持 WAL 也可以避免对 RocksDB 的侵入性改动，这也是 Titan 的重要设计目标之一。

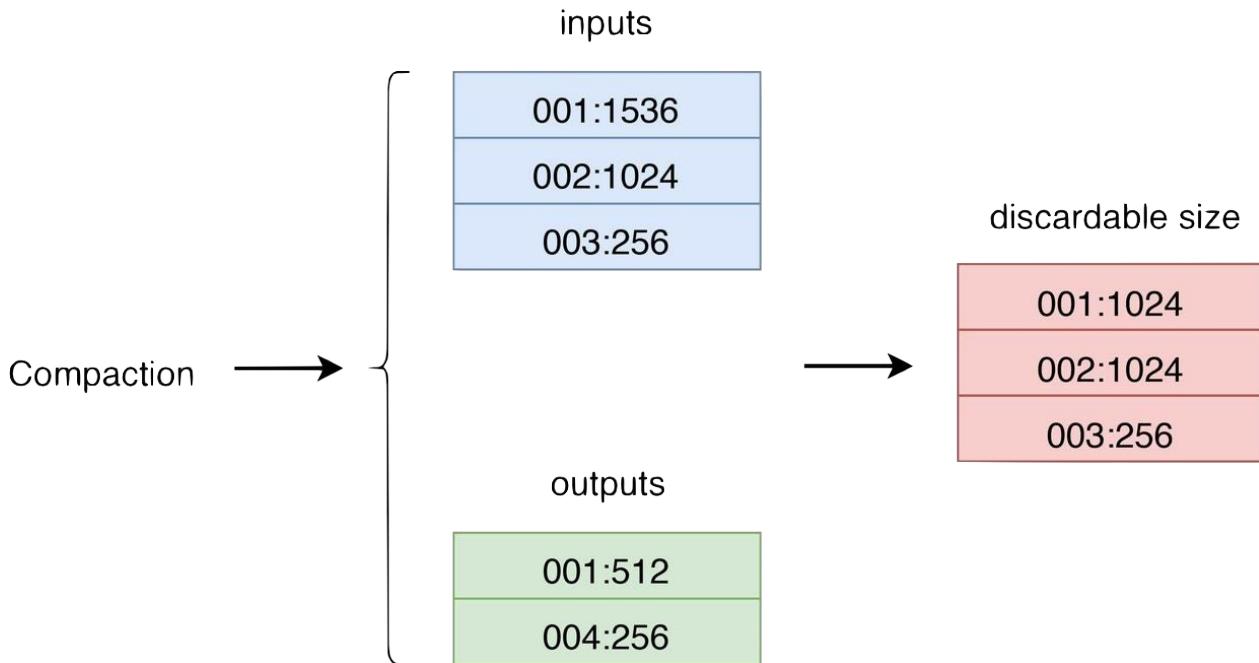
### 8.1.2.3 Garbage Collection

RocksDB 在 LSM-tree Compaction 时对已删除数据进行空间回收。同样 Titan 也具备 Garbage Collection (GC) 组件用于已删除数据的空间回收。在 Titan 中存在两种不同的 GC 方式分别应对不同的适用场景。下面我们将分别介绍「传统 GC」和「Level-Merge GC」的工作原理。

#### 1. 传统 GC

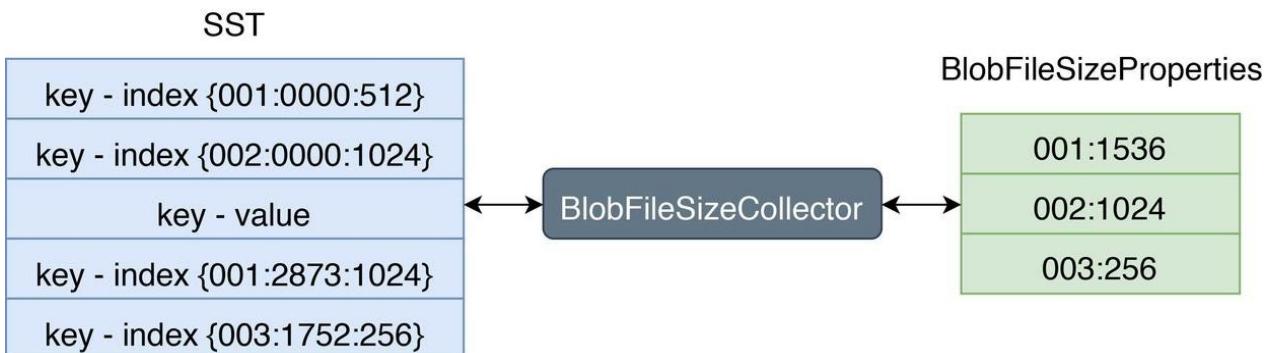
通过定期整合重写删除率满足设定阈值条件的 Blob 文件，我们可以回收已删除数据所占用的存储空间。这种 GC 的原理是非常直观容易理解的，我们只需要考虑何时进行 GC 以及挑选那些文件进行 GC 这两个问题。在 GC 目标文件被选择好后我们只需对相应的文件进行一次重写只保留有效数据即可完成存储空间的回收工作。

首先 Titan 需要决定何时开始进行 GC 操作，显然选择同 RocksDB 一样在 compaction 时丢弃旧版本数据回收空间是最适当的。每当 RocksDB 完成一轮 compaction 并删除了部分过期数据后，在 BlobFile 中所对应的 value 数据也就随之失效。因此 Titan 选择监听 RocksDB 的 compaction 事件来触发 GC 检查，通过搜集比对 compaction 中输出和产出 SST 文件对应的 BlobFile 的统计信息 (BlobFileSizeProperties) 来跟踪对应 BlobFile 的可回收空间大小。



图中 inputs 代表所有参与本轮 compaction 的 SST 文件计算的得到的 BlobFileSizeProperties 统计信息 (BlobFile ID : 有效数据大小) , outputs 则代表新生成的 SST 文件对应的统计信息。通过计算这两组统计信息的变化，我们可以得出每个 BlobFile 可被丢弃的数据大小，图中 discardable size 的第一列是文件 ID 第二列则是对应文件可被回收数据的大小。

接下来我们来关注 BlobFileSizeProperties 统计信息是如何计算得到的。我们知道原生 RocksDB 提供了 TablePropertiesCollector 机制来计算每一个 SST 文件的属性数据。Titan 通过这个扩展功能自定义了 BlobFileSizeCollector 用于计算 SST 中被保存在 BlobFile 中数据的统计信息。



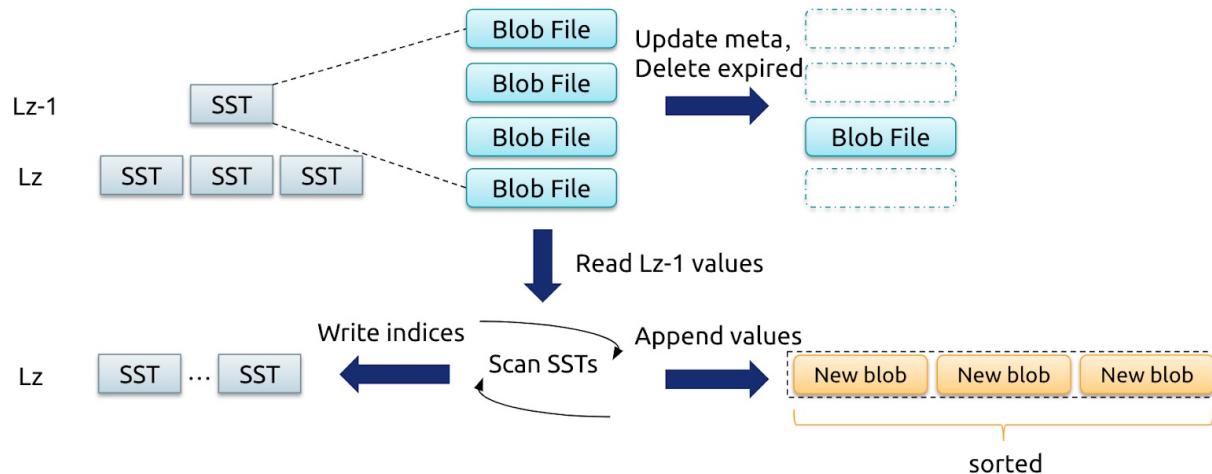
BlobFileSizeCollector 的工作原理非常直观，通过解析 SST 中 KV 分离类型数据的索引信息，它可以得到当前 SST 文件引用了多少个 BlobFile 中的数据以及这些数据的实际大小。

为了更加有效的计算 GC 候选目标 BlobFile，Titan 在内存中为每一个 BlobFile 维护了一份 discardable size 的统计信息。在 RocksDB 每次 compaction 完成后，可以将每一个 BlobFile 文件新增的可回收数据大小累加到内存中对应的 discardable size 上。在重启后，这份统计信息可以从 BlobFile 数据和 SST 文件属性数据重新计算出来。考虑到我们可以容忍一定程度的空间放大（数据暂不回收）来缓解写入放大效应，只有当 BlobFile 可丢弃的数据占全部数据的比例超过一定阈值后才会这个 BlobFile 进行 GC。在 GC 开始时我们只需要从满足删除比例阈值的候选 BlobFile 中选择出 discardable size 最大的若干个进行 GC。

对于筛选出待 GC 的文件集合，Titan 会依次遍历 BlobFile 中每一个 record，使用 record key 到 LSM-tree 中查询 blob index 是否存在或发生过更新。丢弃所有已删除或存在新版本的 blob record，剩余的有效数据则会被重写到新的 BlobFile 中。同时这些存储在新 BlobFile 中数据的 blob index 也会同时被回写到 LSM-tree 中供后续读取。需要注意的是，为了避免删除影响当前活跃的 snapshot 读取，在 GC 完成后旧文件并不能立刻删除。Titan 需要确保没有 snapshot 会访问到旧文件后才可以安全的删除对应的 BlobFile。而这个保障可以通过记录 GC 完成后 RocksDB 最新的 sequence number，并等待所有活跃 snapshot 的 sequence number 超过记录值而达成。

### 1. Level-Merge

传统 GC 的实现直观可靠，但大家可能也发现了在传统 GC 过程中还需要伴随着对 LSM-tree 的大量 record key 查询以及 record index 更新操作。GC 时对 LSM-tree 的大量操作不可避免的会对在线的读取和写入产生压力和负面影响。针对这一问题 Titan 在近期加入了全新的 Level-Merge 策略，它的核心思想是在 LSM-tree compaction 时将 SST 文件相对应的 BlobFile 进行归并重写并生成新的 BlobFile。重写的过程不但避免了 GC 引起的额外 LSM-tree 读写开销，而且通过不断的合并重写过程降低了 BlobFile 之间相互重叠的比例，使得提升了数据物理存储有序性进而提高 Scan 的性能。



Level-Merge 在 RocksDB 对 level  $z-1$  和 level  $z$  的 SST 进行 compaction 时，对所有 KV 对进行一次有序读写，这时就可以对这些 SST 中所使用的 BlobFile 的 value 有序写到新的 BlobFile 中，并在生成新的 SST 时直接保存对应记录的新 blob index。由于 compaction 中被删除的 key 不会被写入到新 BlobFile 中，在整个重新操作完成的同时也就相当于完成了相应 BlobFile 的 GC 操作。考虑到 LSM-tree 中 99% 的数据都落在最后两层，为了避免分层 Level-Merge 时带来的写放大问题，Titan 仅对 LSM-tree 中最后两层数据对应的 BlobFile 进行 Level-Merge。

## 8.2 在 TiDB 集群中开启 Titan

在开启前，我们需要对实际应用进行评估确认 Titan 适合当前业务场景。下面是三个非常重要的评估依据

- Value 大小较大：实际业务中数据大小不会一成不变。满足 value 平均大小较大或者数据中大长度 value 的数据总占比较大的业务场景适合使用 Titan 引擎。目前 Titan 默认设定超过 1KB 的 value 可以被归为大 value，在实际用户场景中发现 512B 以上的 value 同样适用于 Titan。
- 无范围查询或者范围查询性能不敏感：Titan 的数据组织方式决定了数据访问的顺序性较差，相比 RocksDB 在大型范围查询场景下性能较差。在测试中根据不同的业务数据特征，我们发现 Titan 范围查询性能相比 RocksDB 下降 40% 到数倍不等。
- 磁盘空间不敏感：Titan 通过放大磁盘空间占用换取写入放大的降低，Titan 逐行压缩的粒度同 RocksDB 按 block 压缩相比压缩比会低一些。因此通常情况下 Titan 在磁盘空间占用上会比 RocksDB 多。根据经验看在部分极端场景下 Titan 磁盘空间占用可能比 RocksDB 多一倍。

### 8.2.1 开启 Titan 的方式

Titan 对 RocksDB 兼容，也就是说，现有使用 RocksDB 存储引擎的 TiKV 实例可以直接开启 Titan。开启的方法是修改 TiKV 配置并重启 TiKV：

```
[rocksdb.titan]
enabled = true
```

开启 Titan 以后，原有的数据并不会马上移入 Titan 引擎，而是随着前台写入和 RocksDB compaction 的进行，逐步进行 key-value 分离并写入 Titan。可以通过观察 TiKV Details - Titan kv - blob file size 确认数据保存在 Titan 中部分的大小。

如果需要加速数据移入 Titan，可以通过 tikv-ctl 执行一次全量 compaction。请参看 tikv-ctl 文档。

注意 RocksDB 无法读取 Titan 的数据，但用 RocksDB 打开 Titan 数据也不会造成数据损坏。如果在打开过 Titan 的 TiKV 实例上错误地关闭了 Titan（误设置 rocksdb.titan.enabled = false），启动 TiKV 会失败，TiKV log 中出现“*You have disabled titan when its data directory is not empty*”错误。请参看“关闭 Titan”一节。

### 滚动开启 Titan（实验性）

也可以在集群中一个或多个 TiKV 节点中打开 Titan 作为实验，待调整稳定以后再在整个集群开启 Titan。由于 Titan 写入性能和存储方式跟 RocksDB 存在差异，滚动开启 Titan 的过程中可能造成 leader 分布不均匀，可以通过 PD 监控查看是否开启 Titan 的实例 leader count 较高，如果写负载有明显不均可以通过 pd-ctl store weight 降低 Titan 实例的 leader weight 以使 leader count 均衡。

### 8.2.2 参数调整

```
[rocksdb.titan]
max-background-gc (默认值: 1)
```

Titan GC 线程数。当从 TiKV Details - Thread CPU - RocksDB CPU 监控中观察到 Titan GC 线程长期处于满负荷状态时，应该考虑增加 Titan GC 线程池大小。

```
[rocksdb.defaultcf.titan]
min-blob-size (默认值: 1kb)
```

大 value 大小的阈值。当写入的 value 大小小于这个值时，value 会保存在 RocksDB 中，反之则保存在 Titan 的 blob file 中。视乎 value 大小的分布，增大这个值可以使更多 value 保存在 RocksDB，读取这些小 value 的性能会稍好一些；减少这个值可以使更多 value 保存在 Titan 中，进一步减少 RocksDB compaction。

```
[rocksdb.defaultcf.titan]
blob-file-compression (默认值: lz4)
```

Titan 中 value 所使用的压缩算法。Titan 中压缩是以 value 为单元的。

```
[rocksdb.defaultcf.titan]
blob-cache-size (默认值: 0)
```

Titan 中 value 的缓存大小。更大的缓存能提高 Titan 读性能，但过大的缓存会造成 OOM。建议在数据库稳定运行后，根据监控把 RocksDB block cache (storage.block-cache.capacity) 设置为 store size 减去 blob file size 的大小，blob-cache-size 设置为内存大小 \* 50% 减去 block cache 的大小。这是为了保证 block cache 足够缓存整个 RocksDB 的前提下，blob cache 尽量大。

```
[rocksdb.defaultcf.titan]
discardable-ratio (默认值: 0.5)
```

当一个 blob file 中无用数据（相应的 key 已经被更新或删除）比例超过这一阈值时，将会触发 Titan GC，将此文件有用的数据重写到另一个文件。这个值可以估算 Titan 的写放大和空间放大的上界（假设关闭压缩）。公式是：

写放大上界 =  $1 / \text{discardable\_ratio}$

空间放大上界 =  $1 / (1 - \text{discardable\_ratio})$

可以看到，减少这个阈值可以减少空间放大，但是会造成 Titan 更频繁 GC；增加这个值可以减少 Titan GC，减少相应的 IO 带宽和 CPU 消耗，但是会增加磁盘空间占用。

```
[rocksdb]
rate-bytes-per-sec (默认值: 0, 无限制)
```

该选项并不是 Titan 独有的设置。该选项限制 RocksDB compaction 的 IO 速率，以达到在流量高峰时，限制 RocksDB compaction 减少其 IO 带宽和 CPU 消耗对前台读写性能的影响。当开启 Titan 时，该选项限制 RocksDB compaction 和 Titan GC 的 IO 速率总和。当发现在流量高峰时 RocksDB compaction 和 Titan GC 的 IO 和/或 CPU 消耗过大，可以根据磁盘 IO 带宽和实际写入流量适当配置这个选项。

### 8.2.3 关闭 Titan (实验性)

通过设置 rocksdb.defaultcf.titan.blob-run-mode 可以关闭 Titan。blob-run-mode 可以设置为以下几个值之一：

- 当设置为“kNormal”时，Titan 处于正常读写的状态。
- 当设置为“kReadonly”时，新写入的 value 不论大小均会写入 RocksDB。
- 当设置为“kFallback”时，新写入的 value 不论大小均会写入 RocksDB，并且当 RocksDB 进行 compaction 时，会自动把所碰到的存储在 Titan blob file 中的 value 移回 RocksDB。

当需要关闭 Titan 时，可以设置 blob-run-mode = “kFallback”，并通过 tikv-ctl 执行全量 compaction。此后通过监控确认 blob file size 降到 0 以后，可以更改 rocksdb.titan.enabled = false 并重启 TiKV。

关闭 Titan 是实验性功能，非必要不建议使用。

### 8.2.4 Level Merge：提升范围查询性能 (实验性)

TiKV 4.0 中 Titan 提供新的算法提升范围查询性能并降低 Titan GC 对前台写入性能的影响。这个新的算法称为 level merge。Level merge 可以通过以下选项开启：

```
[rocksdb.defaultcf.titan]
level-merge = true
```

开启 level merge 的好处包括：

- 大幅提升 Titan 的范围查询性能。
- 减少了 Titan GC 对前台写入性能的影响，提升写入性能。
- 减少 Titan 空间放大，减少磁盘空间占用（默认配置下的比较）。

相应地，level merge 写放大会比 Titan 稍高，但依然低于原生的 RocksDB。

Level merge 与 Titan 原有的算法目前并不兼容，已经打开 Titan 的 TiKV 实例不可以更改是否使用 Level merge 的设置。

# 第9章 TiFlash 简介与 HTAP 实战

近年来，随着企业数字化转型的不断深入和发展，数据逐渐成为企业最重要的资产。对于每一个企业来说，都需要关注数据从产生、存储、分析到利用，并发挥其巨大价值的整个过程。此外，随着企业之间竞争的不断加剧，对数据的产生到发挥其价值的时间延迟要求越来越短。

作为存储和处理数据最重要的基础软件——数据库系统，一般可以按照负载类型分成 OLTP 型数据库和 OLAP 数据库。在一个企业中，这两种类型的数据库通常是并存的，分别支撑这两种负载类型的应用系统。目前，很多企业的这两种类型的系统之间是通过较为复杂的 ETL 过程“打通的”，数据在时效性上具有比较大的 T+N 延时，这越来越难以满足企业在数据处理和分析方面对时效性的要求。

近几年，HTAP 是比较热的一个概念，它是最有希望解决目前问题的方法。顾名思义，HTAP 是混合 OLTP 和 OLAP 业务，具备同时解决这两种问题能力的系统。2014 年 Gartner 公司给出了严格的定义：混合事务/分析处理 (HTAP) 是一种新兴的应用体系结构，它打破了事务处理和分析之间的“墙”。它支持更多的“信息分析”和“实时业务”的决策。

TiDB 是一个具有优异交易处理能力的分布式 NewSQL 产品，同时也具备了良好的分析能力，又是一款优秀的 HTAP 数据数据库产品。在这部分内容中，首先向大家介绍 TiDB HTAP 的主要特点，然后介绍其实现 HTAP 能力的关键技术之一的 TiFlash 列式存储引擎的架构和基本原理，最后向大家介绍 TiFlash 如何使用。

## 目录

- [9.1 TiDB HTAP 的特点](#)
- [9.2 TiFlash 架构与原理](#)
- [9.3 TiFlash 的使用](#)

## 9.1 TiDB HTAP 的特点

HTAP 是 Hybrid Transactional / Analytical Processing 的缩写。这个词汇在 2014 年由 Gartner 提出。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 TP 和 AP 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。而新型的 HTAP 数据库则可以同时承担交易和分析两种智能，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。

作为一款优秀的 HTAP 数据数据库，TiDB 除了优异的交易处理能力，也具备了良好的分析能力。

### 1. 数据库设计上的矛盾点

传统交易数据库在处理混合负载时有如下两个核心矛盾无法解决：

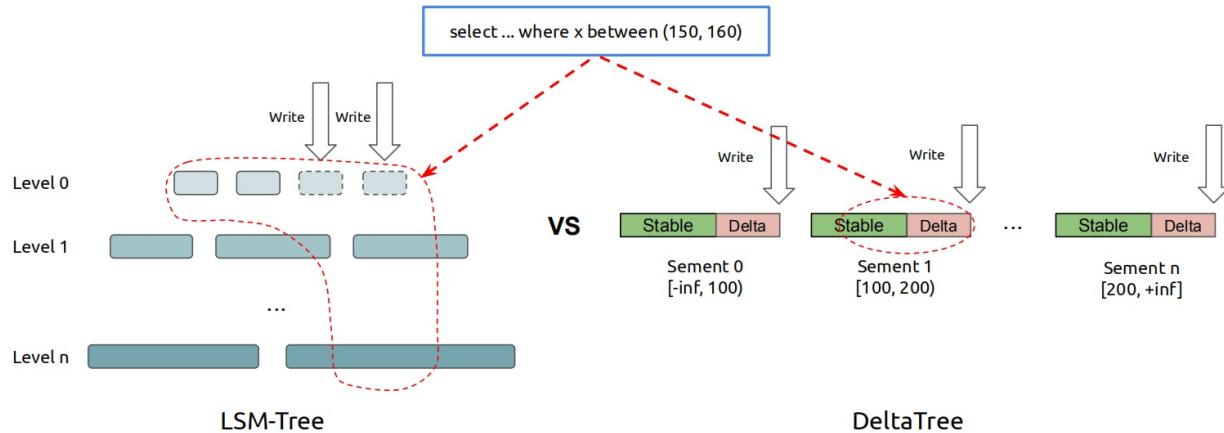
- 行存对于分析场景不友好
- 无法做到业务负载隔离

为了解决上述两个核心矛盾，作为 TiKV 扩展的列存储方案 TiFlash 应运而生，它有如下优势：

- 可更新列式存储设计，在提供高速更新能力的同时，提供高效的批量读取性能
- 配合源于 ClickHouse 的极致向量化计算引擎，更少的废指令，SIMD 加速
- 不影响 TiKV 稳定运行的前提下，提供一致性的读取保证，以及实时查询业务数据的能力
- TiDB 可以智能选择使用行存或者列存

### 2. 可更新列式存储引擎 Delta Tree

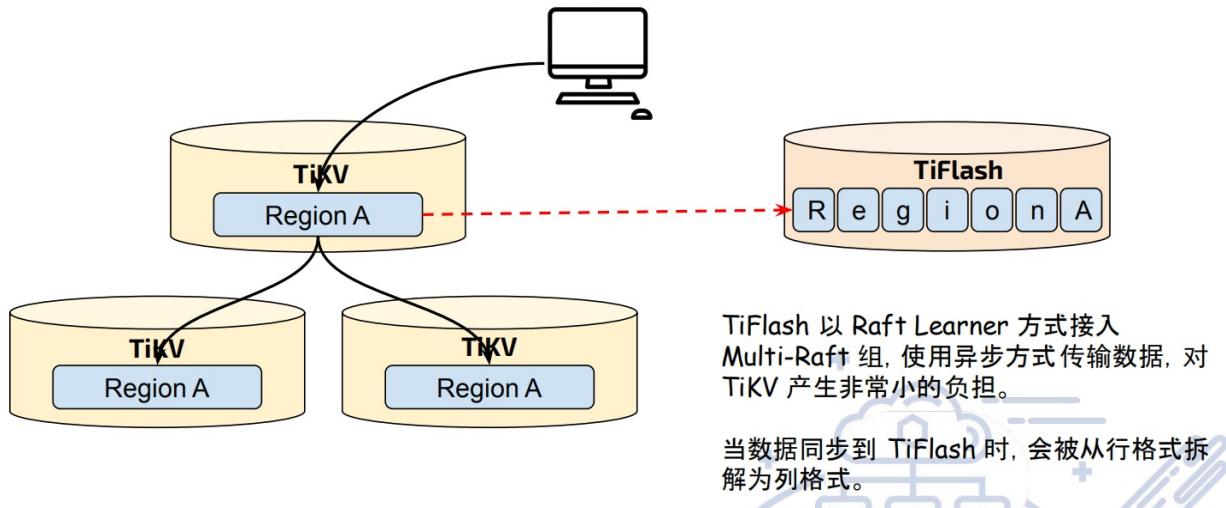
TiFlash 配备了可更新的列式存储引擎。列存更新的主流设计是 Delta Main 方式，基本思想是，由于列存块本身更新消耗大，因此往往设计上使用缓冲层容纳新写入的数据。然后再逐渐和主列存区进行合并。TiFlash 也使用了类似的 Delta Main 设计，从这个意义而言，LSM 也可用于列存更新。具体来说，Delta Tree 利用树状结构和双层 LSM 结合的方式处理更新，以规避单纯使用 LSM 设计时需要进行的多路归并。通过这种方式，TiFlash 在支持更新的同时也具备高速的读性能。



### 3. 实时且一致的复制体系

TiFlash 无缝融入整个 TiDB 的 Multi-Raft 体系。它通过 Raft Learner 进行数据复制，通过这种方式 TiFlash 的稳定性并不会对 TiKV 产生影响。例如 TiFlash 节点宕机或者网络延迟，TiKV 仍然可以继续运行无碍且不会因此产生抖动。于此同时，该复制协议允许在读时进行极轻量的校对以确保数据一致性。另外，TiFlash 可以与 TiKV 一样的方式进行在线扩容缩容，且能自动容错以及负载均衡。

## Raft Learner - Sync

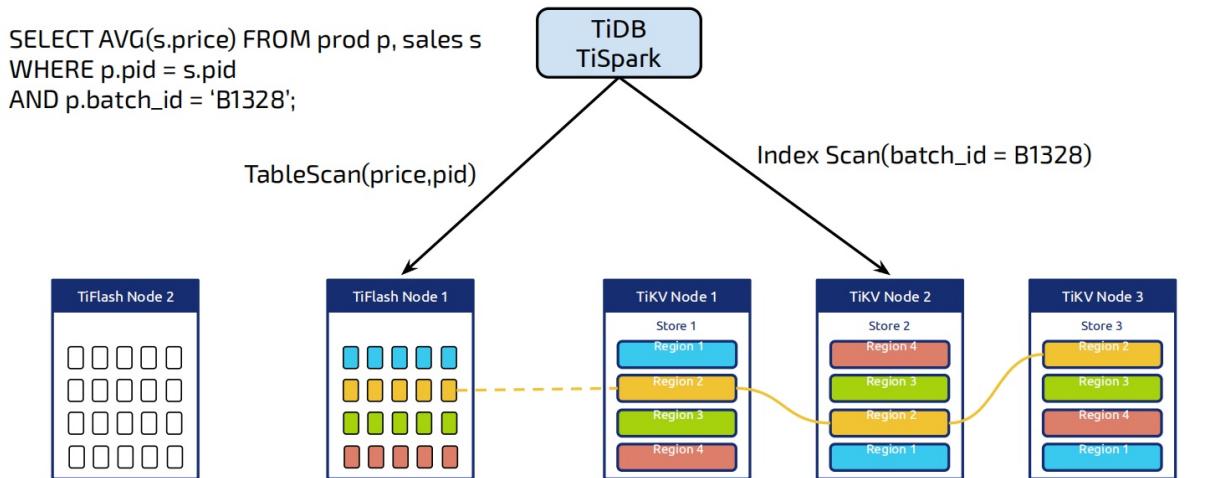


## 4. 完整的业务隔离

由于 TiFlash 的列存复制设计，用户可以选择单独使用与 TiKV 不同的另一组节点存放列存数据。另外不论是 TiDB 还是 TiSpark，计算层都可以强制选择行存或者列存，这样用户可以毫无干扰地查询在线业务数据，为实时 BI 类应用提供强力支持。

## 智能的行列混合模式

如果不使用上述隔离模式进行查询，TiDB 也可经由优化器自主选择行列。这套选择的逻辑与选择索引类似：优化器根据统计信息估算读取数据的规模，并对比选择列存与行存访问开销，做出最优选择。通过这种模式，用户可以在同一套系统方便地同时满足不同类型的业务需求。例如一套物流系统需要同时支持点查某订单信息，也需要进行大规模聚合统计某一时间段内货物派送和分发的汇总信息，利用 TiDB 的行列混合体系可以很简单实现，且完全无需担心不同系统间数据复制带来的不一致。



## 5. 更快的业务接入速度

同时兼备行存和列存的优势，让用户能更容易地接入业务。利用传统手段，用户往往需要将在线数据导出到分析平台才能进行分析，而这中间涉及了复杂的 ETL 或者数据传输管道维护，另外不同系统之间数据如何保持一致，如何进行格式转换也是很费思量的事情。因此，整个业务接入过程往往要花费数天甚至数周。而使用 TiDB 则可以帮助你大大简化这个过程。

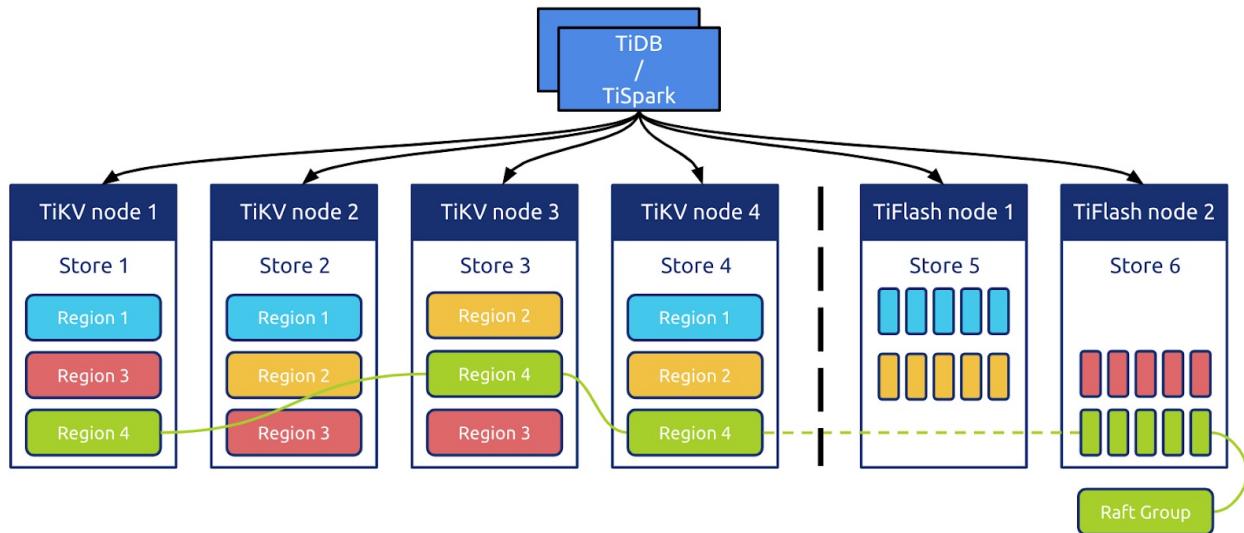
## 6. 未来规划

TiFlash 在未来计划支持不依赖 TiKV 的直接写入，当做 TiKV 的冷备存储等功能，这样 TiDB HTAP 体系将变得更加完整。

## 9.2 TiFlash 架构与原理

相比于行存，TiFlash 根据强 Schema 按列式存储结构化数据，借助 ClickHouse 的向量化计算引擎，带来读取和计算双重性能优势。相较于普通列存，TiFlash 则具有实时更新、分布式自动扩展、SI（Snapshot Isolation）隔离级别读取等优势。本章节将从架构和原理的角度来解读 TiFlash。

### 9.2.1 基本架构

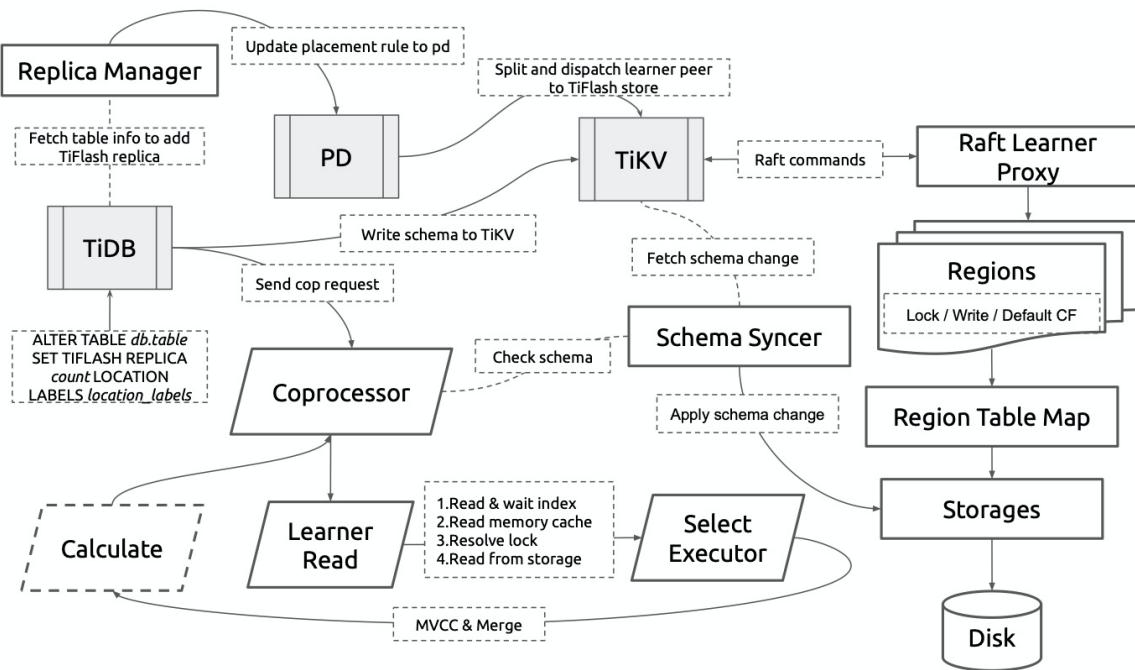


TiFlash 引擎补全了 TiDB 在 OLAP 方面的短板。TiDB 可通过计算层的优化器分析，或者显式指定等方式，将部分运算下推到对应的引擎，达到速度的提升。值得一提的是，在表关联场景下，即便 TiFlash 架构没有 MPP 相关功能，借助 TiDB 的查询优化器、布隆过滤器下推和表广播等手段，相当比例的关联场景仍可享受 TiFlash 加速。

如上图所示，TiFlash 能以 Raft Learner Store 的角色无缝接入 TiKV 的分布式存储体系。TiKV 基于 Key 范围做数据分片并将一个单元命名为 Region，同一 Region 的多个 Peer（副本）分散在不同存储节点上，共同组成一个 Raft Group。每个 Raft Group 中，Peer 按照角色划分主要有 Leader、Follower、Learner。在 TiKV 节点中，Peer 的角色可按照一定的机制切换，但考虑到底层数据的异构性，所有存在于 TiFlash 节点的 Peer 都只能是 Learner（在协议层保证）。TiFlash 对外可以视作特殊的 TiKV 节点，接受 PD 对于 Region 的统一管理调度。TiDB 和 TiSpark 可按照和 TiKV 相同的 Coprocessor 协议下推任务到 TiFlash。

在 TiDB 的体系中，每个 Table 含有 Schema、Index（索引）、Record（实际的行数据）等内容。由于 TiKV 本身没有 Table 的概念，TiDB 需要将 Table 数据按照 Schema 编码为 Key-Value 的形式后写入相应 Region，通过 Multi-Raft 协议同步到 TiFlash，再由 TiFlash 根据 Schema 进行解码拆列和持久化等操作。

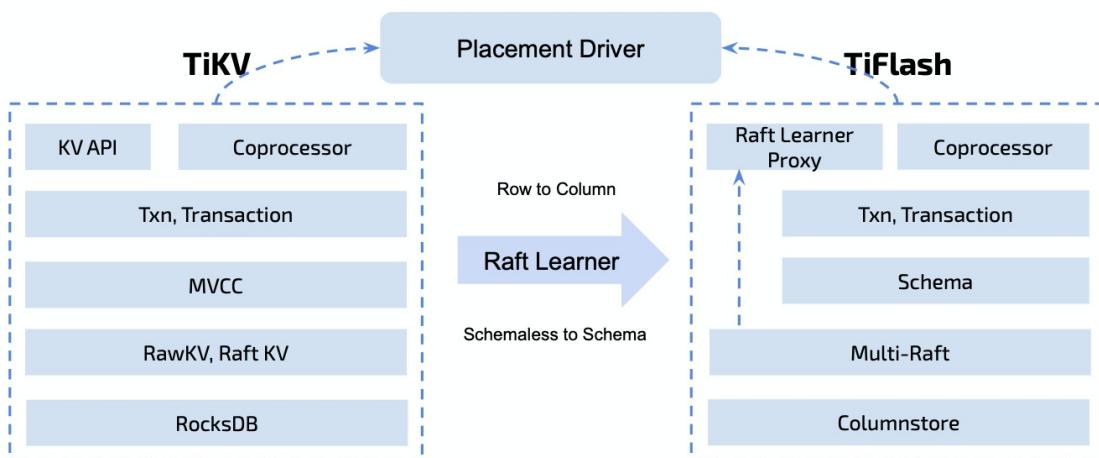
### 9.2.2 原理



这是一张 TiFlash 数据同步到读取的基本流程架构图，以下将按照大类模块分别简单介绍。

## 1. Replica Manager

Index 这种主要面向点查的结构对于 TiFlash 的列式存储是没有意义的。为了避免同步冗余数据并实现按 Table 动态增删 TiFlash 列存副本，需要借助 PD 来有选择地同步 Region。



在同一个集群内的 TiFlash 节点会利用 PD 的 ETCD 选举并维护一个 Replica Manager 来负责与 PD 和 TiDB 交互。当感知到 TiDB 对于 TiFlash 副本的操作后（DDL 语句），会将其转化为 PD 的 Placement Rule，通过 PD 令 TiKV 分裂出指定 Key 范围的 Region，为其添加 Learner Peer 并调度到集群中的 TiFlash 节点。此外，当 Table 的 TiFlash 副本尚未可用时，Replica Manager 还负责向各个 TiFlash 节点收集 Table 的数据同步进度并上报给 TiDB。

## 2. Raft Learner Proxy

基于 Raft Learner 的数据同步机制是整个 TiFlash 存储体系的基石，也是数据实时性、正确性的根本保障。Region 的 Learner Peer 除了不参与投票和选举，仍要维护与其他 TiKV 节点中相同的全套状态机。因此，TiKV 被改造成了一个 [Raft Proxy](#) 库，并由 Proxy 和 TiFlash 协同维护节点内的 Region，其中实际的数据写入发生在 TiFlash 侧。

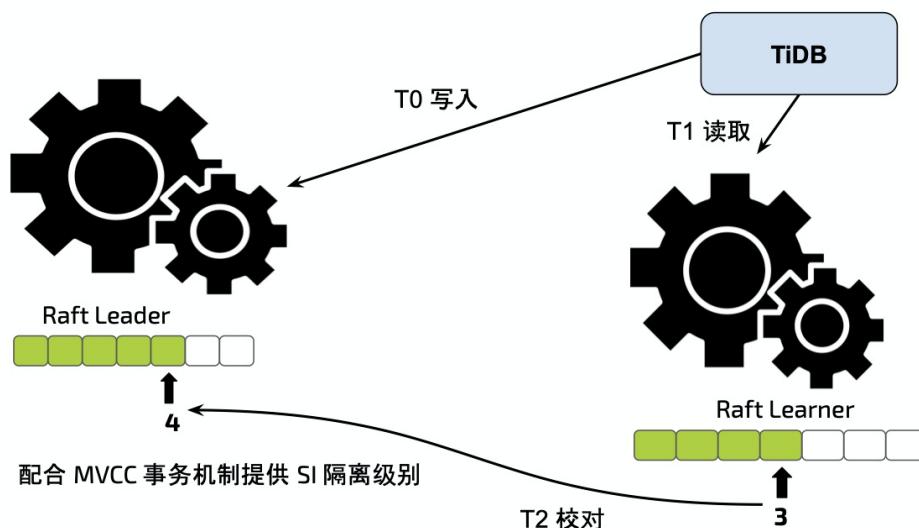
作为固定存在的 Learner，TiFLash 可以进行数据进行二次加工而不用担心对其他节点产生影响。TiFlash 在执行 Raft 命令时就可以做到剔除无效数据，将未提交的数据按 Schema 预解析等优化。同时 TiFlash 实现了对于 Raft 命令的幂等回放以及引擎层的幂等写，可以不记录 WAL 且只在必要的时刻做针对性的持久化，从而简化模型并降低了风险。

TiDB 的事务实现是基于 Percolator 模型的，映射到 TiKV 中则是对 3 个 CF (Column Family) 数据的读写：Write、Default、Lock。TiFlash 也在每个 Region 内部对此做了抽象，不同的是对于 TiFlash 而言，数据写入 CF 只能作为中间过程，最终持久化到存储层需要找到 Region 对应的 Table 再根据 Schema 进行行列结构转换。

### 3. Schema Syncer

每个 TiFlash 节点会依据特定的数据范围异步地向 TiKV 获取集群中 Table 的 Schema 信息，然后检测相关改动并应用变更到存储层。由于行存和列存的格式差异巨大，列存在应用 TiDB 的 Schema 改动时很难做到实时在线处理，对于复杂操作则需要锁表来保证正确性。

### 4. Learner Read / Coprocessor



无论是通过 CH 客户端、TiSpark、CHSpark 还是 TiDB 向 TiFlash 发起查询，都需要 Learner Read 来确保外部一致性。在同一个 Raft Group 中，Index 是永续递增的（任何 Raft 命令都会对其产生修改），可被视作乐观锁。Region 本身含有 Version 和 Conf Version 两种版本号，当发生诸如 Split/Merge/ChangePeer 等操作时，版本号均会产生相应的变化。

所有的查询都需要由上层拆分为一个或多个 Region 的读请求，需要包含 Region 的两种版本号（可从 PD 或 TiKV 获取），Timestamp（用于 Snapshot Read 的时间戳，从 PD 获取），以及 Table 相关信息（Schema Version 从 TiDB 获取）。单次读请求可大致分为以下步骤：

1. 校验并更新本节点的 Schema
2. 向 Region 的 Leader Peer 获得最新的 Index，等待当前节点中 Learner 的 Applied Index 追上
3. 校对 Version 和 Conf Version，检查 Lock CF 中的锁信息
4. 读取内存中的半结构化数据和存储引擎中 Region 范围对应的结构化数据
5. 按照 Timestamp 进行 Snapshot Read 和多路合并
6. Coprocessor 计算（主要借助 ClickHouse 的向量化计算引擎）

### 9.2.3 TiFlash 对 OLAP 查询加速

OLAP 类的查询通常具有以下几个特点：

- 每次查询读取大量的行，但是仅需要少量的列
- 宽表，即每个表包含着大量的列
- 查询通过一张或多张小表关联一张大表，并对大表上的列做聚合

TiFlash 列存引擎针对这类查询有较好的优化效果：

(1) I/O 优化

- 每次查询可以只读取需要的列，减少了 I/O 资源的使用
- 同列数据类型相同，相较于行存可以获得更高的压缩比
- 整体的 I/O 减少，令内存的使用更加高效

(2) CPU 优化

- 列式存储可以很方便地按批处理字段，充分利用 CPU Cache 取得更好的局部性
- 利用向量化处理指令并行处理部分计算

### 9.2.4 TiKV 与 TiFlash 配合

TiFlash 可被当作列存索引使用，获得更精确的统计信息。对于关联查询来说，点查相关的任务可以下推到 TiKV，而需要关联的大批量聚合查询则会下推到 TiFlash，通过两个引擎的配合，达到更快的速度。

### 9.2.5 总结与展望

TiFlash 是 TiDB HTAP 之路上的全新实践。这套架构体系也将伴随生产环境的使用不断演化发展，进而为用户解决更多问题。

## 9.3 TiFlash 的使用

用户可以使用 TiDB 或者 TiSpark 读取 TiFlash，TiDB 适合用于中等规模的 OLAP 计算，而 TiSpark 适合大规模的 OLAP 计算，用户可以根据自己的场景和使用习惯自行选择。

### 9.3.1 按表构建 TiFlash 副本

TiFlash 接入 TiKV 集群后，默认不会开始同步数据，可通过 MySQL 客户端向 TiDB 发送 DDL 命令来为特定的表建立 TiFlash 副本：

```
ALTER TABLE table_name SET TIFLASH REPLICA count;
```

说明：

- count 表示副本数，如果设置为 0 则表示删除 TiFlash 副本
- 对于相同表的多次 DDL 命令，仅保证最后一次能生效

示例 1，为表建立 2 个 TiFlash 副本：

```
ALTER TABLE `tpch50`.`lineitem` SET TIFLASH REPLICA 2;
```

示例 2，删除副本：

```
ALTER TABLE `tpch50`.`lineitem` SET TIFLASH REPLICA 0;
```

可通过如下 SQL 语句查看特定表（通过 WHERE 语句指定，去掉 WHERE 语句则查看所有表）的 TiFlash 副本的状态：

```
SELECT * FROM information_schema.tiflash_replica
WHERE TABLE_SCHEMA = '<db_name>' and TABLE_NAME = '<table_name>';
```

查询结果中的 AVAILABLE 字段表示该表的 TiFlash 副本是否可用。

注意，假设有一张表 t 已经通过上述的 DDL 语句同步到 TiFlash，则通过以下语句创建的表也会自动同步到 TiFlash：

```
CREATE TABLE table_name LIKE t;
```

### 9.3.2 TiDB 读取 TiFlash

TiDB 提供三种读取 TiFlash 副本的方式。如果添加了 TiFlash 副本，而没有做任何 engine 的配置，则默认使用 CBO (Cost Based Optimization) 方式。

#### 1. CBO

对于创建了 TiFlash 副本的表，TiDB 的 CBO 优化器会自动根据代价选择是否使用 TiFlash 副本，具体有没有选择 TiFlash 副本，可以通过 `explain analyze` 语句查看，见下图：

```
mysql> explain analyze select * from t;
+-----+-----+-----+-----+
| id   | count | task      | operator info           | execution info
|      |       |           | memory     |                                |
+-----+-----+-----+-----+
| TableReader_7 | 10000.00 | root      | data:TableScan_6          | time:54.230814ms, loops:2,
| rows:1, rpc time:54.147725ms | 163 Bytes |
| └─TableScan_6 | 10000.00 | cop[tiflash] | table:t, range:[-inf,+inf], keep order:false, stats:pseudo | time:40ms, loops:1, rows:1
|           | N/A      |                                |
+-----+-----+-----+-----+
2 rows in set (0.06 sec)
```

## 2. Engine 隔离

Engine 隔离是通过配置变量来指定所有的查询均使用指定 engine 的副本，可选 engine 为 tikv 和 tiflash，分别有 2 个配置级别：

(1) 会话级别，即 SESSION 级别。如果没有指定，会继承 GLOBAL 的配置。

```
set @@session.tidb_isolation_read_engines="逗号分隔的 engine list";
```

或者

```
set SESSION tidb_isolation_read_engines = "逗号分隔的 engine list";
```

例如让 TiDB 自由选择行存和列存，可按如下配置：

```
set SESSION tidb_isolation_read_engines = "tikv,tiflash";
```

如果希望只读取 TiFlash 的数据（隔离模式），则按如下配置：

```
set SESSION tidb_isolation_read_engines = "tiflash";
```

(2) TiDB 实例级别，即 INSTANCE 级别，和以上配置是交集关系。比如 INSTANCE 配置了 "tikv,tiflash"，而 SESSION 配置了 "tikv"，则只会读取 tikv。如果没有指定，默认继承会话级别配置。在 TiDB 的配置文件添加如下配置项：

```
[isolation-read]
engines = ["tikv", "tiflash"]
```

默认值为 "tikv,tiflash"，即可以同时读取 tikv 和 tiflash 副本，优化器会自动选择。

指定了 engine 后，对于查询中的表没有对应 engine 副本的情况（因为 tikv 副本是必定存在的，因此只有配置了 engine 为 tiflash 而 tiflash 副本不存在这一种情况），查询会报该表不存在该 engine 副本的错。

Engine 隔离的优先级高于优化器选择，即优化器仅会选取指定 engine 的副本。

## 3. 手工 hint

手工 hint 可以强制 TiDB 对于某张或某几张表使用 TiFlash 副本，其优先级高于 CBO 和 engine 隔离，使用方法为：

```
SELECT /*+ read_from_storage(tiflash[t]) */ * FROM t;
```

同样的，对于指定 hint 的表，如果没有 tiflash 副本，查询会报该表不存在该 tiflash 副本的错。



# 第10章 TiDB 安全

本章介绍了 TiDB 的三大安全机制：权限管理，RBAC 以及证书管理和传输加密。权限管理功能提供了对于用户和数据库访问权限的基本管理功能，适用于各种访问控制场景；RBAC 在权限管理功能之上构建了角色这一概念，通过角色可以方便的管理大量的用户和复杂的授权关系；证书管理和数据加密功能提供了除密码之外的用户认证登陆方式，用户通过证书认证登陆后会采用 TLS 加密与服务端的通信数据，避免中间人攻击。

## 目录

- [10.1 权限管理](#)
- [10.2 RBAC](#)
- [10.3 证书管理与数据加密](#)

## 10.1 权限管理

TiDB 的权限管理系统提供了基本的权限访问控制功能，保障数据不被非授权的篡改。

TiDB 的权限管理系统按照 MySQL 的权限管理进行实现，TiDB 支持大部分的 MySQL 的语法和权限类型。

TiDB 的权限管理系统主要包含两部分，用户账户管理和权限管理，在本节会通过示例一一展示。

### 10.1.1 权限管理系统可以做什么

- 权限管理系统可以创建和删除用户，授予和撤销用户权限。
- 只有有相应权限的用户才可以进行操作，比如只有对某个表有写权限的用户，才可以对这个表进行写操作。
- 通过为每个用户设定严格的权限，保障数据不被恶意篡改。

### 10.1.2 权限管理系统原理

在权限管理模块中，有三类对象：用户，被访问的对象（数据库，表）以及权限。

所有对象的具体信息都会被记录在几张系统表中：

- mysql.user
- mysql.tables\_priv
- mysql.db

所有的授权，撤销权限，创建用户，删除用户操作，实际上都是对于这三张用户表的修改操作。TiDB 的权限管理器负责将系统表解析到内存中，方便快速的进行鉴权操作。在进行权限修改操作后，权限管理器会重新加载系统表。

#### 1.mysql.user 表解析：

mysql.user 表的结构如下：

```

CREATE TABLE `user` (
  `Host` char(64) NOT NULL,
  `User` char(32) NOT NULL,
  `authentication_string` text DEFAULT NULL,
  `Select_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Insert_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Update_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Delete_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Drop_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Process_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Grant_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `References_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Alter_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Show_db_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Super_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_tmp_table_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Lock_tables_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Execute_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_view_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Show_view_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_routine_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Alter_routine_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Index_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_user_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Event_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Trigger_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_role_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Drop_role_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Account_locked` enum('N','Y') NOT NULL DEFAULT 'N',
  `Shutdown_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Reload_priv` enum('N','Y') DEFAULT 'N',
  `File_priv` enum('N','Y') DEFAULT 'N',
  PRIMARY KEY (`Host`,`User`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin

```

mysql.user 表主要记录了用户的信息和用户拥有的全局权限，Host 和 User 字段主要用于用户登陆；其中 Host 字段支持通配符功能，用户在登陆时，权限管理器首先会根据登陆指定的用户名，找到 user 表中所有包含这个用户名的行。再通过比对登陆主机的 ip 和这些行的 Host 字段，来确定登陆哪个具体用户；例如用户在 192.168.1.7 登陆 root 用户，mysql.user 表中有 `root @ 172.16.*` 和 `root @ 192.168.1.*` 这两个 User 字段为 root 的用户，那权限管理器会将登陆主机 ip 192.168.1.7 和这两个 Host 进行匹配，从而登陆 `root @ 192.168.1.*`。

## 2.mysql.table, mysql.db 表解析：

mysql.table 表和 mysql.db 表的结构如下：

```

CREATE TABLE `tables_priv` (
  `Host` char(60) NOT NULL,
  `DB` char(64) NOT NULL,
  `User` char(32) NOT NULL,
  `Table_name` char(64) NOT NULL,
  `Grantor` char(77) DEFAULT NULL,
  `Timestamp` timestamp DEFAULT CURRENT_TIMESTAMP,
  `Table_priv` set('Select','Insert','Update','Delete','Create','Drop','Grant','Index','Alter','Create View','Show View','Trigger','References') DEFAULT NULL,
  `Column_priv` set('Select','Insert','Update') DEFAULT NULL,
  PRIMARY KEY (`Host`,`DB`,`User`,`Table_name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin

```

```

CREATE TABLE `db` (
  `Host` char(60) NOT NULL,
  `DB` char(64) NOT NULL,
  `User` char(32) NOT NULL,
  `Select_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Insert_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Update_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Delete_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Drop_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Grant_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `References_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Index_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Alter_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_tmp_table_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Lock_tables_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_view_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Show_view_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Create_routine_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Alter_routine_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Execute_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Event_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  `Trigger_priv` enum('N','Y') NOT NULL DEFAULT 'N',
  PRIMARY KEY (`Host`, `DB`, `User`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin

```

mysql.tables\_priv 和 mysql.db 主要记录了将权限授予给用户的信息，分别对应表权限和数据库权限。

在进行鉴权操作时，执行计划会根据访问到的表，数据库，以及操作类型，汇总出一个所需要权限的 bitmask，然后向权限管理器请求鉴权。权限管理器会根据要鉴权的 User, Host 从内存的权限表中得到对应用户的全局权限，数据库权限和表权限。来逐级检查用户是否拥有所需要的权限。

在进行类似 GRANT, REVOKE 等对用户的权限修改操作时，TiDB 会开启一个内部 sql 事务，用 INSERT, UPDATE, DELETE 修改对应的权限表，然后提交内部事务，如果提交成功，权限管理器会刷新内存中的权限表。

### 10.1.3 权限管理系统操作示例

创建一个用户名为 developer 且能在 192.168.0.\* 这一子网中登陆的用户，密码为 'test\_user'

```
root> CREATE USER 'developer'@'192.168.0.%' IDENTIFIED BY 'test_user';
```

并且授予给 developer 用户在 read\_table 表上的读权限，write\_table 表上的写权限。

```
root> GRANT SELECT ON app.read_table TO 'developer'@'192.168.0.%';
root> GRANT INSERT, UPDATE ON app.write_table TO 'developer'@'192.168.0.%';
```

查看 developer 用户当前的权限。

```
root> SHOW GRANTS FOR 'developer'@'192.168.0.%';
GRANT USAGE ON *.* TO 'developer'@'192.168.0.%'
GRANT Select ON app.read_table TO 'developer'@'192.168.0.%'
GRANT Insert,Update,Delete ON app.write_table TO 'developer'@'192.168.0.%'
```

然后用 developer 登陆，尝试使用权限。

```
developer> SEECT * FROM app.read_table;
Empty set (0.01 sec)
developer> INSERT INTO write_table VALUES (1),(2),(3);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

假如用 developer 试图对 write\_table 进行写操作，TiDB 会提示限检查未通过。

```
developer> INSERT INTO read_table VALUES (1),(2),(3);
ERROR 1142 (42000): INSERT command denied to user 'developer'@'192.168.0.%' for table 'read_table'
```

具有 GRANT\_OPTION 权限的用户可以通过 GRANT, REVOKE 管理其他用户的权限，比如撤销 developer 在 read\_table 表上的读权限。

```
root> REVOKE SELECT ON app.read_table from 'developer'@'192.168.0.%';
```

具有 CREATE USER 权限的用户可以修改其他用户的信息，比如修改密码，删除用户。

```
root> ALTER USER 'developer'@'192.168.0.%' IDENTIFIED BY 'password';
root> DROP USER 'developer'@'192.168.0.%'
```

#### 10.1.4 小结

本节主要介绍了 TiDB 权限相关操作的使用方法；介绍了如何创建一个用户，授予一些权限。如何撤销权限和删除用户。下一节将进一步深入 TiDB 权限管理模块，讲解 RBAC 的原理和使用方法。

## 10.2 RBAC

上一节介绍了 TiDB 的基本权限功能，本小节将介绍另一个权限管理功能-- RBAC。

Role-based access control , RBAC 基于角色的权限访问控制。

区别于 MAC (Mandatory access control) 以及 DAC (Discretionary Access Control)，RBAC 更为中性且更具灵活性。

TiDB 的基于角色的访问控制 (RBAC) 系统的实现类似于 MySQL 8.0 的 RBAC 系统，兼容大部分 MySQL RBAC 系统的语法。

### 10.2.1 RBAC 可以做什么

- 根据业务场景设置角色，集合多个权限。
  - 方便用户权限管理，同时修改多个用户的权限。
  - 用户关注场景，角色关注权限。
  - 进行继承，角色可以授予给另外一个角色。
  - 一个用户可以同时拥有多个角色，可以同时使用这些角色拥有的权限。

### 10.2.2 RBAC 实现原理

- TiDB 的权限管理器，构建出了一个邻接表来记录图结构。在鉴权时，从用户拥有的角色出发，进行深度优先搜索，找到所有与之相关的角色，将这些角色的权限汇总起来，就得到了用户的角色权限。
  - 每个会话 session 中维护了一个 ActiveRole 数组，其中记录着当前哪些角色是启用着，在使用 SET ROLE 时便会对这个数组进行修改，同时权限管理器在用户进行登录时，也会在内存系统表缓存中，找到 default\_roles 中记录的默认启用角色，构建出最开始的 ActiveRole 数组。

主要依赖以下系统表：

- mysql.user 复用用户表，区别是 Account Locked 字段，角色的值是 Y，也就是不能登陆。

- mysql.role\_edges 描述了角色和角色，角色和用户之间的授予关系。例如将角色 r1 授予给 test 后，会出现这样一条记录：

```
+-----+-----+-----+-----+
| FROM_HOST | FROM_USER | TO_HOST | TO_USER | WITH_ADMIN_OPTION |
+-----+-----+-----+-----+
| %         | r1        | %         | test      | N           |
+-----+-----+-----+-----+
```

- mysql.default\_roles 记录每个用户默认启用的角色，启用后的角色才能生效。

```
+-----+-----+-----+-----+
| HOST | USER | DEFAULT_ROLE_HOST | DEFAULT_ROLE_USER |
+-----+-----+-----+-----+
| %    | test | %                | r_1            |
+-----+-----+-----+-----+
```

### 10.2.3 RBAC 操作示例

- 创建角色 r\_1 , r\_2 , 可以一次创建多个，示例：

```
CREATE ROLE `r_1`@`%`, `r_2`@`%`;
```

- 设置 r\_1 为只读角色：

```
GRANT SELECT ON db_1.* TO 'r_1'@'%';
```

- 将 r\_1 角色授予用户 test@'%':

```
grant r_1 to test@'%';
```

- 启用默认角色，在登陆时，默认启用的角色会被自动启用：

```
SET DEFAULT ROLE 'r_1';
```

- 启用当前session角色，仅对当前session生效：

```
SET ROLE 'r_1';
```

- 查看用户角色：

```
SELECT CURRENT_ROLE();
```

- 查看用户角色权限：

```
TiDB > SHOW GRANTS FOR 'test'@'%' USING 'r_1';
+-----+
| Grants for test@%                                |
+-----+
| GRANT USAGE ON *.* TO 'test'@'%'                 |
| GRANT Select ON test.* TO 'test'@'%'              |
| GRANT 'r_1'@'%' TO 'test'@'%'                   |
+-----+
```

- 收回角色：

```
REVOKE 'r_1' FROM 'test'@'%', 'root'@'%';
```

#### 10.2.4 看一个完整的例子

账户 bi\_user 登录，启用只读角色后，才可以查询指定库表权限，会话结束，权限失效。

```

#创建角色 reader
root@127.0.0.1:(none)>create role reader@'%';
Query OK, 0 rows affected (0.012 sec)

#设置角色 reader 只读 mysql.role_edges 权限
root@127.0.0.1:mysql>grant select on mysql.role_edges to reader'%';
Query OK, 0 rows affected (0.017 sec)

#创建用户 bi_user
root@127.0.0.1:(none)>create user bi_user@'%';
Query OK, 0 rows affected (0.011 sec)

#将只读角色 reader 授予 bi_user 用户
root@127.0.0.1:mysql>grant reader to bi_user'%';
Query OK, 0 rows affected (0.014 sec)

# bi_user 登录查看无数据权限
bi_user@127.0.0.1:(none)>show databases;
+-----+
| Database      |
+-----+
| INFORMATION_SCHEMA |
+-----+
1 row in set (0.000 sec)

#查看当前登录用户 bi_user 当前未启用角色
bi_user@127.0.0.1:(none)>SELECT CURRENT_ROLE();
+-----+
| CURRENT_ROLE() |
+-----+
|          |
+-----+
1 row in set (0.000 sec)

#在当前 session 中启用 bi_user 的 reader 角色
bi_user@127.0.0.1:(none)>set role reader;
Query OK, 0 rows affected (0.000 sec)

#查看 bi_user 当前被启用的角色
bi_user@127.0.0.1:(none)>SELECT CURRENT_ROLE();
+-----+
| CURRENT_ROLE() |
+-----+
| `reader`@`%`   |
+-----+
1 row in set (0.000 sec)

#当前登录用户 bi_user 查看 mysql 库中有权限的表
bi_user@127.0.0.1:mysql>select * from role_edges;
+-----+-----+-----+-----+-----+
| FROM_HOST | FROM_USER | TO_HOST | TO_USER | WITH_ADMIN_OPTION |
+-----+-----+-----+-----+-----+
| %        | reader    | %        | bi_user  | N           |
+-----+-----+-----+-----+-----+
1 row in set (0.000 sec)

#当前登录用户 bi_user 执行 delete 表报错，权限校验失败
bi_user@127.0.0.1:mysql>delete from role_edges;
ERROR 1105 (HY000): privilege check fail

#当前登录用户 bi_user 执行查询其他表报错
bi_user@127.0.0.1:mysql>select * from user;
ERROR 1142 (42000): SELECT command denied to user 'bi_user'@'127.0.0.1' for table 'user'

#重新登录 bi_user 权限已经失效
bi_user@127.0.0.1:(none)>use mysql
ERROR 1044 (42000): Access denied for user 'bi_user'@'%' to database 'mysql'

```

## 10.2.5 小结

本小节介绍了 RBAC 的原理和使用方式，企业用户可以用 RBAC 构建出一套灵活的权限管理机制。下一节中将介绍 TiDB 的另一种身份验证方式——证书验证。

## 10.3 证书管理与数据加密

从 TiDB 3.0.8 版本开始，TiDB 支持基于证书鉴权的登录方式。采用这种方式，TiDB 通过验证不同用户提供的客户端证书来确认身份，并在登陆后使用加密连接来传输数据。相比 TiDB 用户常用的用户名密码验证方式，与 MySQL 相兼容的证书鉴权方式更安全，因此越来越多的用户使用证书鉴权来代替用户名密码验证。

### 10.3.1 证书管理可以做什么

TiDB 服务端默认采用非加密连接，因而具备监视信道流量能力的第三方可以知悉 TiDB 服务端与客户端之间发送和接受的数据，包括但不限于查询语句内容、查询结果等。若信道是不可信的，例如客户端是通过公网连接到 TiDB 服务端的，则非加密连接容易造成信息泄露，建议使用加密连接确保安全性。

使用证书验证加密连接后，连接将具有以下安全性质：

- 保密性：流量明文无法被窃听；
- 完整性：流量明文无法被篡改；
- 身份验证（可选）：客户端和服务端能验证双方身份，避免中间人攻击。

### 10.3.2 证书管理的原理

TiDB 证书管理功能中使用的证书，需要符合 X.509 协议。用户先生成服务端密钥，服务端证书，客户端密钥和客户端证书，用户再通过自己的 CA(根证书) 对服务端证书和客户端证书进行签名。

- 服务端验证机制：如果客户端在建立连接时，提供了 CA 证书；则会验证服务端的证书是否是由这个 CA 签发，从而验证服务端身份。
- 客户端验证机制：如果在 TiDB 中配置了 CA 路径，则会在用户登陆时，检查客户端提供的证书是否由 CA 签发，从而验证客户端身份。
- 加密通信：在验证身份之后，会进行密钥协商，之后的数据传输将采用协商后的密钥进行加密。

除了验证证书签名外，TiDB 还支持对于指定用户验证客户端证书的具体内容，包括 subject，issuer，cipher。这个功能的实现是通过将验证的信息写入 mysql.global\_priv 系统表来完成。

在用户登录时，TiDB 获取到客户端证书，会比对相应的验证内容是否符合对相关用户的要求。

如果符合，则可以登陆。

### 10.3.3 证书管理操作示例

#### 1. 制作 CA 证书

目前推荐使用 [OpenSSL](#) 来生成密钥和证书，先执行以下命令来安装 OpenSSL：

```
sudo apt-get install openssl
```

首先要制作一个 CA，生成 CA 密钥：

```
sudo openssl genrsa 2048 > ca-key.pem
```

生成 CA 密钥对应的 CA 证书：

```
sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.pem
```

输入证书信息，示例如下：

```

Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:TiDB admin
Email Address []:s@pingcap.com

```

至此 CA 证书制作完成，在线上使用过程中，CA 密钥最好保存在一个离线安全的服务器上。

## 2. 制作服务端密钥和证书

接下来是制作服务端密钥和证书，用以下命令生成服务端密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.pem -out server-req.pem
```

输入证书信息：

```

Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiKV
Common Name (e.g. server FQDN or YOUR name) []:TiKV Test Server
Email Address []:k@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:

```

生成服务端 RAS 密钥：

```
sudo openssl rsa -in server-key.pem -out server-key.pem
```

使用 CA key 和证书生成服务端证书：

```
sudo openssl x509 -req -in server-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 -out server-cert.pem
```

## 3. 制作客户端密钥和证书

生成客户端密钥和证书也是类似的操作：

```

sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.pem -out client-req.pem
sudo openssl rsa -in client-key.pem -out client-key.pem
sudo openssl x509 -req -in client-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 -out client-cert.pem

```

生成服务端和客户端证书之后，可以通过以下命令来验证证书：

```
openssl verify -CAfile ca-cert.pem server-cert.pem client-cert.pem
```

验证通过会显示以下信息：

```

server-cert.pem: OK
client-cert.pem: OK

```

## 4. 配置 TiDB 启用证书验证

在生成证书之后，需要为 TiDB 配置证书，步骤如下：

修改 TiDB 配置文件中的 [security] 段。这一步指定 CA 证书、服务端密钥和服务端证书存放的路径。

```
[security]
ssl-cert ="path/to/server-cert.pem"
ssl-key ="path/to/server-key.pem"
ssl-ca="path/to/ca-cert.pem"
```

启动 TiDB 日志。如果日志中有以下内容，即代表配置生效：

```
[INFO] [server.go:264] ["secure connection is enabled"] ["client verification enabled=true"]
```

客户端在登陆 TiDB 时，指定客户端密钥和证书路径，不指定 --ssl-ca 则只会加密连接，不会验证服务端身份。

```
mysql -utest -h0.0.0.0 -P4000 --ssl-cert /path/to/client-cert.pem --ssl-key /path/to/client-key.pem --ssl-ca /path/to/ca-cert.pem
```

上面演示了如何制作证书和使用证书进行加密连接，接下来会演示如何使用证书验证用户身份。TiDB 支持在用户登陆时，验证 subject，issuer，cipher 分别是指：

- subject：指定用户在连接时需要提供客户端证书的 subject 内容，对应制作客户端证书时，输入的信息。
- issuer：指定签发用户证书的 CA 证书的 subject 内容，对应制作 CA 证书时，输入的信息。

指定验证用户的 subject 用于验证客户端证书和账号的匹配关系；指定验证用户的 issuer 用户验证用户提供的 CA 证书是否可信。

可以在创建用户时指定验证信息：

```
create user 'u1'@'%' require issuer '/C=US/ST=California/L=San Francisco/O=PingCAP Inc./OU=TiDB/CN=TiDB admin/emailAddress=s@pingcap.com' subject '/C=US/ST=California/L=San Francisco/O=PingCAP Inc./OU=TiDB/CN=tpch-user1/emailAddress=zz@pingcap.com' cipher 'TLS_AES_256_GCM_SHA384'
```

也可以在创建用户后，通过 GRANT 操作指定验证信息：

```
> create user 'u1'@'%';
> grant all on *.* to 'u1'@'%' require issuer '/C=US/ST=California/L=San Francisco/O=PingCAP Inc./OU=TiDB/CN=TiDB admin/emailAddress=s@pingcap.com' subject '/C=US/ST=California/L=San Francisco/O=PingCAP Inc./OU=TiDB/CN=tpch-user1/emailAddress=zz@pingcap.com' cipher 'TLS_AES_256_GCM_SHA384';
```

配置完成后，TiDB 在用户在登录时会验证以下内容：

- 使用 SSL 登录，且证书为 Server 配置的 CA 证书所签发
- 证书 Issuer 信息和权限配置里的信息相匹配
- 证书 Subject 信息和权限配置里的信息相匹配

全部验证通过后方可登录，否则会报 ERROR 1045 (28000): Access denied 错误。

### 10.3.4 更新和替换证书：

证书和密钥通常会周期性更新，下文介绍更新密钥和证书的流程。

CA 证书是客户端和服务端相互校验的依据，所以如果需要替换 CA 证书，则需要生成一个组合证书来在滚动期间同时支持新旧客户端和服务端的验证，并优先将客户端和服务端的 CA 证书都替换完成后再进行其他客户端和服务端的密钥和证书替换。

## 1.更新 CA 证书

以替换 CA 密钥为例（如：ca-key.pem 被盗了），首先将旧的 CA 密钥和证书进行备份：

```
mv ca-key.pem ca-key.old.pem
mv ca-cert.pem ca-cert.old.pem
```

之后生成新的 CA 密钥：

```
sudo openssl genrsa 2048 > ca-key.pem
```

用新的密钥生成新的 CA 证书：

```
sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.new.pem
```

生成组合 CA 证书：

```
cat ca-cert.new.pem ca-cert.old.pem > ca-cert.pem
```

之后使用新生成的组合 CA 证书并重启 TiDB Server，此时服务端可以同时接受和使用新旧 CA 证书。

## 2.更新服务端证书

生成新的服务端密钥和证书：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.new.pem -out server-req.new.pem
sudo openssl rsa -in server-key.new.pem -out server-key.new.pem
```

使用新的组合 CA 为新服务端证书签名：

```
sudo openssl x509 -req -in server-req.new.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 -out server-cert.new.pem
```

之后配置 TiDB 使用新的服务端证书。

## 3.更新客户端证书

生成新的客户端密钥和证书：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.new.pem -out client-req.new.pem
sudo openssl rsa -in client-key.new.pem -out client-key.new.pem
```

注意：这里目标是替换密钥和证书为了保证在线用户不受影响，所以上面这个命令中填写的附加信息必须与已配置的 require subject 信息一致。然后使用新的组合 CA 签名新客户端证书：

```
sudo openssl x509 -req -in client-req.new.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 -out client-cert.new.pem
```

使用新的证书连接 TiDB：

```
mysql -utest -h0.0.0.0 -P4000 --ssl-cert /path/to/client-cert.new.pem --ssl-key /path/to/client-key.new.pem --ssl-ca /path/to/ca-cert.pem
```



## 第11章 TiSpark 简介与实战

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它能够让 Spark SQL 直接运行在分布式存储引擎 TiKV 上，和 TiDB 一起为用户提供了一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 解决方案。

在这部分内容中，首先向大家介绍 TiSpark 的总体架构和基本原理，然后介绍 TiSpark 的部署和使用方法，接下来向大家介绍 TiSpark 如何运行在 PingCAP 为复杂分析型场景而研发的列式存储引擎 TiFlash 的基本知识，最后介绍 TiSpark 融合到已有大数据生态体系的方法。

### 目录

- [11.1 TiSpark 架构与原理](#)
- [11.2 TiSpark 的使用](#)
- [11.3 TiSpark on TiFlash](#)
- [11.4 TiSpark 结合大数据体系](#)

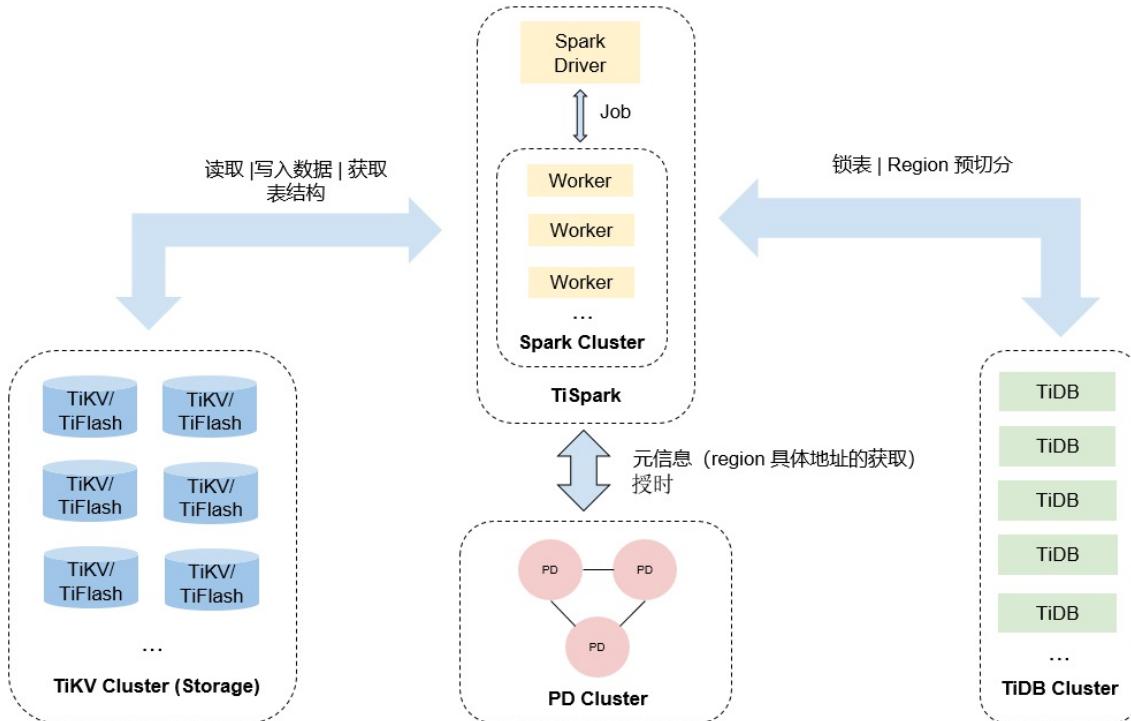
## 11.1 TiSpark 架构与原理

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时结合 TiKV 以及 TiFlash 分布式行列混合集群的优势，和 TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。TiSpark 依赖于 TiKV 集群和 Placement Driver (PD)，也需要你搭建一个 Spark 集群。

本文主要介绍 TiSpark 架构和原理。本文假设你对 Spark 有基本认知。你可以参阅 [Apache Spark 官网](#) 了解 Spark 的相关信息。

### 11.1.1 概述

TiSpark 是将 Spark SQL 直接运行在分布式存储引擎 TiKV 上的 OLAP 解决方案。其架构图如下：



- TiSpark 内置实现 TiKV 和 PD Java Client，让 TiSpark 可以通过 gRPC 与 TiKV 和 PD 通信，从 TiKV / TiFlash 中获取 Key-Value Pair 和表结构用于支持 TiSpark SQL 计算，从 PD 获取数据在 TiKV 上的具体 Region 信息及其副本的物理定位。
- TiSpark 在分布式写入数据时需要通过 TiDB 来进行锁表和 Region 预切分操作，保证数据写入正确性和高效性
- TiSpark Driver 侧：
  - 通过 PD Client 从 PD 中获取 TiDB metadata 信息，并将 TiDB 的 metadata 信息转化 Spark 支持的 metadata 信息。转化成功之后 TiSpark 可以看到 TiDB 的表。
  - 劫持和改写 Spark SQL 的执行计划，添加和 TiKV 兼容的物理算子
  - 通过 PD 定位数据所在的 Region 和获取当前时间戳
    - 将协处理器请求发送至 Region 所在的 TiKV
    - 获取时间戳是为了进行快照读取
  - 将查询任务按 region 拆分
    - 增加并发，加快查询速度
- Spark Executor 侧
  - 定制的物理算子从 TiKV 读取数据
  - 将 TiKV 数据包解码并转化为为 Spark SQL 的行格式

### 11.1.2 富 TiKV Java Client

如上架构所示，TiSpark 需要从 TiKV 中获取表结构信息和底层 Key-Value Pair 信息，那么 TiSpark 如何与 TiKV 通信获取这些信息呢？这里就需要 TiKV Java Client，通过 gRPC 与 TiKV Server 通信调用 TiKV API。

- 解析 TiKV Table Schema 将 TiKV 中的 Schema 信息转化为 Spark SQL 认识的 Schema
- 解析 TiKV 的类型系统
- 从 TiKV 中获取的数据是 Key-Value Pair，需要编码和解码模块负责将 Key-Value Pair 转化为 Spark SQL 可以使用的数据。这里的编解码逻辑和 TiDB 编解码逻辑一致。
- 协处理器支持，可以把谓词，索引，键值域处理计算下推到 TiKV 侧，减少数据传输过程，更能利用 TiKV 的分布式计算能力。在调用协处理的时候也依赖上面类型系统和编码系统，用于构造协处理器调用参数。
- 为了更加精确选择查询计划，提高 SQL 运行效率，TiSpark 中 Java TiKV Client 利用了 TiDB 的统计信息实现了更合理的基于代价的估算。

### 11.1.3 打通 TiKV 和 TiSpark

通过富 Java TiKV Client 可以完成 TiSpark 与 TiKV 通信，获取 TiKV 的相关数据，如何将 TiKV 的数据注入到 Spark 中完成 Spark 程序分布式计算呢？答案是通过修改 Spark Plan 可以完成。Spark 内置了扩展性接口，通过扩展 SparkSessionExtensions 类，Spark 可以实现用户自定义 SQL Plan、语法支持以及元数据解析等。具体可以参见下图 Spark 官网 API 说明。

The screenshot shows the Apache Spark documentation for the `SparkSessionExtensions` class. The title bar includes the package name `org.apache.spark.sql`, the class name `SparkSessionExtensions`, and a "Related Doc: package sql" link. Below the title, there's a "Developer API" button and an "Experimental" badge. The main content area starts with the class definition: `class SparkSessionExtensions extends AnyRef`. It then describes the class as a "Holder for injection points to the `SparkSession`. We make NO guarantee about the stability regarding binary compatibility and source compatibility of methods here." It lists the current extension points: Analyzer Rules, Check Analysis Rules, Optimizer Rules, Planning Strategies, Customized Parser, and (External) Catalog listeners. An example code snippet shows how to use these extensions via the `withExtension` method on a `SparkSession.Builder`. A note at the bottom cautions that injected builders should not touch the session's internals.

```

SparkSession.builder()
    .master("...")
    .conf("...", true)
    .withExtensions { extensions =>
      extensions.injectResolutionRule { session =>
        ...
      }
      extensions.injectParser { (session, parser) =>
        ...
      }
    }
    .getOrCreate()
  
```

Note that none of the injected builders should assume that the `SparkSession` is fully initialized and should not touch the session's internals (e.g. the `SessionState`).

TiSpark 配置文件中有一个重要的配置项：

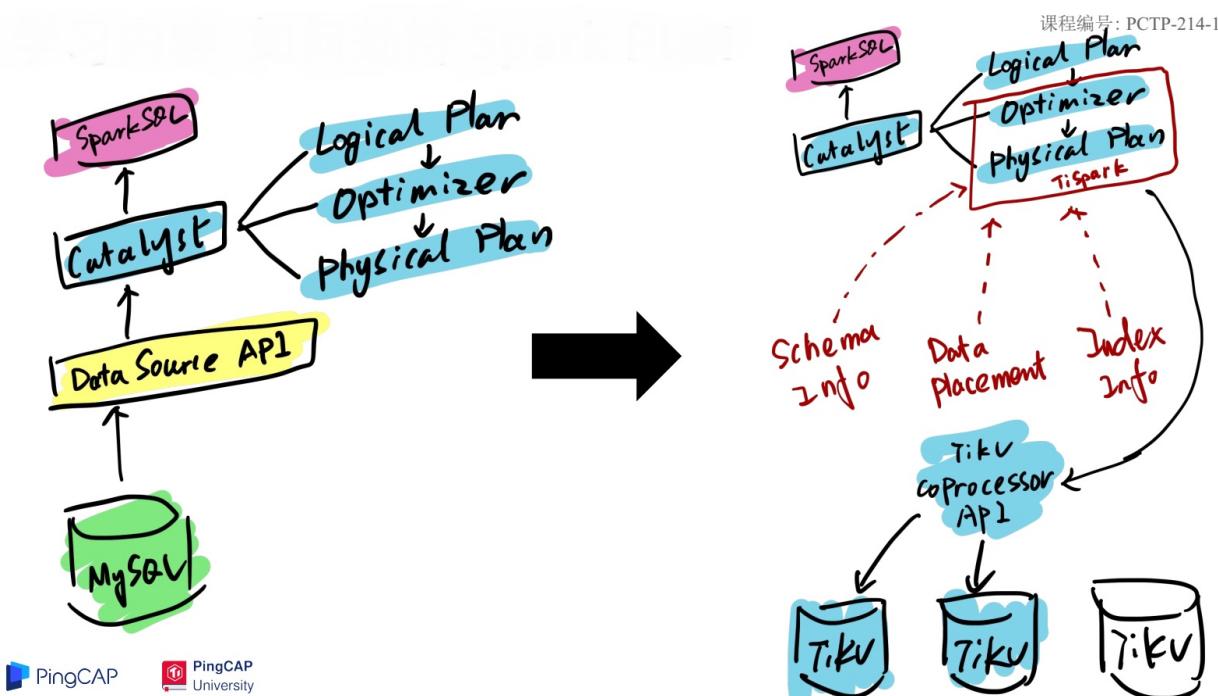
`spark.sql.extensions org.apache.spark.sql.TiExtensions`

此配置为 `TiExtensions`，Spark 在启动的时候会自动加载此类并初始化此类的实例，当此类实例被初始化时，会调用 `SparkSessionExtensions` 相关的扩展方法将 TiSpark 实现的扩展方法注入到 Spark 中。

```

16     tiContextMap.put(sparkSession, tiContext)
17     tiContext
18   }
19 }
20
21 override def apply(e: SparkSessionExtensions): Unit = {
22   e.injectParser(TiParser(getOrCreateTiContext))
23   e.injectResolutionRule(TiDDLRule(getOrCreateTiContext))
24   e.injectResolutionRule(TiResolutionRule(getOrCreateTiContext))
25   e.injectPlannerStrategy(TiStrategy(getOrCreateTiContext))
26 }
27
28 }
```

原生 Spark SQL 执行计划如下图左边所示，一个 Spark SQL 开始运行，通过 Catalyst 解析器被解析为逻辑执行计划，在优化器中生成物理执行计划，最后通过 DataSource 的 API 获取数据。TiSpark 修改 Spark Plan 之后的 SQL 执行过程如下图右边所示。将 SQL 优化器和物理执行计划改写为与 TiKV 相关的交互，在物理执行计划中从 TiKV 获取表结构、数据和索引等信息。改写之后 Spark 看到的接口不变，但是底层实现变成了与 TiKV 交互，既保证了与原来 Spark 程序的兼容又完成 TiKV 数据注入。



#### 11.1.4 聚簇索引

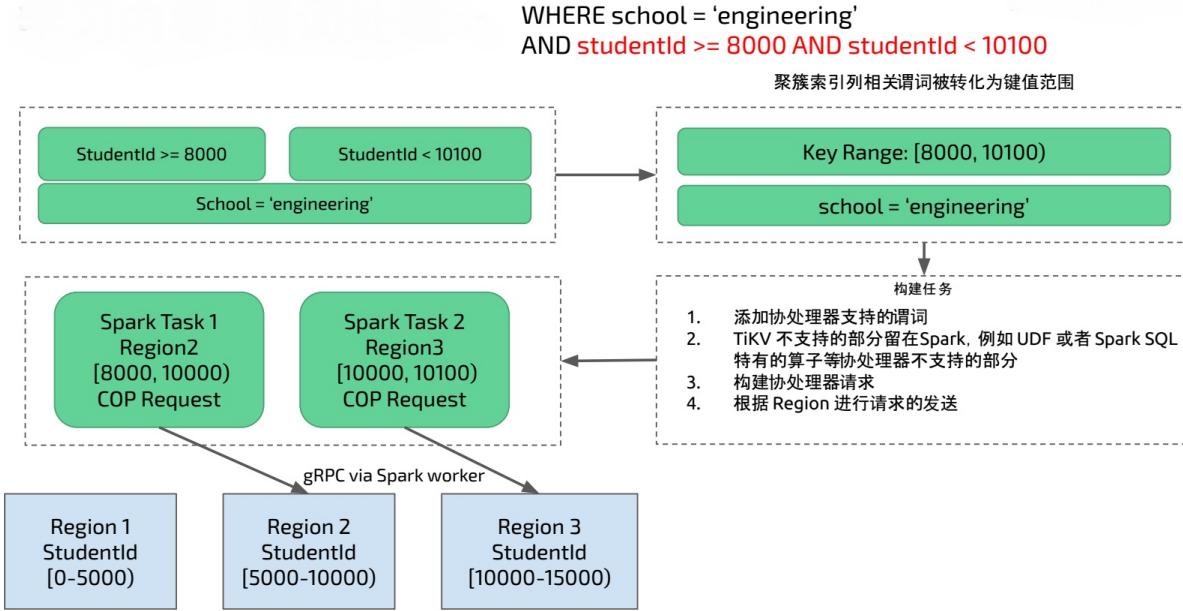
上面 Spark SQL 架构比较抽象，具体来看一个例子：

Spark SQL 运行如下 SQL，其中 student 表是 TiDB 中的表，在 studentID 列上有聚簇索引，在 school 列上有非聚簇索引。聚簇索引会将索引和数据放在一起。

```
SELECT class, avg(score) FROM student
```

```
WHERE school = 'engineering' AND studentId >= 8000 AND studentId < 10100
```

```
GROUP BY class;
```



在上图中 studentID 是一个聚簇索引，TiKV Region 中会包含聚簇索引的范围，比如上图中 Region 2 的 studentId 范围是 [5000 - 10000)，Region 3 的 studentID 范围是 [10000 - 15000)。在 SQL 运行时聚簇索引会转化为对 TiKV 的范围查询，现在要查找范围在 8000 到 10100 的数据，TiSpark 会将对应的请求发送到 Region 2 和 Region 3 做范围查找。TiSpark 会在 Spark Executor 端将 TiKV 支持的谓词发送给 TiKV 协处理器计算，并将 TiKV 计算之后的结果进行汇总和再计算。对于 TiKV 不支持的谓词部分会留在 Spark 中进行计算，从而得到最终 SQL 运行结果。

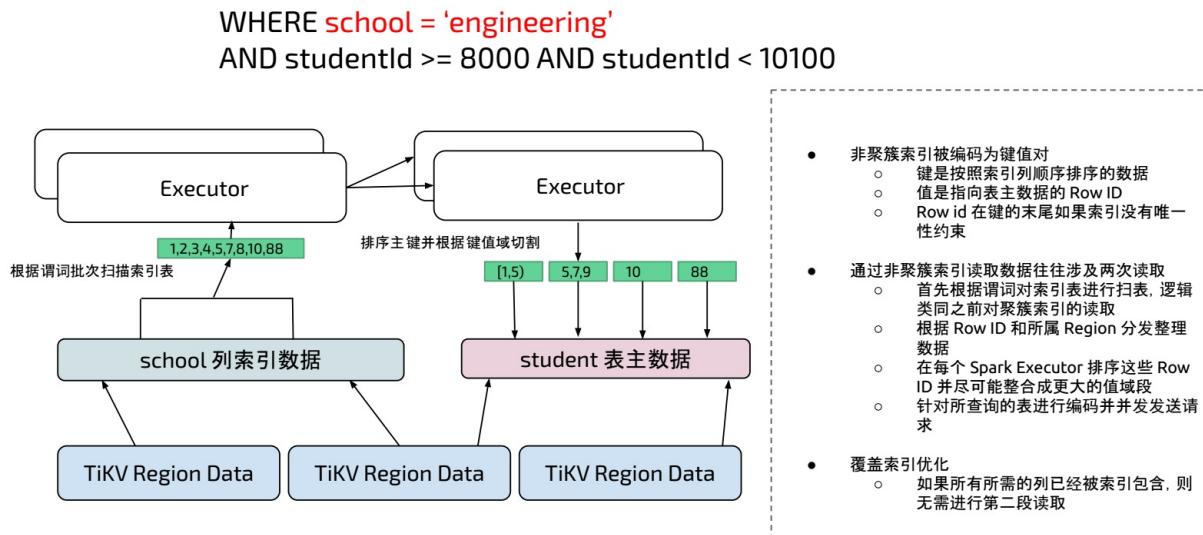
### 11.1.5 非聚簇索引处理

非聚簇索引被编码为键值对，键是按照索引列顺序排序的数据，值是指向表主数据的 Row ID。

通过非聚簇索引读取数据往往涉及两次读取

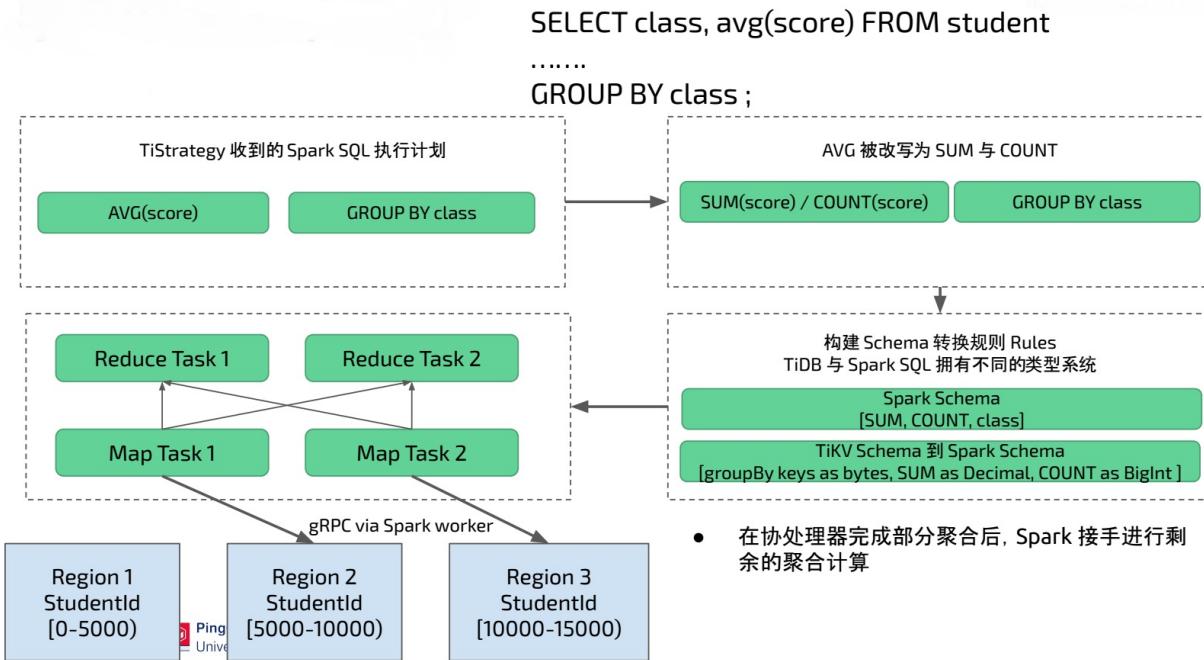
- 首先根据谓词对索引表进行扫表，逻辑类同之前对聚簇索引的读取。
- 根据 Row ID 和所属 Region 分发整理数据
- 在每个 Spark Executor 排序这些 Row ID 并尽可能整合成更大的值域段
- 针对所查询的表进行编码，同时并发发送请求

例如下图中扫描 school 的非聚簇索引表数据，得到 1,2,3,4,5,7,8,10,88 的 Row ID，在 Spark Executor 端对这些 Row ID 排序，再根据 Row ID 对 student 主表进行范围扫描，再将 TiKV 主表返回数据在 Spark 中再计算得到最终结果。



### 11.1.6 聚合处理

TiStrategy 负责改写 TiSpark 的物理执行计划，假设 Spark SQL 中包含 AVG(score) 和 GROUP BY class。TiSpark 会将 AVG 改写为 SUM(score) / COUNT(score)，在改写过程中还需要将 TiSpark 的表达式改写为与 TiDB 兼容的表达式，以保证 TiKV 协处理器同时支持 TiDB 和 TiSpark。在下图的场景中 group by 返回的类型是 bytes，SUM 返回类型是 Decimal，Count 返回类型是 Bigint。而这些 TiDB 表达式的返回值到 Spark 中无法被直接使用，还需要转化为 Spark 中兼容的类型。在 Spark Executor 执行中 Map Task 负责发送协处理器并处理 TiKV 协处理器返回的数据，Reduce Task 将 Map Task 的数据进行聚合得到最终结果。



### 11.1.7 分布式大数据写入

最初 TiSpark 只能通过 TiDB JDBC 的方式将数据写入到 TiDB，这存在可扩展性问题。通过 TiSpark 直接写入 TiKV 则可以解决此问题。在 Spark 中数据一般是以 DataFrame 的形式存在，TiSpark 写入过程中可以将 DataFrame 的数据转化为 TiKV 认识的格式，并通过 TiKV Java Client 将数据写入 TiKV。

- 根据 DataFrame 中数据进行 Region 预切分和分配
- TiKV 数据写入需要支持分布式事务，TiKV 采用 Percolator 协议进行事务操作，操作过程如下：
  - 在 Spark Driver 端开始写数据时申请 TiKV 中主数据预写，对此条数据加锁。
  - 在 Spark Executor 端将 DataFrame 转化为 TiKV 的 Key-Value Pair 格式，并调用 gRPC 进行次数据预写，将 DataFrame 数据存入到 TiKV，此过程如果存在写入冲突可以选择报错或者覆盖写入。
  - 在 Spark Driver 端等待 Spark Executor 预写成功，再将主数据提交。Percolator 提交成功取决于主数据提交状态。
  - 在 Spark Executor 端提交次数据。到此完成了所有两阶段事务提交。

### 11.1.8 总结

TiSpark 实现了富 TiKV Java Client，并通过 Spark 内置扩展接口改写 Spark Plan，将 TiKV 的表结构和数据集成到 Spark 中。非常巧妙的将 TiKV 体系和现有大数据体系融合起来。再通过分析 TiSpark 对聚簇和非聚簇索引的处理，以及协处理器在其中的作用，加深了对 TiSpark 与 TiKV 交互的理解。最后分析 TiSpark 分布式写入 TiKV，完成了 TiSpark 对 TiKV 读和写的总体理解。

## 11.2 TiSpark 的使用

上一节中，针对 TiSpark 的架构和原理进行了详细的介绍，在本节，我们会介绍 TiSpark 的部署和使用方法。

### 11.2.1 TiSpark 的部署

由于 TiSpark 并没有直接修改 Apache Spark 的代码，因此只要是 Apache Spark 2.1 以上版本，就可以找到对应兼容的 TiSpark。具体版本的对应，可以查看官网文档[对应章节](#)。无论是通过 YARN 部署还是通过 Standalone 部署，都可以参考 Apache Spark 官网的[部署环节](#)。通过匹配 Spark 版本，可以从 [TiSpark Release 栏目](#) 下载对应的 TiSpark 版本的 JAR 包。

实际开启和部署 TiSpark 需要确保如下两点：

1. Spark Driver 以及 Executor 可以访问到 TiSpark JAR 包。
2. Spark 开启如下参数（推荐通过修改 spark-defaults.conf 来控制）

```
spark.sql.extensions          org.apache.spark.sql.TiExtensions
spark.tispark.pd.addresses    pd-host1:port1,pd-host2:port2,pd-host3:port3
```

以 Standalone 集群为例，加入 TiSpark 流程如下：

1. 按照上述介绍修改 SPARK\_HOME/spark-defaults.conf，加入上述必要配置。
2. 启动 Spark 应用时引用 TiSpark JAR 包。以 spark-shell 为例：

```
./spark-shell --jars /path/your-tispark.jar
```

### 11.2.2 TiSpark 的基本查询方式

TiSpark 的使用方法与原生 Spark 类似，提供多种方式查询数据。

#### 11.2.2.1 通过 spark-shell 查询数据

使用 spark-shell 启动，通过 Spark 提供的 API 即可访问数据。

示例一：利用 spark-shell 查询 lineitem 表的数据

```
scala>spark.sql("use tpch")
使用 tpch 库
scala>spark.sql("select count(*) from lineitem").show
查询 lineitem 表的总行数
查询结果显示如下：
+-----+
| Count (1) |
+-----+
| 600000000 |
+-----+
```

示例二：利用 Spark SQL 查询复杂 sql

```

scala> spark.sql(
    """select
       |   l_returnflag,
       |   l_linenumber,
       |   sum(l_quantity) as sum_qty,
       |   sum(l_extendedprice) as sum_base_price,
       |   sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
       |   sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
       |   avg(l_quantity) as avg_qty,
       |   avg(l_extendedprice) as avg_price,
       |   avg(l_discount) as avg_disc,
       |   count(*) as count_order
      |from
      |   lineitem
     |where
     |   l_shipdate <= date '1998-12-01' - interval '90' day
    |group by
       |   l_returnflag,
       |   l_linenumber
    |order by
       |   l_returnflag,
       |   l_linenumber
    """.stripMargin).show
scala>
+-----+-----+-----+-----+
|l_returnflag|l_linenumber|sum_qty|sum_base_price|sum_disc_price|
+-----+-----+-----+-----+
|        A|          F| 380456.00| 532348211.65| 505822441.4861|
|        N|          F|   8971.00| 12384801.37| 11798257.2080|
|        N|          O| 742802.00| 1041502841.45| 989737518.6346|
|        R|          F| 381449.00| 534594445.35| 507996454.4067|
+-----+-----+-----+-----+

```

### 11.2.2.2 通过 Spark SQL 查询数据

TiSpark同样支持利用 Spark SQL 查询数据。使用 spark-sql 命令即可进入交互式数据查询页面，接入输入 sql 即可。

示例三：利用 spark-sql 查询 lineitem 表的数据

```

spark-sql> use tpch;
使用 tpch 库
spark-sql> select count(*) from lineitem;
查询lineitem表的总行数
2000
Time taken: 0.673 seconds, Fetched 1 row(s)

```

### 11.2.2.3 利用 JDBC 访问 TiSpark

部署时启动 Thrift 服务器后，可以通过 JDBC 的方式使用 TiSpark。

示例四：利用 beeline 工具使用 JDBC 的方式访问 TiSpark

```

beeline> !connect jdbc:hive2://localhost:10000
1: jdbc:hive2://localhost:10000> use testdb;
+-----+---+
| Result |
+-----+---+
+-----+---+
No rows selected (0.013 seconds)
select count(*) from account;
+-----+---+
| count(1) |
+-----+---+
| 10000000 |
+-----+---+
1 row selected (1.97 seconds)

```

### 11.2.3 TiSpark 的多语言使用

#### 11.2.3.1 使用 PySpark 访问 TiSpark

TiSpark on PySpark 是 TiSpark 用来支持 Python 语言而构建的 Python 包。PySpark 支持直接使用也可以通过 python 的包管理工具来安装使用。

示例一：直接使用 PySpark 访问 TiSpark

```
./bin/pyspark --jars /PATH/tispark-${name_with_version}.jar
# Query as you are in spark-shell
spark.sql("show databases").show()
spark.sql("use tpch_test")
spark.sql("show tables").show()
spark.sql("select count(*) from customer").show()
# Result
+-----+
|count(1)|
+-----+
|      150|
+-----+
```

示例二：利用 pip 安装 pytispark 后使用 TiSpark

首先，利用 pip 来安装 pytispark，相关命令如下：

```
pip install pytispark
```

安装完成之后，创建一个用以查询数据的 python 文件 test.py，文件示例如下：

```
import pytispark.pytispark as pti
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
ti = pti.TiContext(spark)
ti.tidbMapDatabase("tpch_test")
spark.sql("select count(*) from customer").show()
```

创建完成之后使用 spark-submit 来查询数据，相关命令如下：

```
./bin/spark-submit --jars /PATH/tispark-${name_with_version}.jar test.py
# Result:
+-----+
|count(1)|
+-----+
|      150|
+-----+
```

#### 11.2.3.2 使用 TiSparkR 访问 TiSpark

TiSparkR 是 TiSpark 用来支持 R 语言来构建的 R 包。同 PySpark 类似，TiSparkR 同样支持直接使用也可以通过加载 library 的方式使用。

示例三：直接使用 PySpark 访问 TiSpark

```
./bin/sparkR --jars /PATH/tispark-${name_with_version}.jar
sql("use tpch_test")
count <- sql("select count(*) from customer")
head(count)
# Result
+-----+
|count(1)|
+-----+
|      150|
+-----+
```

示例四：利用 SparkR 包的形式使用 TiSpark 首先，创建一个用以查询数据的 R 文件 test.R，文件示例如下：

```
library(SparkR)
sparkR.session()
sql("use tpch_test")
count <- sql("select count(*) from customer")
head(count)
```

创建完成之后使用 spark-submit 来查询数据，相关命令如下：

```
./bin/spark-submit --jars /PATH/tispark-${name_with_version}.jar test.R
# Result:
+-----+
|count(1)|
+-----+
|      150|
+-----+
```

### 11.2.3.3 TiSpark 访问 TiFlash

参见 TiSpark 访问 TiFlash 章节

## 11.3 TiSpark on TiFlash

TiFlash 弥补了 TiSpark 在分析场景下读取 TiKV 会引发性能抖动的缺陷。以往需要通过限制 TiSpark 读取并发度以确保业务不受影响的情况可以在 TiFlash 的帮助下完美解决。TiSpark 访问 TiFlash 的方式与访问 TiKV 几乎一致，也是经过协处理器下推来进行加速：TiFlash 会接受协处理器请求，将每个 Region 的计算结果分别返回，由 TiSpark 进行后续计算和汇总。与 TiKV 不同的是，TiFlash 针对 TiSpark 提供了原生的编码格式支持，这个格式下 TiFlash 无需按照 TiDB 格式进行编码转换，而是直接以原始计算结果的编码格式返回数据。在该模式下，数据由 TiFlash 向 TiSpark 传输的速度将大大加快，例如表连接场景可以受益。

### 11.3.1 TiSpark 读取 TiFlash

TiSpark 本身的配置仍然如前所述，需要下载 TiSpark JAR 并且在配置中添加 TiExtension 以及 PD 地址配置项。另外请确保 TiFlash 的节点可被 TiSpark 访问。

TiSpark 目前提供类似 TiDB 中 engine 隔离的方式读取 TiFlash，方式是通过配置参数：

```
spark.tispark.use.tiflash 为 true (或 false)
```

可以使用以下任意一种方式进行设置：

- 在 spark-defaults.conf 文件中添加

```
spark.tispark.use.tiflash true
```

- 在启动 spark-shell 或 ThriftServer 时，启动命令中添加

```
--conf spark.tispark.use.tiflash = true
```

- Spark Shell 中实时设置：

```
spark.conf.set("spark.tispark.use.tiflash", true)
```

- ThriftServer 通过 beeline 连接后实时设置：

```
set spark.tispark.use.tiflash = true;
```

注意，设为 true 时，所有查询的表都会只读取 TiFlash 副本，设为 false 则只读取 TiKV 副本。设为 true 时，要求查询所用到的表都必须已创建了 TiFlash 副本，对于未创建 TiFlash 副本的表的查询会报错。

### 11.3.2 相关参数优化

- 合并 Region 请求。TiSpark 配合 TiFlash 的场景下，由于每个 Region 请求响应的速度将比 TiKV 中大大提升，如果小 Region 过多，会使得 Spark 的调度速度无法跟上而降低计算效率。在这样的场景下，用户可以尝试开启 Region 请求合并功能。开启之后，TiSpark 将会每次同时在一个请求中包含多个 Region 计算请求。一般推荐在 Region 平均大小小于 48M 的时候可以将请求合并数设为 2。默认情况下，每次 TiSpark Split 请求只包含一个 Region。

```
spark.tispark.partition_per_split 2
```

- 取消调度等待。如果是 TiFlash 和 TiSpark 并非同机部署，或者业务以聚合计算为主，那么推荐将调度等待关闭。这是因为非同机部署并不会有可能产生本地读取优化或者优化可忽略，反而会因为调度等待大大拖慢计算。

```
spark.locality.wait 0s
```



## 11.4 TiSpark 结合大数据体系

作为新兴的存储引擎，TiDB 虽然在架构上比大多数传统主流的大技术体系要相对优雅和先进，但由于技术生态以及迁移成本等问题，不可能在短时间内取而代之，因此必然面临和前辈共处一室的尴尬场景。毕竟谁也拉不下面子把自己辛苦收集和处理的数据再抽取一次丢给对方使用，但老板又说要实现数据协同效应，怎么办呢？下面介绍一些皆大欢喜的解决办法。

### 11.4.1 与 Hive 表混和读写

但凡有些规模和积累的公司一定不会对基于 Hadoop 集群的 Hive 表陌生，毕竟廉价，稳定且成熟。但 Hive 表不适合存放频繁变化的数据，而这却是 TiDB 的强项，这就导致了很多业务公司可能既有 Hadoop 集群又有 TiDB 集群。要实现完美的混和访问，我们希望达到以下三个核心目标：

1. 对业务 SQL 来讲，不能有 Hive/TiDB 表的区分，都按库名+表名进行访问，不要有多余操作。
2. 由于访问 Hive 表可能需要消耗巨大资源，因此最好可以使用与之配套的计算集群资源。
3. 确保 hive-site.xml 能够被 Spark 访问到，例如 hive-site.xml 复制到 SPARK\_HOME/conf 下。hive-site.xml 中包含了 Hive Metastore 相关信息，只有 Spark 可以读取它，才能访问 Hive 中的数据。下节将用一个具体的例子进行阐述。

### 11.4.2 使用 beeline + Livy + Spark + Tispark 实现混访

#### 1. 软件环境清单：

- Livy 服务，版本：基于 0.6 版本的改动版，作用：提交原始任务到 Spark 客户端；
- beeline 客户端，版本：3.1.1，作用：连接 Livy 服务的客户端；
- Spark 客户端，版本：2.4.0，作用：向 yarn 集群提交 Spark 作业；
- Tispark Jar 包，版本：2.1.8，作用：提供与 TiKV 交互的能力；
- YARN 集群，作用：提供计算资源。

#### 2. 函数封装代码：

```
function runMixSQLOnLivy(){
export HIVE_HOME=/usr/local/share/apache-Hive-3.1.1-bin
/usr/local/share/apache-Hive-3.1.1-bin/bin/beeline -n hdfs_user_name -p hdfs_user_pwd --verbose=false --color=false \
-u "jdbc:Hive2://bj0000,bj0001,bj0002:2222;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=mix-livy" \
--Hiveconf livy.session.conf.spark.sql.extensions=org.apache.spark.sql.TiExtensions \
--Hiveconf livy.session.conf.spark.tispark.pd.addresses=10.10.10.11:2379,10.10.10.12:2379,10.10.13:2379 \
--Hiveconf livy.session.conf.spark.jars=hdfs://com-hdfs/user/spark/tispark-core-2.1.4-spark_2.4-jar-with-dependencies.jar \
--Hiveconf livy.session.name=session_name_${RANDOM}_$2 \
--Hiveconf livy.session.queue=your_yarn_queue_name \
-e "$1"
}
```

#### 3. 封装函数说明：

- 上述函数为封装的 shell 脚本函数，接收 2 个参数，\$1 是 SQL 代码，\$2 是可选的参数用以进行任务标识，本身使用了 \$RANDOM 保证 session 名称不重复；
- beeline 本身支持多种用户认证的方式，因此可以根据环境的具体情况变化，详情不在这里讲述。这里使用的是使用 hdfs 用户来认证；
- -u 是指 JDBC URL，用来连接 livy 服务；
- TiDB 相关参数：
  - a) spark.sql.extensions=org.apache.spark.sql.TiExtensions
  - b) spark.tispark.pd.addresses=10.10.10.11:2379,10.10.10.12:2379,10.10.13.10:2379
  - c) spark.jars=hdfs://com-hdfs/user/spark/tispark-core-2.1.4-spark\_2.4-jar-with-dependencies.jar
- livy.session.name 需要保证唯一，因此加了随机数及\$2；
- livy.session.queue 是 YARN 的队列名称；
- 如果不加 -e "\$1" 即可以实现交互查询。

#### 4. 实际效果演示：

##### (1) 查看库：

```
0: jdbc:Hive2://bj0000,bj0001,bj0002:2222> show databases;
+-----+
| databaseName |
+-----+
| sales_db     |
| db_em        |
| db_test       |
| db_shr        |
+-----+
```

上面 sales\_db 是 Hive 库，db\_em 及以下是 TiDB 库。

##### (2) 单独查询 TiDB 库表：

```
0: jdbc:Hive2://bj0000,bj0001,bj0002:2222> select count(1) from db_em.app_war_room_fpyr_rt;
RSC client is executing SQL query: select count(1) from db_em.app_war_room_fpyr_rt, statementId = 91583fc3-0837-
4b26-b579-b438a53f151e, session = SessionHandle [b6933d96-8ec0-47b9-a767-ff2da5c5b2b2]
[Stage 0:>                                              (0 + 1) / 1]
+-----+
| count(1)   |
+-----+
| 224        |
+-----+
1 row selected (6.505 seconds)
```

##### (3) 单独查 Hive 表：

```
0: jdbc:Hive2://bj0000,bj0001,bj0002:2222> select count(1) from sales_db.mdms_tsqa_syyt;
RSC client is executing SQL query: select count(1) from sales_db.mdms_tsqa_syyt, statementId = 324d7d1d-03bc-4d9
b-849f-18967305a454, session = SessionHandle [b6933d96-8ec0-47b9-a767-ff2da5c5b2b2]
[Stage 2:>                                              (0 + 1) / 151]
...
[Stage 3:>                                              (0 + 0) / 1]
+-----+
| count(1)   |
+-----+
| 142380109 |
+-----+
1 row selected (25.169 seconds)
```

##### (4) 混和查询并写入 Hive 表：

```
0: jdbc:Hive2://bj0000,bj0001,bj0002:2222> insert into dc_tmp.test_for_mix select count(1) as cnt from sales_d
b.mdms_tsqa_syyt union select count(1) as cnt from db_em.app_war_room_fpyr_rt;
RSC client is executing SQL query: insert into dc_tmp.test_for_mix select count(1) as cnt from sales_db.mdms_ts
qa_syyt union select count(1) as cnt from db_em.app_war_room_fpyr_rt, statementId = 7e87c512-d5c5-4ba1-bd62-d013
ff16a4e7, session = SessionHandle [b6933d96-8ec0-47b9-a767-ff2da5c5b2b2]
[Stage 5:>                                              (0 + 0) / 151]
.....
[Stage 10:>                                              (0 + 0) / 1]
+-----+
| Result   |
+-----+
+-----+
No rows selected (31.221 seconds)
```

##### (5) 检查结果：

```

0: jdbc:Hive2://bj0000,bj0001,bj0002:2222> select cnt from dc_tmp.test_for_mix;
RSC client is executing SQL query: select cnt from dc_tmp.test_for_mix, statementId = 18247b15-d9fb-4b97-87a1-6f
a9cc3afad8, session = SessionHandle [b6933d96-8ec0-47b9-a767-ff2da5c5b2b2]
[Stage 11:>                                         (0 + 1) / 1]
.....
[Stage 11:>                                         (0 + 1) / 1]
+-----+
|   cnt   |
+-----+
| 142380109 |
| 224      |
+-----+
2 rows selected (2.977 seconds)

```

可以看到是成功的实现了混和访问，对业务逻辑来说，数据在Hive库或者在TiDB库没有任何感知。

### 11.4.3 改进地方：

#### 1. 写回TiDB

TiDB 4.0 实现大事务支持之前，TiSpark 没有理想的方案支持向 TiDB 原生写入数据的方案。用户可以选择的是：

- 使用 Spark 原生的 JDBC 方案，将 TiDB 当做 MySQL 写入数据，具体方案请参考[文档](#)。这个方案的缺陷是，数据必须被拆分为小批次插入，而这些批次之间无法维持事务原子性。换句话说，如果插入在中途失败，那么已经写入的数据并不会自动回滚，而需要人工干预。
- 第二个方案是使用 TiSpark 的[大批写入](#)，这个方案可以导入大量数据且维持事务的原子性，但是由于缺少锁表和大事务支持，并不推荐在生产环境使用。

在 TiSpark 完成对应 TiDB 4.0 大事务对应的 support 后，用户就可以使用 TiSpark 作为一种主要的 TiDB 跑批方案，无论是向 TiDB 写入还是由 TiDB 向其他系统写出。在本文写作的时间点，此功能尚未完成，如有相关需要，请关注官方 [Github 页面](#) 更新 TiSpark 版本。

#### 2. 库重名问题

TiDB 和 Hive 重名的情况，需要为 TiSpark 开启表名前缀模式，该模式会为所有 TiDB 表在 TiSpark 中加入前缀（而并不会改变 TiDB 内实际的表名）。例如，希望 TiDB 表在 TiSpark 中以 tidb\_ 作为前缀使用，则增加如下配置（这并不会实际改变 TiDB 的表名）：

```
spark.tispark.db_prefix "tidb_"
```

#### 11.4.4 TiSpark 与其他系统协同

由于 TiSpark 没有直接修改 Apache Spark 代码，因此 Spark 原生兼容的大多数功能仍可正常运行。可以参考 Apache Spark 如何访问各个系统的文档正常使用，这里不做赘述。

## 第 1 章 部署安装和常见运维

在深入理解和使用一款软件之前，最基本的方式就是先部署一套跑一跑。毕竟是骡子是马，跑出来遛遛才知道。本章，我们将真正地手把手教你玩 TiDB，由浅入深带你走进 TiDB 的世界，主要内容如下：

- TiUP TiDB 官方版本管理工具，带您一分钟部署一套 TiDB 集群。
- TiDB on Kubernetes TiDB 在 Kubernetes 平台上地自动化部署运维指导
- TiDB 集群如何扩容缩容
- TiDB 集群版本升级
- TiDB 集群动态修改配置

## 1.1 TiUP & TiOps

在单机部署一个 TiDB 集群要多久？

之前，我们其实很难回答这个问题，但现在可以很自豪的说「一分钟」。为什么会这么快？因为我们专门为 TiDB 4.0 做了一个全新的组件管理工具：TiUP。

在多机环境部署一个 TiDB 集群要多久？

之前，我们同样很难回答这个问题，但现在，答案仍然是「一分钟」，因为我们可以方便地使用 TiUP cluster 功能来快速部署集群。

TiUP 会管理 TiDB 整个生态里面的组件，无论是核心组件 tidb/tikv/pd/tiflash 等，还是生态工具 prometheus/grafana/drainer/pump 等，都可以通过 TiUP 来进行管理和使用，用户也可以给 TiUP 添加各种组件工具。

本章我们就来介绍 TiUP 的功能和用法。

## 1.1.1 TiUP 简介

在各种系统软件和应用软件的安装管理中，包管理器均有着广泛的应用，包管理工具的出现大大简化了软件的安装和升级维护工作。例如，几乎所有使用 RPM 的 Linux 都会使用 Yum 来进行包管理，而 Anaconda 则可以非常方便地管理 python 的环境和相关软件包。在早期的 TiDB 生态中，没有专门的包管理工具，使用者只能通过相应的配置文件和文件夹命名来手动管理，像 Prometheus 等第三方监控报表工具甚至需要额外的特殊管理，这样大大提升了运维管理难度。

如今，在 TiDB 4.0 的生态系统里，TiUP 作为新的工具，承担着包管理器的角色，管理着 TiDB 生态下众多的组件（例如 TiDB、PD、TiKV）。用户想要运行 TiDB 生态中任何东西的时候，只需要执行 TiUP 一行命令即可，相比以前，极大地降低了管理难度。用户可以访问 <https://tiup.io/> 来查看相应的文档。

## 1. 安装

作为一个包管理工具，TiUP 的安装非常简单，只需要在控制台执行如下命令：

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

该命令将 TiUP 安装在 `$HOME/.tiup` 文件夹下，之后安装的组件以及组件运行产生的数据也会放在该文件夹下。同时，它还会自动将 `$HOME/.tiup/bin` 加入到 Shell Profile 文件的 PATH 环境变量中，这样你就可以直接使用 TiUP 了，譬如查看 TiUP 的版本：

```
tiup --version
```

该文档主要参照 TiUP v0.0.3 版本，由于 TiUP 功能还在不断改进完善中，所以文档可能存在与最新版不一致的地方。

## 2. 功能介绍

TiUP 的使用非常简单，只需要利用 TiUP 的命令或者组件即可。首先执行 `tiup help` 看一下它的用法：

```

-$ tiup help
TiUP is a command-line component management tool that can help to download and install
TiDB platform components to the local system. You can run a specific version of a component via
"tiup <component>[:version]". If no version number is specified, the latest version installed
locally will be used. If the specified component does not have any version installed locally,
the latest stable version will be downloaded from the repository.

Usage:
  tiup [flags] <command> [args...]
  tiup [flags] <component> [args...]

Available Commands:
  install      Install a specific version of a component
  list         List the available TiDB components or versions
  uninstall    Uninstall components or versions of a component
  update       Update tiup components to the latest version
  status        List the status of instantiated components
  clean        Clean the data of instantiated components
  help         Help about any command or component

Available Components:
  playground    Bootstrap a local TiDB cluster
  package       A toolbox to package tiup component
  cluster       Deploy a TiDB cluster for production
  mirrors       Build a local mirrors and download all selected components

Flags:
  -B, --binary <component>[:version]  Print binary path of a specific version of a component <component>[:version]
                                         and the latest version installed will be selected if no version specified
  --binpath string                   Specify the binary path of component instance
  -h, --help                         help for tiup
  --skip-version-check              Skip the strict version check, by default a version must be a valid SemVer str
  ing
  -T, --tag string                  Specify a tag for component instance

Component instances with the same "tag" will share a data directory ($TIUP_HOME/data/$tag):
$ tiup --tag mycluster playground

Examples:
$ tiup playground                      # Quick start
$ tiup playground nightly                # Start a playground with the latest nightly version
$ tiup install <component>[:version]   # Install a component of specific version
$ tiup update --all                     # Update all installed components to the latest version
$ tiup update --nightly                 # Update all installed components to the nightly version
$ tiup update --self                    # Update the "tiup" to the latest version
$ tiup list --refresh                 # Fetch the latest supported components list
$ tiup status                          # Display all running/terminated instances
$ tiup clean <name>                   # Clean the data of running/terminated instance (Kill process if it's running)
$ tiup clean --all                     # Clean the data of all running/terminated instances

Use "tiup [command] --help" for more information about a command.

```

## TiUP 使用方式：

```

tiup [flags] <command> [args...]
tiup [flags] <component> [args...]

```

如上，一个典型的 TiUP 命令分为四个部分：

1. tiup: TiUP 程序名
2. flags: 全局通用选项，可选
3. command 或 component: 运行的命令或组件
4. args: 命令或组件的专有参数，可选

## 目前支持这些命令：

- list: 查询组件列表，知道有哪些组件可以安装，以及这些组件有哪些版本可选

- install: 安装某个组件的特定版本
- update: 升级某个组件到最新的版本
- uninstall: 卸载组件
- status: 查看组件运行状态
- clean: 清理组件实例
- help: 打印帮助信息，后面跟命令则是打印该命令的使用方法

常见的全局通用选项 flags :

- --binary : 打印某个组件的可执行程序文件路径
- --binpath : 指定要运行组件的可执行程序文件路径，这样可以不使用组件的安装路径
- --tag : 指定组件运行实例的 tag 名称，该名称可以认为是该实例的 ID，如果不指定，则会自动生成随机的 tag 名称

通过 `tiup list` 命令可以查看支持的组件，目前已经支持了上十个组件了，包括常用的 `playground/package/cluster` 等。随着时间的推移，组件还会越来越多，同时也希望大家积极参与贡献组件。

如果我们想要知道某个命令或组件的具体用法，可以执行 `tiup help <command|component>` 或者 `tiup <command|component> --help` 或者 `tiup <command|component> -h`。

比如我们想知道 `install` 命令用法，就可以执行 `tiup help install` 或者 `tiup install --help` 或者 `tiup install -h`。

下面我们按照正常使用习惯依次介绍各个命令。

## (1) 查询组件列表 : `tiup list`

当想要用 TiUP 安装东西的时候，首先需要知道有哪些组件可以安装，以及这些组件有哪些版本可以安装，这便是 `list` 命令的功能。相关的命令和参数如下：

```
~$ tiup help list
List the available TiDB components if you don't specify any component name,
or list the available versions of a specific component. Display a list of
local caches by default. You must use --refresh to force TiUP to fetch
the latest list from the mirror server. Use the --installed flag to hide
components or versions which have not been installed.

# Refresh and list all available components
tiup list --refresh

# List all installed components
tiup list --installed

# List all installed versions of TiDB
tiup list tidb --installed

Usage:
  tiup list [component] [flags]

Flags:
  -h, --help      help for list
  --installed    List installed components only.
  --refresh      Refresh local components/version list cache.

Global Flags:
  --skip-version-check  Skip the strict version check, by default a version must be a valid SemVer string
```

支持这几种用法：

- `tiup list` : 查看当前有哪些组件可以安装
- `tiup list <component>` : 查看某个组件有哪些版本可以安装

对于上面两种使用方法，可以组合使用两个 flag：

- `--installed` : 本地已经安装了哪些组件，或者已经安装了某个组件的哪些版本
- `--refresh` : 获取服务器上最新的组件列表，以及它们的版本列表

示例一：查看当前已经安装的所有组件

```
tiup list --installed
```

示例二：从服务器获取 TiKV 所有可安装版本组件列表

```
tiup list tikv --refresh
```

## (2) 安装组件 : tiup install

查看组件列表之后，安装也非常简单，利用 `tiup install` 命令即可。相关的命令和参数如下：

```
$ tiup help install
Install a specific version of a component. The component can be specified
by <component> or <component>:<version>. The latest stable version will
be installed if there is no version specified.

You can install multiple components at once, or install multiple versions
of the same component:

tiup install tidb:v3.0.5 tikv pd
tiup install tidb:v3.0.5 tidb:v3.0.8 tikv:v3.0.9

Usage:
  tiup install <component1>[:version] [component2...N] [flags]

Flags:
  -h, --help    help for install

Global Flags:
  --skip-version-check  Skip the strict version check, by default a version must be a valid SemVer string
```

使用方式：

- `tiup install <component>` : 安装指定组件的最新稳定版
- `tiup install <component>:[version]` : 安装指定组件的指定版本

示例一：使用 TiUP 安装最新稳定版的 TiDB

```
tiup install tidb
```

示例二：使用 TiUP 安装 nightly 版本的 TiDB

```
tiup install tidb:nightly
```

示例三：使用 TiUP 安装 v3.0.6 版本的 TiKV

```
tiup install tikv:v3.0.6
```

## (3) 升级组件 : tiup update

在官方组件提供了新版之后，同样可以利用 TiUP 进行升级。相关的命令和参数如下：

```
$ tiup help update
Update some components to the latest version. Use --nightly
to update to the latest nightly version. Use --all to update all components
installed locally. Use <component>:<version> to update to the specified
version. Components will be ignored if the latest version has already been
installed locally, but you can use --force explicitly to overwrite an
existing installation. Use --self which is used to update TiUP to the
latest version. All other flags will be ignored if the flag --self is given.

$ tiup update --all           # Update all components to the latest stable version
$ tiup update --nightly --all   # Update all components to the latest nightly version
$ tiup update playground:v0.0.3 --force # Overwrite an existing local installation
$ tiup update --self            # Update TiUP to the latest version

Usage:
tiup update [component1][:version] [component2..N] [flags]

Flags:
--all      Update all components
--force    Force update a component to the latest version
-h, --help   help for update
--nightly  Update the components to nightly version
--self     Update tiup to the latest version

Global Flags:
--skip-version-check  Skip the strict version check, by default a version must be a valid SemVer string
```

使用方式上和 install 基本相同，不过它支持几个额外的 flag：

- --all : 升级所有组件
- --nightly : 升级至 nightly 版本
- --self : 升级 TiUP 自己至最新版本
- --force : 强制升级至最新版本

示例一：升级所有组件至最新版本

```
tiup update --all
```

示例二：升级所有组件至 nightly 版本

```
tiup update --all --nightly
```

示例三：升级 TiUP 至最新版本

```
tiup update --self
```

## (4) 运行组件：tiup <component>

安装完成之后可以利用 TiUP 启动相应的组件：

```
tiup [flags] <component>[:version] [args...]
```

该命令需要提供一个组件的名字以及可选的版本，若不提供版本，则使用该组件已安装的最新稳定版。

在组件启动之前，TiUP 会先为它创建一个目录，然后将组件放到该目录中运行。组件会将所有数据生成在该目录中，目录的名字就是该组件运行时指定的 tag 名称。如果不指定 tag，则会随机生成一个 tag 名称，并且在实例终止时自动删除工作目录。

如果我们想要多次启动同一个组件并复用之前的工作目录，就可以在启动时用 `--tag` 指定相同的名字。指定 tag 后，在实例终止时就不会自动删除工作目录，方便下次启动时复用。

示例一：运行 v3.0.8 版本的 TiDB

```
tiup tidb:v3.0.8
```

示例二：指定 tag 运行 TiKV

```
tiup --tag=experiment tikv
```

## (5) 查询组件运行状态 : tiup status

通过 `tiup status` 可以查看组件的运行状态。相关的命令和参数如下：

```
~$ tiup help status
List the status of instantiated components

Usage:
  tiup status [flags]

Flags:
  -h, --help    help for status

Global Flags:
  --skip-version-check  Skip the strict version check, by default a version must be a valid SemVer string
```

运行该命令会得到一个实例列表，每行一个实例。列表中包含这些列：

- Name: 实例的 tag 名称
- Component: 实例的组件名称
- PID: 实例运行的进程 ID
- Status: 实例状态，RUNNING 表示正在运行，TERM 表示已经终止
- Created Time: 实例的启动时间
- Directory: 实例的工作目录，可以通过 `--tag` 指定
- Binary: 实例的可执行程序，可以通过 `--binpath` 指定
- Args: 实例的运行参数

## (6) 清理组件实例 : tiup clean

通过 `tiup clean` 可以清理组件实例，并删除工作目录。如果在清理之前实例还在运行，会先 kill 相关进程。相应的命令和参数如下：

```
~$ tiup help clean
Clean the data of instantiated components

Usage:
  tiup clean [flags]

Flags:
  --all    Clean all data of instantiated components
  -h, --help    help for clean

Global Flags:
  --skip-version-check  Skip the strict version check, by default a version must be a valid SemVer string
```

示例一：清理 tag 名称为 experiment 的组件实例

```
tiup clean experiment
```

示例二：清理所有组件实例

```
tiup clean --all
```

## (4) 卸载组件 : tiup uninstall

TiUP 安装的组件是要占用本地磁盘空间的，如果不想保留那么多老版本的组件，可以先查看当前安装了哪些版本的组件，然后再卸载某个组件。TiUP 支持卸载某个组件的所有版本或者特定版本，也支持卸载所有组件。相应的命令和参数如下：

```
~$ tiup help uninstall
If you specify a version number, uninstall the specified version of
the component. You must use --all explicitly if you want to remove all
components or versions which are installed. You can uninstall multiple
components or multiple versions of a component at once. The --self flag
which is used to uninstall tiup.

# Uninstall tiup
tiup uninstall --self

# Uninstall the specific version a component
tiup uninstall tidb:v3.0.10

# Uninstall all version of specific component
tiup uninstall tidb --all

# Uninstall all installed components
tiup uninstall --all

Usage:
  tiup uninstall <component>[:version] [flags]

Flags:
  --all      Remove all components or versions.
  -h, --help   help for uninstall
  --self     Uninstall tiup and clean all local data

Global Flags:
  --skip-version-check  Skip the strict version check, by default a version must be a valid SemVer string
```

### 示例一：卸载 v3.0.8 版本的 TiDB

```
tiup uninstall tidb:v3.0.8
```

### 示例二：卸载所有版本的 TiKV

```
tiup uninstall tikv --all
```

### 示例三：卸载所有已经安装的组件

```
tiup uninstall --all
```

## 1.1.2 用 TiUP 部署本地测试环境

TiDB 集群是由多个组件构成的分布式系统，一个典型的 TiDB 集群至少由 3 个 PD 节点、3 个 TiKV 节点和 2 个 TiDB 节点构成。通过手工来部署这么多组件对于想要体验 TiDB 的用户甚至是 TiDB 的开发人员来说都是非常耗时且头疼的事情。在上一章节我们介绍了 TiUP 的基础用法，在本章中，我们将介绍 TiUP 中的 playground 和 client 组件，并且将通过这两个组件搭建起一套本地的 TiDB 测试环境。

### 1. 通过 playground 组件启动本地集群

playground 是一个集群组件，它会自动利用 TiUP 下载指定的 TiDB/PD/TiKV 版本，并按照用户指定的组件数量快速启动本地集群。

根据上一章节的知识，我们可以通过 list 命令先查看 playground 提供了哪些版本，相关的命令如下：

```
tiup list playground
```

接着我们可以通过 install 命令来安装最新版本的 playground，相关的命令如下：

```
tiup install playground
```

安装完成之后，可以通过 `tiup playground` 来启动一个默认的集群。playground 提供了一键启动集群的方法，大大简化了搭建的集群的时间。相关命令和参数如下：

```
~$ tiup help playground
Bootstrap a TiDB cluster in your local host, the latest release version will be chosen
if you don't specified a version.

Examples:
$ tiup playground nightly                                     # Start a TiDB nightly version local cluster
$ tiup playground v3.0.10 --db 3 --pd 3 --kv 3               # Start a local cluster with 10 nodes
$ tiup playground nightly --monitor                           # Start a local cluster with monitor system
$ tiup playground --pd.config ~/config/pd.toml              # Start a local cluster with specified configuration file,
$ tiup playground --db.binpath /xx/tidb-server                # Start a local cluster with component binary path

Usage:
tiup playground [version] [flags]

Flags:
--db int           TiDB instance number (default 1)
--db.binpath string TiDB instance binary path
--db.config string TiDB instance configuration file
-h, --help          help for tiup
--host string      Playground cluster host (default "127.0.0.1")
--kv int           TiKV instance number (default 1)
--kv.binpath string TiKV instance binary path
--kv.config string TiKV instance configuration file
--monitor          Start prometheus component
--pd int           PD instance number (default 1)
--pd.binpath string PD instance binary path
--pd.config string PD instance configuration file
```

从帮助信息上可以看出，playground 在启动时可以通过参数做很多定制化工作：

- 指定各组件的个数
- 指定各组件的可执行程序和配置文件
- 使用 `--host` 修改默认的 host，譬如修改为机器的对外 IP 地址，服务就可以被其他机器访问
- 使用 `--monitor` 启动 Prometheus 组件，提供集群监控能力

最简单地，你可以通过如下命令快速启动一个集群：

```
tiup playground
```

上述命令实际上做了以下事情：

- 因为没有指定版本，TiUP 会先查找 playground 的最新版本，假设当前最新版为 v0.0.6，则该命令相当于 `tiup playground:v0.0.6`
- 如果 playground 组件的 v0.0.6 版本没有安装，TiUP 会先将其安装，然后再启动运行实例
- 因为 playground 没有指定 TiDB/PD/TiKV 各组件的版本，默认情况下，它会使用各组件的最新 release 版本，假设当前为 v4.0.0，则该命令相当于 `tiup playground:v0.0.6 v4.0.0`
- 因为 playground 也没有指定各组件的个数，默认情况下，它会启动由 1 个 TiDB、1 个 TiKV 和 1 个 PD 构成的最小化集群
- playground 实际上也是调用 TiUP 命令来启动 TiDB/PD/TiKV 组件，譬如调用 `tiup tidb:v4.0.0` 来启动 TiDB 实例，当然，在真正执行时它还会额外指定一些参数
- 在依次启动完各个组件后，playground 会告诉你启动成功，并告诉你一些有用的信息，譬如如何通过 MySQL 客户端连接集群、如何访问 dashboard

在上一章节中我们知道，由于没有使用 `--tag` 选项，TiUP 会为该实例随机生成一个 tag 名称，将该实例的运行数据都放在用 tag 名称命名的文件夹下，并且在实例运行终止时自动删除文件夹。如果想要在多次启动时复用数据，可以通过指定 tag 名称的方法来启动，譬如：

```
tiup --tag=my-cluster playground
```

示例一：使用 TiUP 启动 v3.0.9 版本的集群

```
tiup playground v3.0.9
```

示例二：使用 TiUP 启动 nightly 版本的集群

```
tiup playground nightly
```

示例三：指定 TiKV 组件的个数为 3 个，同时启动 Prometheus 监控

```
tiup playground --kv=3 --monitor
```

示例四：各组件使用本机的对外 IP 地址 `x.x.x.x` 提供服务

```
tiup playground --host x.x.x.x
```

## 2. 通过 playground 搭建测试集群

作为一个分布式系统，最基础的 TiDB 测试集群通常由 2 个 TiDB 组件、3 个 TiKV 组件和 3 个 PD 组件来构成。通过 playground，我们可以快速搭建出上述的一套基础测试集群，相关的命令如下：

```
tiup playground --db=2 --kv=3 --pd=3
```

相比于之前需要手动搭建各个组件和修改各种配置，playground 功能极大的减少了搭建时间和成本。同时，我们还可以通过 `--monitor` 选项来为测试集群增加监控功能，相关的命令如下：

```
tiup playground --db=2 --kv=3 --pd=3 --monitor
```

在集群搭建成功后，playground 会提供使用 MySQL 客户端连接集群的命令信息：

```
CLUSTER START SUCCESSFULLY, Enjoy it ^-^
To connect TiDB: mysql --host 127.0.0.1 --port 4000 -u root
```

除了使用 playground 输出的连接命令外，还可以通过 TiUP 提供的 client 组件来连接到测试集群：

```
tiup client
```

client 组件非常聪明，它会自动探测到当前启动了哪些集群，并展示一个类似 DOS 图形化界面的列表，让你选择连接哪个集群。

当然，你也可以通过指定 tag 名称来连接到特定的集群：

```
tiup client <tag>
```

### 1.1.3 用 TiUP 部署生产集群

上一节介绍了如何使用 TiUP 结合组件 `playground` 快速启动一个本地集群，这样的集群可以用来本地测试，但该集群显然不能用于生产环境。因此我们推出了用于快速部署生产环境的 `cluster` 组件，该组件像 `playground` 部署本地集群一样快速部署生产集群，对比 `playground`，它提供了更强大的集群管理功能，包括对集群的升级，缩容，扩容甚至操作审计等。下面我们一起看看这个组件的用法。

#### 1. 安装 cluster 组件

当你想用某个软件的时候，第一步当然是安装它，由于 TiUP 是一个包管理工具，安装它的组件并不是什么难事，比如安装 `tiup-cluster` 只需要执行：

```
tiup install cluster
```

然后我们可以通过 `--help` 指令看该组件支持的功能：

```
tiup cluster --help
Deploy a TiDB cluster for production

Usage:
  cluster [flags]
  cluster [command]

Available Commands:
  deploy      Deploy a cluster for production
  start       Start a TiDB cluster
  stop        Stop a TiDB cluster
  restart     Restart a TiDB cluster
  scale-in    Scale in a TiDB cluster
  scale-out   Scale out a TiDB cluster
  destroy     Destroy a specified cluster
  upgrade    Upgrade a specified TiDB cluster
  exec        Run shell command on host in the tidb cluster
  display    Display information of a TiDB cluster
  list        List all clusters
  audit       Show audit log of cluster operation
  import      Import an exist TiDB cluster from TiDB-Ansible
  edit-config Edit TiDB cluster config
  reload      Reload a TiDB cluster's config and restart if needed
  help        Help about any command

Flags:
  -h, --help      help for cluster

Use "cluster [command] --help" for more information about a command.
```

TiDB 集群需要用到的操作可以说应有尽有：部署，启动，停止，重启，缩容，扩容，升级...

#### 2. 部署集群

部署集群使用的命令为 `tiup cluster deploy`：

```

tiup cluster deploy --help
Deploy a cluster for production. SSH connection will be used to deploy files, as well as creating system users for running the service.

Usage:
  cluster deploy <cluster-name> <version> <topology.yaml> [flags]

Flags:
  -h, --help           help for deploy
  -i, --identity_file string   The path of the SSH identity file. If specified, public key authentication will be used.
  --user string        The user name to login via SSH. The user must has root (or sudo) privilege. (default "root")
  -y, --yes            Skip confirming the topology

```

该命令需要我们提供集群的名字，集群使用的 TiDB 版本，以及一个集群的拓扑文件，拓扑文件的编写参考[示例](#)。以一个最简单的拓扑为例：

```

---
pd_servers:
- host: 172.16.5.134
  name: pd-134
- host: 172.16.5.139
  name: pd-139
- host: 172.16.5.140
  name: pd-140

tidb_servers:
- host: 172.16.5.134
- host: 172.16.5.139
- host: 172.16.5.140

tikv_servers:
- host: 172.16.5.134
- host: 172.16.5.139
- host: 172.16.5.140

grafana_servers:
- host: 172.16.5.134

monitoring_servers:
- host: 172.16.5.134

```

将该文件保存为 `/tmp/topology.yaml`。假如我们想要使用 TiDB 的 `v3.0.12` 版本，集群名字命名为 `prod-cluster`，则执行：

```
tiup cluster deploy prod-cluster v3.0.12 /tmp/topology.yaml
```

执行过程中会再次确认拓扑结构并提示输入目标机器上的 root 密码：

```

Please confirm your topology:
TiDB Cluster: prod-cluster
TiDB Version: v3.0.12
Type      Host        Ports      Directories
----      ---        ----      -----
pd        172.16.5.134 2379/2380  deploy/pd-2379,data/pd-2379
pd        172.16.5.139 2379/2380  deploy/pd-2379,data/pd-2379
pd        172.16.5.140 2379/2380  deploy/pd-2379,data/pd-2379
tikv     172.16.5.134 20160/20180 deploy/tikv-20160,data/tikv-20160
tikv     172.16.5.139 20160/20180 deploy/tikv-20160,data/tikv-20160
tikv     172.16.5.140 20160/20180 deploy/tikv-20160,data/tikv-20160
tidb     172.16.5.134 4000/10080  deploy/tidb-4000
tidb     172.16.5.139 4000/10080  deploy/tidb-4000
tidb     172.16.5.140 4000/10080  deploy/tidb-4000
prometheus 172.16.5.134 9090    deploy/prometheus-9090,data/prometheus-9090
grafana   172.16.5.134 3000    deploy/grafana-3000

Attention:
 1. If the topology is not what you expected, check your yaml file.
 2. Please confirm there is no port/directory conflicts in same host.

Do you want to continue? [y/N]:

```

输入密码后 `tiup-cluster` 便会下载需要的组件并部署到对应的机器上，当看到以下提示时说明部署成功：

```
Deployed cluster `prod-cluster` successfully
```

### 3. 查看集群列表

集群一旦部署之后我们就能够通过 `tiup cluster list` 在集群列表中看到它：

```

[root@localhost ~]# tiup cluster list
Starting /root/.tiup/components/cluster/v0.4.5/cluster list
Name      User  Version  Path          PrivateKey
----      ---   -----  ---          -----
prod-cluster  tidb  v3.0.12  /root/.tiup/storage/cluster/clusters/prod-cluster  /root/.tiup/storage/cluster/clusters/prod-cluster/ssh/id_rsa

```

### 4. 启动集群

上一步部署成功后，我们可以执行命令将该集群启动起来，如果忘记了已经部署的集群的名字，可以使用 `tiup cluster list` 查看，启动集群的命令：

```
tiup cluster start prod-cluster
```

### 5. 查看集群状态

我们经常想知道集群中每个组件的运行状态，如果挨个机器上去看的话显然很低效，这个时候就轮到 `tiup cluster display` 登场了，它的用法很简单：

```
[root@localhost ~]# tiup cluster display prod-cluster
Starting /root/.tiup/components/cluster/v0.4.5/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v3.0.12
ID          Role      Host        Ports      Status     Data Dir      Deploy Dir
--          ----      ----        -----      -----     -----      -----
172.16.5.134:3000  grafana   172.16.5.134  3000      Up         -           deploy/grafana-3000
172.16.5.134:2379  pd        172.16.5.134  2379/2380  Healthy|L  data/pd-2379  deploy/pd-2379
172.16.5.139:2379  pd        172.16.5.139  2379/2380  Healthy    data/pd-2379  deploy/pd-2379
172.16.5.140:2379  pd        172.16.5.140  2379/2380  Healthy    data/pd-2379  deploy/pd-2379
172.16.5.134:9090  prometheus 172.16.5.134  9090      Up         data/prometheus-9090  deploy/prometheus-9090
172.16.5.134:4000  tidb      172.16.5.134  4000/10080  Up         -           deploy/tidb-4000
172.16.5.139:4000  tidb      172.16.5.139  4000/10080  Up         -           deploy/tidb-4000
172.16.5.140:4000  tidb      172.16.5.140  4000/10080  Up         -           deploy/tidb-4000
172.16.5.134:20160  tikv      172.16.5.134  20160/20180 Up         data/tikv-20160  deploy/tikv-20160
172.16.5.139:20160  tikv      172.16.5.139  20160/20180 Up         data/tikv-20160  deploy/tikv-20160
172.16.5.140:20160  tikv      172.16.5.140  20160/20180 Up         data/tikv-20160  deploy/tikv-20160
```

对于普通的组件，Status 列会显示 "Up" 或者 "Down" 表示该服务是否正常，对于 PD，Status 会显示 Healthy 或者 "Down"，同时可能会带有 |L 表示该 PD 是 Leader。

## 6. 缩容

有时候业务量降低了，集群再占有原来的资源显得有些浪费，我们会想安全地释放某些节点，减小集群规模，于是需要缩容：

```
[root@localhost ~]# tiup cluster scale-in --help
Scale in a TiDB cluster

Usage:
  cluster scale-in <cluster-name> [flags]

Flags:
  -h, --help            help for scale-in
  -N, --node strings    Specify the nodes
  --transfer-timeout int Timeout in seconds when transferring PD and TiKV store leaders (default 300)
  -y, --yes             Skip the confirmation of destroying

Global Flags:
  --ssh-timeout int    Timeout in seconds to connect host via SSH, ignored for operations that don't need an SSH connection. (default 5)
```

它需要指定至少两个参数，一个是集群名字，另一个是节点 ID，节点 ID 可以参考上一节使用 `tiup cluster display` 命令获取。比如我想要将 `172.16.5.140` 上的 TiKV 干掉，于是可以执行：

```
tiup cluster scale-in prod-cluster -N 172.16.5.140:20160
```

通过 `tiup cluster display` 可以看到该 TiKV 已经被标记为 Offline：

TiDB Cluster: prod-cluster						
ID	Role	Host	Ports	Status	Data Dir	Deploy Dir
--	---	---	-----	-----	-----	-----
172.16.5.134:3000	grafana	172.16.5.134	3000	Up	-	deploy/grafana-3000
172.16.5.134:2379	pd	172.16.5.134	2379/2380	Healthy L	data/pd-2379	deploy/pd-2379
172.16.5.139:2379	pd	172.16.5.139	2379/2380	Healthy	data/pd-2379	deploy/pd-2379
172.16.5.140:2379	pd	172.16.5.140	2379/2380	Healthy	data/pd-2379	deploy/pd-2379
172.16.5.134:9090	prometheus	172.16.5.134	9090	Up	data/prometheus-9090	deploy/prometheus-9090
172.16.5.134:4000	tidb	172.16.5.134	4000/10080	Up	-	deploy/tidb-4000
172.16.5.139:4000	tidb	172.16.5.139	4000/10080	Up	-	deploy/tidb-4000
172.16.5.140:4000	tidb	172.16.5.140	4000/10080	Up	-	deploy/tidb-4000
172.16.5.134:20160	tikv	172.16.5.134	20160/20180	Up	data/tikv-20160	deploy/tikv-20160
172.16.5.139:20160	tikv	172.16.5.139	20160/20180	Up	data/tikv-20160	deploy/tikv-20160
172.16.5.140:20160	tikv	172.16.5.140	20160/20180	Offline	data/tikv-20160	deploy/tikv-20160

待 PD 将其数据调度到其他 TiKV 后，该节点会被自动删除。

## 7. 扩容

与缩容相反，随着业务的增长，原来的集群资源不够用时，我们需要向集群中添加资源，`scale-out` 用法如下：

```
[root@localhost ~]# tiup cluster scale-out --help
Scale out a TiDB cluster

Usage:
  cluster scale-out <cluster-name> <topology.yaml> [flags]

Flags:
  -h, --help           help for scale-out
  -i, --identity_file string   The path of the SSH identity file. If specified, public key authentication will be used.
  --user string         The user name to login via SSH. The user must has root (or sudo) privilege. (default "root")
  -y, --yes            Skip confirming the topology

Global Flags:
  --ssh-timeout int   Timeout in seconds to connect host via SSH, ignored for operations that don't need an SSH connection. (default 5)
```

该命令需要提供一个已经存在的集群名字，以及一个增量的拓扑文件，比如，我要扩容一台 TiKV，那就创建一个叫 `scale.yaml` 的文件：

```
...
tikv_servers:
  - host: 172.16.5.140
```

然后执行：

```
tiup cluster scale-out prod-cluster /tmp/scale.yaml
```

然后再 `display` 就可以看到新的节点了！

## 8. 升级

软件升级是软件生命周期中常见的操作，对于一套集群软件来说，升级的同时保证服务可用是一件最基本也是最有挑战的事情，它涉及到繁杂的运维操作，好在 `tiup cluster upgrade` 简化了这个操作，从此升级 TiDB 集群只需要一行简单的命令：

```
tiup cluster upgrade prod-cluster v4.0.0-rc
```

这样就能把 `prod-cluster` 这个版本升级到 `v4.0.0-rc` 了。

## 9. 更新配置

有时候我们会想要动态更新组件的配置，`tiup-cluster` 为每个集群保存了一份当前的配置，如果想要编辑这份配置，则执行 `tiup cluster edit-config <cluster-name>`，例如：

```
tiup cluster edit-config prod-cluster
```

然后 `tiup-cluster` 会使用 `vi` 打开配置文件供编辑，编辑完之后保存即可。此时的配置并没有应用到集群，如果想要让它生效，还需要执行：

```
tiup cluster reload prod-cluster
```

该操作会将配置发送到目标机器，重启集群，使配置生效。如果只修改了某个组件的配置（比如 `TiDB`），可以只重启该组件：

```
tiup cluster reload prod-cluster -R tidb
```

## 10. 其他

除了上面介绍的以外，`tiup-cluster` 还有很多功能等待探索，TiUP 自身尽可能提供了帮助信息，可以在任何命令后加上 `--help` 来查看具体的用法，比如我们知道有一个子命令叫 `import` 但是不知道它是干什么的，也不知道它怎么用，于是：

```
[root@localhost ~]# tiup cluster import -h
Import an exist TiDB cluster from TiDB-Ansible

Usage:
  cluster import [flags]

Flags:
  -d, --dir string      The path to TiDB-Ansible directory
  -h, --help             help for import
  --inventory string    The name of inventory file (default "inventory.ini")
  -r, --rename NAME     Rename the imported cluster to NAME

Global Flags:
  --ssh-timeout int    Timeout in seconds to connect host via SSH, ignored for operations that don't need an SSH connection. (default 5)
```

这样就很容易看出这个命令是用来导入一个之前的 `TiDB-Ansible` 集群的，它的基本用法应该是 `tiup cluster import --dir=<ansible-dir>`

有了这个技巧，相信你可以很快玩转 TiUP 世界。

## 1.1.4 TiUP cluster 部署生产环境集群

### 操作系统版本要求

- 仅支持 CentOS 7.3 及以上 Linux 操作系统。

### 生产环境服务器数量和配置参数

组件	CPU	内存	硬盘类型	磁盘大小	网络	机器数量
PD	>= 4 Core	>= 8 GB	SSD	>= 300 GB	>= 1 块万兆网卡	3
TiDB	>= 16 Core	>= 32 GB	SAS/SSD	>= 300 GB	>= 1 块万兆网卡	2
TiKV	>= 16 Core	>= 32 GB	SSD	<= 2 TB	>= 1 块万兆网卡	3
Prometheus	>= 8 Core	>= 16 GB	SAS/SSD	>= 300 GB	>= 1 块千兆网卡或者万兆网卡	1

### 注意事项

- TiDB
  - 机器数量  $\geq 2$  台，若每台机器资源较丰富，则建议部署多个 tidb-server
  - 磁盘建议 SAS/SSD，建议容量  $\geq 300$  GB
- TiKV
  - 机器数量  $\geq 3$  台，若每台机器有多块磁盘，建议部署多个 tikv-server
  - 部署推荐配置 label，配置后系统才会将根据 label 将数据分布在不同的机房、机架、机器防止有单个机房或者机架或者机器宕机时，系统才具备容灾能力
  - 磁盘建议采用 SSD，其中建议 PCI-E SSD 容量  $\leq 2$  TB，普通 SSD 容量  $\leq 1.5$  TB
- PD
  - 机器数量  $\geq 3$  台
  - 磁盘建议采用 SSD，容量建议  $\geq 300$  GB
- Prometheus
  - 机器数量 1 台，若资源较紧张，可与其他组件混合部署
- Other
  - 若资源较紧张时 TiDB、PD、Prometheus 可混合部署在同一台服务器上
  - 若对性能、可靠性有更高的要求，建议按组件分别部署
  - 生产环境强烈推荐使用更高的配置

### 生产环境网络要求

TiDB 正常运行需要网络环境提供如下的网络端口配置，管理员可根据实际 TiDB 组件部署的需要进行调整：

组件	默认端口	说明
TiDB	4000	应用及 DBA 工具访问通信端口
TiDB	10080	TiDB 状态信息上报通信端口
TiKV	20160	TiKV 通信端口
PD	2379	提供 TiDB 和 PD 通信端口
PD	2380	PD 集群节点间通信端口
Pump	8250	Pump 通信端口
Drainer	8249	Drainer 通信端口
Prometheus	9090	Prometheus 服务通信端口
Pushgateway	9091	TiDB , TiKV , PD 监控聚合和上报端口
Node_exporter	9100	TiDB 集群每个节点的系统信息上报通信端口
Blackbox_exporter	9115	Blackbox_exporter 通信端口，用于 TiDB 集群端口监控
Grafana	3000	Web 监控服务对外服务和客户端(浏览器)访问端口
Grafana	8686	grafana_collector 通信端口，用于将 Dashboard 导出为 PDF 格式
Kafka_exporter	9308	Kafka_exporter 通信端口，用于监控 binlog kafka 集群

## topology 配置

```

---
pd_servers:
  - ip: 10.9.1.1
  - ip: 10.9.1.2
  - ip: 10.9.1.3

tidb_servers:
  - ip: 10.9.1.2
  - ip: 10.9.1.3

tikv_servers:
  - ip: 10.9.1.4
    ## The value of label can be customized, for example: 'zone=z1,rack=r1,host=h1' or 'a=a1,b=b1,c=c1', etc
    ## Can only set in tikv_servers
    # label: host=h1
  - ip: 10.9.1.5
  - ip: 10.9.1.6

monitoring_server:
  - ip: 10.9.1.7

grafana_server:
  - ip: 10.9.1.7

```

label 可以自定义，例如：`zone=z1,rack=r1,host=h1` 或 `a=a1,b=b1,c=c1`。label 只能在 tikv\_servers 上设置。

## 部署方法

参照上一节的部署和运维管理操作，即可使用 TiOps 在生产环境部署一套可用的 TiDB 集群。

## 1.2 TiDB on Kubernetes

使用传统的自动化工具带来了很高的部署和运维成本。TiDB 的分层架构对于分布式系统是比较常见的，各个组件都可以根据业务需求独立水平伸缩，并且 TiKV 和 TiDB 都可以独立使用。比如，在 TiKV 之上可以构建兼容 Redis 协议的 KV 数据库，而 TiDB 也可以对接 LevelDB 这样的 KV 存储引擎。

但是，这种多组件的分布式系统增加了手工部署和运维的成本。一些传统的自动化部署和运维工具如 Puppet/Chef/SaltStack/Ansible，由于缺乏全局状态管理，不能及时对各种异常情况做自动故障转移，并且很难发挥分布式系统的弹性伸缩能力。其中有些还需要写大量的 DSL 甚至与 Shell 脚本一起混合使用，可移植性较差，维护成本比较高。

在云时代，容器成为应用分发部署的基本单位，而谷歌基于内部使用数十年的容器编排系统 Borg 经验推出的开源容器编排系统 Kubernetes 成为当前容器编排技术事实上的标准。如今各大云厂商都开始提供托管的 Kubernetes 集群，部署在 Kubernetes 平台的应用可以不用绑定在特定云平台，轻松实现在各种云平台之间的迁移，其容器化打包和发布方式也解决了对操作系统环境的依赖。

TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或私有部署的 Kubernetes 集群上。

本章我们将详细介绍 TiDB-Operator 的特性、原理、部署与升级，并指导读者如何在 Kubernetes 集群上展开以下操作：

- 访问集群与查看监控
- 使用 BR 备份，恢复 TiDB 集群
- 使用 Lighting 导入工具
- 使用 TiDB 工具

## 1.2.1 TiDB Operator 简介及原理

——化繁为简的艺术

### 1.2.1.1 背景

小时候，想象着 2020 年机器人早就普及了，机器人管家叫我起床，机器人厨师端上早餐，机器人司机开车送我去上班，路上还有机器人交警，到了公司是机器人姐姐接待，下班去看的是机器人牙医而且一点也不疼。

“快醒醒，客户爸爸问你 TiDB 集群怎么还没搞好？”

“天啊，为什么都 2020 年了还要人手动去维护分布式集群？为什么这群人要把 TiDB 设计成这么多组件？为什么 TiDB 集群半夜挂了不派个机器人去修复？”

“兄 dei，K8s 了解一下？”

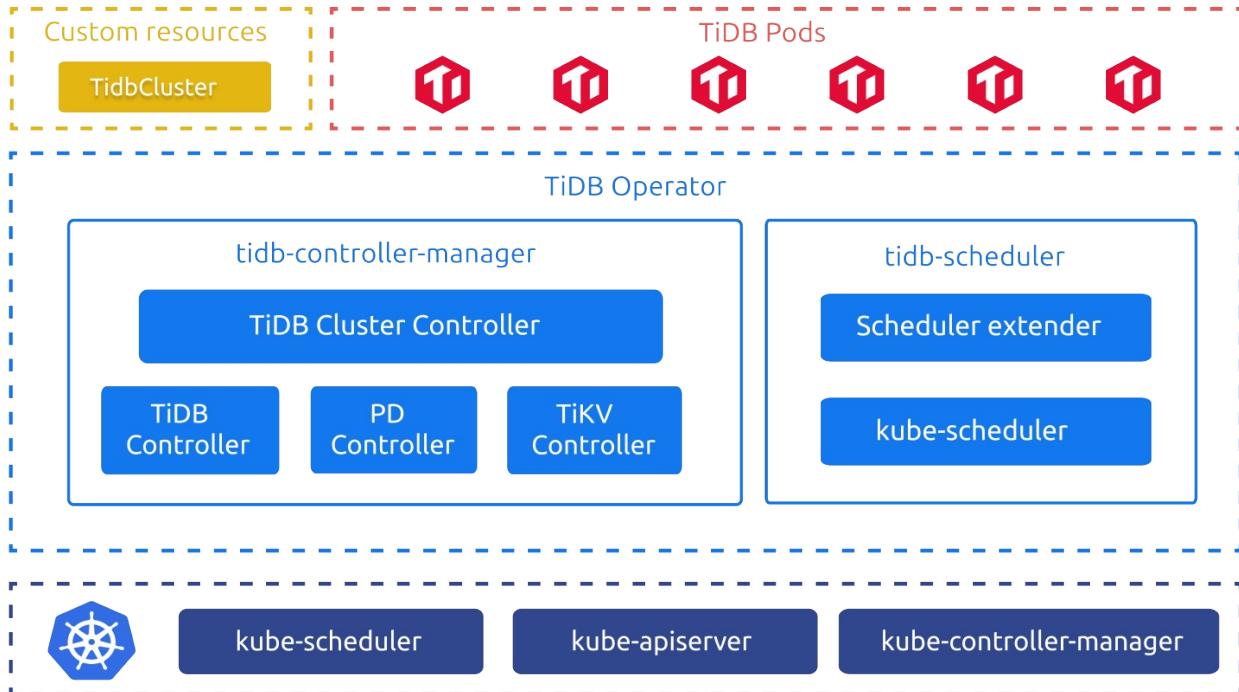
### 1.2.1.2 简介

TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或私有部署的 Kubernetes 集群上。

——来自 PingCAP 官方定义

TiDB Operator 像“牧羊人”一样，持续的监督并管理着 TiDB 各组件“羊群”以恰当的状态运行在主机集群“牧场”上。现在运维人员只要告诉 Operator “What to do”，而由 Operator 来决定 “How to do”。在最新版本 TiDB Operator 甚至可以根据实际情况来决定 “What to do”，比如：auto-scaler。真正实现了自动化运维，减轻运维人员维护压力，提高服务能力。

### 1.2.1.3 TiDB Operator 架构



### 1.2.1.4 TiDB Operator 组件

- TiDB Cluster 定义：CRD (`CustomResourceDefinition`) 定义了 `TidbCluster` 等自定义资源，使得 Kubernetes 世界认识 TiDB Cluster 并让其与 `Deployment`、`StatefulSet` 一同享受 Kubernetes 的头等公民待遇。目前 TiDB Operator v1.1.0 版本包含的 CRD 有：`TidbCluster`、`Backup`、`Restore`、`BackupSchedule`、`TidbMonitor`、`TidbInitializer` 以及 `TidbClusterAutoScaler`。

- 控制器：`tidb-controller-manager` 包含了一组自定义控制器，控制器通过循环不断比对被控制对象的期望状态与实际状态，并通过自定义的逻辑驱动被控制对象达到期望状态。
- 调度器：`tidb-scheduler` 是一个 Kubernetes 调度器扩展，它为 Kubernetes 调度器注入 TiDB 集群特有的调度逻辑，比如：为保证高可用，任一 Node 不能调度超过 TiDB 集群半数以上的 TiKV 实例。

### 1.2.1.5 自定义资源

- TiDB Cluster 资源：`CR ( CustomResource )` 声明了 TiDB Cluster 自定义资源对象，它声明了 `TidbCluster` 对象的期望状态，并被控制器逻辑不断进行处理，同时将实际运行状态记录下来。

### 1.2.1.6 Kubernetes 控制平面

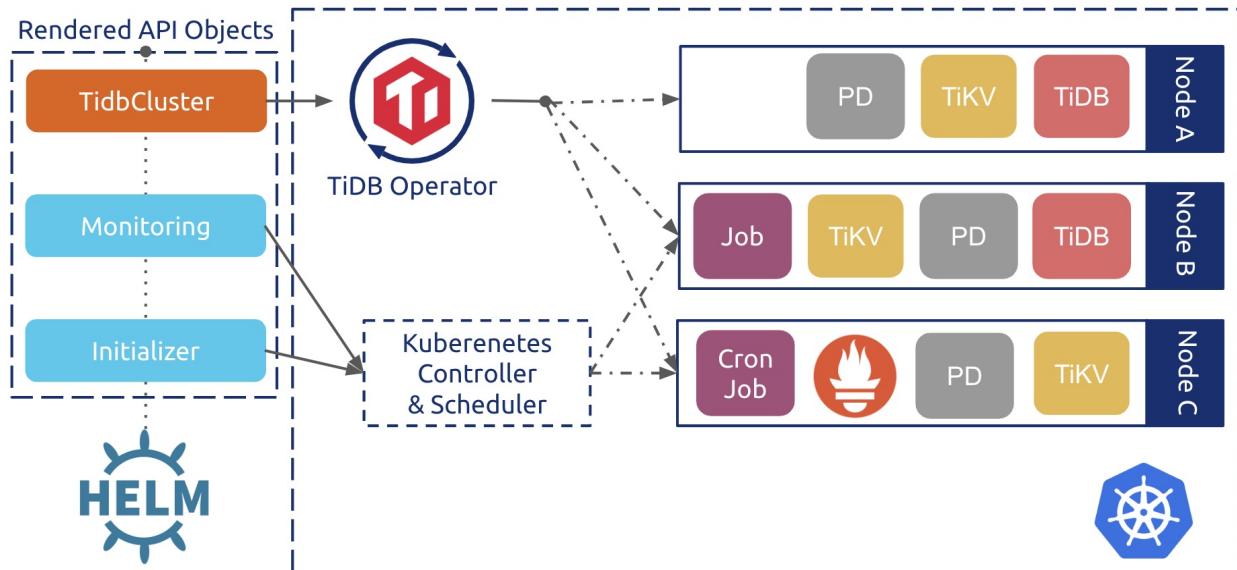
- `kube-apiserver`：Kubernetes 控制平面的前端，所有组件通过 API Server 获取或更新对象信息。
- `kube-controller-manager`：`TidbCluster` 等 CR 封装了 `StatefulSet`、`Deployment`、`CronJob` 等原生对象，所以依然需要 K8s 原生控制器来进行控制逻辑。
- `kube-scheduler`：调度 TiDB Cluster 的 Pod，`filtering` 阶段，`kube-scheduler` 筛选出的节点会再经过 `tidb-scheduler` 筛选一次，然后 `kube-scheduler` 再进行 `scoring` 选择最合适的节点进行 Pod 调度。

举个栗子（脑洞）：

小明买了辆家用汽车，去修车厂改造成变形金刚。

这辆家用汽车类似 `Deployment`、`StatefulSet` 这类原厂的标准化组件。改造成独一无二的变形金刚需要设计图纸（`CRD`）。同时原来车上的零件需要实现新的功能（自定义控制器），轮子不仅能转还可以当关节，后备箱不仅能装东西还可以变成脚。修车厂（K8s 控制平面）根据设计图纸和控制逻辑真的造出了一个变形金刚（`CR`）。这个变形金刚可以根据环境的不同而改变形态（调度器），甚至还可以在战斗损坏后进行修复（调和）。

### 1.2.1.7 原理浅析



TiDB Operator 中使用 Helm Chart 封装了 TiDB 集群定义。整体的控制流程如下：

- 用户通过 Helm 创建 `TidbCluster` 对象和相应的一系列 Kubernetes 原生对象，比如执行定时备份的 `cronJob`；
- TiDB Operator 会通过 Kubernetes API Server watch `TidbCluster` 以及其他相关对象，基于集群的实际状态不断调整 PD、TiKV、TiDB 的 `StatefulSet` 和 `Service` 对象；
- Kubernetes 的原生控制器根据 `StatefulSet`、`Deployment`、`CronJob` 等对象创建更新或删除对应的 `Pod`；
- PD、TiKV、TiDB 的 `Pod` 声明中会指定使用 `tidb-scheduler` 调度器，`tidb-scheduler` 会在调度对应 `Pod` 时应用 TiDB 的特定调度逻辑。

基于上述的声明式控制流程，TiDB Operator 能够自动进行集群节点健康检查和故障恢复。部署、升级、扩缩容等操作也可以通过修改 `TidbCluster` 对象声明“一键”完成。

## 1.2.2 TiDB-Operator 部署本地测试环境

### 1.2.2.1 背景介绍

本小节介绍如何在个人电脑（Linux 或 MacOS）上采用 kind 方式在 Kubernetes 上部署 TiDB Operator 和 TiDB 集群。部署包含三个关键环节：

1. 通过 kind 部署 K8s 集群
2. 在 K8s 集群上部署 TiDB Operator
3. 在 K8s 集群中部署 TiDB 集群

在部署前，请确认资源满足以下要求：

- 内存 4GB+、CPU 至少 2 cores
- Docker 17.03+
- Go 1.10+
- net.ipv4.ip\_forward 设置为 1

### 1.2.2.2 通过 kind 部署 K8s 集群

1. 下载自动化部署程序

```
# cd /root & git clone --depth=1 https://github.com/pingcap/tidb-operator && cd tidb-operator
```

1. 通过程序创建集群

```
# cd /root/tidb-operator && hack/kind-cluster-build.sh
```

执行成功后会有如下关键提示信息：

```
##### success create cluster:[kind] #####
To start using your cluster, run:
  kubectl config use-context kind-kind
```

1. 将 K8s 集群相关命令路径加入 PATH 路径

```
# export PATH=$PATH:/root/tidb-operator/output/bin/
```

1. 验证 K8s 环境是否符合要求

```
# kubectl cluster-info
Kubernetes master is running at https://127.0.0.1:32771
KubeDNS is running at https://127.0.0.1:32771/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

输出以上信息，则说明 K8s 服务符合要求。

```
# helm version
Client: &version.Version{SemVer:"v2.9.1", GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.9.1", GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710", GitTreeState:"clean"}
```

输出以上信息，则说明 Helm 客户端与服务端都符合要求。

### 1.2.2.3 在 K8s 集群上部署 TiDB Operator

#### 1. 通过 helm 安装 TiDB Operator

创建 TiDB CRD

```
# kubectl apply -f https://raw.githubusercontent.com/pingcap/tidb-operator/master/manifests/crd.yaml && kubectl get crd tidbclusters.pingcap.com
customresourcedefinition.apirextensions.k8s.io/tidbclusters.pingcap.com unchanged
customresourcedefinition.apirextensions.k8s.io/backups.pingcap.com unchanged
customresourcedefinition.apirextensions.k8s.io/restores.pingcap.com unchanged
customresourcedefinition.apirextensions.k8s.io/backupschedules.pingcap.com unchanged
customresourcedefinition.apirextensions.k8s.io/tidbmonitors.pingcap.com unchanged
customresourcedefinition.apirextensions.k8s.io/tidbinitializers.pingcap.com unchanged
customresourcedefinition.apirextensions.k8s.io/tidbclusterautoscalers.pingcap.com unchanged
NAME                CREATED AT
tidbclusters.pingcap.com  2020-03-06T13:38:32Z
```

下载 TiDB Operator 的 Helm chart 文件：

```
# mkdir -p /root/chart/
从 https://github.com/pingcap/tidb-operator/releases 下载 tidb-operator-chart-v1.0.6.tgz 文件放到 /root/chart/ 路径下
# cd /root/chart/ && tar xvf tidb-operator-chart-v1.0.6.tgz
```

将 /root/tidb-operator/charts/tidb-operator/values.yaml 文件内的 scheduler.kubeSchedulerImageName 值修改为 registry.cn-hangzhou.aliyuncs.com/google\_containers/kube-scheduler 以加快镜像拉取速度。

#### 1. 安装 TiDB Operator

```
# helm install --namespace=tidb-admin --name=tidb-operator /root/tidb-operator/charts/tidb-operator -f /root/tidb-operator/charts/tidb-operator/values.yaml
NAME: tidb-operator
LAST DEPLOYED: Fri Mar 6 14:24:09 2020
NAMESPACE: tidb-admin
STATUS: DEPLOYED
...
```

#### 1. 验证 Operator 运行状态

```
# kubectl get pods -n tidb-admin
NAME                               READY   STATUS    RESTARTS   AGE
tidb-controller-manager-85d8d498bf-2n8km  1/1     Running   0          19s
tidb-scheduler-7c67d6c77b-qd54r        2/2     Running   0          19s
```

以上信息显示 Operator 运行正常。

### 1.2.2.4 在 K8s 集群中部署 TiDB 集群

#### 1. 下载 TiDB Cluster 的 helm chart 文件

```
# mkdir -p /root/chart/
从 https://github.com/pingcap/tidb-operator/releases 下载 tidb-cluster-chart-v1.0.6.tgz 文件放到 /root/chart/ 路径下
```

#### 1. 安装 TiDB Cluster

```
# cd /root/chart/ && tar xvf tidb-cluster-chart-v1.0.6.tgz
# helm install --namespace dba-test --name=test /root/tidb-operator/charts/tidb-cluster -f /root/tidb-operator/charts/tidb-cluster/values.yaml
NAME: test
LAST DEPLOYED: Fri Mar 6 14:50:25 2020
NAMESPACE: dba-test
STATUS: DEPLOYED
```

以上信息显示 TiDB Cluster 部署正常。

### 1. 观察 TiDB 的 POD 状态

```
# kubectl get pods -n dba-test
NAME                      READY   STATUS    RESTARTS   AGE
test-discovery-668b48577c-lqqbz  1/1    Running   0          7m37s
test-monitor-5b586d8cb-227qx   3/3    Running   0          7m37s
test-pd-0                   1/1    Running   0          7m37s
test-pd-1                   1/1    Running   0          7m37s
test-pd-2                   1/1    Running   1          7m37s
test-tidb-0                 2/2    Running   0          6m18s
test-tidb-1                 2/2    Running   0          6m18s
test-tikv-0                  1/1    Running   0          6m58s
test-tikv-1                  1/1    Running   0          6m58s
test-tikv-2                  1/1    Running   0          6m58s
```

以上信息显示 TiDB Cluster 所有 Pod 全部运行正常。

### 1. 访问 TiDB 集群

```
# nohup kubectl port-forward svc/test-tidb 4000:4000 --namespace=dba-test &
# yum install -y mysql
# mysql -h 127.0.0.1 -uroot -P4000
mysql -h 127.0.0.1 -P 4000 -uroot
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.7.25-TiDB-v3.0.5 MySQL Community Server (Apache License 2.0)
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
MySQL [(none)]>
```

显示以上输出显示 TiDB 集群部署成功。

## 1.2.3 用 TiDB-Operator 部署生产环境集群

本章节介绍如何使用 TiDB-Operator 部署生产环境 TiDB 集群，包括在私有云和公有云上部署。

在公有云上部署 TiDB 集群，包含以下四部分内容：

- [在 AWS EKS 上部署 TiDB 集群](#)
- [在 GCP GKE 上部署 TiDB 集群](#)
- [在阿里云上部署 TiDB 集群](#)
- [在京东云上部署 TiDB 集群](#)

在私有云上部署 TiDB 集群，包含集群的环境和资源需求，如何为 TiDB 集群配置 PV，部署 TiDB Operator 和 TiDB 集群，如何进行节点维护以及删除 TiDB 集群的步骤。

### 1.2.3.1 在公有云上部署 TiDB 集群

本章节介绍如何在公有云上部署 TiDB 集群，包含以下四部分内容：

- [在 AWS EKS 上部署 TiDB 集群](#)
- [在 GCP GKE 上部署 TiDB 集群](#)
- [在阿里云上部署 TiDB 集群](#)
- [在京东云上部署 TiDB 集群](#)

### 1.2.3.1.1 在 AWS EKS 上部署 TiDB 集群

本节介绍如何使用个人电脑（Linux 或 macOS 系统）在 AWS EKS (Elastic Kubernetes Service) 上部署 TiDB 集群。

## 1. 环境配置准备

部署前，请确认已安装以下软件并完成配置：

- `awscli` >= 1.16.73，控制 AWS 资源

要与 AWS 交互，必须配置 `awscli`。最快的方式是使用 `aws configure` 命令：

```
aws configure
```

替换下面的 AWS Access Key ID 和 AWS Secret Access Key：

```
AWS Access Key ID [None]: IOSF0DNN7EXAMPLE
AWS Secret Access Key [None]: wJaXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
Default region name [None]: us-west-2
Default output format [None]: json
```

注意：

Access key 必须至少具有以下权限：创建 VPC、创建 EBS、创建 EC2 和创建 Role。

- `terraform` >= 0.12
- `kubectl` >= 1.11
- `helm` >= 2.11.0 且 < 3.0.0
- `jq`
- `aws-iam-authenticator`，AWS 权限鉴定工具，确保安装在 `PATH` 路径下。

最简单的安装方法是下载编译好的二进制文件 `aws-iam-authenticator`，如下所示。

Linux 用户下载二进制文件：

```
curl -o aws-iam-authenticator https://amazon-eks.s3-us-west-2.amazonaws.com/1.12.7/2019-03-27/bin/linux/amd64/
aws-iam-authenticator
```

macOS 用户下载二进制文件：

```
curl -o aws-iam-authenticator https://amazon-eks.s3-us-west-2.amazonaws.com/1.12.7/2019-03-27/bin/darwin/amd64/
aws-iam-authenticator
```

二进制文件下载完成后，执行以下操作：

```
chmod +x ./aws-iam-authenticator && \
sudo mv ./aws-iam-authenticator /usr/local/bin/aws-iam-authenticator
```

## 2. 部署集群

默认部署会创建一个新的 VPC、一个 t2.micro 实例作为堡垒机，并包含以下 ec2 实例作为工作节点的 EKS 集群：

- 3 台 m5.xlarge 实例，部署 PD

- 3 台 c5d.4xlarge 实例，部署 TiKV
- 2 台 c5.4xlarge 实例，部署 TiDB
- 1 台 c5.2xlarge 实例，部署监控组件

使用如下命令部署集群。

从 Github 克隆代码并进入指定路径：

```
git clone --depth=1 https://github.com/pingcap/tidb-operator && \
cd tidb-operator/deploy/aws
```

使用 `terraform` 命令初始化并部署集群：

```
terraform init
```

```
terraform apply
```

**注意：**

`terraform apply` 过程中必须输入 "yes" 才能继续。

整个过程可能至少需要 10 分钟。`terraform apply` 执行成功后，控制台会输出类似如下的信息：

```
Apply complete! Resources: 67 added, 0 changed, 0 destroyed.

Outputs:

bastion_ip = [
  "34.219.204.217",
]
default-cluster_monitor-dns = a82db513ba84511e9af170283460e413-1838961480.us-west-2.elb.amazonaws.com
default-cluster_tidb-dns = a82df6d13a84511e9af170283460e413-d3ce3b9335901d8c.elb.us-west-2.amazonaws.com
eks_endpoint = https://9A9A5ABB8303DDD35C0C2835A1801723.y14.us-west-2.eks.amazonaws.com
eks_version = 1.12
kubeconfig_filename = credentials/kubeconfig_my-cluster
region = us-west-21
```

可以通过 `terraform output` 命令再次获取上面的输出信息。

**注意：**

1.14 版本以前的 EKS 不支持自动开启 NLB 跨可用区负载均衡，因此默认配置下会出现各台 TiDB 实例压力不均衡额状况。生产环境下，强烈建议参考 [AWS 官方文档](#)手动开启 NLB 的跨可用区负载均衡。

### 3. 访问数据库

`terraform apply` 完成后，可先通过 `ssh` 远程连接到堡垒机，再通过 MySQL client 来访问 TiDB 集群。

所需命令如下（用上面的输出信息替换 `<...>` 部分内容）：

```
ssh -i credentials/<eks_name>.pem centos@<bastion_ip>
```

```
mysql -h <default-cluster_tidb-dns> -P 4000 -u root
```

`eks_name` 默认为 `my-cluster`。如果 DNS 名字无法解析，请耐心等待几分钟。

还可以通过 `kubectl` 和 `helm` 命令使用 `kubeconfig` 文件 `credentials/kubeconfig_<eks_name>` 和 EKS 集群交互，主要有两种方式，如下所示。

- 指定 --kubeconfig 参数：

```
kubectl --kubeconfig credentials/kubeconfig_<eks_name> get po -n <default_cluster_name>
```

```
helm --kubeconfig credentials/kubeconfig_<eks_name> ls
```

- 或者，设置 KUBECONFIG 环境变量：

```
export KUBECONFIG=$PWD/credentials/kubeconfig_<eks_name>
```

```
kubectl get po -n <default_cluster_name>
```

```
helm ls
```

## 4. Grafana 监控

可以通过浏览器访问 `<monitor-dns>:3000` 地址查看 Grafana 监控指标。

Grafana 默认登录信息：

- 用户名：admin
- 密码：admin

## 5. 升级 TiDB 集群

要升级 TiDB 集群，可在 `terraform.tfvars` 文件中设置 `default_cluster_version` 变量到更高版本，然后运行 `terraform apply`。

例如，要升级 TiDB 集群到 4.0.1，则修改 `default_cluster_version` 为 `v4.0.1`：

```
default_cluster_version= "v4.0.1"
```

注意：

升级过程会持续一段时间，可以通过 `kubectl --kubeconfig credentials/kubeconfig_<eks_name> get po -n <default_cluster_name> --watch` 命令持续观察升级进度。

## 6. 扩容 TiDB 集群

若要扩容 TiDB 集群，可在 `terraform.tfvars` 文件中设置 `default_cluster_tikv_count` 或者 `default_cluster_tidb_count` 变量，然后运行 `terraform apply`。

例如，可以将 `default_cluster_tidb_count` 从 2 改为 4 以扩容 TiDB：

```
default_cluster_tidb_count = 4
```

注意：

- 由于扩容过程中无法确定会缩掉哪个节点，目前还不支持 TiDB 集群的扩容。
- 扩容过程会持续几分钟，可以通过 `kubectl --kubeconfig credentials/kubeconfig_<eks_name> get po -n <default_cluster_name> --watch` 命令持续观察进度。

## 7. 自定义

可以按需在 `terraform.tfvars` 文件中设置各个变量，例如集群名称和镜像版本等。

### 自定义 AWS 相关的资源

由于 TiDB 服务通过 [Internal Elastic Load Balancer](#) 暴露，默认情况下，会创建一个 Amazon EC2 实例作为堡垒机，访问创建的 TiDB 集群。堡垒机上预装了 MySQL 和 Sysbench，所以可以通过 SSH 方式登陆到堡垒机后通过 ELB 访问 TiDB。如果的 VPC 中已经有了类似的 EC2 实例，可以通过设置 `create_bastion` 为 `false` 禁掉堡垒机的创建。

TiDB 版本和组件数量也可以在 `terraform.tfvars` 中修改，可以按照自己的需求配置。

### 自定义 TiDB 参数配置

Terraform 脚本中为运行在 EKS 上的 TiDB 集群提供了合理的默认配置。有自定义需求时，可以在 `clusters.tf` 中通过 `override_values` 参数为每个 TiDB 集群指定一个 `values.yaml` 文件来自定义集群参数配置。

作为例子，默认集群使用了 `./default-cluster.yaml` 作为 `values.yaml` 配置文件，并在配置中打开了"配置文件滚动更新"特性。

值得注意的是，在 EKS 上部分配置项无法在 `values.yaml` 中进行修改，包括集群版本、副本数、`NodeSelector` 以及 `Tolerations`。`NodeSelector` 和 `Tolerations` 由 Terraform 直接管理以确保基础设施与 TiDB 集群之间的一致性。集群版本和副本数可以通过 `cluster.tf` 文件中的 `tidb-cluster` module 参数来修改。

**注意：**

自定义 `values.yaml` 配置文件中，不建议包含如下配置（`tidb-cluster` module 默认固定配置）：

```
pd:
  storageClassName: ebs-gp2
tikv:
  storageClassName: local-storage
tidb:
  service:
    type: LoadBalancer
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-internal: '0.0.0.0/0'
      service.beta.kubernetes.io/aws-load-balancer-type: nlb
      service.beta.kubernetes.io/aws-load-balancer-cross-zone-load-balancing-enabled: >'true'
  separateSlowLog: true
monitor:
  storage: 100Gi
  storageClassName: ebs-gp2
  persistent: true
grafana:
  config:
    GF_AUTH_ANONYMOUS_ENABLED: "true"
  service:
    type: LoadBalancer
```

### 自定义 TiDB Operator

可以通过在 `terraform.tfvars` 中设置 `operator_values` 参数传入自定义的 `values.yaml` 内容来配置 TiDB Operator。示例如下：

```
operator_values = "./operator_values.yaml"
}
```

## 8. 管理多个 TiDB 集群

一个 `tidb-cluster` 模块的实例对应一个 TiDB 集群，可以通过编辑 `clusters.tf` 添加新的 `tidb-cluster` 模块实例来新增 TiDB 集群，示例如下：

```
module example-cluster {
  source = "../modules/aws/tidb-cluster"

  # The target EKS, required
  eks = local.eks
  # The subnets of node pools of this TiDB cluster, required
  subnets = local.subnets
  # TiDB cluster name, required
  cluster_name      = "example-cluster"

  # Helm values file
  override_values = file("example-cluster.yaml")
  # TiDB cluster version
  cluster_version           = "v3.0.0"
  # SSH key of cluster nodes
  ssh_key_name               = module.key-pair.key_name
  # PD replica number
  pd_count                   = 3
  # TiKV instance type
  pd_instance_type            = "t2.xlarge"
  # TiKV replica number
  tikv_count                 = 3
  # TiKV instance type
  tikv_instance_type          = "t2.xlarge"
  # The storage class used by TiKV, if the TiKV instance type do not have local SSD, you should change it to storage
  class
    # TiDB replica number
    tidb_count                = 2
    # TiDB instance type
    tidb_instance_type         = "t2.xlarge"
    # Monitor instance type
    monitor_instance_type       = "t2.xlarge"
    # The version of tidb-cluster helm chart
    tidb_cluster_chart_version = "v1.0.0"
    # Decides whether or not to create the tidb-cluster helm release.
    # If this variable is set to false, you have to
    # install the helm release manually
    create_tidb_cluster_release = true
}
```

注意：

`cluster_name` 必须是唯一的。

可以通过 `kubectl` 获取新集群的监控系统地址与 TiDB 地址。假如希望让 Terraform 脚本输出这些地址，可以通过在 `outputs.tf` 中增加相关的输出项实现：

```
output "example-cluster_tidb-hostname" {
  value = module.example-cluster.tidb_hostname
}

output "example-cluster_monitor-hostname" {
  value = module.example-cluster.monitor_hostname
}
```

修改完成后，执行 `terraform init` 和 `terraform apply` 创建集群。

最后，只要移除 `tidb-cluster` 模块调用，对应的 TiDB 集群就会被销毁，EC2 资源也会随之释放。

## 9. 仅管理基础设施

通过调整配置，可以控制 Terraform 脚本只创建 Kubernetes 集群和 TiDB Operator。操作步骤如下：

- 修改 `clusters.tf` 中 TiDB 集群的 `create_tidb_cluster_release` 配置项：

```
module "default-cluster" {
  ...
  create_tidb_cluster_release = false
}
```

如上所示，当 `create_tidb_cluster_release` 设置为 `false` 时，Terraform 脚本不会创建和修改 TiDB 集群，但仍会创建 TiDB 集群所需的计算和存储资源。此时，可以使用 Helm 等工具来独立管理集群。

注意：

在已经部署的集群上将 `create_tidb_cluster_release` 调整为 `false` 会导致已安装的 TiDB 集群被删除，对应的 TiDB 集群对象也会随之被删除。

## 10. 销毁集群

可以通过如下命令销毁集群：

```
terraform destroy
```

注意：

- 该操作会销毁 EKS 集群以及部署在该 EKS 集群上的所有 TiDB 集群。
- 如果不再需要存储卷中的数据，在执行 `terraform destroy` 后，需要在 AWS 控制台手动删除 EBS 卷。

## 11. 管理多个 Kubernetes 集群

本节详细介绍了如何管理多个 Kubernetes 集群（EKS），并在每个集群上部署一个或更多 TiDB 集群。

上述文档中介绍的 Terraform 脚本组合了多个 Terraform 模块：

- `tidb-operator` 模块，用于创建 EKS 集群并在 EKS 集群上安装配置 TiDB Operator。
- `tidb-cluster` 模块，用于创建 TiDB 集群所需的资源池并部署 TiDB 集群。
- EKS 上的 TiDB 集群专用的 `vpc` 模块、`key-pair` 模块和 `bastion` 模块

管理多个 Kubernetes 集群的最佳实践是为每个 Kubernetes 集群创建一个单独的目录，并在新目录中自行组合上述 Terraform 模块。这种方式能够保证多个集群间的 Terraform 状态不会互相影响，也便于自由定制和扩展。下面是一个例子：

```
mkdir -p deploy/aws-staging
vim deploy/aws-staging/main.tf
```

`deploy/aws-staging/main.tf` 的内容可以是：

```
provider "aws" {
  region = "us-west-1"
}

# 创建一个 ssh key, 用于登录堡垒机和 Kubernetes 节点
module "key-pair" {
  source = "../modules/aws/key-pair"

  name = "another-eks-cluster"
  path = "${path.cwd}/credentials/"
}

# 创建一个新的 VPC
module "vpc" {
  source = "../modules/aws/vpc"
```

```

    vpc_name = "another-eks-cluster"
}

# 在上面的 VPC 中创建一个 EKS 并部署 tidb-operator
module "tidb-operator" {
  source = "../modules/aws/tidb-operator"

  eks_name          = "another-eks-cluster"
  config_output_path = "credentials/"
  subnets           = module.vpc.private_subnets
  vpc_id            = module.vpc.vpc_id
  ssh_key_name      = module.key-pair.key_name
}

# 特殊处理，确保 helm 操作在 EKS 创建完毕后进行
resource "local_file" "kubeconfig" {
  depends_on      = [module.tidb-operator.eks]
  sensitive_content = module.tidb-operator.eks.kubeconfig
  filename        = module.tidb-operator.eks.kubeconfig_filename
}

provider "helm" {
  alias     = "eks"
  insecure = true
  install_tiller = false
  kubernetes {
    config_path = local_file.kubeconfig.filename
  }
}

# 在上面的 EKS 集群上创建一个 TiDB 集群
module "tidb-cluster-a" {
  source = "../modules/aws/tidb-cluster"
  providers = {
    helm = "helm.eks"
  }

  cluster_name = "tidb-cluster-a"
  eks          = module.tidb-operator.eks
  ssh_key_name = module.key-pair.key_name
  subnets      = module.vpc.private_subnets
}

# 在上面的 EKS 集群上创建另一个 TiDB 集群
module "tidb-cluster-b" {
  source = "../modules/aws/tidb-cluster"
  providers = {
    helm = "helm.eks"
  }

  cluster_name = "tidb-cluster-b"
  eks          = module.tidb-operator.eks
  ssh_key_name = module.key-pair.key_name
  subnets      = module.vpc.private_subnets
}

# 创建一台堡垒机
module "bastion" {
  source = "../modules/aws/bastion"

  bastion_name          = "another-eks-cluster-bastion"
  key_name               = module.key-pair.key_name
  public_subnets         = module.vpc.public_subnets
  vpc_id                 = module.vpc.vpc_id
  target_security_group_id = module.tidb-operator.eks.worker_security_group_id
  enable_ssh_to_workers = true
}

# 输出 tidb-cluster-a 的 TiDB 服务地址
output "cluster-a_tidb-dns" {
  description = "tidb service endpoints"
  value       = module.tidb-cluster-a.tidb_hostname
}

```

```
# 输出 tidb-cluster-b 的监控地址
output "cluster-b_monitor-dns" {
  description = "tidb service endpoint"
  value        = module.tidb-cluster-b.monitor_hostname
}

# 输出堡垒机 IP
output "bastion_ip" {
  description = "Bastion IP address"
  value        = module.bastion.bastion_ip
}
```

上面的例子很容易进行定制，比如，假如不需要堡垒机，便可以删去对 `bastion` 模块的调用。同时，项目中提供的 Terraform 模块均设置了合理的默认值，因此在调用这些 Terraform 模块时，可以略去大部分的参数。

可以参考默认的 Terraform 脚本来定制每个模块的参数，也可以参考每个模块的 `variables.tf` 文件来了解所有可配置的参数。

另外，这些 Terraform 模块可以很容易地集成到自己的 Terraform 工作流中。假如对 Terraform 非常熟悉，这也是我们推荐的一种使用方式。

注意：

- 由于 Terraform 本身的限制 ([hashicorp/terraform#2430](#))，在自己的 Terraform 脚本中，也需要保留上述例子中对 `helm provider` 的特殊处理。
- 创建新目录时，需要注意与 Terraform 模块之间的相对路径，这会影响调用模块时的 `source` 参数。
- 假如想在 tidb-operator 项目之外使用这些模块，需要确保 `modules` 目录中的所有模块的相对路径保持不变。

假如不想自己写 Terraform 代码，也可以直接拷贝 `deploy/aws` 目录来创建新的 Kubernetes 集群。但要注意不能拷贝已经运行过 `terraform apply` 的目录（已经有 Terraform 的本地状态）。这种情况下，推荐在拷贝前克隆一个新的仓库。

### 1.2.3.1.2 在 GCP GKE 上部署 TiDB 集群

本节介绍如何使用个人电脑（Linux 或 macOS 系统）在 GCP GKE 上部署 TiDB 集群。

警告：

当前多磁盘聚合功能[存在一些已知问题](#)，不建议在生产环境中每节点配置一块以上磁盘。我们正在修复此问题。

## 1. 环境准备

部署前，确认已安装以下软件：

- [Git](#)
- [Google Cloud SDK](#)
- [Terraform](#) >= 0.12
- [kubectl](#) >= 1.14
- [Helm](#) >= 2.11.0 且 < 3.0.0
- [jq](#)

## 2. 配置

为保证部署顺利，需要提前进行一些配置。在开始配置 Google Cloud SDK、API、Terraform 前，先下载以下资源：

```
git clone --depth=1 https://github.com/pingcap/tidb-operator && \
cd tidb-operator/deploy/gcp
```

### 配置 Google Cloud SDK

安装 Google Cloud SDK 后，需要执行 `gcloud init` 进行[初始化](#)。

### 配置 API

如果使用的 GCP 项目是新项目，需确保以下 API 已启用：

```
gcloud services enable cloudresourcemanager.googleapis.com \
cloudbilling.googleapis.com iam.googleapis.com \
compute.googleapis.com container.googleapis.com
```

### 配置 Terraform

要执行 Terraform 脚本，需要设置以下 3 个环境变量。可以等待 Terraform 提示再输入，也可以提前在 `.tfvars` 文件中定义变量。

- `GCP_CREDENTIALS_PATH`：GCP 证书文件路径。
  - 建议另建一个服务账号给 Terraform 使用，参考[创建与管理服务账号文档](#)。`./create-service-account.sh` 会创建最低权限的服务账号。
  - 参考[服务账号密钥文档](#)来创建服务账号密钥。下面脚本中的步骤详细说明了如何使用 `deploy/gcp` 目录中提供的脚本执行此操作。或者，如果自己创建服务账号和密钥，可以在创建时选择 `JSON` 类型的密钥。下载的包含私钥的 `JSON` 文档即所需的证书文件。
- `GCP_REGION`：创建资源所在的区域，例如：`us-west1`。

- `GCP_PROJECT` : GCP 项目的名称。

要使用以上 3 个环境变量来配置 Terraform，可执行以下步骤：

- (1) 将 `GCP_REGION` 替换为的 GCP Region。

```
```bash
echo GCP_REGION=\"us-west1\" >> terraform.tfvars
```

```

- (2) 将 `GCP_PROJECT` 替换为的 GCP 项目名称，确保连接的是正确的 GCP 项目。

```
```bash
echo "GCP_PROJECT=$(gcloud config get-value project)" >> terraform.tfvars
```

```

- (3) 初始化 Terraform。

```
```bash
terraform init
```

```

- (4) 为 Terraform 创建一个有限权限的服务账号，并设置证书路径。

```
```bash
./create-service-account.sh
```

```

Terraform 自动加载和填充匹配 `terraform.tfvars` 或 `*.auto.tfvars` 文件的变量。相关详细信息，请参阅 [Terraform 文档](#)。

上述步骤会使用 `GCP_REGION` 和 `GCP_PROJECT` 填充 `terraform.tfvars` 文件，使用 `GCP_CREDENTIALS_PATH` 填充 `credentials.auto.tfvars` 文件。

## 3. 部署 TiDB 集群

本小节介绍如何部署 TiDB 集群。

- (1) 确定实例类型。

- 如果只是想试一下 TiDB，又不想花费太高成本，可以采用轻量级的配置：

```
```bash
cat small.tfvars >> terraform.tfvars
```

```

- 如果要对生产环境的部署进行 benchmark 测试，则建议采用生产级的配置：

```
```bash
cat prod.tfvars >> terraform.tfvars
```

```

`prod.tfvars` 会默认创建一个新的 VPC，两个子网和一个 `f1-micro` 实例作为堡垒机，以及使用以下实例类型作为工作节点的 GKE 集群：

- \* 3 台 n1-standard-4 实例：部署 PD
- \* 3 台 n1-highmem-8 实例：部署 TiKV
- \* 3 台 n1-standard-16 实例：部署 TiDB
- \* 3 台 n1-standard-2 实例：部署监控组件

如上所述，生产环境的部署需要 91 个 CPU，超过了 GCP 项目的默认配额。可以参考 [配额](<https://cloud.google.com/compute/quotas>) 来增加项目配额。扩容同样需要更多 CPU。

```
> **注意：**
>
> 工作节点的数量取决于指定 Region 中可用区的数量。大部分 Region 有 3 个可用区，但是 `us-central1` 有 4 个可用区。参考 [Regions and Zones](https://cloud.google.com/compute/docs/regions-zones/) 查看更多信息。参考 [自定义](#自定义) 部分来自定义区域集群的节点池。
```

```

## (2) 启动脚本来部署 TiDB 集群：

```
```bash
terraform apply
```
> **注意：**
>
> 如果未提前设置上文所述的 3 个环境变量，执行 `terraform apply` 过程中会有提示出现，要求对 3 个变量进行设置。详情请参考 [配置 Terraform](#配置-terraform)。
```

```

整个过程可能至少需要 10 分钟。`terraform apply` 执行成功后，会输出类似如下的信息：

```
...
Apply complete! Resources: 23 added, 0 changed, 0 destroyed.
```

Outputs:

```
how_to_connect_to_default_cluster_tidb_from_bastion = mysql -h 172.31.252.20 -P 4000 -u root
how_to_ssh_to_bastion = gcloud compute ssh tidb-cluster-bastion --zone us-west1-b
how_to_set_reclaim_policy_of_pv_for_default_tidb_cluster_to_delete = kubectl --kubeconfig ./credentials/kubeconfig _tidb-cluster get pvc -n tidb-cluster -o jsonpath='{.items[*].spec.volumeName}'|fmt -1 | xargs -I {} kubectl --kubecfg /.../credentials/kubeconfig_tidb-cluster patch pv {} -p '{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
kubeconfig_file = ./credentials/kubeconfig_tidb-cluster
monitor_lb_ip = 35.227.134.146
monitor_port = 3000
region = us-west1
tidb_version = v3.0.1
```

```

## 4. 访问 TiDB 数据库

`terraform apply` 运行完成后，可执行以下步骤来访问 TiDB 数据库。注意用 [部署 TiDB 集群](#) 小节的输出信息替换 `<>` 部分的内容。

### (1) 通过 ssh 远程连接到堡垒机。

```
```bash
gcloud compute ssh <gke-cluster-name>-bastion --zone <zone>
```

```

(2) 通过 MySQL 客户端来访问 TiDB 集群。

```
```bash
mysql -h <tidb_ilb_ip> -P 4000 -u root
```

> **注意：**
>
> 通过 MySQL 连接 TiDB 前，需要先安装 MySQL 客户端。

```

## 5. 与 GKE 集群交互

可以通过 `kubectl` 和 `helm` 使用 `kubeconfig` 文件 `credentials/kubeconfig_<gke_cluster_name>` 和 GKE 集群交互。交互方式主要有以下两种。

注意：

`gke_cluster_name` 默认为 `tidb-cluster`，可以通过 `variables.tf` 中 `gke_name` 修改。

- 指定 `--kubeconfig` 参数：

```
kubectl --kubeconfig credentials/kubeconfig_<gke_cluster_name> get po -n <tidb_cluster_name>
```

```
helm --kubeconfig credentials/kubeconfig_<gke_cluster_name> ls
```

- 设置 `KUBECONFIG` 环境变量：

```
export KUBECONFIG=$PWD/credentials/kubeconfig_<gke_cluster_name>
```

```
kubectl get po -n <tidb_cluster_name>
```

```
helm ls
```

## 6. 升级 TiDB 集群

要升级 TiDB 集群，可执行以下步骤：

(1) 编辑 `terraform.tfvars` 文件，将 `tidb_version` 变量的值修改为更高版本。 (2) 运行 `terraform apply`。

例如，要将 TiDB 集群升级到 4.0.0-rc.2，可设置 `tidb_version` 为 `v4.0.0-rc.2`：

```
tidb_version = "v4.0.0-rc.2"
```

升级过程会持续一段时间。可以通过以下命令来持续观察升级进度：

```
kubectl --kubeconfig credentials/kubeconfig_<gke_cluster_name> get po -n <tidb_cluster_name> --watch
```

然后，可以[访问数据库](#)并通过 `tidb_version()` 确认 TiDB 集群是否升级成功：

```
select tidb_version();

*****
1. row *****
tidb_version(): Release Version: v3.0.0-rc.2
Git Commit Hash: 06f3f63d5a87e7f0436c0618cf524fea7172eb93
Git Branch: HEAD
UTC Build Time: 2019-05-28 12:48:52
GoVersion: go version go1.12 linux/amd64
Race Enabled: false
TiKV Min Version: 2.1.0-alpha.1-ff3dd160846b7d1aed9079c389fc188f7f5ea13e
Check Table Before Drop: false
1 row in set (0.001 sec)
```

## 7. 管理多个 TiDB 集群

一个 `tidb-cluster` 模块的实例对应一个 GKE 集群中的 TiDB 集群。要添加一个新的 TiDB 集群，可执行以下步骤：

- (1) 编辑 `tidbclusters.tf` 文件来添加一个 `tidb-cluster` 模块。

例如：

```
```hcl
module "example-tidb-cluster" {
  providers = {
    helm = "helm.gke"
  }
  source          = "../modules/gcp/tidb-cluster"
  cluster_id      = module.tidb-operator.cluster_id
  tidb_operator_id = module.tidb-operator.tidb_operator_id
  gcp_project     = var.GCP_PROJECT
  gke_cluster_location = local.location
  gke_cluster_name   = <gke-cluster-name>
  cluster_name       = <example-tidb-cluster>
  cluster_version    = "v3.0.1"
  kubeconfig_path    = local.kubeconfig
  tidb_cluster_chart_version = "v1.0.0"
  pd_instance_type   = "n1-standard-1"
  tikv_instance_type = "n1-standard-4"
  tidb_instance_type = "n1-standard-2"
  monitor_instance_type = "n1-standard-1"
  pd_node_count      = 1
  tikv_node_count    = 2
  tidb_node_count    = 1
  monitor_node_count = 1
}
```

> **注意：**
>
> - 每个集群的 `cluster_name` 必须是唯一的。
> - 为任一组件实际创建的总节点数等于配置文件中的节点数乘以该 Region 中可用区的个数。
```

可以通过 `kubectl` 获取创建的 TiDB 集群和监控组件的地址。如果希望 Terraform 脚本打印此信息，可在 `outputs.tf` 中添加一个 `output` 配置项，如下所示：

```
```hcl
output "how_to_connect_to_example_tidb_cluster_from_bastion" {
  value = module.example-tidb-cluster.how_to_connect_to_tidb_from_bastion
}
```

```

上述配置可使该脚本打印出用于连接 TiDB 集群的命令。

- (2) 修改完成后，执行以下命令来创建集群。

```
```bash
terraform init
```
```bash
terraform apply
```
```

```

## 8. 扩容

如果需要扩容 TiDB 集群，可执行以下步骤：

(1) 按需在文件 `terraform.tfvars` 中设置 `tikv_count`、`tidb_count` 变量。 (2) 运行 `terraform apply`。

**警告：**

由于缩容过程中无法确定哪个节点会被删除，因此目前不支持集群缩容。通过修改 `tikv_count` 来进行缩容可能会导致数据丢失。

扩容过程会持续几分钟，可以通过以下命令来持续观察进度：

```
kubectl --kubeconfig credentials/kubeconfig_<gke_cluster_name> get po -n <tidb_cluster_name> --watch
```

例如，可以将 `tidb_count` 从 1 改为 2 来扩容 TiDB：

```
tidb_count      = 2
```

**注意：**

增加节点数量会在每个可用区都增加节点。

## 9. 自定义

可以在 `terraform.tfvars` 文件来指定值。

### 自定义 GCP 资源

GCP 允许 `n1-standard-1` 或者更大的实例类型挂载本地 SSD，这提供了更好的自定义特性。

### 自定义 TiDB 参数配置

Terraform 脚本为 GKE 中的 TiDB 集群提供了默认设置。也可以在 `tidbclusters.tf` 中为每个 TiDB 集群指定一个覆盖配置 `override_values` 或者覆盖配置文件 `override_values_file`。如果同时配置两个变量，`override_values` 配置将生效，该自定义配置会覆盖默认设置，示例如下：

```
override_values = <<EOF
discovery:
  image: pingcap/tidb-operator:v1.0.1
  imagePullPolicy: IfNotPresent
  resources:
    limits:
      cpu: 250m
      memory: 150Mi
    requests:
      cpu: 30m
      memory: 30Mi
EOF
```

```
override_values_file = "./test-cluster.yaml"
```

集群默认使用 `deploy/modules/gcp/tidb-cluster` 模块中的 `values/default.yaml` 作为覆盖配置文件。

在 GKE 中，某些值不支持在 `values.yaml` 中自定义，包括集群版本、副本数、`NodeSelector` 以及 `Tolerations`。`NodeSelector` 和 `Tolerations` 由 Terraform 直接管理，以确保基础设施与 TiDB 集群之间的一致性。

如果需要自定义集群版本和副本数，可以修改 `tidbclusters.tf` 文件中每个 `tidb-cluster` module 的参数。

注意：

自定义配置中，不建议在 `values.yaml` 中包含以下配置（`tidb-cluster` module 默认固定配置）：

```
pd:
  storageClassName: pd-ssd
tikv:
  storageClassName: local-storage
tidb:
  service:
    type: LoadBalancer
    annotations:
      cloud.google.com/load-balancer-type: "Internal"
  separateSlowLog: true
monitor:
  storageClassName: pd-ssd
  persistent: true
grafana:
  config:
    GF_AUTH_ANONYMOUS_ENABLED: "true"
  service:
    type: LoadBalancer
```

## 自定义 TiDB Operator

如果要自定义 TiDB Operator，可以使用 `operator_helm_values` 变量来指定覆盖配置或者使用 `operator_helm_values_file` 变量来指定覆盖配置文件。如果同时配置两个变量，`operator_helm_values` 配置将生效，该自定义配置会传递给 `tidb-operator` 模块，示例如下：

```
operator_helm_values = <<EOF
controllerManager:
  resources:
    limits:
      cpu: 250m
      memory: 150Mi
    requests:
      cpu: 30m
      memory: 30Mi
EOF

operator_helm_values_file = "./test-operator.yaml"
```

## 自定义日志

GKE 使用 [Fluentd](#) 作为其默认的日志收集工具，然后将日志转发到 Stackdriver。Fluentd 进程可能会占用大量资源，消耗大量的 CPU 和 RAM。[Fluent Bit](#) 是一种性能更高，资源占用更少的替代方案。与 Fluentd 相比，更建议在生产环境中使用 Fluent Bit。可参考[在 GKE 集群中设置 Fluent Bit 的示例](#)。

## 自定义节点池

集群是按区域 (regional) 而非按可用区 (zonal) 来创建的。也就是说，GKE 向每个可用区复制相同的节点池，以实现更高的可用性。但对于 Grafana 这样的监控服务来说，通常没有必要维护相同的可用性。可以通过 `gcloud` 手动删除节点。

注意：

GKE 节点池通过实例组管理。如果使用 `gcloud compute instances delete` 命令删除某个节点，GKE 会自动重新创建节点并将其添加到集群。

如果需要从监控节点池中删掉一个节点，可采用如下步骤：

(1) 获取托管的实例组和所在可用区。

```
```bash
gcloud compute instance-groups managed list | grep monitor
```

```

输出结果类似：

```
```
gke-tidb-monitor-pool-08578e18-grp us-west1-b zone gke-tidb-monitor-pool-08578e18 0 0 gke-tidb-mo
nitor-pool-08578e18 no
gke-tidb-monitor-pool-7e31100f-grp us-west1-c zone gke-tidb-monitor-pool-7e31100f 1 1 gke-tidb-mo
nitor-pool-7e31100f no
gke-tidb-monitor-pool-78a961e5-grp us-west1-a zone gke-tidb-monitor-pool-78a961e5 1 1 gke-tidb-mo
nitor-pool-78a961e5 no
```

```

第一列是托管的实例组，第二列是所在的可用区。

(2) 获取实例组中的实例名字。

```
```bash
gcloud compute instance-groups managed list-instances <the-name-of-the-managed-instance-group> --zone <zone>
```

```

示例：

```
```bash
gcloud compute instance-groups managed list-instances gke-tidb-monitor-pool-08578e18-grp --zone us-west1-b
```

```

输出结果类似：

```
```
NAME ZONE STATUS ACTION INSTANCE_TEMPLATE VERSION
_NAME LAST_ERROR
gke-tidb-monitor-pool-08578e18-c7vd us-west1-b RUNNING NONE gke-tidb-monitor-pool-08578e18
```

```

(3) 通过指定托管的实例组和实例的名称来删掉该实例。

例如：

```
```bash
gcloud compute instance-groups managed delete-instances gke-tidb-monitor-pool-08578e18-grp --instances=gke-tidb-monit
or-pool-08578e18-c7vd --zone us-west1-b
```

```

## 10. 销毁 TiDB 集群

如果不想要继续使用 TiDB 集群，可以通过如下命令进行销毁：

```
terraform destroy
```

注意：

在执行 `terraform destroy` 过程中，可能会发生错误：Error reading Container Cluster "tidb": Cluster "tidb" has status "RECONCILING" with message""。当 GCP 升级 Kubernetes master 节点时会出现该问题。一旦问题出现，就无法删除集群，需要等待 GCP 升级结束，再次执行 `terraform destroy`。

## 删除磁盘

如果不再需要之前的数据，并且想要删除正在使用的磁盘，有以下两种方法可以完成此操作：

- 手动删除：在 Google Cloud Console 中删除磁盘，或使用 `gcloud` 命令行工具执行删除操作。
- 自动删除：在执行 `terraform destroy` 之前将 Kubernetes 的 PV (Persistent Volume) 回收策略设置为 `Delete`，具体操作为在 `terraform destroy` 之前运行以下 `kubectl` 命令：

```
kubectl --kubeconfig /path/to/kubeconfig/file get pvc -n namespace-of-tidb-cluster -o jsonpath='{.items[*].spec.volumeName}' | fmt -1 | xargs -I {} kubectl --kubeconfig /path/to/kubeconfig/file patch pv {} -p '{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
```

上述命令将获取 TiDB 集群命名空间中的 PVC (Persistent Volume Claim)，并将绑定的 PV 的回收策略设置为 `Delete`。在执行 `terraform destroy` 过程中删除 PVC 时，也会将磁盘删除。

下面是一个名为 `change-pv-reclaimpolicy.sh` 的脚本。相对于仓库根目录来说，它在 `deploy/gcp` 目录，简化了上述过程。

```
./change-pv-reclaimpolicy.sh /path/to/kubeconfig/file <tidb-cluster-namespace>
```

## 11. 管理多个 Kubernetes 集群

本节介绍管理多个 Kubernetes 集群的最佳实践，其中每个 Kubernetes 集群都可以部署一个或多个 TiDB 集群。

在 TiDB 的案例中，Terraform 模块通常结合了几个子模块：

- `tidb-operator`：为 TiDB 集群提供 [Kubernetes Control Plane](#) 并部署 TiDB Operator。
- `tidb-cluster`：在目标 Kubernetes 集群中创建资源池并部署 TiDB 集群。
- 一个 `vpc` 模块，一个 `bastion` 模块和一个 `project-credentials` 模块：专门用于 GKE 上的 TiDB 集群。

管理多个 Kubernetes 集群的最佳实践有以下两点：

(1) 为每个 Kubernetes 集群创建一个新目录。(2) 根据具体需求，使用 Terraform 脚本将上述模块进行组合。

如果采用了最佳实践，集群中的 Terraform 状态不会相互干扰，并且可以很方便地管理多个 Kubernetes 集群。示例如下（假设已在项目根目录）：

```
mkdir -p deploy/gcp-staging && \
vim deploy/gcp-staging/main.tf
```

deploy/gcp-staging/main.tf 中的内容类似：

```
provider "google" {
  credentials = file(var.GCP_CREDENTIALS_PATH)
  region     = var.GCP_REGION
  project    = var.GCP_PROJECT
}

// required for taints on node pools
provider "google-beta" {
  credentials = file(var.GCP_CREDENTIALS_PATH)
  region     = var.GCP_REGION
  project    = var.GCP_PROJECT
```

```

}

locals {
  gke_name      = "another-gke-name"
  credential_path = "${path.cwd}/credentials"
  kubeconfig     = "${local.credential_path}/kubeconfig_${var.gke_name}"
}

module "project-credentials" {
  source = "../modules/gcp/project-credentials"

  path = local.credential_path
}

module "vpc" {
  source          = "../modules/gcp/vpc"
  create_vpc     = true
  gcp_project    = var.GCP_PROJECT
  gcp_region     = var.GCP_REGION
  vpc_name        = "${locals.gke_name}-vpc-network"
  private_subnet_name = "${locals.gke_name}-private-subnet"
  public_subnet_name = "${locals.gke_name}-public-subnet"
}

module "tidb-operator" {
  source          = "../modules/gcp/tidb-operator"
  gke_name        = locals.gke_name
  vpc_name        = module.vpc.vpc_name
  subnetwork_name = module.vpc.private_subnetwork_name
  gcp_project    = var.GCP_PROJECT
  gcp_region     = var.GCP_REGION
  kubeconfig_path = local.kubeconfig
  tidb_operator_version = "v1.0.0"
}

module "bastion" {
  source          = "../modules/gcp/bastion"
  vpc_name        = module.vpc.vpc_name
  public_subnet_name = module.vpc.public_subnetwork_name
  gcp_project    = var.GCP_PROJECT
  bastion_name   = "${locals.gke_name}-tidb-bastion"
}

# HACK: 强制使 Helm 依赖 GKE 集群
data "local_file" "kubeconfig" {
  depends_on = [module.tidb-operator.cluster_id]
  filename   = module.tidb-operator.kubeconfig_path
}

resource "local_file" "kubeconfig" {
  depends_on = [module.tidb-operator.cluster_id]
  content    = data.local_file.kubeconfig.content
  filename   = module.tidb-operator.kubeconfig_path
}

provider "helm" {
  alias      = "gke"
  insecure   = true
  install_tiller = false
  kubernetes {
    config_path = local_file.kubeconfig.filename
  }
}

module "tidb-cluster-a" {
  providers = {
    helm = "helm.gke"
  }
  source          = "../modules/gcp/tidb-cluster"
  gcp_project    = var.GCP_PROJECT
  gke_cluster_location = var.GCP_REGION
  gke_cluster_name   = locals.gke_name
  cluster_name     = "tidb-cluster-a"
}

```

```

cluster_version      = "v3.0.1"
kubeconfig_path     = module.tidb-operator.kubeconfig_path
tidb_cluster_chart_version = "v1.0.0"
pd_instance_type    = "n1-standard-1"
tikv_instance_type  = "n1-standard-4"
tidb_instance_type  = "n1-standard-2"
monitor_instance_type = "n1-standard-1"
}

module "tidb-cluster-b" {
  providers = {
    helm = "helm.gke"
  }
  source          = "../modules/gcp/tidb-cluster"
  gcp_project     = var.GCP_PROJECT
  gke_cluster_location = var.GCP_REGION
  gke_cluster_name   = locals.gke_name
  cluster_name      = "tidb-cluster-b"
  cluster_version   = "v3.0.1"
  kubeconfig_path   = module.tidb-operator.kubeconfig_path
  tidb_cluster_chart_version = "v1.0.0"
  pd_instance_type  = "n1-standard-1"
  tikv_instance_type = "n1-standard-4"
  tidb_instance_type = "n1-standard-2"
  monitor_instance_type = "n1-standard-1"
}

output "how_to_ssh_to_bastion" {
  value= module.bastion.how_to_ssh_to_bastion
}

output "connect_to_tidb_cluster_a_from_bastion" {
  value = module.tidb-cluster-a.how_to_connect_to_default_cluster_tidb_from_bastion
}

output "connect_to_tidb_cluster_b_from_bastion" {
  value = module.tidb-cluster-b.how_to_connect_to_default_cluster_tidb_from_bastion
}

```

如上述代码所示，可以在每个模块调用中省略几个参数，因为有合理的默认值，并且可以轻松地自定义配置。例如，如果不需要调用堡垒机模块，将其删除即可。

如果要自定义每个字段，可使用以下两种方法中的一种：

- 直接修改 `*.tf` 文件中 `module` 的参数配置。
- 参考每个模块的 `variables.tf` 文件，了解所有可修改的参数，并在 `terraform.tfvars` 中设置自定义值。

注意：

- 创建新目录时，请注意其与 Terraform 模块的相对路径，这会影响模块调用期间的 `source` 参数。
- 如果要在 tidb-operator 项目之外使用这些模块，务必确保复制整个 `modules` 目录并保持目录中每个模块的相对路径不变。
- 由于 Terraform 的限制（[参见 hashicorp/terraform # 2430](#)），上面示例中添加了 **HACK:** 强制使 Helm 依赖 GKE 集群部分对 Helm provider 进行处理。如果自己编写 tf 文件，需要包含这部分内容。

如果不愿意编写 Terraform 代码，还可以复制 `deploy/gcp` 目录来创建新的 Kubernetes 集群。但需要注意，不能复制已被执行 `terraform apply` 命令的目录。在这种情况下，建议先克隆新的仓库再复制目录。

### 1.2.3.1.3 在阿里云上部署 TiDB 集群

本文介绍了如何使用个人电脑（Linux 或 macOS 系统）在阿里云上部署 TiDB 集群。

## 1. 环境需求

- `aliyun-cli` >= 3.0.15 并且配置 `aliyun-cli`

注意：

Access Key 需要具有操作相应资源的权限。

- `kubectl` >= 1.12
- `helm` >= 2.11.0
- `jq` >= 1.6
- `terraform` 0.12.\*

你可以使用阿里云的[云命令行](#)服务来进行操作，云命令行中已经预装并配置好了所有工具。

## 权限

完整部署集群需要具备以下权限：

- AliyunECSFullAccess
- AliyunESSFullAccess
- AliyunVPCFullAccess
- AliyunSLBFullAccess
- AliyunCSFullAccess
- AliyunEIPFullAccess
- AliyunECIFullAccess
- AliyunVPNGatewayFullAccess
- AliyunNATGatewayFullAccess

## 2. 概览

默认配置下，会创建：

- 一个新的 VPC
- 一台 ECS 实例作为堡垒机
- 一个托管版 ACK（阿里云 Kubernetes）集群以及一系列 worker 节点：
  - 属于一个伸缩组的 2 台 ECS 实例（2 核 2 GB）托管版 Kubernetes 的默认伸缩组中必须至少有两台实例，用于承载整个的系统服务，例如 CoreDNS
  - 属于一个伸缩组的 3 台 `ecs.g5.large` 实例，用于部署 PD
  - 属于一个伸缩组的 3 台 `ecs.i2.2xlarge` 实例，用于部署 TiKV
  - 属于一个伸缩组的 2 台 `ecs.c5.4xlarge` 实例用于部署 TiDB
  - 属于一个伸缩组的 1 台 `ecs.c5.xlarge` 实例用于部署监控组件
  - 一块 100 GB 的云盘用作监控数据存储

除了默认伸缩组之外的其它所有实例都是跨可用区部署的。而伸缩组（Auto-scaling Group）能够保证集群的健康实例数等于期望数值。因此，当发生节点故障甚至可用区故障时，伸缩组能够自动为我们创建新实例来确保服务可用性。

## 3. 安装部署

(1) 设置目标 Region 和阿里云密钥（也可以在运行 `terraform` 命令时根据命令提示输入）：

```
```shell
export TF_VAR_ALICLOUD_REGION=<YOUR_REGION> && \
export TF_VAR_ALICLOUD_ACCESS_KEY=<YOUR_ACCESS_KEY> && \
export TF_VAR_ALICLOUD_SECRET_KEY=<YOUR_SECRET_KEY>
```

```

用于部署集群的各变量的默认值存储在 `variables.tf` 文件中，如需定制可以修改此文件或在安装时通过 `"-var` 参数覆盖。

(2) 使用 Terraform 进行安装：

```
```shell
git clone --depth=1 https://github.com/pingcap/tidb-operator && \
cd tidb-operator/deploy/aliyun
```

```

```
```shell
terraform init
```

```

`apply` 过程中需要输入 `yes` 来确认执行：

```
```shell
terraform apply
```

```

假如在运行 `terraform apply` 时出现报错，可根据报错信息（例如缺少权限）进行修复后再次运行 `terraform apply`。

整个安装过程大约需要 5 至 10 分钟，安装完成后会输出集群的关键信息（想要重新查看这些信息，可以运行 `terraform output`）：

```
```
Apply complete! Resources: 3 added, 0 changed, 1 destroyed.
```

Outputs:

```
bastion_ip = 47.96.174.214
cluster_id = c2d9b20854a194f158ef2bc8ea946f20e
kubeconfig_file = /tidb-operator/deploy/aliyun/credentials/kubeconfig
monitor_endpoint = 121.199.195.236:3000
region = cn-hangzhou
ssh_key_file = /tidb-operator/deploy/aliyun/credentials/my-cluster-keyZ.pem
tidb_endpoint = 172.21.5.171:4000
tidb_version = v3.0.0
vpc_id = vpc-bp1v8i5rwsc7yh8dwyp5
```

```

(3) 用 `kubectl` 或 `helm` 对集群进行操作：

```
```shell
export KUBECONFIG=$PWD/credentials/kubeconfig
```

```

```
```shell
kubectl version
```

```

```
```shell
helm ls
```

```

## 4. 连接数据库

通过堡垒机可连接 TiDB 集群进行测试，相关信息在安装完成后的输出中均可找到：

```
ssh -i credentials/<cluster_name>-key.pem root@<bastion_ip>
```

```
mysql -h <tidb_slb_ip> -P 4000 -u root
```

## 5. 监控

访问 `<monitor_endpoint>` 就可以查看相关的 Grafana 监控面板。相关信息可在安装完成后的输出中找到。默认帐号密码为：

- 用户名：admin
- 密码：admin

警告：

出于安全考虑，假如你已经或将要配置 VPN 用于访问 VPC，强烈建议将 `deploy/modules/aliyun/tidb-cluster/values/default.yaml` 文件里 `monitor.grafana.service.annotations` 中的 `service.beta.kubernetes.io/alicloud-loadbalancer-address-type` 设置为 `intranet` 以禁止监控服务的公网访问。

## 6. 升级 TiDB 集群

在 `terraform.tfvars` 中设置 `tidb_version` 参数，并再次运行 `terraform apply` 即可完成升级。

升级操作可能会执行较长时间，可以通过以下命令来持续观察进度：

```
kubectl get pods --namespace <tidb_cluster_name> -o wide --watch
```

## 7. TiDB 集群水平伸缩

按需在 `terraform.tfvars` 中设置 `tikv_count` 和 `tidb_count` 数值，再次运行 `terraform apply` 即可完成 TiDB 集群的水平伸缩。

## 8. 销毁集群

```
terraform destroy
```

假如 Kubernetes 集群没有创建成功，那么在 `destroy` 时会出现报错，无法进行正常清理。此时需要手动将 Kubernetes 资源从本地状态中移除：

```
terraform state list
```

```
terraform state rm module.ack.alicloud_cs_managed_kubernetes.k8s
```

销毁集群操作需要执行较长时间。

注意：

监控组件挂载的云盘需要在阿里云管理控制台中手动删除。

## 9. 配置

## 配置 TiDB Operator

可以参考 `variables.tf` 内的变量来配置 TiDB Operator，大多数配置项均能按照 `variable` 的注释理解语义后进行修改。需要注意的是，`operator_helm_values` 配置项允许为 TiDB Operator 提供一个自定义的 `values.yaml` 配置文件，示例如下：

- 在 `terraform.tfvars` 中设置 `operator_helm_values`：

```
operator_helm_values = "./my-operator-values.yaml"
```

- 在 `main.tf` 中设置 `operator_helm_values`：

```
operator_helm_values = file("./my-operator-values.yaml")
```

同时，在默认配置下 Terraform 脚本会创建一个新的 VPC，假如要使用现有的 VPC，可以在 `variable.tf` 中设置 `vpc_id`。注意，当使用现有 VPC 时，没有设置 vswitch 的可用区将不会部署 Kubernetes 节点。

## 配置 TiDB 集群

TiDB 集群会使用 `./my-cluster.yaml` 作为集群的 `values.yaml` 配置文件，修改该文件即可配置 TiDB 集群。支持的配置项可参考 [Kubernetes 上的 TiDB 集群配置](#)。

## 10. 管理多个 TiDB 集群

需要在一个 Kubernetes 集群下管理多个 TiDB 集群时，需要编辑 `./main.tf`，按实际需要新增 `tidb-cluster` 声明，示例如下：

```
module "tidb-cluster-dev" {
  source = "../modules/aliyun/tidb-cluster"
  providers = {
    helm = helm.default
  }

  cluster_name = "dev-cluster"
  ack        = module.tidb-operator

  pd_count          = 1
  tikv_count        = 1
  tidb_count        = 1
  override_values    = file("dev-cluster.yaml")
}

module "tidb-cluster-staging" {
  source = "../modules/aliyun/tidb-cluster"
  providers = {
    helm = helm.default
  }

  cluster_name = "staging-cluster"
  ack        = module.tidb-operator

  pd_count          = 3
  tikv_count        = 3
  tidb_count        = 2
  override_values    = file("staging-cluster.yaml")
}
```

注意，多个 TiDB 集群之间 `cluster_name` 必须保持唯一。下面是 `tidb-cluster` 模块的所有可配置参数：

| 参数名                        | 说明   | 默认值               |
|----------------------------|--|-------------------|
| ack                        | 封装目标 Kubernetes 集群信息的结构体，必填                    | nil               |
| cluster_name               | TiDB 集群名，必填且必须唯一                               | nil               |
| tidb_version               | TiDB 集群版本                                      | v3.0.1            |
| tidb_cluster_chart_version | tidb-cluster helm chart 的版本                    | v1.0.1            |
| pd_count                   | PD 节点数   | 3                 |
| pd_instance_type           | PD 实例类型  | ecs.g5.large      |
| tikv_count                 | TiKV 节点数                                       | 3                 |
| tikv_instance_type         | TiKV 实例类型                                      | ecs.i2.2xlarge    |
| tidb_count                 | TiDB 节点数                                       | 2                 |
| tidb_instance_type         | TiDB 实例类型                                      | ecs.c5.4xlarge    |
| monitor_instance_type      | 监控组件的实例类型                                      | ecs.c5.xlarge     |
| override_values            | TiDB 集群的 values.yaml 配置文件，通常通过 file() 函数从文件中读取 | nil               |
| local_exec_interpreter     | 执行命令行指令的解释器                                    | ["/bin/sh", "-c"] |

## 11. 管理多个 Kubernetes 集群

推荐针对每个 Kubernetes 集群都使用单独的 Terraform 模块进行管理（一个 Terraform Module 即一个包含 .tf 脚本的目录）。

deploy/aliyun 实际上是将 deploy/modules 中的数个可复用的 Terraform 脚本组合在了一起。当管理多个集群时（下面的操作在 tidb-operator 项目根目录下进行）：

(1) 首先针对每个集群创建一个目录，如：

```
```shell
mkdir -p deploy/aliyun-staging
```
```

(2) 参考 deploy/aliyun 的 main.tf，编写自己的脚本，下面是一个简单的例子：

```

```hcl
provider "alicloud" {
  region      = <YOUR_REGION>
  access_key  = <YOUR_ACCESS_KEY>
  secret_key  = <YOUR_SECRET_KEY>
}

module "tidb-operator" {
  source      = "../modules/aliyun/tidb-operator"

  region      = <YOUR_REGION>
  access_key  = <YOUR_ACCESS_KEY>
  secret_key  = <YOUR_SECRET_KEY>
  cluster_name = "example-cluster"
  key_file    = "ssh-key.pem"
  kubeconfig_file = "kubeconfig"
}

provider "helm" {
  alias      = "default"
  insecure   = true
  install_tiller = false
  kubernetes {
    config_path = module.tidb-operator.kubeconfig_filename
  }
}

module "tidb-cluster" {
  source = "../modules/aliyun/tidb-cluster"
  providers = {
    helm = helm.default
  }

  cluster_name = "example-cluster"
  ack         = module.tidb-operator
}

module "bastion" {
  source = "../modules/aliyun/bastion"

  bastion_name      = "example-bastion"
  key_name          = module.tidb-operator.key_name
  vpc_id            = module.tidb-operator.vpc_id
  vswitch_id        = module.tidb-operator.vswitch_ids[0]
  enable_ssh_to_worker = true
  worker_security_group_id = module.tidb-operator.security_group_id
}
```

```

上面的脚本可以自由定制，比如，假如不需要堡垒机则可以移除 `module "bastion"` 相关声明。

你也可以直接拷贝 `deploy/aliyun` 目录，但要注意不能拷贝已经运行了 `terraform apply` 的目录，建议重新 clone 仓库再进行拷贝。

## 12. 使用限制

目前，`pod cidr`，`service cidr` 和节点型号等配置在集群创建后均无法修改。

### 1.2.3.1.4 在京东云上部署 TiDB 集群

#### 1. 创建 Kubernetes 集群

参考官方文档 <https://docs.jdcloud.com/cn/jcs-for-kubernetes/create-to-cluster>

#### 2. 连接集群

参考官方文档 <https://docs.jdcloud.com/cn/jcs-for-kubernetes/connect-to-cluster>

#### 3. 安装 Helm

(1) 通过 <https://github.com/helm/helm/releases> 找到要下载的 Helm 版本，TiDB Operator 要求 Helm 版本 < 3.0：

```
wget https://get.helm.sh/helm-v2.16.1-linux-amd64.tar.gz
```

(2) 解压缩：

```
tar -zxvf helm-v2.16.1-linux-amd64.tar.gz
```

(3) 在解压后的目录中找到二进制文件，将其移动到所需的位置并添加执行权限：

```
mv linux-amd64/helm /usr/local/bin/helm
chmod +x /usr/local/bin/helm
```

(4) 运行以下命令：

```
helm help
```

(5) 为 Tiller 添加权限，详见 [Role-based Access Control](#)，新建 rbac-config.yaml，内容如下：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
```

(6) 初始化 Helm 并安装 Tiller 服务：

```
helm init --upgrade --service-account tiller
```

如果无法下载镜像，可以用 `--tiller-image` 参数替换镜像地址。

(7) 运行以下命令：

```
helm version
```

出现以下信息，确认安装成功：

```
Client: &version.Version{SemVer:"v2.16.1", GitCommit:"bbdfe5e7803a12bbdf97e94cd847859890cf4050", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.16.1", GitCommit:"bbdfe5e7803a12bbdf97e94cd847859890cf4050", GitTreeState:"clean"}
```

(8) 配置 PingCAP 官方 chart 仓库：

```
helm repo add pingcap https://charts.pingcap.org/
```

## 4. 安装 TiDB Operator

TiDB Operator 使用 [CRD \(Custom Resource Definition\)](#) 扩展 Kubernetes，所以要使用 TiDB Operator，必须先创建 `TidbCluster` 自定义资源类型。只需要在你的 Kubernetes 集群上创建一次即可。

```
kubectl apply -f https://raw.githubusercontent.com/pingcap/tidb-operator/master/manifests/crd.yaml && kubectl get crd tidbclusters.pingcap.com
```

创建 `TidbCluster` 自定义资源类型后，在 Kubernetes 集群上安装 TiDB Operator。

(1) 获取你要安装的 `tidb-operator` chart 中的 `values.yaml` 文件：

```
mkdir -p /home/tidb/tidb-operator && \
helm inspect values pingcap/tidb-operator --version=<chart-version> > /home/tidb/tidb-operator/values-tidb-operator.yaml
```

注意：

`<chart-version>` 在后续文中代表 chart 版本，例如 `v1.0.0`，可以通过 `helm search -l tidb-operator` 查看当前支持的版本

(2) 配置 TiDB Operator：

TiDB Operator 里面会用到 `k8s.gcr.io/kube-scheduler` 镜像，如果下载不了该镜像，可以通过修改 `/home/tidb/tidb-operator/values-tidb-operator.yaml` 文件中的 `scheduler.kubeSchedulerImageName` 替换镜像。

(3) 安装 TiDB Operator：

```
helm install pingcap/tidb-operator --name=tidb-operator --namespace=tidb-admin --version=<chart-version> -f /home/tidb/tidb-operator/values-tidb-operator.yaml && \
kubectl get po -n tidb-admin -l app.kubernetes.io/name=tidb-operator
```

## 5. 自定义 TiDB Operator

通过修改 `/home/tidb/tidb-operator/values-tidb-operator.yaml` 中的配置自定义 TiDB Operator。后续文档使用 `values.yaml` 指代 `/home/tidb/tidb-operator/values-tidb-operator.yaml`。

TiDB Operator 有两个组件：

- `tidb-controller-manager`

- `tidb-scheduler`

这两个组件是无状态的，通过 `Deployment` 部署。你可以在 `values.yaml` 中自定义资源 `limit`、`request` 和 `replicas`。

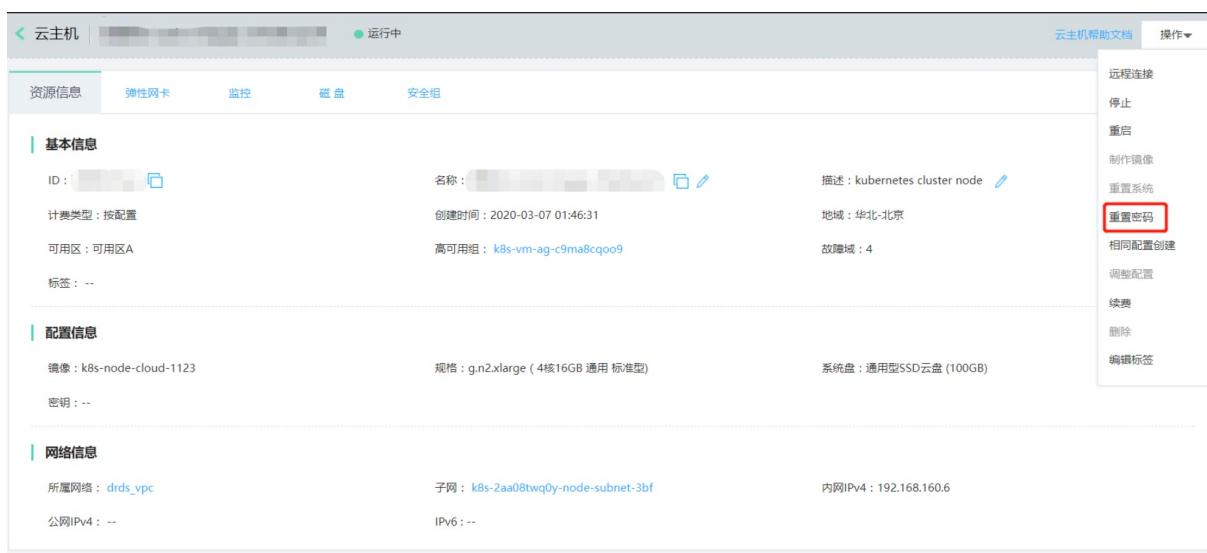
修改为 `values.yaml` 后，执行下面命令使配置生效：

```
helm upgrade tidb-operator pingcap/tidb-operator --version=<chart-version> -f /home/tidb/tidb-operator/values-tidb-operator.yaml
```

## 6. 设置 ulimit

TiDB 分布式数据库默认会使用很多文件描述符，工作节点和上面的 Docker 进程的 `ulimit` 必须设置大于等于 1048576：

- 重置京东云 Kubernetes 集群所有 Node 的登录密码



- 控制台登陆 Node
- 设置工作节点的 `ulimit` 值，详情可以参考[如何设置 ulimit](#)

```
sudo vim /etc/security/limits.conf
```

设置 root 账号的 `soft` 和 `hard` 的 `nofile` 大于等于 1048576。

- 设置 Docker 服务的 `ulimit`

```
sudo vim /etc/systemd/system/docker.service
```

设置 `LimitNOFILE` 大于等于 1048576

- 修改完后重启 Docker 服务

```
systemctl daemon-reload
systemctl restart docker
```

注意：

`LimitNOFILE` 需要显式设置为 1048576 或者更大，而不是默认的 `infinity`，由于 `systemd` 的 bug，`infinity` 在某些版本中指的是 65536。

## 7. 配置 TiDB 集群

## 配置 StorageClass

京东云为 Kubernetes 集群提供了自定义卷插件 [kubernetes.io/jdcloud-ebs](https://kubernetes.io/jdcloud-ebs)，将 provisioner 定义为京东云自定义卷插件，可以使用京东云云硬盘为 Kubernetes 集群提供持久化存储。目前，在 Kubernetes 集群服务中，提供三种 StorageClass：

```
kubectl get storageclass
NAME          PROVISIONER           AGE
default       kubernetes.io/jdcloud-ebs   39d
jdcloud-hdd   kubernetes.io/jdcloud-ebs   39d
jdcloud-ssd   kubernetes.io/jdcloud-ebs   39d
```

您也可以创建自定义的 StorageClass：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: mystorageclass-hdd1
provisioner: kubernetes.io/jdcloud-ebs
parameters:
  zones: cn-north-1a, cn-north-1b
  fstype: ext4
reclaimPolicy: Retain
```

参数说明：

(1) provisioner：设置参数值 kubernetes.io/jdcloud-ebs，且不可修改，标识使用京东云云硬盘 Provisioner 插件创建。

(2) reclaimPolicy：由 storage class 动态创建的 Persistent Volume 会在 reclaimPolicy 字段中指定回收策略，可以是 Delete 或者 Retain。如果 storageClass 对象被创建时没有指定 reclaimPolicy，默認為 Delete。

(3) parameters

type：设置参数值为 ssd(gp1、ssd.io1 或 hdd.std1)，分别对应京东云的通用型 SSD 云盘、性能型 SSD 云盘和容量型 HDD 云盘。

| StorageClass type | 云硬盘类型  | 容量范围          | 步长    |
|-------------------|--------|---------------|-------|
| hdd.std1          | 容量型hdd | [20-16000]GiB | 10GiB |
| ssd(gp1           | 通用型ssd | [20-16000]GiB | 10GiB |
| ssd.io1           | 性能型ssd | [20-16000]GiB | 10GiB |

fstype：设置文件系统类型，可选参数值为 xfs 和 ext4，如未指定 fstype，将使用 ext4 作为默认的文件系统类型，例如：fstype=ext4。

更多参数说明，参考参数说明 <https://docs.jdcloud.com/cn/jcs-for-kubernetes/deploy-storageclass>。

## 获取 Values 文件

通过下面命令获取待安装的 tidb-cluster chart 的 values.yaml 配置文件：

```
mkdir -p /home/tidb/<release-name> && \
helm inspect values pingcap/tidb-cluster --version=<chart-version> > /home/tidb/<release-name>/values-<release-name>.yaml
```

注意：

- `/home/tidb` 可以替换为你想用的目录。
- `release-name` 将会作为 Kubernetes 相关资源（例如 Pod，Service 等）的前缀名，可以起一个方便记忆的名字，要求全局唯一，通过 `helm ls -q` 可以查看集群中已经有的 `release-name`。
- `chart-version` 是 `tidb-cluster` chart 发布的版本，可以通过 `helm search -l tidb-cluster` 查看当前支持的版本。
- 下文会用 `values.yaml` 指代 `/home/tidb/<release-name>/values-<release-name>.yaml`。

## 集群拓扑

默认部署的集群拓扑是：3 个 PD Pod，3 个 TiKV Pod，2 个 TiDB Pod 和 1 个监控 Pod。在该部署拓扑下根据数据高可用原则，TiDB Operator 扩展调度器要求 Kubernetes 集群中至少有 3 个节点。如果 Kubernetes 集群节点个数少于 3 个，将会导致有一个 PD Pod 处于 Pending 状态，而 TiKV 和 TiDB Pod 也都不会被创建。

Kubernetes 集群节点个数少于 3 个时，为了使 TiDB 集群能启动起来，可以将默认部署的 PD 和 TiKV Pod 个数都减小到 1 个，或者将 `values.yaml` 中 `schedulerName` 改为 Kubernetes 内置调度器 `default-scheduler`。

警告：

`default-scheduler` 仅适用于演示环境，改为 `default-scheduler` 后，TiDB 集群的调度将无法保证数据高可用，另外一些其它特性也无法支持，例如 [TiDB Pod StableScheduling](#) 等。

其它更多配置参数请参考 [TiDB 集群部署配置文档](#)。

## 8. 部署 TiDB 集群

注意：

京东云硬盘支持创建的磁盘大小范围为 `[20-16000]GiB`，步长 `10G`，`values.yaml` 里 `PD`、`TiKV`、`Monitor`、`Drainer` 默认的磁盘大小不满足京东云盘的最小磁盘要求，需要修改为磁盘范围内的大小才可以正确创建 PV。

如果要使用京东云 Kubernetes 集成的负载均衡服务，需要修改 `values.yaml` 中 `tidb.service.annotations`，设置 `service.beta.kubernetes.io/jdccloud-load-balancer-spec`，因为 TiDB 的 Service 有两个 Port，因此需要配置两个 listeners：

```
tidb:
  service:
    type: LoadBalancer
    exposeStatus: true
    annotations:
      service.beta.kubernetes.io/jdccloud-load-balancer-spec: |
        version: "v1" # 【版本号】只支持 "v1"
        loadBalancerType: nlb # 【必填项】要创建的 JD LB 的类型，创建后不支持变更
        internal: true # true 表示 LB 实例不会绑定公网 IP，只内部使用；false 表示为外部服务，会绑定公网 IP。修改可能会触发 IP 的创建，绑定或者解绑，不会自动删除
        listeners: # 每个 port 对应的 LB 的 listener 的配置，数量必须和 ports 的数量一致
          - protocol: "tcp" # 修改可能触发删除重建，导致服务短暂中断，listener 的协议，alb: Tcp, Http,Https,Tls;nlb: Tcp;dnlb: Tcp
            connectionIdleTimeSeconds: 1800 # 连接超时时间，alb/nlb 有效
            backend: # 关于 JD LB 的 backend 的通用配置
              connectionDrainingSeconds: 300 # 【nlb】移除 target 前，连接的最大保持时间，默认 300s，取值范围 [0-3600] (Optional)
              sessionStickyTimeout: 300 # 【nlb】会话保持超时时间，sessionStickiness 开启时生效，默认 300s，取值范围 [1-3600] (Optional)
              algorithm: "IpHash" # 调度算法，取值范围为 [IpHash, RoundRobin, LeastConn] (取值范围的含义分别为：源 IP Hash, 加权轮询, 加权最小连接)，默认为 RoundRobin (加权轮询) (Optional), nlb : ; dnlb: ; alb
            - protocol: "tcp"
              connectionIdleTimeSeconds: 1800
              backend:
                connectionDrainingSeconds: 300
                sessionStickyTimeout: 300
                algorithm: "IpHash"
```

关于 LoadBalance 的更多参数参考官方文档 <https://docs.jdcloud.com/cn/jcs-for-kubernetes/deploy-service-new>

创建 Secret：

```
kubectl create secret generic <tidb-secretname> --from-literal=root=<password> --namespace=<namespace>
```

修改 `values.yaml` 的 `tidb` 下的 `passwordSecretName` 为 `<tidb-secretname>` 设置 TiDB 的初始密码。

`values.yaml` 文件修改好以后，使用以下命令创建 TiDB 集群：

```
helm install pingcap/tidb-cluster --name=<release-name> --namespace=<namespace> --version=<chart-version> -f /home/tidb/<release-name>/values-<release-name>.yaml
```

注意：

`namespace` 是命名空间，你可以起一个方便记忆的名字，比如和 `release-name` 相同的名称。

通过下面命令可以查看 Pod 状态：

```
kubectl get po -n <namespace> -l app.kubernetes.io/instance=<release-name>
```

单个 Kubernetes 集群中可以利用 TiDB Operator 部署管理多套 TiDB 集群，重复以上命令并将 `release-name` 替换成不同名字即可。不同集群既可以在相同 `namespace` 中，也可以在不同 `namespace` 中，可根据实际需求进行选择。

TiDB 集群创建好后，通过下面的命令查看 TiDB Service 的 ClusterIP：

```
$kubectl -n jddb get svc -l app.kubernetes.io/instance=jddb
```

| NAME                  | TYPE         | CLUSTER-IP      | EXTERNAL-IP    | PORT(S)                        |
|-----------------------|--------------|-----------------|----------------|--------------------------------|
| jddb-discovery        | ClusterIP    | 192.168.189.43  | <none>         | 10261/TCP                      |
| jddb-grafana          | NodePort     | 192.168.190.132 | <none>         | 3000:32445/TCP                 |
| jddb-monitor-reloader | NodePort     | 192.168.188.141 | <none>         | 9089:30732/TCP                 |
| jddb-pd               | ClusterIP    | 192.168.188.210 | <none>         | 2379/TCP                       |
| jddb-pd-peer          | ClusterIP    | None            | <none>         | 2380/TCP                       |
| jddb-prometheus       | NodePort     | 192.168.186.63  | <none>         | 9090:30415/TCP                 |
| jddb-tidb             | LoadBalancer | 192.168.188.201 | 192.168.176.10 | 4000:30487/TCP,10080:32045/TCP |
| jddb-tidb-peer        | ClusterIP    | None            | <none>         | 10080/TCP                      |
| jddb-tikv-peer        | ClusterIP    | None            | <none>         | 20160/TCP                      |

其中 `jddb-tidb` 即是 TiDB 的 Service，集群内部的 Pod 可以通过 CLUSTER-IP 访问 TiDB 服务，同 VPC 下的云主机可以通过 EXTERNAL-IP 访问 TiDB 集群。

### 1.2.3.2 私有云部署

本章将介绍如何在私有云上部署 TiDB 集群，包含以下内容：

- 集群环境、资源需求
- PV 配置
- 部署 TiDB Operator
- 部署 TiDB 集群
- 节点维护
- 删除 TiDB 集群

### 1.2.3.2.1 集群环境资源需求

本文介绍在 Kubernetes 上部署 TiDB 集群的软硬件环境需求。

## 1. 软件版本要求

| 软件名称       | 版本                                |
|------------|-----------------------------------|
| Docker     | Docker CE 18.09.6                 |
| Kubernetes | v1.12.5+                          |
| CentOS     | CentOS 7.6，内核要求为 3.10.0-957 或之后版本 |

## 2. 内核参数设置

| 配置项                               | 设置值     |
|-----------------------------------|---------|
| net.core.somaxconn                | 32768   |
| vm.swappiness                     | 0       |
| net.ipv4.tcp_syncookies           | 0       |
| net.ipv4.ip_forward               | 1       |
| fs.file-max                       | 1000000 |
| fs.inotify.max_user_watches       | 1048576 |
| fs.inotify.max_user_instances     | 1024    |
| net.ipv4.conf.all.rp_filter       | 1       |
| net.ipv4.neigh.default.gc_thresh1 | 80000   |
| net.ipv4.neigh.default.gc_thresh2 | 90000   |
| net.ipv4.neigh.default.gc_thresh3 | 100000  |

同时还需要关闭每个部署 Kubernetes 节点的 swap，执行如下命令：

```
swapoff -a
```

执行如下命令检查 swap 是否已经关闭：

```
free -m
```

如果执行命令后输出显示 swap 一列全是 0，则表明 swap 已经关闭。

此外，为了永久性地关闭 swap，还需要将 `/etc/fstab` 中 swap 相关的条目全部删除。

在上述内容都设置完成后，还需要检查是否给机器配置了 [SMP IRQ Affinity](#)，也就是将各个设备对应的中断号分别绑定到不同的 CPU 上，以防止所有中断请求都落在同一个 CPU 上而引发性能瓶颈。对于 TiDB 集群来说，网卡处理包的速度对集群的吞吐率影响很大。因此下文主要描述如何将网卡中断号绑定到特定的 CPU 上，充分利用多核的优势来提高集群的吞吐率。首先可以通过以下命令来查看网卡对应的中断号：

```
cat /proc/interrupts|grep <iface-name>|awk '{print $1,$NF}'
```

以上命令输出的第一列是中断号，第二列是设备名称。如果是多队列网卡，上面的命令会显示多行信息，网卡的每个队列对应一个中断号。通过以下命令可以查看该中断号被绑定到哪个 CPU 上：

```
cat /proc/irq/<irq_num>/smp_affinity
```

上面命令输出 CPU 序号对应的十六进制值。输出结果欠直观。具体计算方法可参见 [SMP IRQ Affinity](#) 文档。

```
cat /proc/irq/<irq_num>/smp_affinity_list
```

上面命令输出 CPU 序号对应的十进制值，输出结果较为直观。

如果多队列网卡对应的所有中断号都已被绑定到不同的 CPU 上，那么该机器的 SMP IRQ Affinity 配置是正确的。如果中断号都落在同一个 CPU 上，则需要进行调整。调整的方式有以下两种：

- 方法一：开启 [irqbalance](#) 服务。在 centos7 系统上的开启命令如下：

```
systemctl start irqbalance
```

- 方法二：禁用 [irqbalance](#)，自定义中断号和 CPU 的绑定关系。详情参见脚本 [set\\_irq\\_affinity.sh](#)。

上文所描述的是处理多队列网卡和多核心的场景。单队列网卡和多核的场景则有不同的处理方式。在这种场景下，可以使用 [RPS/RFS](#) 在软件层面模拟实现硬件的网卡多队列功能 (RSS)。此时不能使用方法一所述的 [irqbalance](#) 服务，而是通过使用方法二提供的脚本来设置 RPS。RFS 的配置可以参考[这里](#)。

### 3. 硬件和部署要求

与使用 binary 方式部署 TiDB 集群一致，要求选用 Intel x86-64 架构的 64 位通用硬件服务器，使用万兆网卡。关于 TiDB 集群在物理机上的具体部署需求，参考 [TiDB 软件和硬件环境建议配置](#)。

对于服务器 disk、memory、CPU 的选择要根据对集群的容量规划以及部署拓扑来定。线上 Kubernetes 集群部署为了保证高可用，一般需要部署三个 master 节点、三个 etcd 节点以及若干个 worker 节点。同时，为了充分利用机器资源，master 节点一般也充当 worker 节点（也就是 master 节点上也可以调度负载）。通过 kubelet 设置[预留资源](#)来保证机器上的系统进程以及 Kubernetes 的核心进程在工作负载很高的情况下仍然有足够的资源来运行，从而保证整个系统的稳定。

下面按 3 Kubernetes master + 3 etcd + 若干 worker 节点部署方案进行分析。Kubernetes 的多 master 节点高可用部署可参考[官方文档](#)。

### 4. Kubernetes 系统资源要求

- 每台机器需要一块比较大的 SAS 盘（至少 1T），这块盘用来存 Docker 和 kubelet 的数据目录。Docker 的数据主要包括镜像和容器日志数据，kubelet 主要占盘的数据是 [emptyDir](#) 所使用的数据。
- 如果需要部署 Kubernetes 集群的监控系统，且监控数据需要落盘，则也需要考虑为 Prometheus 准备一块 SAS 盘，后面日志监控系统也需要大的 SAS 盘，同时考虑到机器采购最好是同构的这一因素，因此每台机器最好有两块大的 SAS 盘。

注意：

生产环境建议给这两种类型的盘做 RAID5，至于使用多少块来做 RAID5 可自己决定。

- etcd 的分布建议是和 Kubernetes master 节点保持一致，即有多少个 master 节点就部署多少个 etcd 节点。etcd 数据建议使用 SSD 盘存放。

### 5. TiDB 集群资源需求

TiDB 分布式数据库由 PD、TiKV、TiDB 三个组件组成，在做容量规划的时候一般按照可以支持多少套 TiDB 集群来算。这里按照标准的 TiDB 集群（3 个 PD + 3 个 TiKV + 2 个 TiDB）来算，下面是对每个组件规划的一种建议：

- PD 组件：PD 占用资源较少，这种集群规模下分配 2C 4GB 即可，占用少量本地盘。

为了便于管理，可以将所有集群的 PD 都放在 master 节点，比如需要支持 5 套 TiDB 集群，则可以规划 3 个 master 节点，每个节点支持部署 5 个 PD 实例，5 个 PD 实例使用同一块 SSD 盘即可（两三百 GB 的盘即可）。通过 bind mount 的方式在这块 SSD 上创建 5 个目录作为挂载点，操作方式见 [Sharing a disk filesystem by multiple filesystem PVs](#)。

如果后续要添加更多机器支持更多的 TiDB 集群，可以在 master 上用这种方式继续增加 PD 实例。如果 master 上资源耗尽，可以找其它的 worker 节点机器用同样的方式添加 PD 实例。这种方式的好处就是方便规划和管理 PD 实例，坏处就是由于 PD 实例过于集中，这些机器中如果有两台宕机会导致所有的 TiDB 集群不可用。

因此建议从所有集群里面的机器都拿出一块 SSD 盘像 master 节点一样提供 PD 实例。比如总共 7 台机器，要支持 7 套 TiDB 标准集群的情况下，则需要每台机器上都能支持部署 3 个 PD 实例，如果后续有集群需要通过扩容机器增加容量，也只需要在新的机器上创建 PD 实例。

- TiKV 组件：因为 TiKV 组件的性能很依赖磁盘 I/O 且数据量一般较大，因此建议每个 TiKV 实例独占一块 NVMe 的盘，资源配置为 8C 32GB。如果想要在一个机器上支持部署多个 TiKV 实例，则建议参考这些参数去选择合适的机器，同时在规划容量的时候应当预留出足够的 buffer。
- TiDB 组件：TiDB 组件因为不占用磁盘，因此在规划的时候只需要考虑其占用的 CPU 和内存资源即可，这里也按 8C 32 GB 的容量来计算。

## 6. TiDB 集群规划示例

通过上面的分析，这里给出一个支持部署 5 套规模为 3 个 PD + 3 个 TiKV + 2 个 TiDB 集群的例子，其中 PD 配置为 2C 4GB，TiDB 配置为 8C 32GB，TiKV 配置为 8C 32GB。Kubernetes 节点有 7 个，其中有 3 个节点既是 master 又是 worker 节点，另外 4 个是纯 worker 节点。各节点上部署组件情况如下：

- 每台 master 节点：

- 1 etcd (2C 4GB) + 2 PD (2 \* 2C 2 \* 4GB) + 3 TiKV (3 \* 8C 3 \* 32GB) + 1 TiDB (8C 32GB)，总共是 38C 140GB
- 两块 SSD 盘，一块给 etcd，另外一块给 2 个 PD 实例
- 做了 RAID5 的 SAS 盘，给 Docker 和 kubelet 做数据盘
- 三块 NVMe 盘给 TiKV 实例

- 每台 worker 节点：

- 3 PD (3 \* 2C 3 \* 4GB) + 2 TiKV (2 \* 8C 2 \* 32GB) + 2 TiDB (2 \* 8C 2 \* 32GB)，总共是 38C 140GB
- 一块 SSD 盘给三个 PD 实例
- 做了 RAID5 的 SAS 盘，给 Docker 和 kubelet 做数据盘
- 两块 NVMe 盘给 TiKV 实例

从上面的分析来看，要支持 5 套 TiDB 集群容量共需要 7 台物理机，其中 3 台为 master 兼 worker 节点，其余 4 台为 worker 节点，机器配置需求如下：

- master 兼 worker 节点：48C 192GB；2 块 SSD 盘，一块做了 RAID5 的 SAS 盘，三块 NVMe 盘
- worker 节点：48C 192GB；1 块 SSD 盘，一块做了 RAID5 的 SAS 盘，两块 NVMe 盘

使用上面的机器配置，除去各个组件占用的资源外，还有比较多的预留资源。如果要考虑加监控和日志组件，则可以用同样的方法去规划需要采购的机器类型以及配置。

另外，在生产环境的使用上尽量不要在 master 节点部署 TiDB 实例，或者尽可能少地部署 TiDB 实例。这里的主要考虑点是网卡带宽，因为 master 节点网卡满负荷工作会影响到 worker 节点和 master 节点之间的心跳信息汇报，导致比较严重的问题。



## 1.2.3.2.2 Kubernetes 上的持久化存储配置

### 1. 背景介绍

TiDB 分布式数据库中 PD、TiKV、监控等组件以及 TiDB Binlog 和备份等工具都需要使用将数据持久化的存储。Kubernetes 上的数据持久化需要使用 [PersistentVolume \(PV\)](#)。Kubernetes 提供多种[存储类型](#)，主要分为两大类：

- 网络存储

存储介质不在当前节点，而是通过网络方式挂载到当前节点。一般有多副本冗余提供高可用保证，在节点出现故障时，对应网络存储可以再挂载到其它节点继续使用。

- 本地存储

存储介质在当前节点，通常能提供比网络存储更低的延迟，但没有多副本冗余，一旦节点出故障，数据就有可能丢失。如果是 IDC 服务器，节点故障可以一定程度上对数据进行恢复，但公有云上使用本地盘的虚拟机在节点故障后，数据是无法找回的。

PV 一般由系统管理员或 volume provisioner 自动创建，PV 与 Pod 是通过 [PersistentVolumeClaim \(PVC\)](#) 进行关联的。普通用户在使用 PV 时并不需要直接创建 PV，而是通过 PVC 来申请使用 PV，对应的 volume provisioner 根据 PVC 创建符合要求的 PV，并将 PVC 与该 PV 进行绑定。

**警告：**

为了数据安全，任何情况下都不要直接删除 PV，除非对 volume provisioner 原理非常清楚。

### 2. TiDB 分布式数据库推荐存储类型

TiKV 自身借助 Raft 实现了数据复制，出现节点故障后，PD 会自动进行数据调度补齐缺失的数据副本，同时 TiKV 要求存储有较低的读写延迟，所以生产环境强烈推荐使用本地 SSD 存储。

PD 同样借助 Raft 实现了数据复制，但作为存储集群元信息的数据库，并不是 IO 密集型应用，所以一般本地普通 SAS 盘或网络 SSD 存储（例如 AWS 上 gp2 类型的 EBS 存储卷，GCP 上的持久化 SSD 盘）就可以满足要求。

监控组件以及 TiDB Binlog、备份等工具，由于自身没有做多副本冗余，所以为保证可用性，推荐用网络存储。其中 TiDB Binlog 的 pump 和 drainer 组件属于 IO 密集型应用，需要较低的读写延迟，所以推荐用高性能的网络存储（例如 AWS 上的 io1 类型的 EBS 存储卷，GCP 上的持久化 SSD 盘）。

在利用 TiDB Operator 部署 TiDB 分布式数据库或者备份工具的时候，需要持久化存储的组件都可以通过 values.yaml 配置文件中对应的 `storageClassName` 设置存储类型。不设置时默认都使用 `local-storage`。

### 3. 网络 PV 配置

Kubernetes 1.11 及以上的版本支持[网络 PV 的动态扩容](#)，但用户需要为相应的 `StorageClass` 开启动态扩容支持。

```
kubectl patch storageclass <storage-class-name> -p '{"allowVolumeExpansion": true}'
```

开启动态扩容后，通过下面方式对 PV 进行扩容：

- (1) 修改 PVC 大小

假设之前 PVC 大小是 10 Gi, 现在需要扩容到 100 Gi

```
```shell
kubectl patch pvc -n <namespace> <pvc-name> -p '{"spec": {"resources": {"requests": {"storage": "100Gi"}}}}'
```

```

## (2) 查看 PV 扩容成功

扩容成功后，通过 `kubectl get pvc -n <namespace> <pvc-name>` 显示的大小仍然是初始大小，但查看 PV 大小会显示已经扩容到预期的大小。

```
```shell
kubectl get pv | grep <pvc-name>
```

```

## 4. 本地 PV 配置

Kubernetes 当前支持静态分配的本地存储。可使用 [local-static-provisioner](#) 项目中的 `local-volume-provisioner` 程序创建本地存储对象。创建流程如下：

(1) 参考 Kubernetes 提供的[操作文档](#)，在集群节点中预分配本地存储。

(2) 部署 `local-volume-provisioner` 程序。

```
```shell
kubectl apply -f https://raw.githubusercontent.com/pingcap/tidb-operator/master/manifests/local-dind/local-volume-provisioner.yaml
```

```

通过下面命令查看 Pod 和 PV 状态：

```
```shell
kubectl get po -n kube-system -l app=local-volume-provisioner && \
kubectl get pv | grep local-storage
```

```

`local-volume-provisioner` 会为发现目录 (discovery directory) 下的每一个挂载点创建一个 PV。注意，在 GKE 上，默认只能创建大小为 375 GiB 的本地卷。

更多信息，可参阅 [Kubernetes 本地存储和 local-static-provisioner 文档](#)。

## 最佳实践

- Local PV 的路径是本地存储卷的唯一标识符。为了保证唯一性并避免冲突，推荐使用设备的 UUID 来生成唯一的路径
- 如果想要 IO 隔离，建议每个存储卷使用一块物理盘会比较恰当，在硬件层隔离
- 如果想要容量隔离，建议每个存储卷一个分区或者每个存储卷使用一块物理盘来实现

更多信息，可参阅 local-static-provisioner 的[最佳实践文档](#)。

## 示例

如果监控、TiDB Binlog、备份等组件都使用本地盘存储数据，可以挂载普通 SAS 盘，并分别创建不同的 `StorageClass` 供这些组件使用，具体操作如下：

- 给监控数据使用的盘，可以参考[步骤](#)挂载盘，创建目录，并将新建的目录以 bind mount 方式挂载到 `/mnt/disks` 目录，后续创建 `local-storage` `StorageClass`。

注意：

该步骤中创建的目录个数取决于规划的 TiDB 集群数量。1 个目录会对应创建 1 个 PV。每个 TiDB 集群的监控数据会使用 1 个 PV。

- 给 TiDB Binlog 和备份数据使用的盘，可以参考[步骤](#)挂载盘，创建目录，并将新建的目录以 bind mount 方式挂载到 `/mnt/backup` 目录，后续创建 `backup-storage` `StorageClass`。

注意：

该步骤中创建的目录个数取决于规划的 TiDB 集群数量、每个集群内的 Pump 数量及备份方式。1 个目录会对应创建 1 个 PV。每个 Pump 会使用 1 个 PV，每个 drainer 会使用 1 个 PV，每次 [Ad-hoc 全量备份](#) 会使用 1 个 PV，所有 [定时全量备份](#) 会共用 1 个 PV。

- 给 PD 数据使用的盘，可以参考[步骤](#)挂载盘，创建目录，并将新建的目录以 bind mount 方式挂载到 `/mnt/sharedssd` 目录，后续创建 `shared-ssd-storage` `StorageClass`。

注意：

该步骤中创建的目录个数取决于规划的 TiDB 集群数量及每个集群内的 PD 数量。1 个目录会对应创建 1 个 PV。每个 PD 会使用一个 PV。

- 给 TiKV 数据使用的盘，可通过[普通挂载](#)方式将盘挂载到 `/mnt/ssd` 目录，后续创建 `ssd-storage` `StorageClass`。

盘挂载完成后，需要根据上述磁盘挂载情况修改 [local-volume-provisioner yaml 文件](#)，配置发现目录并创建必要的 `StorageClass`。以下是根据上述挂载修改的 yaml 文件示例：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: "local-storage"
provisioner: "kubernetes.io/no-provisioner"
volumeBindingMode: "WaitForFirstConsumer"
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: "ssd-storage"
provisioner: "kubernetes.io/no-provisioner"
volumeBindingMode: "WaitForFirstConsumer"
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: "shared-ssd-storage"
provisioner: "kubernetes.io/no-provisioner"
volumeBindingMode: "WaitForFirstConsumer"
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: "backup-storage"
provisioner: "kubernetes.io/no-provisioner"
volumeBindingMode: "WaitForFirstConsumer"
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: local-provisioner-config
  namespace: kube-system
data:
  nodeLabelsForPV: |
    - kubernetes.io/hostname
  storageClassMap: |
    shared-ssd-storage:
      hostDir: /mnt/sharedssd
      mountDir: /mnt/sharedssd
    ssd-storage:
      hostDir: /mnt/ssd
      mountDir: /mnt/ssd
    local-storage:
      hostDir: /mnt/disks
      mountDir: /mnt/disks
    backup-storage:
```

```

hostDir: /mnt/backup
mountDir: /mnt/backup
---
.....
volumeMounts:
.....
- mountPath: /mnt/ssd
  name: local-ssd
  mountPropagation: "HostToContainer"
- mountPath: /mnt/sharedssd
  name: local-sharedssd
  mountPropagation: "HostToContainer"
- mountPath: /mnt/disks
  name: local-disks
  mountPropagation: "HostToContainer"
- mountPath: /mnt/backup
  name: local-backup
  mountPropagation: "HostToContainer"
volumes:
.....
- name: local-ssd
  hostPath:
    path: /mnt/ssd
- name: local-sharedssd
  hostPath:
    path: /mnt/sharedssd
- name: local-disks
  hostPath:
    path: /mnt/disks
- name: local-backup
  hostPath:
    path: /mnt/backup
.....

```

最后通过 `kubectl apply` 命令部署 `local-volume-provisioner` 程序。

```
kubectl apply -f https://raw.githubusercontent.com/pingcap/tidb-operator/master/manifests/local-dind/local-volume-provisioner.yaml
```

后续创建 TiDB 集群或备份等组件的时候，再配置相应的 `StorageClass` 供其使用。

## 5. 数据安全

一般情况下 PVC 在使用完删除后，与其绑定的 PV 会被 provisioner 清理回收再放入资源池中被调度使用。为避免数据意外丢失，可在全局配置 `StorageClass` 的回收策略 (reclaim policy) 为 `Retain` 或者只将某个 PV 的回收策略修改为 `Retain`。`Retain` 模式下，PV 不会自动被回收。

- 全局配置

`StorageClass` 的回收策略一旦创建就不能再修改，所以只能在创建时进行设置。如果创建时没有设置，可以再创建相同 provisioner 的 `StorageClass`，例如 GKE 上默认的 pd 类型的 `StorageClass` 默认保留策略是 `Delete`，可以再创建一个名为 `pd-standard` 的保留策略是 `Retain` 的存储类型，并在创建 TiDB 集群时将相应组件的 `storageClassName` 修改为 `pd-standard`。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: pd-standard
parameters:
  type: pd-standard
provisioner: kubernetes.io/gce-pd
reclaimPolicy: Retain
volumeBindingMode: WaitForFirstConsumer
```

- 配置单个 PV

```
kubectl patch pv <pv-name> -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

注意：

TiDB Operator 默认会自动将 PD 和 TiKV 的 PV 保留策略修改为 `Retain` 以确保数据安全。

PV 保留策略是 `Retain` 时，如果确认某个 PV 的数据可以被删除，需要通过下面的操作来删除 PV 以及对应的数据：

(1) 删除 PV 对应的 PVC 对象：

```
```shell
kubectl delete pvc <pvc-name> --namespace=<namespace>
```

```

(2) 设置 PV 的保留策略为 `Delete`，PV 会被自动删除并回收：

```
```shell
kubectl patch pv <pv-name> -p '{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
```

```

要了解更多关于 PV 的保留策略可参考[修改 PV 保留策略](#)。

### 1.2.3.2.3 在 Kubernetes 上部署 TiDB Operator

本文介绍如何在 Kubernetes 上部署 TiDB Operator。

## 1. 准备环境

TiDB Operator 部署前，请确认以下软件需求：

- Kubernetes v1.12 或者更高版本
- Helm 版本  $\geq v2.11.0 \&& < v3.0.0$

注意：

- 尽管 TiDB Operator 可以使用网络卷持久化 TiDB 数据，TiDB 数据自身会存多副本，再走额外的网络卷性能会受到很大影响。强烈建议搭建[本地卷](#)以提高性能。
- 跨多可用区的网络卷需要 Kubernetes v1.12 或者更高版本。

## 安装 Helm 服务端

在集群中应用 helm 服务端组件 tiller 所需的 RBAC 规则并安装 tiller：

```
kubectl apply -f https://raw.githubusercontent.com/pingcap/tidb-operator/master/manifests/tiller-rbac.yaml && \
helm init --service-account=tiller --upgrade
```

通过下面命令确认 tiller Pod 进入 running 状态：

```
kubectl get po -n kube-system -l name=tiller
```

如果 Kubernetes 集群没有启用 RBAC，那么可以直接使用下列命令安装 tiller：

```
helm init --upgrade
```

Kubernetes 应用在 Helm 中被打包为 chart。PingCAP 维护的 helm chart 仓库是 <https://charts.pingcap.org/>，可以通过下面的命令添加该仓库：

```
helm repo add pingcap https://charts.pingcap.org/
```

添加完成后，可以使用 helm search 搜索 PingCAP 提供的 chart：

```
helm search pingcap -l
```

| NAME                  | CHART VERSION | APP VERSION | DESCRIPTION                             |
|-----------------------|---------------|-------------|---|
| pingcap/tidb-backup   | v1.0.0        |             | A Helm chart for TiDB Backup or Restore |
| pingcap/tidb-cluster  | v1.0.0        |             | A Helm chart for TiDB Cluster           |
| pingcap/tidb-operator | v1.0.0        |             | tidb-operator Helm chart for Kubernetes |

当新版本的 chart 发布后，可以使用 helm repo update 命令更新本地对于仓库的缓存：

```
helm repo update
```

## 2. 安装 TiDB Operator

TiDB Operator 使用 [CRD \(Custom Resource Definition\)](#) 扩展 Kubernetes，所以要使用 TiDB Operator，必须先创建 `TidbCluster` 自定义资源类型。只需要在你的 Kubernetes 集群上创建一次即可：

```
kubectl apply -f https://raw.githubusercontent.com/pingcap/tidb-operator/master/manifests/crd.yaml && \
kubectl get crd tidbclusters.pingcap.com
```

创建 `TidbCluster` 自定义资源类型后，接下来在 Kubernetes 集群上安装 TiDB Operator。

(1) 获取你要安装的 `tidb-operator` chart 中的 `values.yaml` 文件：

```
```shell
mkdir -p /home/tidb/tidb-operator && \
helm inspect values pingcap/tidb-operator --version=<chart-version> > /home/tidb/tidb-operator/values-tidb-operator.yaml
```

> **注意：**
>
> `<chart-version>` 在后续文档中代表 chart 版本，例如 `v1.0.0`，可以通过 `helm search -l tidb-operator` 查看当前支持的版本。
```

(2) 配置 TiDB Operator

TiDB Operator 里面会用到 `k8s.gcr.io/kube-scheduler` 镜像，如果下载不了该镜像，可以修改 `/home/tidb/tidb-operator/values-tidb-operator.yaml` 文件中的 `scheduler.kubeSchedulerImageName` 为 `registry.cn-hangzhou.aliyuncs.com/google\_containers/kube-scheduler`。

(3) 安装 TiDB Operator

```
```shell
helm install pingcap/tidb-operator --name=tidb-operator --namespace=tidb-admin --version=<chart-version> -f /home/tidb/tidb-operator/values-tidb-operator.yaml && \
kubectl get po -n tidb-admin -l app.kubernetes.io/name=tidb-operator
```

```

## 3. 自定义 TiDB Operator

通过修改 `/home/tidb/tidb-operator/values-tidb-operator.yaml` 中的配置自定义 TiDB Operator。后续文档使用 `values.yaml` 指代 `/home/tidb/tidb-operator/values-tidb-operator.yaml`。

TiDB Operator 有两个组件：

- `tidb-controller-manager`
- `tidb-scheduler`

这两个组件是无状态的，通过 `Deployment` 部署。你可以在 `values.yaml` 中自定义资源 `limit`、`request` 和 `replicas`。

修改为 `values.yaml` 后，执行下面命令使配置生效：

```
helm upgrade tidb-operator pingcap/tidb-operator --version=<chart-version> -f /home/tidb/tidb-operator/values-tidb-operator.yaml
```

## 4. 手动下载 chart 安装

如果安装环境无法访问 PingCAP chart 仓库，可以通过 `wget http://charts.pingcap.org/tidb-cluster-<chart-version>.tgz` 下载指定版本 chart，进行安装。

假设下载的 chart 放在 `/home/tidb/` 目录，版本为 `v1.0.6`：

```
cd /home/tidb/ && tar -xvf tidb-operator-chart-v1.0.6.tgz
```

修改 /home/tidb/tidb-operator/values.yaml 文件，安装 TiDB Operator：

```
helm install /home/tidb/tidb-operator --namespace=tidb-admin --name=tidb-operator -f /home/tidb/tidb-operator/values.yaml
```

可以通过下面命令验证 TiDB Operator 运行状态：

```
kubectl get pods -n tidb-admin
```

| NAME                                     | READY | STATUS  | RESTARTS | AGE |
|--|-------|---------|----------|-----|
| tidb-controller-manager-85d8d498bf-2n8km | 1/1   | Running | 0        | 19s |
| tidb-scheduler-7c67d6c77b-qd54r          | 2/2   | Running | 0        | 19s |

以上信息显示 TiDB Operator 运行正常。

### 1.2.3.2.4 部署 TiDB 集群

#### (1) 下载 TiDB Cluster 的 helm chart 文件

```
# mkdir -p /root/charts/
从 https://github.com/pingcap/tidb-operator/releases 下载 tidb-cluster-chart-v1.0.6.tgz 文件放到 /root/charts/ 路径
下
```

#### (2) 安装 TiDB Cluster

```
# cd /root/charts/ && tar xvf tidb-cluster-chart-v1.0.6.tgz
# helm install --namespace dba-test --name=test /root/charts/tidb-cluster -f /root/charts/tidb-cluster/values.yaml
1
NAME: test
LAST DEPLOYED: Sat Mar 7 05:27:57 2020
NAMESPACE: dba-test
STATUS: DEPLOYED
...
...
```

以上信息显示 TiDB Cluster 部署正常。

#### (3) 观察 TiDB Cluster 所有 Pod 状态

```
# kubectl get pods -n dba-test
NAME                      READY   STATUS    RESTARTS   AGE
test-discovery-854fb5b46c-hbg4q   1/1    Running   0          4m41s
test-monitor-66589f9748-q28lp    3/3    Running   0          4m41s
test-pd-0                     1/1    Running   1          4m40s
test-pd-1                     1/1    Running   0          4m40s
test-pd-2                     1/1    Running   0          4m40s
test-tidb-0                   2/2    Running   0          2m13s
test-tidb-1                   2/2    Running   0          2m13s
test-tikv-0                   1/1    Running   0          2m45s
test-tikv-1                   1/1    Running   0          2m45s
test-tikv-2                   1/1    Running   0          2m45s
```

以上信息显示 TiDB Cluster 所有 Pod 全部运行正常。

#### (4) 访问 TiDB Cluster

```
# kubectl get svc -n dba-test
NAME           TYPE      CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
test-discovery ClusterIP  10.102.244.238  <none>       10261/TCP    3m58s
test-grafana   NodePort   10.104.130.193  <none>       3000:32326/TCP 3m58s
test-monitor-reloader NodePort   10.106.105.144  <none>       9089:30818/TCP 3m58s
test-pd         ClusterIP  10.96.183.196   <none>       2379/TCP     3m58s
test-pd-peer   ClusterIP  None            <none>       2380/TCP     3m58s
test-prometheus NodePort   10.107.17.45   <none>       9090:31800/TCP 3m58s
test-tidb      NodePort   10.104.37.71   <none>       4000:30169/TCP,10080:30286/TCP 3m58s
test-tidb-peer ClusterIP  None            <none>       10080/TCP    90s
test-tikv-peer ClusterIP  None            <none>       20160/TCP    2m2s
```

找到 test-tidb 这个 Service 的 CLUSTER-IP，通过其访问 TiDB Cluster：

```
# mysql -h 10.104.37.71 -uroot -P4000
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.7.25-TiDB-v3.0.5 MySQL Community Server (Apache License 2.0)

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

访问 TiDB Cluster 成功。

### 1.2.3.2.5 维护 TiDB 集群所在的 Kubernetes 节点

#### 1. TiDB 与 PD 所在服务器异常下线

由于节点维护需要至少 4 台服务器才能操作完成，本节基于 kind 模拟 5 个节点，讲解第 5 个 node 服务器故障时的维护方法。

PD 和 TiDB 实例的迁移较快，可以采取主动驱逐实例到其它节点上的策略进行节点维护：

```
# kubectl get pod --all-namespaces -o wide | grep worker5|grep dba-test
dba-test      test-pd-2           1/1     Running   0          5m18s   10.244.5.4   kind-wor
ker5         <none>            <none>
```

以上显示 kind-worker5 节点运行了一个 PD 实例，使用 `kubectl cordon` 命令防止新的 Pod 调度到待维护节点上：

```
# kubectl cordon kind-worker5
node/kind-worker5 cordoned
```

使用 `kubectl drain` 命令将待维护节点上的数据库实例迁移到其它节点上：

```
# kubectl drain kind-worker5 --ignore-daemonsets --delete-local-data
node/kind-worker5 already cordoned
WARNING: ...
pod/test-pd-2 evicted
```

等待一会儿，检查 kind-worker5 节点上的 Pod：

```
# kubectl get pods -n dba-test|grep kind-worker5
```

无输出，表明 kind-worker5 节点上的 Pod 已全部迁移走出去。再观察 TiDB 集群的 Pod 状态：

```
# kubectl get pods -n dba-test
NAME                  READY   STATUS    RESTARTS   AGE
test-discovery-854fb5b46c-c81ng  1/1     Running   0          21m
test-monitor-59468bcd58-btbpc   3/3     Running   0          10m
test-pd-0                1/1     Running   2          21m
test-pd-1                1/1     Running   0          21m
test-pd-2                1/1     Running   0          4m50s
test-tidb-0              2/2     Running   0          19m
test-tidb-1              2/2     Running   0          19m
test-tikv-0               1/1     Running   0          20m
test-tikv-1               1/1     Running   0          20m
test-tikv-2               1/1     Running   0          20m
```

集群所有 Pod 已在正常运行，集群已恢复正常。

此时，分两种情况：

1. 短期维护（对节点无破坏性）；
2. 长期维护，服务器需要长时间修复，甚至需要重装系统。

短期维护场景下，维护完毕后，将节点解除调度限制即可：

```
# kubectl uncordon kind-worker5
```

长期维护场景下，将节点下线，修复之后，再由 K8s 运维专家将服务器上线到 K8s 集群：

```
# kubectl delete node kind-worker5
node "kind-worker5" deleted
```

## 2. TiKV 所在服务器异常下线

本节讲解 TiKV node 需要下线维护的场景。

注意：至少要确保有 4 个 TiKV 实例正常运行才能操作成功（默认副本数为3）。

使用 `kubectl cordon` 命令防止新的 Pod 调度到待维护节点上：

```
# kubectl cordon kind-worker4
node/kind-worker4 cordoned
```

查看待维护节点上的 TiKV 实例：

```
# kubectl get pods -n dba-test -owide|grep tikv|grep kind-worker4
test-tikv-2           1/1     Running   0      5m28s   10.244.1.7   kind-worker4   <none>       <
none>
```

查看 TiKV 实例的 `store-id`：

```
# kubectl get tc test -ojson -n dba-test| jq '.status.tikv.stores | .[] | select (.podName == "test-tikv-2") | .id'
"115"
```

开启 PD 访问：

```
# nohup kubectl port-forward svc/test-pd 2379:2379 -n dba-test &
[1] 8968
```

使用 `pd-ctl` 下线 TiKV 实例：

```
# pd-ctl -d store delete 115
Success!
```

等待 TiKV store 状态（`state_name`）转化为 `Tombstone`：

```
# pd-ctl -d store 115|grep state_name
"state_name": "Tombstone"
```

接下来，还需要解除 TiKV 实例与节点本地盘的绑定：

查询 Pod 使用的 PVC：

```
# kubectl get -n dba-test pod test-tikv-2 -ojson | jq '.spec.volumes | .[] | select (.name == "tikv") | .persistentVolumeClaim.claimName'
"tikv-test-tikv-2"
```

删除该 PVC：

```
# kubectl delete pvc/tikv-test-tikv-2 -n dba-test
persistentvolumeclaim "tikv-test-tikv-2" deleted
```

删除 TiKV 实例：

```
# kubectl delete pod/test-tikv-2 -n dba-test
pod "test-tikv-2" deleted
```

观察该 TiKV 实例是否正常调度到其它节点上：

```
# kubectl get pods -n dba-test -owide|grep tikv
test-tikv-0           1/1     Running   0          29m    10.244.2.6   kind-worker2   <none>      <no
ne>
test-tikv-1           1/1     Running   0          29m    10.244.4.6   kind-worker3   <none>      <no
ne>
test-tikv-2           1/1     Running   0          34s    10.244.3.6   kind-worker    <none>      <no
ne>
test-tikv-3           1/1     Running   0          25m    10.244.3.5   kind-worker    <none>      <no
ne>
```

此时，test-tikv-2 已经从 kind-worker4 迁移到 kind-worker了。

此时，分两种情况：

1. 短期维护（对节点无破坏性）；
2. 长期维护，服务器需要长时间修复，甚至需要重装系统。

短期维护场景，维护完毕后，将节点解除调度限制即可：

```
# kubectl uncordon kind-worker4
```

长期维护场景，将节点下线，修复之后，再由 K8s 运维专家将服务器上线到 K8s 集群：

```
# kubectl delete node kind-worker4
node "kind-worker4" deleted
```

## 1.2.3.2.6 删除 TiDB 集群

本小节演示删除名为“test”的TiDB集群。

### 1. 通过 helm 删除 TiDB Cluster

```
# helm list
NAME      REVISION  UPDATED             STATUS    CHART          APP VERSION  NAMESPACE
test      1          Sat Mar 7 22:30:16 2020  DEPLOYED  tidb-cluster-v1.0.6
tidb-operator  1        Sat Mar 7 05:02:15 2020  DEPLOYED  tidb-operator-v1.0.6
n
# helm delete test --purge
release "test" deleted
```

请注意：删除操作非常危险，请删除指定TiDB集群，千万不要误删除。

### 2. 删除 PVC

```
# kubectl delete pvc -n dba-test -l app.kubernetes.io/instance=test,app.kubernetes.io/managed-by=tidb-operator
persistentvolumeclaim "pd-test-pd-0" deleted
persistentvolumeclaim "pd-test-pd-1" deleted
persistentvolumeclaim "pd-test-pd-2" deleted
persistentvolumeclaim "pd-test-pd-3" deleted
persistentvolumeclaim "tikv-test-tikv-0" deleted
persistentvolumeclaim "tikv-test-tikv-1" deleted
persistentvolumeclaim "tikv-test-tikv-2" deleted
```

请注意：删除操作非常危险，请删除指定TiDB集群的PVC，千万不要误删除。

### 3. 删除 PV

```
# kubectl get pv -l app.kubernetes.io/namespace=dba-test,app.kubernetes.io/managed-by=tidb-operator,app.kubernetes.io
/instance=test -o name|xargs -I {} kubectl patch {} -p '{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
persistentvolume/local-pv-2c956bbd patched
persistentvolume/local-pv-3a4dae53 patched
persistentvolume/local-pv-3c7e9ebb patched
persistentvolume/local-pv-5ebe9899 patched
persistentvolume/local-pv-682d37c9 patched
persistentvolume/local-pv-af00e20c patched
persistentvolume/local-pv-d4cf548e patched
```

一分钟之后，检查PV状态：

```
# kubectl get pv
NAME      CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS    CLAIM      STORAGECLASS  REASON  AGE
local-pv-2c956bbd  1468Mi  RWO        Delete        Available  local-storage  64s
local-pv-3a4dae53  1468Mi  RWO        Delete        Available  local-storage  72s
local-pv-3c7e9ebb  1468Mi  RWO        Delete        Available  local-storage  54s
local-pv-5cb252d7  1974Mi  RWO        Delete        Available  local-storage  24h
local-pv-5ebe9899  1468Mi  RWO        Delete        Available  local-storage  58s
local-pv-682d37c9  1468Mi  RWO        Delete        Available  local-storage  72s
local-pv-af00e20c  1974Mi  RWO        Delete        Available  local-storage  24h
local-pv-d4cf548e  1468Mi  RWO        Delete        Available  local-storage  58s
local-pv-eb0e3c9f  1974Mi  RWO        Delete        Available  local-storage  24h
```

PV资源已全部释放。至此，“test”TiDB集群已经删除完毕。



## 1.2.4 访问 Kubernetes 上的 TiDB 集群及其监控

### 1.2.4.1 访问 Kubernetes 上的 TiDB 集群

在 Kubernetes 集群内访问 TiDB 时，使用 TiDB service 域名 `<release-name>-tidb.<namespace>` 即可。若需要在集群外访问，则需将 TiDB 服务端口暴露出去。在 `tidb-cluster` Helm chart 中，通过 `values.yaml` 文件中的 `tidb.service` 字段进行配置：

```
tidb:
  service:
    type: NodePort
    # externalTrafficPolicy: Cluster
    # annotations:
    #   cloud.google.com/load-balancer-type: Internal
```

#### 1. NodePort

在没有 LoadBalancer 时，可选择通过 NodePort 暴露。NodePort 有两种模式：

- `externalTrafficPolicy=Cluster`：集群所有的机器都会给 TiDB 分配 TCP 端口，此为默认值

使用 `cluster` 模式时，可以通过任意一台机器的 IP 加同一个端口访问 TiDB 服务，如果该机器上没有 TiDB Pod，则相应请求会转发到有 TiDB Pod 的机器上。

注意：

该模式下 TiDB 服务获取到的请求源 IP 是主机 IP，并不是真正的客户端源 IP，所以基于客户端源 IP 的访问权限控制在该模式下不可用。

- `externalTrafficPolicy=Local`：只有运行 TiDB 的机器会分配 TCP 端口，用于访问本地的 TiDB 实例

使用 `Local` 模式时，建议打开 `tidb-scheduler` 的 `stableScheduling` 特性。`tidb-scheduler` 会尽可能在升级过程中将新 TiDB 实例调度到原机器，这样集群外的客户端便不需要在 TiDB 重启后更新配置。

#### • 查看 NodePort 模式下对外暴露的 IP/PORT

查看 Service 分配的 Node Port，可通过获取 TiDB 的 Service 对象来获知：

```
kubectl -n <namespace> get svc <release-name>-tidb -ojsonpath=".spec.ports[?(.name=='mysql-client')].nodePort{`\n`}"
```

查看可通过哪些节点的 IP 访问 TiDB 服务，有两种情况：

- `externalTrafficPolicy` 为 `Cluster` 时，所有节点 IP 均可
- `externalTrafficPolicy` 为 `Local` 时，可通过以下命令获取指定集群的 TiDB 实例所在的节点

```
kubectl -n <namespace> get pods -l "app.kubernetes.io/component=tidb,app.kubernetes.io/instance=<release-name>" -ojsonpath="{range .items[*]}{.spec.nodeName}{`\n`}{end}"
```

#### • LoadBalancer

若运行在有 LoadBalancer 的环境，比如 GCP/AWS 平台，建议使用云平台的 LoadBalancer 特性。

访问 [Kubernetes Service 文档](#)，了解更多 Service 特性以及云平台 Load Balancer 支持。

### 1.2.4.2 Kubernetes 上的 TiDB 集群监控

基于 Kubernetes 环境部署的 TiDB 集群监控可以大体分为两个部分：对 TiDB 集群本身的监控、对 Kubernetes 集群及 TiDB Operator 的监控。本小节将对两者进行简要说明。

## 1. TiDB 集群的监控

TiDB 通过 Prometheus 和 Grafana 监控 TiDB 集群。在通过 TiDB Operator 创建新的 TiDB 集群时，对于每个 TiDB 集群，会同时创建、配置一套独立的监控系统，与 TiDB 集群运行在同一 Namespace，包括 Prometheus 和 Grafana 两个组件。

监控数据默认没有持久化，如果由于某些原因监控容器重启，已有的监控数据会丢失。可以在 `values.yaml` 中设置 `monitor.persistent` 为 `true` 来持久化监控数据。开启此选项时应将 `storageClass` 设置为一个当前集群中已有的存储，并且此存储应当支持将数据持久化，否则仍然会存在数据丢失的风险。

### 1. 查看监控面板

Grafana 服务默认通过 `NodePort` 暴露，如果 Kubernetes 集群支持负载均衡器，你可以在 `values.yaml` 中将 `monitor.grafana.service.type` 修改为 `LoadBalancer`，然后在执行 `helm upgrade` 后通过负载均衡器访问面板。

如果不需要使用 Grafana，可以在部署时在 `values.yaml` 中将 `monitor.grafana.create` 设置为 `false` 来节省资源。这一情况下需要使用其他已有或新部署的数据可视化工具直接访问监控数据来完成可视化。

### 1. 访问监控数据

Prometheus 服务默认通过 `NodePort` 暴露，如果 Kubernetes 集群支持负载均衡器，你可以在 `values.yaml` 中将 `monitor.prometheus.service.type` 修改为 `LoadBalancer`，然后在执行 `helm upgrade` 后通过负载均衡器访问监控数据。

### 1. Kubernetes 的监控

随集群部署的 TiDB 监控只关注 TiDB 本身各组件的运行情况，并不包括对容器资源、宿主机、Kubernetes 组件和 TiDB Operator 等的监控。对于这些组件或资源的监控，需要在整个 Kubernetes 集群维度部署监控系统来实现。

#### 1. 宿主机监控

对宿主机及其资源的监控与传统的服务器物理资源监控相同。

如果你的现有基础设施中已经有针对物理服务器的监控系统，只需要通过常规方法将 Kubernetes 所在的宿主机添加到现有监控系统中即可；如果没有可用的监控系统，或者希望部署一套独立的监控系统用于监控 Kubernetes 所在的宿主机，也可以使用你熟悉的任意监控系统。

新部署的监控系统可以运行于独立的服务器、直接运行于 Kubernetes 所在的宿主机，或运行于 Kubernetes 集群内，不同部署方式除在安装配置与资源利用上存在少许差异，在使用上并没有重大区别。

常见的可用于监控服务器资源的开源监控系统有：

- [CollectD](#)
- [Nagios](#)
- [Prometheus & node\\_exporter](#)
- [Zabbix](#)

一些云服务商或专门的性能监控服务提供商也有各自的免费或收费的监控解决方案可以选择。

我们推荐通过 [Prometheus Operator](#) 在 Kubernetes 集群内部署基于 [Node Exporter](#) 和 Prometheus 的宿主机监控系统，这一方案同时可以兼容并用于 Kubernetes 自身组件的监控。

#### 1. Kubernetes 组件监控

对 Kubernetes 组件的监控可以参考[官方文档](#)提供的方案，也可以使用其他兼容 Kubernetes 的监控系统来进行。

一些云服务商可能提供了自己的 Kubernetes 组件监控方案，一些专门的性能监控服务商也有各自的 Kubernetes 集成方案可以选择。

由于 TiDB Operator 实际上是运行于 Kubernetes 中的容器，选择任一可以覆盖对 Kubernetes 容器状态及资源进行监控的监控系统即可覆盖对 TiDB Operator 的监控，无需再额外部署监控组件。

我们推荐通过 [Prometheus Operator](#) 部署基于 [Node Exporter](#) 和 Prometheus 的宿主机监控系统，这一方案同时可以兼容并用于对宿主机资源的监控。

### 1.2.4.3 报警配置

#### 1. TiDB 集群报警

在随 TiDB 集群部署 Prometheus 时，会自动导入一些默认的报警规则，可以通过浏览器访问 Prometheus 的 Alerts 页面查看当前系统中的所有报警规则和状态。

我们目前暂不支持报警规则的自定义配置，如果确实需要修改报警规则，可以手动下载 charts 进行修改。

默认的 Prometheus 和报警配置不能发送报警消息，如需发送报警消息，可以使用任意支持 Prometheus 报警的工具与其集成。推荐通过 [AlertManager](#) 管理与发送报警消息。

如果在你的现有基础设施中已经有可用的 AlertManager 服务，可以在 `values.yaml` 文件中修改 `monitor.prometheus.alertmanagerURL` 配置其地址供 Prometheus 使用；如果没有可用的 AlertManager 服务，或者希望部署一套独立的服务，可以参考官方的[说明](#)部署。

#### 1. Kubernetes 报警

如果使用 Prometheus Operator 部署针对 Kubernetes 宿主机和服务的监控，会默认配置一些告警规则，并且会部署一个 AlertManager 服务，具体的设置方法请参阅 [kube-prometheus](#) 的说明。

如果使用其他的工具或服务对 Kubernetes 宿主机和服务进行监控，请查阅该工具或服务提供商的对应资料。

## 1.2.5 使用 BR 工具备份恢复 TiDB 集群

本文详细描述了如何将 TiDB 集群数据备份到 AWS 的 S3 存储上以及从 AWS S3 上拉取数据恢复集群。本文档中的“备份”，均是指全量备份（Ad-hoc 全量备份和定时全量备份）。底层通过使用 [BR](#) 获取集群的逻辑备份，然后再将备份数据上传到 AWS 的存储上。

本文使用的备份、恢复方式基于 TiDB Operator v1.1 及以上版本的 Custom Resource Definition(CRD) 实现。

### AWS 账号权限授予的三种方式

本文测试了以下三种权限授予方式：

(1) 通过传入 AWS 账号的 AccessKey 和 SecretKey 进行授权：

AWS 的客户端支持读取进程环境变量中的 `AWS_ACCESS_KEY_ID` 以及 `AWS_SECRET_ACCESS_KEY` 来获取与之相关联的用户或者角色的权限。

(2) 通过将 [IAM](#) 绑定 Pod 进行授权：通过将用户的 IAM 角色与所运行的 Pod 资源进行绑定，使 Pod 中运行的进程获得角色所拥有的权限，这种授权方式是由 [kube2iam](#) 提供。

注意：

- 使用该授权模式时，可以参考 [kube2iam 文档](#) 在 Kubernetes 集群中创建 kube2iam 环境，并且部署 TiDB Operator 以及 TiDB 集群。
- 该模式不适用于 `hostNetwork` 网络模式，请确保参数 `spec.tikv.hostNetwork` 的值为 `false`。

(3) 通过将 [IAM](#) 绑定 ServiceAccount 进行授权：

通过将用户的 IAM 角色与 Kubeneters 中的 `serviceAccount` 资源进行绑定，从而使得使用该 ServiceAccount 账号的 Pod 都拥有该角色所拥有的权限，这种授权方式由 [EKS Pod Identity Webhook](#) 服务提供。

- 使用该授权模式时，可以参考 [AWS 官方文档](#) 创建 EKS 集群，并且部署 TiDB Operator 以及 TiDB 集群。

### Ad-hoc 全量备份

Ad-hoc 全量备份通过创建一个自定义的 `Backup` Custom Resource (CR) 对象来描述一次备份。TiDB Operator 根据这个 `Backup` 对象来完成具体的备份过程。如果备份过程中出现错误，程序不会自动重试，此时需要手动处理。

目前 Ad-hoc 全量备份已经兼容以上三种授权模式，本文档提供如下备份示例。示例假设对部署在 Kubernetes `test1` 这个 namespace 中的 TiDB 集群 `demo1` 进行数据备份，下面是具体操作过程。

#### Ad-hoc 全量备份环境准备

##### 通过 AccessKey 和 SecretKey 授权

(1) 下载文件 `backup-rbac.yaml`，并执行以下命令在 `test1` 这个 namespace 中创建备份需要的 RBAC 相关资源：

```
kubectl apply -f backup-rbac.yaml -n test1
```

(2) 创建 `s3-secret` secret。该 secret 存放用于访问 S3 兼容存储的凭证。

```
kubectl create secret generic s3-secret --from-literal=access_key=xxx --from-literal=secret_key=yyy --namespace=test1
```

(3) 创建 `backup-demo1-tidb-secret` secret。该 secret 存放用于访问 TiDB 集群的用户所对应的密码。

```
kubectl create secret generic backup-demo1-tidb-secret --from-literal=password=<password> --namespace=test1
```

## 通过 IAM 绑定 Pod 授权

(1) 下载文件 [backup-rbac.yaml](#)，并执行以下命令在 `test1` 这个 namespace 中创建备份需要的 RBAC 相关资源：

```
kubectl apply -f backup-rbac.yaml -n test1
```

(2) 创建 `backup-demo1-tidb-secret` secret。该 secret 存放用于访问 TiDB 集群的用户所对应的密码：

```
kubectl create secret generic backup-demo1-tidb-secret --from-literal=password=<password> --namespace=test1
```

(3) 创建 IAM 角色：

可以参考 [AWS 官方文档](#) 来为账号创建一个 IAM 角色，并且通过 [AWS 官方文档](#) 为 IAM 角色赋予需要的权限。由于 `Backup` 需要访问 AWS 的 S3 存储，所以这里给 IAM 赋予了 `AmazonS3FullAccess` 的权限。

(4) 绑定 IAM 到 TiKV Pod：

在使用 BR 备份的过程中，TiKV Pod 和 BR Pod 一样需要对 S3 存储进行读写操作，所以这里需要给 TiKV Pod 打上 annotation 来绑定 IAM 角色。

```
kubectl edit tc demo1 -n test1
```

找到 `spec.tikv.annotations`，增加 annotation `arn:aws:iam::123456789012:role/user`，然后退出编辑，等到 TiKV Pod 重启后，查看 Pod 是否加上了这个 annotation。

**注意：**

`arn:aws:iam::123456789012:role/user` 为步骤 (3) 中创建的 IAM 角色。

## 通过 IAM 绑定 ServiceAccount 授权

(1) 下载文件 [backup-rbac.yaml](#)，并执行以下命令在 `test1` 这个 namespace 中创建备份需要的 RBAC 相关资源：

```
kubectl apply -f backup-rbac.yaml -n test2
```

(2) 创建 `backup-demo1-tidb-secret` secret。该 secret 存放用于访问 TiDB 集群的 root 账号和密钥：

```
kubectl create secret generic backup-demo1-tidb-secret --from-literal=password=<password> --namespace=test1
```

(3) 在集群上为服务帐户启用 IAM 角色：

可以参考 [AWS 官方文档](#) 开启所在的 EKS 集群的 IAM 角色授权。

(4) 创建 IAM 角色：

可以参考 [AWS 官方文档](#) 创建一个 IAM 角色，为角色赋予 `AmazonS3FullAccess` 的权限，并且编辑角色的 `Trust relationships`。

(5) 绑定 IAM 到 ServiceAccount 资源上：

```
kubectl annotate sa tidb-backup-manager -n eks.amazonaws.com/role-arn=arn:aws:iam::123456789012:role/user --namespace=test1
```

注意：

`arn:aws:iam::123456789012:role/user` 为步骤 (4) 中创建的 IAM 角色。

(6) 将 ServiceAccount 绑定到 TiKV Pod：

```
kubectl edit tc demo1 -n test1
```

将 `spec.tikv.serviceAccount` 修改为 `tidb-backup-manager`，等到 TiKV Pod 重启后，查看 Pod 的 `serviceAccountName` 是否有变化。

## 使用 BR 备份数据到 Amazon S3 的存储

- 创建 `Backup` CR，通过 `accessKey` 和 `secretKey` 授权的方式备份集群：

```
kubectl apply -f backup-aws-s3.yaml
```

`backup-aws-s3.yaml` 文件内容如下：

```
---
apiVersion: pingcap.com/v1alpha1
kind: Backup
metadata:
  name: demo1-backup-s3
  namespace: test1
spec:
  backupType: full
  br:
    cluster: demo1
    clusterNamespace: test1
    # enableTLSClient: false
    # logLevel: info
    # statusAddr: <status-addr>
    # concurrency: 4
    # rateLimit: 0
    # timeAgo: <time>
    # checksum: true
    # sendCredToTikv: true
  from:
    host: <tidb-host-ip>
    port: <tidb-port>
    user: <tidb-user>
    secretName: backup-demo1-tidb-secret
  s3:
    provider: aws
    secretName: s3-secret
    region: us-west-1
    bucket: my-bucket
    prefix: my-folder
```

- 创建 `Backup` CR，通过 IAM 绑定 Pod 授权的方式备份集群：

```
kubectl apply -f backup-aws-s3.yaml
```

`backup-aws-s3.yaml` 文件内容如下：

```
---
apiVersion: pingcap.com/v1alpha1
kind: Backup
metadata:
  name: demo1-backup-s3
  namespace: test1
  annotations:
    iam.amazonaws.com/role: arn:aws:iam::123456789012:role/user
spec:
  backupType: full
  br:
    cluster: demo1
    sendCredToTikv: false
    clusterNamespace: test1
    # enableTLSClient: false
    # logLevel: info
    # statusAddr: <status-addr>
    # concurrency: 4
    # rateLimit: 0
    # timeAgo: <time>
    # checksum: true
  from:
    host: <tidb-host-ip>
    port: <tidb-port>
    user: <tidb-user>
    secretName: backup-demo1-tidb-secret
  s3:
    provider: aws
    region: us-west-1
    bucket: my-bucket
    prefix: my-folder
```

- 创建 `Backup` CR，通过 IAM 绑定 ServiceAccount 授权的方式备份集群：

```
kubectl apply -f backup-aws-s3.yaml
```

`backup-aws-s3.yaml` 文件内容如下：

```

---
apiVersion: pingcap.com/v1alpha1
kind: Backup
metadata:
  name: demo1-backup-s3
  namespace: test1
spec:
  backupType: full
  serviceAccount: tidb-backup-manager
  br:
    cluster: demo1
    sendCredToTikv: false
    clusterNamespace: test1
    # enableTLSClient: false
    # logLevel: info
    # statusAddr: <status-addr>
    # concurrency: 4
    # rateLimit: 0
    # timeAgo: <time>
    # checksum: true
  from:
    host: <tidb-host-ip>
    port: <tidb-port>
    user: <tidb-user>
    secretName: backup-demo1-tidb-secret
  s3:
    provider: aws
    region: us-west-1
    bucket: my-bucket
    prefix: my-folder

```

以上三个示例分别使用三种授权模式将数据导出到 Amazon S3 存储上。Amazon S3 的 `acl`、`endpoint`、`storageClass` 配置项均可以省略。

Amazon S3 支持以下几种 access-control list (ACL) 策略：

- `private`
- `public-read`
- `public-read-write`
- `authenticated-read`
- `bucket-owner-read`
- `bucket-owner-full-control`

如果不设置 ACL 策略，则默认使用 `private` 策略。这几种访问控制策略的详细介绍参考 [AWS 官方文档](#)。

Amazon S3 支持以下几种 `storageClass` 类型：

- `STANDARD`
- `REDUCED_REDUNDANCY`
- `STANDARD_IA`
- `ONEZONE_IA`
- `GLACIER`
- `DEEP_ARCHIVE`

如果不设置 `storageClass`，则默认使用 `STANDARD_IA`。这几种存储类型的详细介绍参考 [AWS 官方文档](#)。

创建好 `Backup` CR 后，可通过如下命令查看备份状态：

```
kubectl get bk -n test1 -o wide
```

更多 `Backup` CR 字段的详细解释：

- `.spec.metadata.namespace`：`Backup` CR 所在的 namespace。
- `.spec.from.host`：待备份 TiDB 集群的访问地址。

- `.spec.from.port` : 待备份 TiDB 集群的访问端口。
- `.spec.from.user` : 待备份 TiDB 集群的访问用户。
- `.spec.from.tidbSecretName` : 待备份 TiDB 集群 `.spec.from.user` 用户的密码所对应的 secret。

更多支持的兼容 S3 的 provider 如下：

- `alibaba` : Alibaba Cloud Object Storage System (OSS) formerly Aliyun
- `digitalocean` : Digital Ocean Spaces
- `dreamhost` : Dreamhost DreamObjects
- `ibmcos` : IBM COS S3
- `minio` : Minio Object Storage
- `netease` : Netease Object Storage (NOS)
- `wasabi` : Wasabi Object Storage
- `other` : Any other S3 compatible provider

## 定时全量备份

用户通过设置备份策略来对 TiDB 集群进行定时备份，同时设置备份的保留策略以避免产生过多的备份。定时全量备份通过自定义的 `BackupSchedule` CR 对象来描述。每到备份时间点会触发一次全量备份，定时全量备份底层通过 Ad-hoc 全量备份来实现。下面是创建定时全量备份的具体步骤：

### 定时全量备份环境准备

同 [Ad-hoc 全量备份环境准备](#)。

### 使用 BR 定时备份数据到 Amazon S3 的存储

- 创建 `BackupSchedule` CR，开启 TiDB 集群定时全量备份，通过 `accessKey` 和 `secretKey` 授权的方式备份集群：

```
kubectl apply -f backup-scheduler-aws-s3.yaml
```

`backup-scheduler-aws-s3.yaml` 文件内容如下：

```

---
apiVersion: pingcap.com/v1alpha1
kind: BackupSchedule
metadata:
  name: demo1-backup-schedule-s3
  namespace: test1
spec:
  #maxBackups: 5
  #pause: true
  maxReservedTime: "3h"
  schedule: "*/*/*/*/*"
  backupTemplate:
    backupType: full
    br:
      cluster: demo1
      clusterNamespace: test1
      # enableTLSClient: false
      # logLevel: info
      # statusAddr: <status-addr>
      # concurrency: 4
      # rateLimit: 0
      # timeAgo: <time>
      # checksum: true
      # sendCredToTikv: true
    from:
      host: <tidb-host-ip>
      port: <tidb-port>
      user: <tidb-user>
      secretName: backup-demo1-tidb-secret
  s3:
    provider: aws
    secretName: s3-secret
    region: us-west-1
    bucket: my-bucket
    prefix: my-folder

```

- 创建 `BackupSchedule` CR，开启 TiDB 集群定时全量备份，通过 IAM 绑定 Pod 授权的方式备份集群：

```
kubectl apply -f backup-scheduler-aws-s3.yaml
```

`backup-scheduler-aws-s3.yaml` 文件内容如下：

```

---
apiVersion: pingcap.com/v1alpha1
kind: BackupSchedule
metadata:
  name: demo1-backup-schedule-s3
  namespace: test1
  annotations:
    iam.amazonaws.com/role: arn:aws:iam::123456789012:role/user
spec:
  #maxBackups: 5
  #pause: true
  maxReservedTime: "3h"
  schedule: "*/*/*/*/*"
  backupTemplate:
    backupType: full
    br:
      cluster: demo1
      sendCredToTikv: false
      clusterNamespace: test1
      # enableTLSClient: false
      # logLevel: info
      # statusAddr: <status-addr>
      # concurrency: 4
      # rateLimit: 0
      # timeAgo: <time>
      # checksum: true
    from:
      host: <tidb-host-ip>
      port: <tidb-port>
      user: <tidb-user>
      secretName: backup-demo1-tidb-secret
    s3:
      provider: aws
      region: us-west-1
      bucket: my-bucket
      prefix: my-folder

```

- 创建 `BackupSchedule` CR，开启 TiDB 集群定时全量备份，通过 IAM 绑定 ServiceAccount 授权的方式备份集群：

```
kubectl apply -f backup-scheduler-aws-s3.yaml
```

`backup-scheduler-aws-s3.yaml` 文件内容如下：

```

---
apiVersion: pingcap.com/v1alpha1
kind: BackupSchedule
metadata:
  name: demo1-backup-schedule-s3
  namespace: test1
spec:
  #maxBackups: 5
  #pause: true
  maxReservedTime: "3h"
  schedule: "*/*/*/*/*"
  serviceAccount: tidb-backup-manager
  backupTemplate:
    backupType: full
    br:
      cluster: demo1
      sendCredToTikv: false
      clusterNamespace: test1
      # enableTLSClient: false
      # logLevel: info
      # statusAddr: <status-addr>
      # concurrency: 4
      # rateLimit: 0
      # timeAgo: <time>
      # checksum: true
    from:
      host: <tidb-host-ip>
      port: <tidb-port>
      user: <tidb-user>
      secretName: backup-demo1-tidb-secret
  s3:
    provider: aws
    region: us-west-1
    bucket: my-bucket
    prefix: my-folder

```

定时全量备份创建完成后，可以通过以下命令查看定时全量备份的状态：

```
kubectl get bks -n test1 -o wide
```

查看定时全量备份下面所有的备份条目：

```
kubectl get bk -l tidb.pingcap.com/backup-schedule=demo1-backup-schedule-s3 -n test1
```

从以上两个示例可知，`BackupSchedule` 的配置由两部分组成。一部分是 `BackupSchedule` 独有的配置，另一部分是 `BackupTemplate`。`BackupTemplate` 指定 S3 兼容存储相关的配置，该配置与 Ad-hoc 全量备份到兼容 S3 的存储配置完全一样，可参考[使用 BR 备份数据到 Amazon S3 的存储](#)。下面介绍 `BackupSchedule` 独有的配置项：

- `.spec.maxBackups`：一种备份保留策略，决定定时备份最多可保留的备份个数。超过该数目，就会将过时的备份删除。如果将该项设置为 `0`，则表示保留所有备份。
- `.spec.maxReservedTime`：一种备份保留策略，按时间保留备份。例如将该参数设置为 `24h`，表示只保留最近 24 小时内的备份条目。超过这个时间的备份都会被清除。时间设置格式参考 [func ParseDuration](#)。如果同时设置最大备份保留个数和最长备份保留时间，则以最长备份保留时间为准。
- `.spec.schedule`：Cron 的时间调度格式。具体格式可参考 [Cron](#)。
- `.spec.pause`：该值默认为 `false`。如果将该值设置为 `true`，表示暂停定时调度。此时即使到了调度时间点，也不会进行备份。在定时备份暂停期间，备份 Garbage Collection (GC) 仍然正常进行。将 `true` 改为 `false` 则重新开启定时全量备份。

## 使用 BR 工具恢复 AWS 上的备份数据

本节详细描述了如何将存储在 Amazon S3 存储的备份数据恢复到 TiDB 集群。

## 环境准备

从以下三种授权方式中选择一种。

### 通过 AccessKey 和 SecretKey 授权

(1) 下载文件 `backup-rbac.yaml`，并执行以下命令在 `test2` 这个 namespace 中创建备份需要的 RBAC 相关资源：

```
kubectl apply -f backup-rbac.yaml -n test2
```

(2) 创建 `s3-secret` secret。该 secret 存放用于访问 S3 兼容存储的凭证。

```
kubectl create secret generic s3-secret --from-literal=access_key=xxx --from-literal=secret_key=yyy --namespace=test2
```

(3) 创建 `restore-demo2-tidb-secret` secret。该 secret 存放用于访问 TiDB 集群的 root 账号和密钥。

```
kubectl create secret generic restore-demo2-tidb-secret --from-literal=password=<password> --namespace=test2
```

### 通过 IAM 绑定 Pod 授权

(1) 下载文件 `backup-rbac.yaml`，并执行以下命令在 `test2` 这个 namespace 中创建备份需要的 RBAC 相关资源：

```
kubectl apply -f backup-rbac.yaml -n test2
```

(2) 创建 `restore-demo2-tidb-secret` secret。该 secret 存放用于访问 TiDB 集群的 root 账号和密钥：

```
kubectl create secret generic restore-demo2-tidb-secret --from-literal=password=<password> --namespace=test2
```

(3) 创建 IAM 角色：

可以参考 [AWS 官方文档](#) 来为账号创建一个 IAM 角色，并且通过 [AWS 官方文档](#) 为 IAM 角色赋予需要的权限。由于 `Restore` 需要访问 AWS 的 S3 存储，所以这里给 IAM 赋予了 `AmazonS3FullAccess` 的权限。

(4) 绑定 IAM 到 TiKV Pod：

在使用 BR 备份的过程中，TiKV Pod 和 BR Pod 一样需要对 S3 存储进行读写操作，所以这里需要给 TiKV Pod 打上 annotation 来绑定 IAM 角色。

```
kubectl edit tc demo2 -n test2
```

找到 `spec.tikv.annotations`，增加 annotation `arn:aws:iam::123456789012:role/user`，然后退出编辑，等到 TiKV Pod 重启后，查看 Pod 是否加上了这个 annotation。

注意：

`arn:aws:iam::123456789012:role/user` 为步骤(3)中创建的 IAM 角色。

### 通过 IAM 绑定 ServiceAccount 授权

(1) 下载文件 `backup-rbac.yaml`，并执行以下命令在 `test2` 这个 namespace 中创建备份需要的 RBAC 相关资源：

```
kubectl apply -f backup-rbac.yaml -n test2
```

(2) 创建 `restore-demo2-tidb-secret` secret。该 secret 存放用于访问 TiDB 集群的 root 账号和密钥：

```
kubectl create secret generic restore-demo2-tidb-secret --from-literal=password=<password> --namespace=test2
```

(3) 在集群上为服务帐户启用 IAM 角色：

可以参考 [AWS 官方文档](#) 开启所在的 EKS 集群的 IAM 角色授权。

(4) 创建 IAM 角色：

可以参考 [AWS 官方文档](#) 创建一个 IAM 角色，为角色赋予 `AmazonS3FullAccess` 的权限，并且编辑角色的 `Trust relationships`。

(5) 绑定 IAM 到 ServiceAccount 资源上：

```
kubectl annotate sa tidb-backup-manager -n eks.amazonaws.com/role-arn=arn:aws:iam::123456789012:role/user --namespace=test2
```

(6) 将 ServiceAccount 绑定到 TiKV Pod：

```
kubectl edit tc demo2 -n test2
```

将 `spec.tikv.serviceAccount` 修改为 `tidb-backup-manager`，等到 TiKV Pod 重启后，查看 Pod 的 `serviceAccountName` 是否有变化。

注意：

`arn:aws:iam::123456789012:role/user` 为步骤 (4) 中创建的 IAM 角色。

## 将指定备份数据恢复到 TiDB 集群

- 创建 `Restore` CR，通过 `accessKey` 和 `secretKey` 授权的方式恢复集群：

```
kubectl apply -f resotre-aws-s3.yaml
```

`restore-aws-s3.yaml` 文件内容如下：

```

---
apiVersion: pingcap.com/v1alpha1
kind: Restore
metadata:
  name: demo2-restore-s3
  namespace: test2
spec:
  br:
    cluster: demo2
    clusterNamespace: test2
    # enableTLSClient: false
    # logLevel: info
    # statusAddr: <status-addr>
    # concurrency: 4
    # rateLimit: 0
    # timeAgo: <time>
    # checksum: true
    # sendCredToTikv: true
  to:
    host: <tidb-host-ip>
    port: <tidb-port>
    user: <tidb-user>
    secretName: restore-demo2-tidb-secret
  s3:
    provider: aws
    secretName: s3-secret
    region: us-west-1
    bucket: my-bucket
    prefix: my-folder

```

- 创建 `Restore` CR，通过 IAM 绑定 Pod 授权的方式备份集群：

```
kubectl apply -f restore-aws-s3.yaml
```

`restore-aws-s3.yaml` 文件内容如下：

```

---
apiVersion: pingcap.com/v1alpha1
kind: Restore
metadata:
  name: demo2-restore-s3
  namespace: test2
  annotations:
    iam.amazonaws.com/role: arn:aws:iam::123456789012:role/user
spec:
  br:
    cluster: demo2
    sendCredToTikv: false
    clusterNamespace: test2
    # enableTLSClient: false
    # logLevel: info
    # statusAddr: <status-addr>
    # concurrency: 4
    # rateLimit: 0
    # timeAgo: <time>
    # checksum: true
  to:
    host: <tidb-host-ip>
    port: <tidb-port>
    user: <tidb-user>
    secretName: restore-demo2-tidb-secret
  s3:
    provider: aws
    region: us-west-1
    bucket: my-bucket
    prefix: my-folder

```

- 创建 `Restore` CR，通过 IAM 绑定 ServiceAccount 授权的方式备份集群：

```
kubectl apply -f restore-aws-s3.yaml
```

`restore-aws-s3.yaml` 文件内容如下：

```
---
apiVersion: pingcap.com/v1alpha1
kind: Restore
metadata:
  name: demo2-restore-s3
  namespace: test2
spec:
  serviceAccount: tidb-backup-manager
  br:
    cluster: demo2
    sendCredToTikv: false
    clusterNamespace: test2
    # enableTLSClient: false
    # logLevel: info
    # statusAddr: <status-addr>
    # concurrency: 4
    # rateLimit: 0
    # timeAgo: <time>
    # checksum: true
  to:
    host: <tidb-host-ip>
    port: <tidb-port>
    user: <tidb-user>
    secretName: restore-demo2-tidb-secret
  s3:
    provider: aws
    region: us-west-1
    bucket: my-bucket
    prefix: my-folder
```

创建好 `Restore` CR 后，可通过以下命令查看恢复的状态：

```
kubectl get rt -n test2 -o wide
```

更多 `Restore` CR 字段的详细解释：

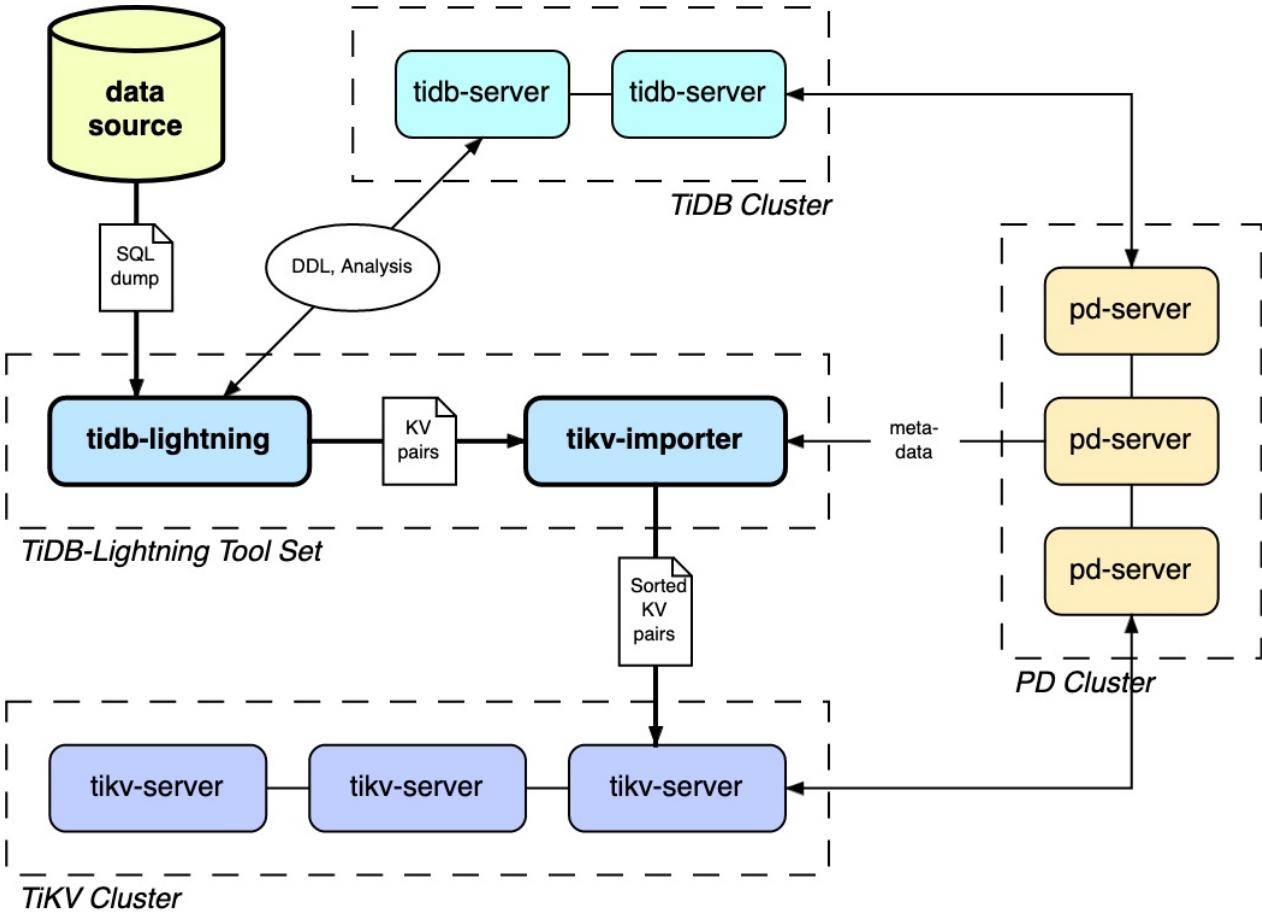
- `.spec.metadata.namespace`：`Restore` CR 所在的 namespace。
- `.spec.to.host`：待恢复 TiDB 集群的访问地址。
- `.spec.to.port`：待恢复 TiDB 集群的访问端口。
- `.spec.to.user`：待恢复 TiDB 集群的访问用户。
- `.spec.to.tidbSecretName`：待恢复 TiDB 集群 `.spec.to.user` 用户的密码所对应的 secret。

## 1.2.6 在 Kubernetes 集群上使用 Lightning 导入数据

### 1. 背景介绍

Mydumper + Loader 使用多线程导入导出数据时需要经过 TiDB SQL 语义解析，导致 TiDB 计算能力成为新的瓶颈。所以又一个想法孕育而出——导入数据不经过 SQL 解析，直接转换成 KV 键值对写入 TiKV 集群。

TiDB Lightning 整体架构：



TiDB Lightning 主要包含两个部分：

- `tidb-lightning` (“前端”)：主要完成适配工作，通过读取数据源，在下游 TiDB 集群建表、将数据转换成键值对 (KV 对) 发送到 `tikv-importer`、检查数据完整性等。
- `tikv-importer` (“后端”)：主要完成将数据导入 TiKV 集群的工作，对 `tidb-lightning` 写入的键值对进行缓存、排序、切分操作并导入到 TiKV 集群。

在 Kubernetes 上，`tikv-importer` 位于 TiDB 集群的 Helm chart 内，被部署为一个副本数为 1 (`replicas=1`) 的 `StatefulSet`；`tidb-lightning` 位于单独的 Helm chart 内，被部署为一个 `Job`。

为了使用 TiDB Lightning 恢复数据，`tikv-importer` 和 `tidb-lightning` 都必须分别部署。

### 2. 部署 tikv-importer

`tikv-importer` 可以在一个现有的 TiDB 集群上启用，或者在新建 TiDB 集群时启用。

- 在新建一个 TiDB 集群时启用 `tikv-importer`：

(1) 在 `tidb-cluster` 的 `values.yaml` 文件中将 `importer.create` 设置为 `true`。

(2) 部署该集群。

```
helm install pingcap/tidb-cluster --name=<tidb-cluster-release-name> --namespace=<namespace> -f values.yaml --version=<chart-version>
```

- 配置一个现有的 TiDB 集群以启用 tikv-importer：

(1) 在该 TiDB 集群的 `values.yaml` 文件中将 `importer.create` 设置为 `true`。

(2) 升级该 TiDB 集群。

```
helm upgrade <tidb-cluster-release-name> pingcap/tidb-cluster -f values.yaml --version=<chart-version>
```

### 3. 部署 tidb-lightning

#### (1) 配置 TiDB Lightning

使用如下命令获得 TiDB Lightning 的默认配置。

```
helm inspect values pingcap/tidb-lightning --version=<chart-version> > tidb-lightning-values.yaml
```

`tidb-lightning` Helm chart 支持恢复本地或远程的备份数据。

- 本地模式

本地模式要求 Mydumper 备份数据位于其中一个 Kubernetes 节点上。要启用该模式，你需要将 `dataSource.local.nodeName` 设置为该节点名称，将 `dataSource.local.hostPath` 设置为 Mydumper 备份数据目录路径，该路径中需要包含名为 `metadata` 的文件。

- PVC 模式

PVC 模式要求 Mydumper 备份数据位于和要恢复到的目标 TiDB 集群在同一 namespace 下的一个 PVC 上。要启用该模式，你需要将 `dataSource.adhoc.pvcName` 设置为 Mydumper 备份数据所在的 PVC。

- 远程模式

与本地模式不同，远程模式需要使用 `rclone` 将 Mydumper 备份 tarball 文件从网络存储中下载到 PV 中。远程模式能在 `rclone` 支持的任何云存储下工作，目前已经有以下存储进行了相关测试：[Google Cloud Storage \(GCS\)](#)、[AWS S3](#) 和 [Ceph Object Storage](#)。

- 确保 `values.yaml` 中的 `dataSource.local.nodeName` 和 `dataSource.local.hostPath` 被注释掉。
- 新建一个包含 `rclone` 配置的 `secret`。`rclone` 配置示例如下。一般只需要配置一种云存储。有关其他的云存储，请参考 [rclone 官方文档](#)。

```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: cloud-storage-secret
type: Opaque
stringData:
  rclone.conf: |
    [s3]
    type = s3
    provider = AWS
    env_auth = false
    access_key_id = <my-access-key>
    secret_access_key = <my-secret-key>
    region = us-east-1
    [ceph]
    type = s3
    provider = Ceph
    env_auth = false
    access_key_id = <my-access-key>
    secret_access_key = <my-secret-key>
    endpoint = <ceph-object-store-endpoint>
    region = :default-placement
    [gcs]
    type = google cloud storage
    # 该服务账号必须被授予 Storage Object Viewer 角色。
    # 该内容可以通过 `cat <service-account-file.json> | jq -c .` 命令获取。
    service_account_credentials = <service-account-json-file-content>
```

```

使用你的实际配置替换上述配置中的占位符，并将该文件存储为 `secret.yaml`。然后通过 `kubectl apply -f secret.yaml -n <namespace>` 命令创建该 Secret。

- 将 `dataSource.remote.storageClassName` 设置为 Kubernetes 集群中现有的一个存储类型。

## (2) 部署 TiDB Lightning

```
helm install pingcap/tidb-lightning --name=<tidb-lightning-release-name> --namespace=<namespace> --set failFast=true -f tidb-lightning-values.yaml --version=<chart-version>
```

## 4. Demo 演示

### 通过 Mydumper 执行全量逻辑备份

- 环境信息：

在 namespace test-cluster 下有两套集群：cluster-1、cluster-2。

查看 cluster-1 集群 Mysql 数据：

```
'select * from cloud.test_tbl;'  
+----+-----+-----+  
| id | title      | author | date       |  
+----+-----+-----+  
1	K8s	shonge	2020-03-07
2	operator	shonge	2020-03-07
3	kubernetes	shonge	2020-03-07
+----+-----+-----+
```

- 创建备份所需的 secret：

```
kubectl create secret generic backup-secret --namespace=test-backup --from-literal=user=root --from-literal=password=<root_password>
```

```
helm install pingcap/tidb-backup --version=v1.1.0-beta.2 --name backup-cluster-1 --namespace test-backup --set-string clusterName=cluster-1,storage.size=500Gi
```

- 确认备份任务完成：

```
kubectl -n test-backup get job -l app.kubernetes.io/instance=backup-cluster-1
NAME                  COMPLETIONS  DURATION   AGE
basic-fullbackup-202003080800  1/1        3s         3m32s
```

- 检查备份文件：

查找备份 PV 挂载路径。

```
kubectl -n test-cluster get pvc -l app.kubernetes.io/instance=backup-cluster-1
NAME      STATUS  VOLUME      CAPACITY  ACCESS MODES  STORAGECLASS  AGE
fullbackup-202003080800  Bound   local-pv-2a2853fb  77Gi     RWO   local-storage  62m

kubectl describe pv local-pv-2a2853fb
Name:           local-pv-2a2853fb
Labels:         kubernetes.io/hostname=tidb-operator-worker2
Annotations:   pv.kubernetes.io/bound-by-controller: yes
               pv.kubernetes.io/provisioned-by: local-volume-provisioner-tidb-operator-worker2-9d6bdbba-89ff-4180
               -9917-35b4dda3a3db
Finalizers:    [kubernetes.io/pv-protection]
StorageClass:  local-storage
Status:        Bound
Claim:         test-cluster/fullbackup-202003080800
Reclaim Policy: Delete
Access Modes:  RWO
VolumeMode:   Filesystem
Capacity:    500Gi
Node Affinity:
  Required Terms:
    Term 0:  kubernetes.io/hostname in [tidb-operator-worker2]
Message:
Source:
  Type: LocalVolume (a persistent volume backed by local storage on a node)
  Path: /mnt/disks/20
Events: <none>
```

查看备份文件，以 kind worker node 为例：

```
docker exec -ti tidb-operator-worker2 ls /mnt/disks/20/fullbackup-202003080800
cloud-schema-create.sql          mysql.opt_rule_blacklist-schema.sql
cloud.test_tbl-schema.sql       mysql.role_edges-schema.sql
cloud.test_tbl.sql              mysql.stats_buckets-schema.sql
metadata                         mysql.stats_feedback-schema.sql
mysql-schema-create.sql         mysql.stats_histograms-schema.sql
mysql.GLOBAL_VARIABLES-schema.sql mysql.stats_histograms.sql
mysql.GLOBAL_VARIABLES.sql       mysql.stats_meta-schema.sql
mysql.bind_info-schema.sql      mysql.stats_meta.sql
mysql.columns_priv-schema.sql   mysql.stats_top_n-schema.sql
mysql.db-schema.sql             mysql.tables_priv-schema.sql
mysql.default_roles-schema.sql  mysql.tidb-schema.sql
mysql.expr_pushdown_blacklist-schema.sql mysql.tidb.sql
mysql.gc_delete_range-schema.sql mysql.user-schema.sql
mysql.gc_delete_range_done-schema.sql mysql.user.sql
mysql.global_priv-schema.sql    test-schema-create.sql
mysql.help_topic-schema.sql
```

## 使用 Lightning 恢复数据

- 在 cluster-2 开启 importer

```
helm upgrade cluster-2 --set-string importer.create=true pingcap/tidb-cluster
```

- 部署 lightning 开始恢复数据

```
helm install pingcap/tidb-lightning --version=v1.1.0-beta.2 --name restore-cluster-1 --namespace test-cluster --set-string dataSource.adhoc.pvcName='fullbackup-202003080800',targetTidbCluster.name='cluster-2'
```

- 检查恢复任务状态

```
kubectl -n test-cluster get job -l app.kubernetes.io/name='restore-cluster-1-tidb-lightning'  
NAME                      COMPLETIONS   DURATION   AGE  
restore-cluster-1-tidb-lightning   1/1          3s         9m3s
```

- 访问 cluster-2 TiDB 服务，确认数据恢复情况

```
MySQL [(none)]> select * from cloud.test_tbl;  
+----+-----+-----+  
| id | title | author | date |  
+----+-----+-----+  
1	K8s	shonge	2020-03-07
2	operator	shonge	2020-03-07
3	kubernetes	shonge	2020-03-07
+----+-----+-----+  
3 rows in set (0.01 sec)
```

## 1.2.7 Kubernetes 上的 TiDB 工具指南

有一些开源工具可用以在 Kubernetes 上运维 TiDB。这些工具在 Kubernetes 上使用时，有一些特殊的操作要求。本节详细描述 Kubernetes 上的 TiDB 相关的工具及其使用方法。

### 1.2.7.1 在 Kubernetes 上使用 PD Control

[PD Control](#) 是 PD 的命令行工具。

使用 PD Control 操作 Kubernetes 上的 TiDB 集群，首先需要使用 `kubectl exec` 进入 PD 所在的容器：

```
kubectl exec -n <namespace> -it <pd-pod-name> sh
```

进入容器后，就可以通过 `127.0.0.1:2379` 访问到 PD 服务，从而直接使用 `pd-ctl` 命令的默认参数执行操作，如：

```
./pd-ctl config show
```

### 1.2.7.2 在 Kubernetes 上使用 TiKV Control

[TiKV Control](#) 是 TiKV 的命令行工具。

在使用 TiKV Control 操作 Kubernetes 上的 TiDB 集群时，针对 TiKV Control 的不同操作模式，有不同的操作步骤。

- 远程模式：此模式下 TiKV Control 需要通过网络访问 TiKV 服务或 PD 服务，因此需要先使用 `kubectl port-forward` 打开本地到 PD 服务以及目标 TiKV 节点的连接：

```
kubectl port-forward -n <namespace> svc/<cluster-name>-pd 2379:2379 &>/tmp/portforward-pd.log &
```

```
kubectl port-forward -n <namespace> <tikv-pod-name> 20160:20160 &>/tmp/portforward-tikv.log &
```

打开连接后，即可通过本地的对应端口访问 PD 服务和 TiKV 节点：

```
tikv-ctl --host 127.0.0.1:20160 <subcommand>
```

```
tikv-ctl --pd 127.0.0.1:2379 compact-cluster
```

- 本地模式：本地模式需要访问 TiKV 的数据文件，并停止正在运行的 TiKV 实例。因此，需要先使用[诊断模式](#)关闭 TiKV 实例自动重启功能，然后关闭 TiKV 进程，再使用 `tkctl debug` 命令在目标 TiKV Pod 中启动一个包含 `tikv-ctl` 可执行文件的新容器来执行操作。步骤如下：

1. 进入诊断模式：

```
kubectl annotate pod <tikv-pod-name> -n <namespace> runmode=debug
```

2. 关闭 TiKV 进程：

```
kubectl exec <tikv-pod-name> -n <namespace> -c tikv -- kill -s TERM 1
```

3. 启动 debug 容器：

```
tkctl debug <tikv-pod-name> -c tikv
```

4. 开始使用 `tikv-ctl` 的本地模式。需要注意的是 `tikv` 容器的根文件系统在 `/proc/1/root` 下，因此执行命令时也需要调整数据目录的路径：

```
tikv-ctl --db /path/to/tikv/db size -r 2
```

Kubernetes 上 TiKV 实例在 debug 容器中的默认 db 路径是 `/proc/1/root/var/lib/tikv/db size -r 2`

### 1.2.7.3 在 Kubernetes 上使用 TiDB Control

[TiDB Control](#) 是 TiDB 的命令行工具。

使用 TiDB Control 时，需要从本地访问 TiDB 节点和 PD 服务，因此建议使用 `kubectl port-forward` 打开到集群中 TiDB 节点和 PD 服务的连接：

```
kubectl port-forward -n <namespace> svc/<cluster-name>-pd 2379:2379 &>/tmp/portforward-pd.log &
```

```
kubectl port-forward -n <namespace> <tidb-pod-name> 10080:10080 &>/tmp/portforward-tidb.log &
```

接下来便可开始使用 `tidb-ctl` 命令：

```
tidb-ctl schema in mysql
```

### 1.2.7.4 使用 Helm

[Helm](#) 是一个 Kubernetes 的包管理工具。下面的介绍是基于 $2.9.0 \leq \text{Helm} < 3.0.0$  的 Helm 版本。其安装步骤如下：

1. 安装 Helm 客户端

```
curl https://raw.githubusercontent.com/kubernetes/helm/release-2.16/scripts/get | bash
```

如果使用 macOS，可以用以下命令通过 Homebrew 安装 Helm：

```
brew install helm@2
brew link --force helm@2
```

2. 安装 Helm 服务端

在集群中应用 helm 服务端组件 `tiller` 所需的 RBAC 规则并安装 `tiller`：

```
kubectl apply -f https://raw.githubusercontent.com/pingcap/tidb-operator/master/manifests/tiller-rbac.yaml && \
helm init --service-account=tiller --upgrade
```

通过下面命令确认 `tiller` Pod 进入 running 状态：

```
kubectl get po -n kube-system -l name=tiller
```

如果 Kubernetes 集群没有启用 RBAC，那么可以直接使用下列命令安装 `tiller`：

```
helm init --upgrade
```

Kubernetes 应用在 helm 中被打包为 chart。PingCAP 针对 Kubernetes 上的 TiDB 部署运维提供了三个 Helm chart：

- `tidb-operator`：用于部署 TiDB Operator；
- `tidb-cluster`：用于部署 TiDB 集群；
- `tidb-backup`：用于 TiDB 集群备份恢复；

这些 chart 都托管在 PingCAP 维护的 helm chart 仓库 <https://charts.pingcap.org/> 中，可以通过下面的命令添加：

```
helm repo add pingcap https://charts.pingcap.org/
```

添加完成后，可以使用 `helm search` 搜索 PingCAP 提供的 chart：

```
helm search pingcap -l
```

| NAME                  | CHART VERSION | APP VERSION | DESCRIPTION                             |
|-----------------------|---------------|-------------|---|
| pingcap/tidb-backup   | v1.0.0        |             | A Helm chart for TiDB Backup or Restore |
| pingcap/tidb-cluster  | v1.0.0        |             | A Helm chart for TiDB Cluster           |
| pingcap/tidb-operator | v1.0.0        |             | tidb-operator Helm chart for Kubernetes |

当新版本的 chart 发布后，可以使用 `helm repo update` 命令更新本地对于仓库的缓存：

```
helm repo update
```

Helm chart 往往都有很多可配置参数，通过命令行进行配置比较繁琐，因此推荐使用 YAML 文件的形式来编写这些配置项。基于 Helm 社区约定俗称的命名方式，我们在文档中将用于配置 chart 的 YAML 文件称为 `values.yaml` 文件。

Helm 的常用操作有部署（`helm install`）、升级（`helm upgrade`）、销毁（`helm del`）、查询（`helm ls`）。在执行部署和升级操作时，必须指定使用的 chart 名字（`chart-name`）和部署后的应用名（`release-name`）。可以指定一个或多个 `values.yaml` 文件来配置 chart。假如对 chart 有特定的版本需求，需要通过 `--version` 参数指定 `chart-version`（默认为最新的 GA 版本）。部署和升级的命令形式具体如下：

- 执行安装：

```
helm install <chart-name> --name=<release-name> --namespace=<namespace> --version=<chart-version> -f <values-file>
```

- 执行升级（升级可以是修改 `chart-version` 升级到新版本的 chart，也可以是修改 `values.yaml` 文件更新应用配置）：

```
helm upgrade <release-name> <chart-name> --version=<chart-version> -f <values-file>
```

假如要删除 helm 部署的应用，可以执行：

```
helm del --purge <release-name>
```

最后，在执行上述部署、升级、销毁等操作前，可以通过 `helm ls` 可以查看集群中已部署的应用：

```
helm ls
```

Helm3 已经 GA，经测试通过 Helm3 可以直接进行部署。更多 helm 的相关文档，请参考 [Helm 官方文档](#)。

## 1.2.7.5 使用 Terraform

Terraform 是一个基础设施即代码（Infrastructure as Code）管理工具。它允许用户使用声明式的风格描述自己的基础设施，并针对描述生成执行计划来创建或调整真实世界的计算资源。Kubernetes 上的 TiDB 使用 Terraform 来在公有云上创建和管理 TiDB 集群。

你可以参考 [Terraform 官方文档](#) 来安装 Terraform。

## 1.2.8 升级 TiDB Operator

本文介绍如何升级 TiDB Operator。升级 TiDB Operator 和自定义 TiDB Operator 类似，修改 `values.yaml` 中的镜像版本，然后执行 `helm upgrade`，例如：

```
helm upgrade tidb-operator pingcap/tidb-operator --version=<chart-version> -f /home/tidb/tidb-operator/values-tidb-operator.yaml
```

当新版本 tidb-operator 发布，只要更新 `values.yaml` 中的 `operatorImage` 然后执行上述命令就可以。但是安全起见，最好从新版本 tidb-operator chart 中获取新版本 `values.yaml` 并和旧版本 `values.yaml` 合并生成新的 `values.yaml`，然后升级。

TiDB Operator 是用来管理 TiDB 集群的，也就是说，如果 TiDB 集群已经启动并正常运行，你甚至可以停掉 TiDB Operator，而 TiDB 集群仍然能正常工作，直到你需要维护 TiDB 集群，比如进行伸缩、升级等操作。

生产环境 TiDB Operator 升级要慎重，最好选择业务流量低的时候进行升级。

### 1.2.8.1 升级 Kubernetes

当你的 Kubernetes 集群有版本升级，请确保 `kubeSchedulerImageTag` 与之匹配。默认情况下，这个值是由 Helm 在安装或者升级过程中生成的，要修改它你需要执行 `helm upgrade`。



## 1.3.1 基于 TiUP cluster 的集群扩缩容

在 TiUP cluster 之前，扩缩容是通过 Ansible 实现，但操作颇为繁琐，在易用性上没有很好的符合预期。现在用 TiUP cluster 只需要一两条命令就可以优雅的完成扩缩容操作。

### 1.3.1.1 扩容

扩容的内部逻辑如同部署类似，TiUP cluster 会先保证节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。其中 PD 节点的扩容会通过 join 方式加入到集群中，并且会更新与 PD 有关联的服务的配置；其他服务直接启动加入到集群中。所有服务在扩容时都会做正确性验证，最终返回是否扩容成功。

例如在集群 `tidb-test` 中扩容一个 TiKV 的节点和一个 PD 节点：

- 新建 `scale.yaml` 文件，添加 TiKV 和 PD 节点 IP。

注意：

注意新建一个拓扑文件，文件中只写入扩容节点的描述信息，不要包含已存在的节点。

```
---
pd_servers:
  - ip: 172.16.5.140
tikv_servers:
  - ip: 172.16.5.140
```

- 执行扩容操作。TiUP cluster 根据 `scale.yaml` 文件中声明的端口、目录等信息在集群中添加相应的节点。

```
$ tiup cluster scale-out tidb-test scale.yaml
```

```
[root@localhost ~]# tiup cluster scale-out --help
Scale out a TiDB cluster

Usage:
  cluster scale-out <cluster-name> <topology.yaml> [flags]

Flags:
  -h, --help           help for scale-out
  -i, --identity_file string   The path of the SSH identity file. If specified, public key authentication will be used.
  --user string         The user name to login via SSH. The user must has root (or sudo) privilege. (default "root")
  -y, --yes            Skip confirming the topology

Global Flags:
  --ssh-timeout int   Timeout in seconds to connect host via SSH, ignored for operations that don't need an SSH connection. (default 5)
```

执行完成之后可以通过 `tiup cluster display tidb-test` 命令检查扩容后的集群状态。

### 1.3.1.2 缩容

有时候业务量降低了，集群再占有原来的资源显得有些浪费，我们会想安全地释放某些节点，减小集群规模，于是需要缩容。缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关数据文件。由于 TiKV 和 Binlog 组件的下线是异步的（需要先通过 API 执行移除操作）并且下线过程耗时较长（需要持续观察节点是否已经下线成功），所以对 TiKV 和 Binlog 组件做了特殊处理。

- 对 TiKV 及 Binlog 组件的操作

- TiUP cluster 通过 API 将其下线后直接退出而不等待下线完成
- 等之后再执行集群操作相关的命令时会检查是否存在已经下线完成的 TiKV 或者 Binlog 节点。如果不存在，则继续执行指定的操作；如果存在，则执行如下操作：
  - 停止已经下线掉的节点的服务
  - 清理已经下线掉的节点的相关数据文件
  - 更新集群的拓扑，移除已经下线掉的节点
- 对其他组件的操作
  - PD 组件的下线通过 API 将指定节点从集群中 `delete` 掉（这个过程很快），然后停掉指定 PD 的服务并且清除该节点的相关数据文件
  - 下线其他组件时，直接停止并且清除节点的相关数据文件
- 缩容需要指定至少两个参数，一个是集群名字，另一个是节点 ID。比如我想要将 172.16.5.140 上的 TiKV 干掉。首先我们通过 `display` 命令查看当前集群节点的信息。

```
[root@localhost ~]# tiup cluster display prod-cluster
Starting /root/.tiup/components/cluster/v0.4.5/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v3.0.12
ID          Role       Host        Ports      Status     Data Dir   Deploy Dir
--          ----       ---        ----      -----     -----    -----
172.16.5.134:3000  grafana   172.16.5.134 3000    Up         -          deploy/grafana-3000
172.16.5.134:2379  pd        172.16.5.134 2379/2380 Healthy|L data/pd-2379  deploy/pd-2379
172.16.5.139:2379  pd        172.16.5.139 2379/2380 Healthy    data/pd-2379  deploy/pd-2379
172.16.5.140:2379  pd        172.16.5.140 2379/2380 Healthy    data/pd-2379  deploy/pd-2379
172.16.5.134:9090  prometheus 172.16.5.134 9090    Up         data/prometheus-9090  deploy/prometheus-9090
172.16.5.134:4000  tidb      172.16.5.134 4000/10080 Up         -          deploy/tidb-4000
172.16.5.139:4000  tidb      172.16.5.139 4000/10080 Up         -          deploy/tidb-4000
172.16.5.140:4000  tidb      172.16.5.140 4000/10080 Up         -          deploy/tidb-4000
172.16.5.134:20160  tikv      172.16.5.134 20160/20180 Up         data/tikv-20160  deploy/tikv-20160
172.16.5.139:20160  tikv      172.16.5.139 20160/20180 Up         data/tikv-20160  deploy/tikv-20160
172.16.5.140:20160  tikv      172.16.5.140 20160/20180 Offline    data/tikv-20160  deploy/tikv-20160
```

### 1. 执行缩容操作。

```
$ tiup cluster scale-in prod-cluster -N 172.16.5.140:20160
```

```
[root@localhost ~]# tiup cluster scale-in --help
Scale in a TiDB cluster

Usage:
  cluster scale-in <cluster-name> [flags]

Flags:
  -h, --help           help for scale-in
  -N, --node strings   Specify the nodes
  --transfer-timeout int  Timeout in seconds when transferring PD and TiKV store leaders (default 300)
  -y, --yes            Skip the confirmation of destroying

Global Flags:
  --ssh-timeout int   Timeout in seconds to connect host via SSH, ignored for operations that don't need an SSH connection. (default 5)
```

### 1. 执行完成之后可以通过 `tiup cluster display prod-cluster` 命令检查缩容后的集群状态。

```
[root@localhost ~]# tiup cluster display prod-cluster
Starting /root/.tiup/components/cluster/v0.4.5/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v3.0.12
ID          Role      Host        Ports      Status     Data Dir      Deploy Dir
--          ----      ----        -----      -----     -----      -----
172.16.5.134:3000  grafana   172.16.5.134  3000      Up         -          deploy/grafana-3000
172.16.5.134:2379  pd        172.16.5.134  2379/2380  Healthy|L  data/pd-2379  deploy/pd-2379
172.16.5.139:2379  pd        172.16.5.139  2379/2380  Healthy    data/pd-2379  deploy/pd-2379
172.16.5.140:2379  pd        172.16.5.140  2379/2380  Healthy    data/pd-2379  deploy/pd-2379
172.16.5.134:9090  prometheus 172.16.5.134  9090      Up         data/prometheus-9090  deploy/prometheus-9090
172.16.5.134:4000  tidb      172.16.5.134  4000/10080  Up         -          deploy/tidb-4000
172.16.5.139:4000  tidb      172.16.5.139  4000/10080  Up         -          deploy/tidb-4000
172.16.5.140:4000  tidb      172.16.5.140  4000/10080  Up         -          deploy/tidb-4000
172.16.5.134:20160  tikv      172.16.5.134  20160/20180  Up         data/tikv-20160  deploy/tikv-20160
172.16.5.139:20160  tikv      172.16.5.139  20160/20180  Up         data/tikv-20160  deploy/tikv-20160
172.16.5.140:20160  tikv      172.16.5.140  20160/20180  Offline    data/tikv-20160  deploy/tikv-20160
```

## 1.3.2 基于 TiDB Operator 的集群扩缩容

本节介绍如何对基于 TiDB Operator 部署的 TiDB 集群进行水平和垂直扩缩容。

### 1. 环境准备

- 参考[本地测试环境](#)或[生产环境](#)的部署指引，使用 TiDB Operator 部署 TiDB 集群。
- 在本例中，release 的名称为 `test`，在 `dba-test` 命名空间下，chart 文件放置在 `/home/charts/tidb-cluster` 路径下。

初始 TiDB 集群配置：

- 3 个 PD 节点
- 2 个 TiDB 节点
- 3 个 TiKV 节点

### 2. 水平扩缩容

#### 水平扩缩容原理

TiDB 集群水平扩缩容操作指的是通过增加或减少节点的数量，来达到集群扩缩容的目的。扩缩容 TiDB 集群时，会按照填入的 `replicas` 值，对 PD、TiKV、TiDB 进行顺序扩缩容操作。扩容操作按照节点编号由小到大增加节点，缩容操作按照节点编号由大到小删除节点。

#### 水平扩缩容操作步骤

- (1) 修改集群的 `value.yaml` 文件中的 `pd.replicas`、`tidb.replicas`、`tikv.replicas` 至期望值（本例中为 3、3、4）。
- (2) 执行 `helm upgrade` 命令进行扩缩容：

```
```shell
# helm upgrade test /home/charts/tidb-cluster -f /home/charts/tidb-cluster/values.yaml
Release "test" has been upgraded. Happy Helming!
LAST DEPLOYED: Sun Mar  8 15:39:57 2020
NAMESPACE: dba-test
STATUS: DEPLOYED
```

```

- (3) 查看集群水平扩缩容状态：

```
```shell
# watch kubectl get po -n dba-test
```

当所有组件的 Pod 数量都达到了预设值（本例中为 3 个 PD 节点、3 个 TiDB 节点、4 个 TiKV 节点），并且都进入 `Running` 状态后，水平扩容完成。

```shell
NAME          READY   STATUS    RESTARTS   AGE
test-discovery-668b48577c-zw4jh  1/1    Running   0          116m
test-monitor-86797cd996-9ggfh   3/3    Running   0          116m
test-pd-0        1/1    Running   0          116m
test-pd-1        1/1    Running   0          116m
test-pd-2        1/1    Running   1          116m
test-tidb-0      2/2    Running   0          112m
test-tidb-1      2/2    Running   0          112m
test-tidb-2      2/2    Running   0          2m52s
test-tikv-0       1/1    Running   0          114m
test-tikv-1       1/1    Running   0          114m
test-tikv-2       1/1    Running   0          114m
test-tikv-3       1/1    Running   0          2m52s
```

```

注意：

- PD、TiKV 组件在滚动升级的过程中不会触发扩容操作。
- TiKV 组件在缩容过程中会调用 PD 接口将对应 TiKV 标记为下线，然后将其上所有 Region 迁移到其它 TiKV 节点，在数据迁移期间 TiKV Pod 依然是 `Running` 状态，数据迁移完成后对应 Pod 才会被删除，缩容时间与待缩容的 TiKV 上的数据量有关，可以通过 `kubectl get tidbcluster -n <namespace> <release-name> -o json | jq '.status.tikv.stores'` 查看 TiKV 是否处于下线 `offline` 状态。
- PD、TiKV 组件在缩容过程中被删除的节点的 PVC 会保留，并且由于 PV 的 `Reclaim Policy` 设置为 `Retain`，即使 PVC 被删除，数据依然可以找回。
- TiKV 组件不支持在缩容过程中进行扩容操作，强制执行此操作可能导致集群状态异常。假如异常已经发生，可以参考 [TiKV Store 异常进入 Tombstone 状态](#) 进行解决。

## 3. 垂直扩缩容

### 垂直扩缩容原理

垂直扩缩容操作指的是通过增加或减少节点的资源限制，来达到集群扩缩容的目的。垂直扩缩容本质上是按照节点编号由大到小的顺序，滚动升级节点的过程。

### 垂直扩缩容操作步骤

- (1) 修改 `values.yaml` 文件中的 `tidb.resources`、`tikv.resources`、`pd.resources` 至期望值。
- (2) 执行 `helm upgrade` 命令进行升级：

```
```shell
# helm upgrade test /home/charts/tidb-cluster -f /home/charts/tidb-cluster/values.yaml
Release "test" has been upgraded. Happy Helming!
LAST DEPLOYED: Sun Mar  8 15:57:03 2020
NAMESPACE: dba-test
STATUS: DEPLOYED
```

```

- (3) 查看升级进度：

```
```shell
# watch kubectl -n <namespace> get pod -o wide
```

```

当所有 Pod 都重建完毕进入 `Running` 状态后，垂直扩容完成。

注意：

- 如果在垂直扩容时修改了资源的 `requests` 字段，由于 PD、TiKV 使用了 `Local PV`，升级后还需要调度回原节点，如果原节点资源不够，则会导致 Pod 一直处于 `Pending` 状态而影响服务。
- TiDB 作为一个可水平扩展的数据库，推荐通过增加节点个数发挥 TiDB 集群可水平扩展的优势，而不是类似传统数据库升级节点硬件配置来实现垂直扩容。



## 1.4.1 基于 TiUP cluster 的集群滚动更新

滚动升级功能借助 TiDB 的分布式能力，升级过程中尽量保证对前端业务透明、无感知。升级时会先检查各个组件的配置文件是否合理，如果配置有问题，则报错退出；如果配置没有问题，则工具会逐个节点升级。其中对不同节点有不同的操作。

### 1.4.1.1 不同节点的操作

- 升级 PD
  - 优先升级非 Leader 节点
  - 所有非 Leader 节点升级完成后再升级 Leader 节点
    - 工具会向 PD 发送一条命令将 Leader 迁移到升级完成的节点上
    - 当 Leader 已经切换到其他节点之后，再对旧的 Leader 节点做升级操作
  - 同时升级过程中，若发现有不健康的节点时工具会中止本次升级并退出，此时需要由人工判断、修复后再执行升级。
- 升级 TiKV
  - 先在 PD 中添加一个迁移对应 TiKV 上 region leader 的调度，通过迁移 Leader 确保升级过程中不影响前端业务。
  - 等待迁移 Leader 完成之后，再对该 TiKV 节点进行升级更新
  - 等更新后的 TiKV 正常启动之后再移除迁移 Leader 的调度
- 升级其他服务
  - 正常停止服务更新

### 1.4.1.2 版本升级操作

上面提到的逻辑都是在工具中集成好的功能，以升级到 4.0.0-rc 版本为例，我们实际上只需要一条命令就可以实现整个集群的版本升级。

```
$ tiup cluster upgrade tidb-test v4.0.0-rc
```

## 1.4.2 基于 TiDB-Operator 的集群滚动更新

TiDB 集群的滚动更新会按 PD、TiKV、TiDB 的顺序进行。每升级每一个组件，会先删除旧版本的 Pod，再创建新版本的 Pod。新版本的 Pod 正常运行后，再处理下一个 Pod。

滚动升级流程会自动处理 PD、TiKV 的 Leader 迁移。因此，在多节点的部署拓扑下（最小环境：PD \* 3、TiKV \* 3、TiDB \* 2），滚动更新 TiKV、PD 不会影响业务正常运行。

滚动升级流程会自动处理 TiDB 的 DDL Owner 迁移。对于有连接重试功能的客户端，滚动更新 TiDB 同样不会影响业务。如果客户端无法进行重试，滚动更新 TiDB 则会导致连接到被关闭节点的数据库连接失效，造成部分业务请求失败。对于这类业务，推荐在客户端添加重试功能，或者在低峰期进行 TiDB 的滚动升级操作。

滚动更新既可以用于升级 TiDB 版本，也可以用于更新集群配置。

### 1.4.2.1 升级 TiDB 版本

1. 修改集群的 `values.yaml` 文件中的 `tidb.image`、`tikv.image`、`pd.image` 的值为新版本镜像；
2. 执行 `helm upgrade` 命令进行升级：

```
helm upgrade <release-name> pingcap/tidb-cluster -f values.yaml --version=<chart-version>
```

3. 查看升级进度：

```
watch kubectl -n <namespace> get pod -o wide
```

所有 Pod 都重建完毕进入 `Running` 状态即表示升级完成。

### 1.4.2.2 更新 TiDB 集群配置

默认条件下，配置文件的修改不会自动应用到 TiDB 集群中，只有在实例重启时，才会重新加载新的配置文件。

您可以开启配置文件自动更新特性，在每次配置文件更新时，自动执行滚动更新，将修改后的配置应用到集群中。操作步骤如下：

1. 修改集群的 `values.yaml` 文件，将 `enableConfigMapRollout` 的值设为 `true`；
2. 根据需求修改 `values.yaml` 中需要调整的集群配置项；
3. 执行 `helm upgrade` 命令进行升级：

```
helm upgrade <release-name> pingcap/tidb-cluster -f values.yaml --version=<chart-version>
```

4. 查看升级进度：

```
watch kubectl -n <namespace> get pod -o wide
```

所有 Pod 都重建完毕进入 `Running` 状态即表示升级完成。

注意：

- 将 `enableConfigMapRollout` 特性从关闭状态打开时，即使没有配置变更，也会触发一次 PD、TiKV、TiDB 的滚动更新。

### 1.4.2.3 强制升级 TiDB 集群

如果 PD 集群因为 PD 配置错误、PD 镜像 tag 错误、NodeAffinity 等原因不可用，[TiDB 集群扩容](#)、[升级 TiDB 版本](#)和[更新 TiDB 集群配置](#)这三种操作都无法成功执行。

这种情况下，可使用 `force-upgrade` (TiDB Operator 版本 > v1.0.0-beta.3) 强制升级集群以恢复集群功能。步骤如下：

- 首先为集群设置 annotation：

```
kubectl annotate --overwrite tc <release-name> -n <namespace> tidb.pingcap.com/force-upgrade=true
```

- 然后执行对应操作中的 `helm upgrade` 命令：

```
helm upgrade <release-name> pingcap/tidb-cluster -f values.yaml --version=<chart-version>
```

## 1.5 动态配置变更

本章节主要介绍如何使用动态配置变更来完成对集群的参数修改。

### 1.5.1 背景

在 TiDB v4.0 版本之前，通过配置变更对集群进行调优一直比较繁琐，主要存在的问题如下：

- 集群缺少配置管理的统一入口
- 大多数配置参数不能在线变更需要重启实例

### 1.5.2 介绍

动态配置变更特性是 TiDB v4.0 版本的新特性，可以通过 pd-ctl component 查看和在线修改 TiDB、TiKV、PD 组件的配置参数。用户可以利用动态配置变更对各组件进行性能调优而无需重启集群组件。

#### (1) 开启动态配置变更

4.0 版本默认开启该参数，可通过修改 TiDB，TiKV，PD 配置文件中的 `enable-dynamic-config = false` 关闭该功能。

#### (2) 查看使用说明

```
/ # ./pd-ctl component --help
manipulate components' configs

Usage:
pd-ctl component [command]

Available Commands:
delete      delete component config with a given component ID (e.g. 127.0.0.1:20160)
ids         get all component IDs with a given component (e.g. tikv)
set          set the component config (set option with value)
show        show component config with a given component ID (e.g. 127.0.0.1:20160)

Global Flags:
-h, --help      Help message.
-u, --pd string  Address of pd. (default "http://127.0.0.1:2379")
```

#### (3) 修改全局配置

可通过 `component set <component> <key> <value>` 进行设置，其中 `component` 为组件类型，目前支持 `tidb`，`tikv`，`pd` 三种类型，`key` 为参数名称，`value` 为参数值。

示例如下：

```
>> component set tikv gc.batch-keys 1024
```

上述命令会将所有 TiKV 实例的 GC 的参数 `batch-keys` 设置为 1024。

#### (4) 修改实例配置

可通过 `component set <address> <key> <value>` 进行设置，其中 `address` 为实例的 IP 地址加端口，如 `127.0.0.1:20160`。

示例如下：

```
>> component set 127.0.0.1:20160 gc.batch-keys 1024
```

上述命令仅将 `127.0.0.1:20160` 这个 TiKV 实例的 GC 的参数 `batch-keys` 设置为 1024。

#### (5) 查看配置

可通过 `component show <address>` 查看具体实例的配置。

示例如下：

```
>> component show 127.0.0.1:20160
```

## 1.5.3 操作

### (1) 创建 `tidbcluster` 集群

```
apiVersion: pingcap.com/v1alpha1
kind: TidbCluster
metadata:
  name: basic
spec:
  version: nightly
  timezone: UTC
  pvReclaimPolicy: Delete
  pd:
    baseImage: pingcap/pd
    replicas: 1
    requests:
      storage: "1Gi"
      config: {}
  tikv:
    baseImage: pingcap/tikv
    replicas: 3
    requests:
      storage: "1Gi"
      config: {}
  tidb:
    baseImage: pingcap/tidb
    replicas: 1
    service:
      type: NodePort
    config: {}
```

### (2) 进入 PD Pod

- 查看当前配置

```
/ # ./pd-ctl component show basic-tikv-0.basic-tikv-peer.test-cluster.svc:20160 | grep batch-keys
batch-keys = 512
```

- 动态变更配置

```
/ # ./pd-ctl component set tikv gc.batch-keys 1024
Success!
```

该命令返回 `Success!` 表示已经成功更新到配置中心。由于配置的分发是通过各组件拉取的，需要最多等待 30s 即可验证配置是否生效。

- 验证变更配置

```
/ # ./pd-ctl component show basic-tikv-0.basic-tikv-peer.test-cluster.svc:20160 | grep batch-keys
batch-keys = 512
```

综上，可以看到通过使用 pd-ctl，能够快速对集群的大部分参数进行在线修改，达到对系统性能调优的目的。

## 2 TiDB 备份恢复和导入导出工具

尽管 TiDB 通过 Raft 一致性算法确保自身数据有多副本，作为生产环境数据库系统的托底方案，数据库管理员依然需要确保能够正确、高效地备份和恢复数据。TiDB 备份恢复和导入导出工具集应运而生。

本章将介绍以下几种 TiDB 备份恢复和导入导出工具：

- 增量数据订阅工具 **CDC**：识别、捕捉和输出 TiDB/TIKV 集群的数据变更。它既可以作为 TiDB 增量数据同步工具，将 TiDB 集群的增量数据同步至下游数据库；也提供开放数据协议，支持把数据发布到第三方系统。
- **TiDB 数据导入工具 Lightning**：支持将 Mydumper 和 CSV 文件格式快速导入目标 TiDB 集群。
- 分布式备份和恢复工具 **BR**：专门为 TiDB 量身打造的快速备份和恢复工具，支持从 TiKV 直接导出 SST 文件，并支持将 SST 文件快速恢复到目标 TiKV 集群。
- 分布式导出工具 **Dumpling**：类 Mydumper 工具，完全采用 Golang 重新实现，与 TiDB 生态高度集成。它支持并发、高速导出 TiDB 数据，支持 SQL 和 CSV 等多种数据输出格式，并支持直接导出数据到 S3 等云存储系统。

通过阅读过本章内容，读者将对 TiDB 备份恢复和导入导出工具集各组件的工作原理和使用方法有一个全面了解。

## 2.1 4.0 增量数据订阅 CDC

TiDB 4.0 引入的新组件 Change Data Capture (CDC) 是用来识别、捕捉和交付 TiDB/TiKV 数据变更的工具系统。在 TiDB 生态链上，CDC 作为 TiDB 的数据出口有着非常重要的地位，其作用包括：构建 TiDB 主从和灾备系统；链接 TiDB 和其它异构数据库；提供开放数据协议，支持把数据发布到第三方系统。

本章将从以下几个方面介绍 CDC：

- 比较 CDC 和 TiDB Binlog，揭示 CDC 在设计、功能、易用性和性能等多方面的优势。
- 介绍 CDC 的主要组件及其工作原理。
- 提供一个简要的实操指南，带读者快速了解如何借助 CDC 在两个 TiDB 集群之间实现数据同步。

希望读者阅读过本章内容后，能够对 CDC 的原理和适用场景有一个全面的了解。

## 2.1.1 CDC 解决什么问题

在 TiCDC 工具出现之前，数据同步功能是由 TiDB-Tools 工具中的 TiDB-Binlog 来实现的。TiDB-Binlog 通过收集各个 TiDB 实例产生的 binlog，并按照事务提交的时间排序后同步到下游。TiCDC 则是通过 TiKV 的 kv change Logs 来实现的，在数据同步过程中的处理上，两者有本质的区别。

### 易用性

TiCDC 更易使用：

- TiCDC 部署只需一个二进制文件，非常简洁
- TiCDC 可以全部使用 SQL 管理，不需要另外组件，而且自带管理界面

### 性能

TiCDC 性能更好：

- TiCDC 提供更良好的扩展性，可以应对超大规模 TiDB 集群的使用场景，在这一点上 TiDB-Binlog 要弱于 TiCDC，Pump 集群虽然具有一定的扩展性，但是 Drainer 是单节点归并排序，无法应对超大规模 TiDB 集群
- TiDB-Binlog 在极端情况下可能会丢失 Commit Binlog，需要反查 TiKV 事务状态，同步延迟可达到 10 分钟，而 TiCDC 的同步延迟通常在毫秒级别
- 目前版本 TiDB-Binlog 的实现强依赖于 TiDB Transaction 模型，会阻扰已知的一些优化，比如 Big Transaction 功能、不从 PD 获取 Commit Timestamp 等

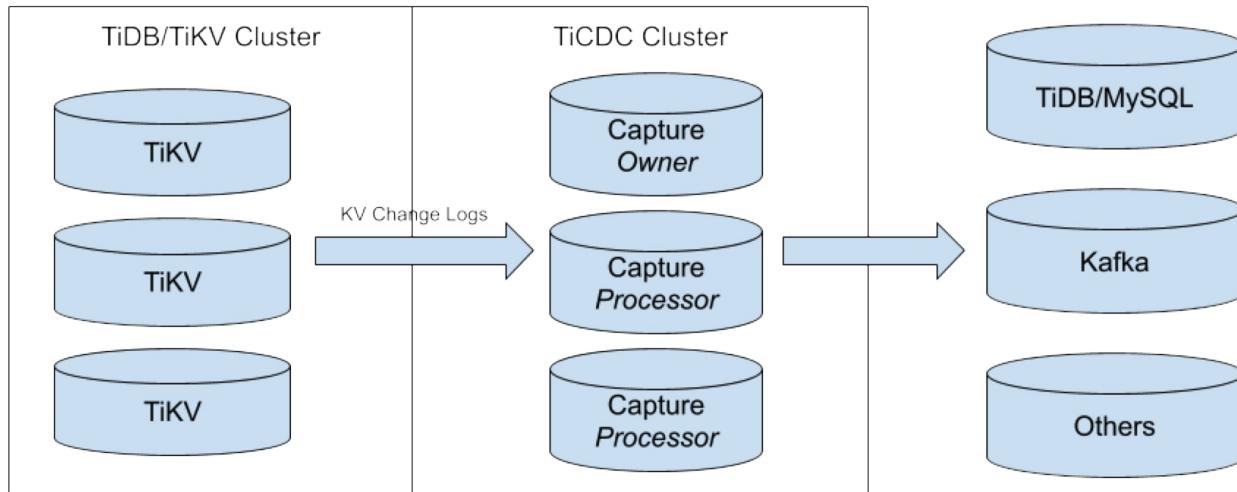
### 可用性

TiCDC 可用性更好：

- TiCDC 多个节点写下游不会有单点瓶颈，直接在 watch KV 层变更，有天然数据安全性保证
- TiCDC 各节点无状态，通过 PD 的 etcd 保存元数据信息，因此可以很方便实现数据高可用及服务高可用
- TiDB-Binlog 需要单独解决数据安全和服务高可用问题，实现代价很高，且目前都未实现

## 2.1.2 CDC 工作原理

TiCDC (TiDB Change Data Capture) 是用来识别、捕捉和输出 TiDB/TiKV 集群上数据变更的工具系统。它既可以作为 TiDB 增量数据同步工具，将 TiDB 集群的增量数据同步至下游数据库，也提供开放数据协议，支持把数据发布到第三方系统。相较于 TiDB Binlog，TiCDC 不依赖 TiDB 事务模型保证数据同步的一致性，系统可水平扩展且天然支持高可用。



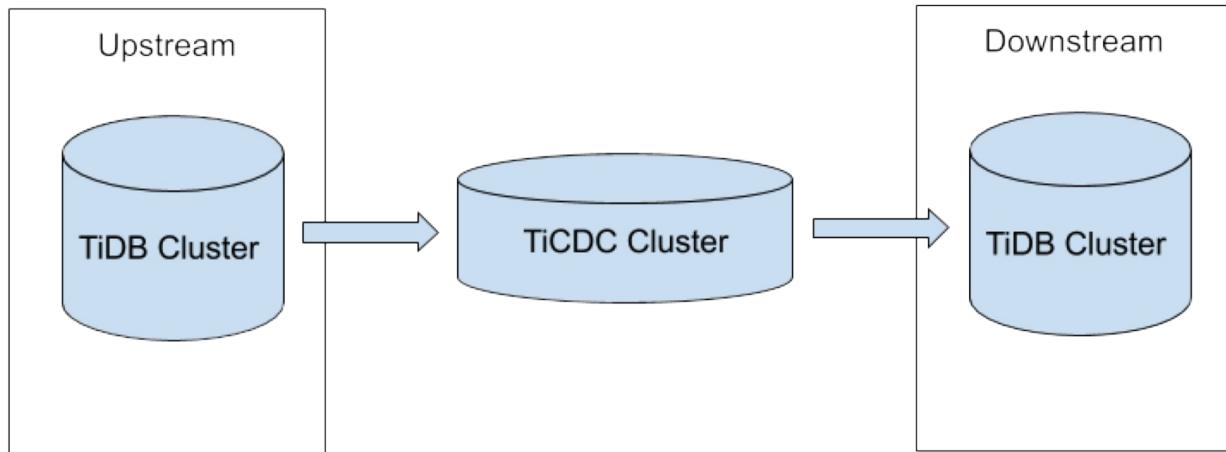
TiCDC 的系统架构如上图所示：

- **TiKV**：输出 KV 变更日志 (KV Change Logs)。KV 变更日志是 TiKV 生成的 row changed events，隐藏了大部分内部实现细节。TiKV 负责拼装 KV 变更日志，并输出到 TiCDC 集群。
- **Capture**：TiCDC 的运行进程。一个 TiCDC 集群通常由多个 capture 节点组成，每个节点负责拉取一部分 KV 变更日志，排序后输出到下游组件。一个 TiCDC 集群里有两种 capture 角色：owner 和 processor。
  - 一个 TiCDC 集群有且仅有一个 owner，它负责集群内部调度。若 owner 出现异常，则其它 capture 节点会自发选举出新的 owner。
  - processor 在实现上是 capture 内部的逻辑线程。一个 capture 节点可以运行多个 processor，每个 processor 负责同步若干个表的数据变更。processor 所在的 capture 节点若出现异常，则 TiCDC 集群会把同步任务重新分配给其它 capture 节点。
  - capture 运行过程中的一些数据会持久化到 PD 内部的 etcd 中，这些数据包括同步任务的配置和状态、各 capture 节点的信息、owner 选举信息以及 processor 的同步状态等。
  - 前面多次提到的“同步任务”，也被称作 change feed，指的是由一个用户启动的从上游 TiDB 同步数据变更到下游组件的任务。用户在创建同步任务时需要指定上下游的连接方式，列出需要同步的数据库和表名，还可以指定从上游哪个 TSO 开始同步 KV 变更日志。TiCDC 集群支持创建并运行多个同步任务，并同时向多个不同的下游组件输出变更日志。通常一个同步任务会被拆分为多个子任务，每个子任务也被称作 task。每个子任务负责若干个表的变更日志同步，这些子任务会分发到不同 capture 节点并行处理。
- **Sink**：TiCDC 内部负责将已经排序的数据变更同步到下游的组件。TiCDC 支持同步数据变更到多种下游组件。
  - 支持同步数据到 TiDB 和 MySQL。为保证数据正确性，表结构定义须满足两个条件：必须要有主键或者唯一索引；若不存在主键，则构成唯一索引的每一个字段都应该被明确定义为 NOT NULL。
  - 支持按照 TiCDC 开发数据协议 (open protocol) 输出数据到 Kafka。其他系统可以订阅 Kafka 上的数据变更。
  - 未来将支持输出数据变更到多种文件存储系统上。

## 2.1.3 CDC 实操指南

本节将介绍如何使用 TiCDC 在两个 TiDB 集群之间实现数据同步。

### 1. 部署结构



部署结构如上图所示。这里我们假定：

- 上游 TiDB 集群的 PD 节点是 `10.1.1.10:2379`
- 下游 TiDB 集群的 SQL 节点是 `10.3.1.30:4000`
- TiCDC 集群由 3 个 capture 节点构成，分别是：
  - `10.2.1.20:8300`
  - `10.2.1.21:8300`
  - `10.2.1.22:8300`

### 2. 部署集群

#### (1) 选择部署目标服务器

- 推荐使用 CentOS 7.3 及以上版本的 Linux 操作系统，以及 x86\_64 架构 (amd64)
- 编译 TiCDC 需要 Go >= 1.13
- 服务器之间内网互通

#### (2) 准备二进制文件

从 [Github](#) 下载源码，并运行 `make` 执行编译，编译好的文件会出现在 `bin` 目录下。

#### (3) 启动集群

在每一台 TiCDC 服务器上分别运行以下命令启动服务：

```
$ cdc server --pd=http://10.1.1.10:2379 --status-addr=127.0.0.1:8300
```

命令参数说明：

- `pd`：上游 TiDB 集群的 PD 节点地址
- `status-addr`：本地 capture 节点地址

至此，一个 TiCDC 集群就搭建成功了，它现在已经开始监听上游 TiKV 的变更日志了。

运行以下命令可以查看 capture 节点列表：

```
$ cdc cli capture list --pd=http://10.1.1.10:2379
[
  {
    "id": "5d1fd3bd-efc9-4cdf-9e8a-6d955f65b3b0",
    "is-owner": true
  },
  {
    "id": "629ec61e-16a3-466c-8fd4-2b2b457dabf7",
    "is-owner": false
  },
  {
    "id": "c5cd08b4-f601-456f-995c-62c97044444b",
    "is-owner": false
  }
]
```

上述命令会返回当前集群包含的全部 capture 节点。不难发现，集群中有且仅有一个 owner，其余节点都是 processor。

### 3. 创建同步任务

运行以下命令创建一个同步任务：

```
$ cdc cli changefeed create --pd=http://10.1.1.10:2379 --sink-uri="mysql://user:password@10.3.1.30:4000/" --config=~/cdc-config.toml --start-ts=0
```

命令参数说明：

- `pd`：上游 TiDB 集群的 PD 节点地址
- `sink-uri`：下游 TiDB 集群的 DSN
- `config`：同步任务配置文件，允许指定需要同步的数据库和表，以及需要跳过的 TSO
- `start-ts`：指定一个 TSO 作为数据同步的起点，若不指定或置为0，则默认使用当前最新的 TSO 作为起点

下面是一个同步任务配置文件示例：

```
ignore-txn-commit-ts = []
filter-case-sensitive = false

[filter-rules]
ignore-dbs = ["test", "mysql", "information_schema", ]

[[filter-rules.do-tables]]
db-name = "sns"
tbl-name = "user"

[[filter-rules.do-tables]]
db-name = "sns"
tbl-name = "following"
```

从中可以看到，`test`、`mysql` 和 `information_schema` 等三个数据库的变更日志会被过滤掉，只有 `sns.user` 和 `sns.following` 两个表会被同步到下游。

### 4. 查询同步任务状态

运行以下命令可以查询同步任务列表：

```
$ cdc cli changefeed list --pd=http://10.1.1.10:2379
[
  {
    "id": "004a0ea8-2ef1-45b4-8ce1-b3281e7dc24d"
  }
]
```

若要查询同步任务的配置信息和同步状态，则需要给出对应的同步任务 ID：

```
$ cdc cli changefeed query --pd=http://10.1.1.10:2379 --changefeed-id=004a0ea8-2ef1-45b4-8ce1-b3281e7dc24d
{
    "info": {
        "sink-uri": "mysql://root:123456@127.0.0.1:3306/",
        "opts": {},
        "create-time": "2020-03-13T16:17:33.965778+08:00",
        "start-ts": 415259021527482369,
        "target-ts": 0,
        "admin-job-type": 0,
        "config": {
            "filter-case-sensitive": false,
            "filter-rules": null,
            "ignore-txn-commit-ts": null
        }
    },
    "status": {
        "resolved-ts": 415259037347348481,
        "checkpoint-ts": 415259036823060481,
        "admin-job-type": 0
    }
}
```

还可以查看子任务：

```
$ cdc cli processor query --pd=http://10.1.1.10:2379 --changefeed-id=004a0ea8-2ef1-45b4-8ce1-b3281e7dc24d --capture-id=5d1fd3bd-efc9-4cdf-9e8a-6d955f65b3b0
{
    "status": {
        "table-infos": [
            {
                "id": 45,
                "start-ts": 415259021527482369
            }
        ],
        "table-p-lock": null,
        "table-c-lock": null,
        "admin-job-type": 0
    },
    "position": {
        "checkpoint-ts": 415259059393658881,
        "resolved-ts": 415259059917946881
    }
}
```

## 5. HTTP 接口

TiCDC 提供了 HTTP 接口，帮助实现一些基础的查询和运维功能。

运行如下命令可以查询某个 capture 节点的服务状态：

```
$ curl http://10.2.1.20:8300/status
{
    "version": "0.0.1",
    "git_hash": "",
    "id": "4a54c85b-fc1d-4897-9934-1be3b9aa6a45",
    "pid": 31652
}
```

上述输出结果中，`id` 是本地 TiCDC 服务对应的 capture ID，`pid` 则是本地进程 ID。

有时候需要驱逐当前的 owner 节点以主动触发 TiCDC 集群选举新的 owner，运行以下命令驱逐当前 owner 节点：

```
$ curl -X POST http://10.2.1.20:8300/capture/owner/resign
```

请注意，上述命令需要向当前的 owner 节点发出请求，该请求对 processor 节点无效。

也可以停止、恢复或者删除指定的同步任务，命令如下：

```
$ curl -X POST -d "admin-job=X&cf-id=136a3bee-621c-42d5-80ec-4c1aaf6ddb53" http://10.2.1.20:8300/capture/owner/admin
```

参数 `admin-job` 表示不同的任务类型：

- `admin-job=1` 表示停止任务。停止任务后所有 processor 会结束同步并退出。同步任务的配置和同步进度都会保留，后续可以恢复任务。
- `admin-job=2` 表示恢复任务。同步任务将继续同步。
- `admin-job=3` 表示删除任务。将结束所有同步 processor，并清理同步任务配置。同步状态将被保留，后续只提供查询功能，无法恢复任务。

请注意，上述命令也需要向当前的 owner 节点发出请求。

最后，可以运行以下命令获取调试信息（譬如 owner 和 processors 的状态以及 etcd 上存储的内容）：

```
$ curl http://10.2.1.20:8300/debug/info
```

## 2.2 TiDB 数据导入工具 Lightning

TiDB Lightning 是一个将全量数据高速导入到 TiDB 集群的工具，速度可达到传统执行 SQL 导入方式的 3 倍以上、大约每小时 300 GB。Lightning 有以下两个主要的使用场景：一是大量新数据的快速导入、二是全量数据的备份恢复。目前，Lightning 支持 Mydumper 或 CSV 输出格式的数据源。

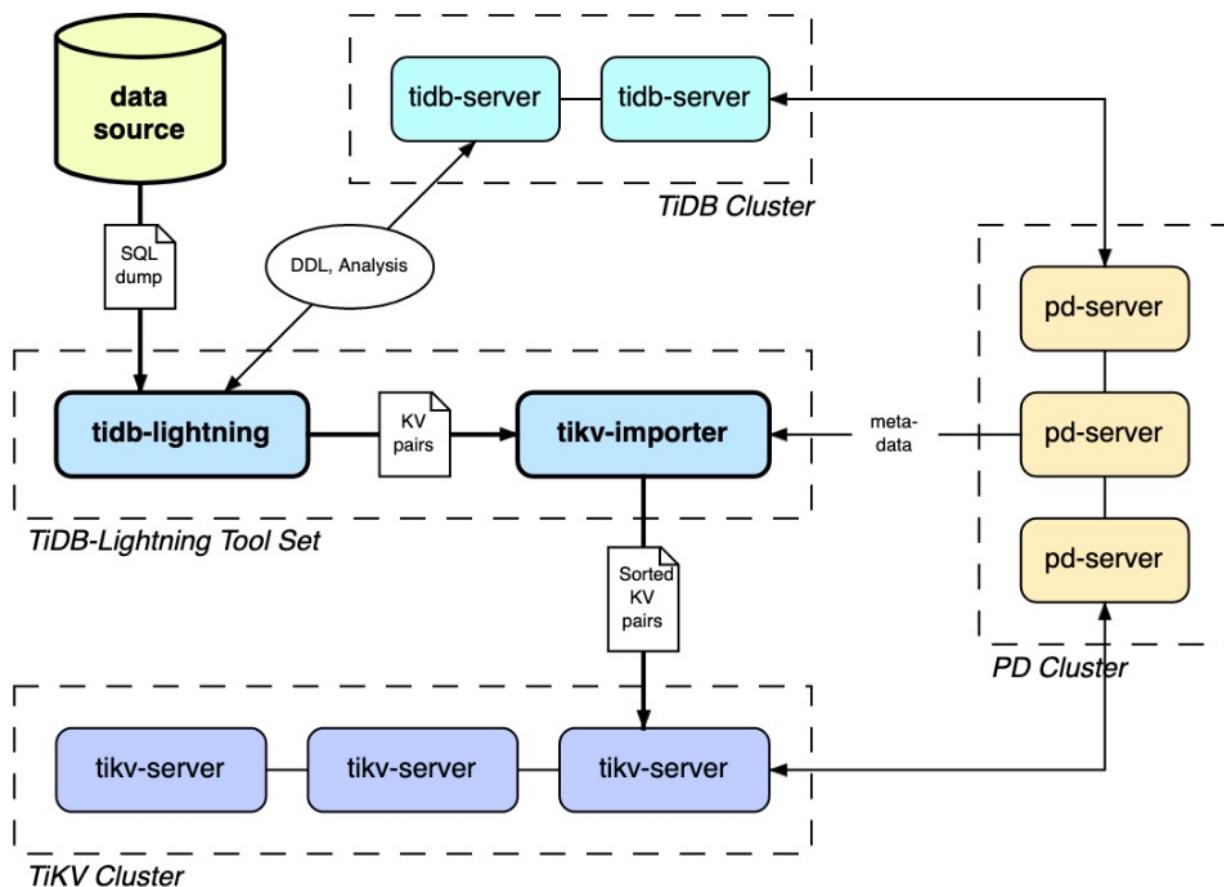
## 2.2.1 Lightning 工作原理

TiDB Lightning 工具支持高速导入 Mydumper 和 CSV 文件格式的数据文件到 TiDB 集群，导入速度可达每小时 300 GB，是传统 SQL 导入方式的 3 倍多。它有两个主要的目标使用场景：大量新数据的快速导入，以及全量数据恢复。

本节将介绍 TiDB Lightning 工具的工作原理。

### 1. 整体架构

#### 组件概览



如上图所示，TiDB Lightning 工具包含两个组件：

- **tidb-lightning**（前端）：负责导入过程的管理和适配工作。读取数据文件，在目标 TiDB 集群上建表，并将数据文件转换成键值对发送到 tikv-importer。等 tikv-importer 的处理工作完成后，tidb-lightning 还需要执行数据校验等收尾工作。
- **tikv-importer**（后端）：负责将数据导入到目标 TiKV 集群。从 tidb-lightning 接收到键值之后，tikv-importer 会执行缓存、排序和切分等操作，最后导入 TiKV 集群。

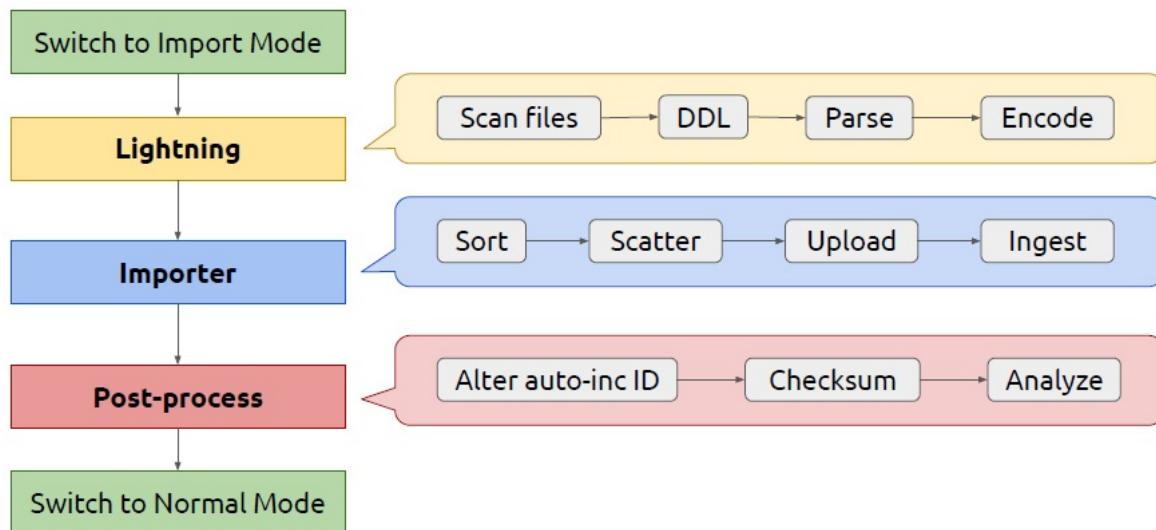
那么，为什么要把一个流程拆分成两个组件呢？

- tidb-lightning 与 TiDB 密不可分，tikv-importer 则与 TiKV 紧密相连。在实现上，tidb-lightning 复用了 TiDB 的代码，tikv-importer 则引用 TiKV 为库。这样一来，tidb-lightning 与 tikv-importer 之间就出现了编程语言冲突：TiDB 使用 Go 实现，而 TiKV 则使用 Rust。拆分为各自独立的组件更方便开发，而双方都需要的键值对可以透过 gRPC 传递。
- 分开 tidb-lightning 和 tikv-importer 也使得横向扩展更为灵活。例如，前端可以运行多个 tidb-lightning，传送键值对给同一个后端 tikv-importer。

总体而言，TiDB Lightning 工具的设计思路是，绕过 SQL 层，在线下将数据文件转化为键值对，并生成排好序的 SST 文件，直接推送到 TiKV 层的 RocksDB 里。这种批处理方式可以绕过 TiDB 层复杂的 SQL 和事务处理，省却 TiKV 层线上排序等耗时步骤，提升数据导入过程的整体效率。

## 数据导入过程

# Import Process



1. 导入数据之前，tidb-lightning 会自动将 TiKV 集群切换为“导入模式”（import mode）以优化写入效率。
2. tidb-lightning 会在目标 TiDB 集群上创建好空数据库和表。
3. 每张表的数据文件都会被分割为多个连续的批次，这样就能实现大表（200 GB 以上）的并行数据导入了。
4. tidb-lightning 会并发读取数据文件，转换成与目标 TiDB 集群相同编码的键值对，然后发送到 tikv-importer。tidb-lightning 通过 gRPC 传递键值对给 tikv-importer。tikv-importer 会为每一个批次的键值对准备一个“引擎文件”（engine file）。
5. 当一个引擎文件数据写入完毕，tikv-importer 便开始对目标 TiKV 集群进行 Region 分裂和调度，然后执行数据导入。有两种引擎文件：数据引擎与索引引擎，它们分别对应两种键值对：行数据和次级索引。通常，行数据在数据文件里是完全有序的，而次级索引则是无序的。因此，数据引擎文件在对应 Region 写入完成后会被立即上传，而索引引擎文件只有在整张表所有 Region 编码完成后才会执行导入。
6. 一张表的两种引擎文件都完成了导入之后，tidb-lightning 会对比本地数据文件及目标 TiDB 集群数据的 Checksum，确保数据完整性；然后，让 TiDB 运行 ANALYZE TABLE 命令更新表和索引的统计信息，为后续 TiDB 生成正确的 SQL 执行计划做好准备。同时，tidb-lightning 会调整表的 AUTO\_INCREMENT 值防止后续新增数据时发生冲突。表的自增 ID 是通过行数的上界估计值得到的，与表的数据文件总大小成正比。因此，最后的自增 ID 通常比实际行数大得多。考虑到 TiDB 中自增 ID 不一定是连续分配的，这种状况是可接受的。
7. 在所有步骤完毕后，tidb-lightning 会自动将 TiKV 切换回“普通模式”（normal mode），此后 TiDB 集群才可以正常对外提供服务。

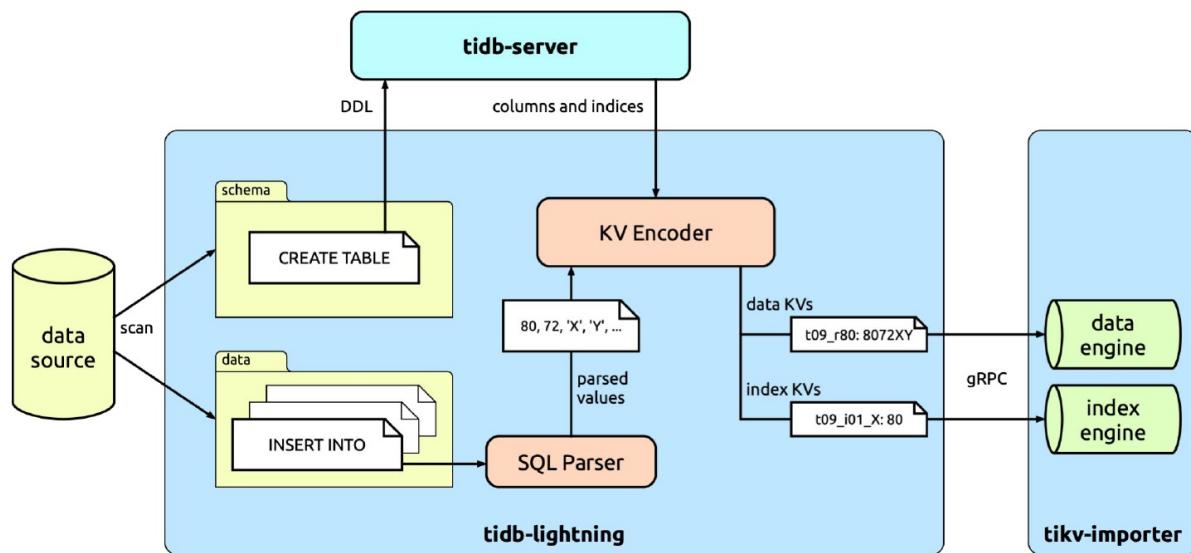
## 导入模式

一旦目标 TiKV 集群切换到导入模式，整个数据导入阶段该集群将被 tidb-lightning 独占，无法对外提供正常服务。tidb-lightning 会修改下列集群配置以提高数据导入速度：

- 增加 TiKV 后台任务数，以并行接收更多的 SST 文件。
- 移除 write stall triggers，使写速度优先于读速度。

数据导入完成后，tidb-lightning 会自动把 TiKV 集群切换回“普通模式”。

## 2. tidb-lightning 架构



## 工作原理

tidb-lightning 会扫描数据文件，区分出结构文件（包含 `CREATE TABLE` 语句）和数据文件（包含 `INSERT` 语句）。结构文件的内容会直接发送到 TiDB，用于建立数据库和表。然后，tidb-lightning 会并发处理数据文件。这里，我们来具体看一下一张表的导入处理过程。

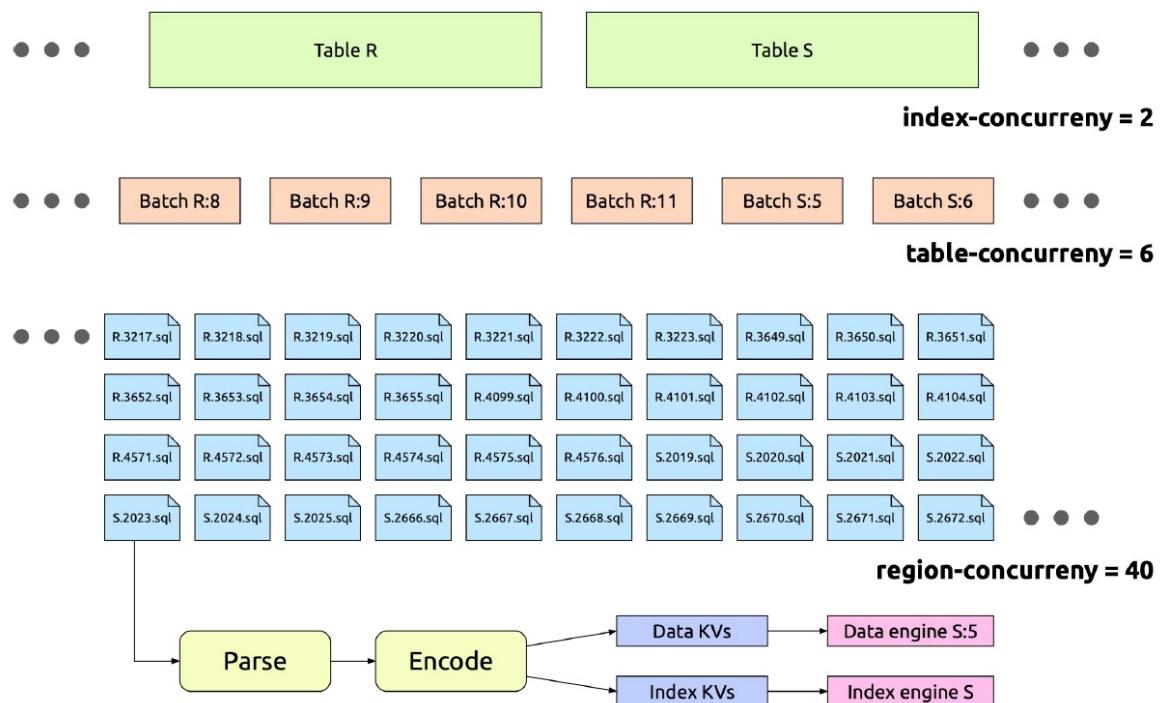
每张表的数据文件内容都是规律的 `INSERT` 语句，如下所示：

```
INSERT INTO `tbl` VALUES (1, 2, 3), (4, 5, 6), (7, 8, 9);
INSERT INTO `tbl` VALUES (10, 11, 12), (13, 14, 15), (16, 17, 18);
INSERT INTO `tbl` VALUES (19, 20, 21), (22, 23, 24), (25, 26, 27);
```

tidb-lightning 会找出每一行的位置，并分配一个行号，这样即使没有定义主键的表也能够区分每一行。tidb-lightning 会直接借助 TiDB 实例把 SQL 转换为键值对，称为“键值编码器”（KV encoder）。与外部的 TiDB 集群不同，键值编码器是寄存在 tidb-lightning 进程内的，并使用内存存储；每执行完一个 `INSERT` 之后，tidb-lightning 可以直接读取内存获取转换后的键值对（这些键值对包含数据及索引），并发送到 tikv-importer。

## 并发设置

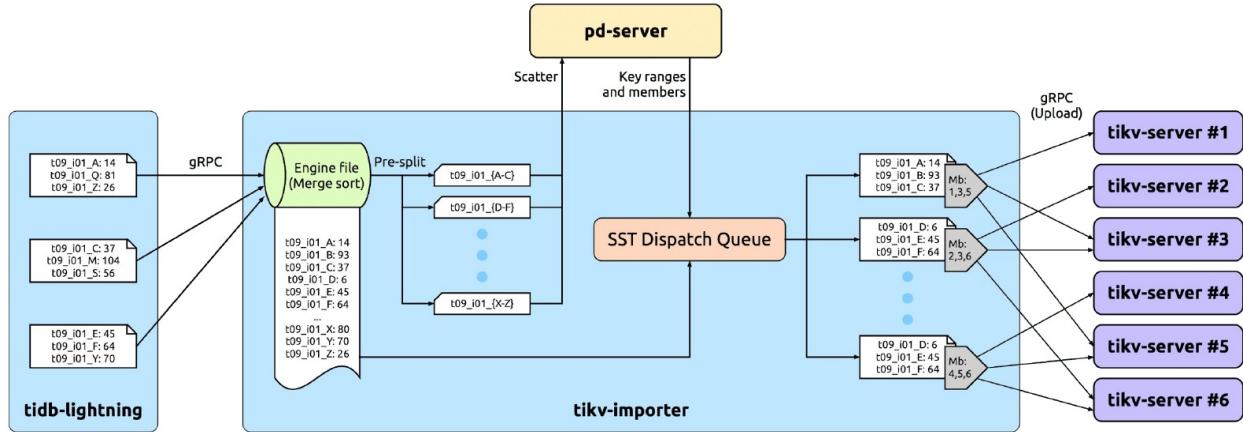
tidb-lightning 把数据文件拆分成多个能并发执行的小任务。下面的配置选项可以帮助调节这些任务的并发度：



- **batch-size** : 对于很大的表，比如超过 5 TB 的表，如果一次性导入，可能会因为 tikv-importer 磁盘空间不足导致失败。tidb-lightning 会按照 `batch-size` 的配置对一个大表进行切分，导入过程中每个批次使用单独的引擎文件。`batch-size` 不应该小于 100 GB，太小的话会使 region balance 和 leader balance 值升高，导致 Region 在 TiKV 不同节点之间频繁调度，浪费网络资源。
- **table-concurrency** : 同时导入的批次个数。如上所述，每个表会按照 `batch-size` 切分成多个批次。
- **index-concurrency** : 并行的索引引擎文件个数。`table-concurrency + index-concurrency` 的总和必须小于 tikv-importer 的 `max-open-engines` 配置。
- **io-concurrency** : 并发访问磁盘的 I/O 线程数。由于磁盘内部缓存容量有限，过高的并发度容易引发频繁的 cache miss，导致 I/O 延迟增大。因此，不建议将该值调整得太大。
- **block-size** : 默认值为 64 KB。tidb-lightning 会一次性读取一个 `block-size` 大小的数据文件，然后进行编码。
- **region-concurrency** : 每个批次的内部线程数。每个线程要执行读文件、编码和发送到 tikv-importer 等步骤。
  - 读文件会消耗 I/O 资源，需要调节 `io-concurrency` 控制并发读取。
  - 编码过程的瓶颈主要在 CPU，需要适当调整 `region-concurrency` 配置。
  - 举例来说，若一次编码处理耗时 50 毫秒，那么每秒只能进行 20 次编码。若 `block-size` 为 64 KB，则单一 CPU 核每秒最多完成 1.28 MB 数据的编码处理。当 `region-concurrency` 设置为 60，则整体编码处理的极限速度约为每秒 75 MB。

### 3. tikv-importer 架构

#### 工作原理



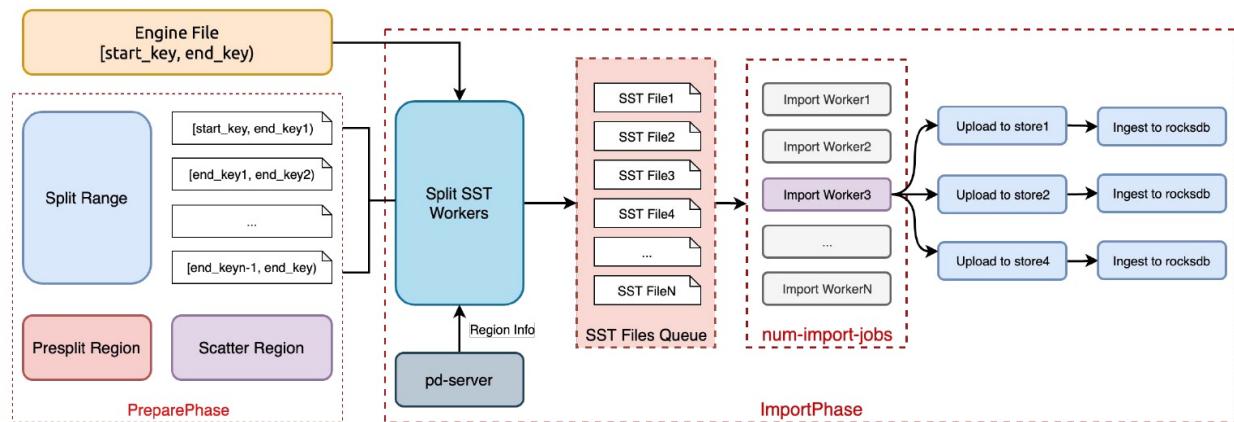
因异步操作的缘故，tikv-importer 得到的原始键值对注定是无序的。所以，tikv-importer 要做的第一件事就是要排序。这需要给每个表划定准备排序的储存空间，我们称之为引擎文件。

对大数据排序是个解决了很多遍的问题，我们在此使用现有的答案：直接使用 RocksDB。一个引擎文件就相等于一个本地 RocksDB，并大量写入操作做了配置上的优化。排序就相当于将键值对全部写入到引擎文件里，然后 RocksDB 就会自动帮我们合并、排序，最终得到 SST 格式的文件。

SST 文件包含整个表的数据和索引，和 TiKV 的储存单位 Region 比起来实在太大了。所以接下来要切分成合适的大小（默认为 96 MB）。tikv-importer 会根据要导入的数据范围预先把 Region 分裂好，然后借助 PD 把这些分裂出来的 Region 分散调度到不同的 TiKV 实例上。

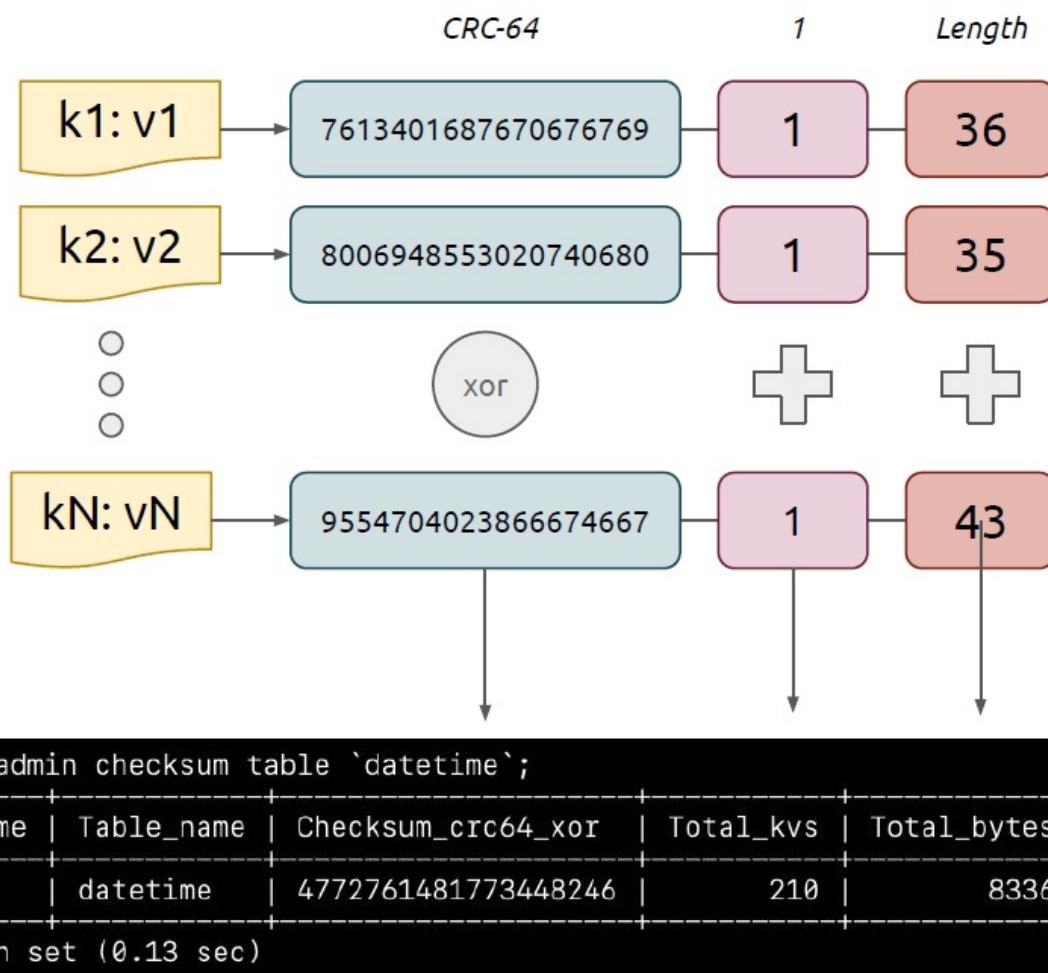
最后，tikv-importer 将 SST 上传到对应 Region 的每个副本上，通过 Leader 发起 Ingest 命令，把 SST 文件导入到 Raft group 里。这样就完成了一个 Region 的导入过程。

## 并发设置



- `max-open-engines`：表示 tikv-importer 上可以同时打开的最大引擎文件数量。如果运行单个 tidb-lightning 实例，该配置不应小于 tidb-lightning 的 `index-concurrency` + `table-concurrency`；多个 Lightning 实例并行运行的状况下，不能小于所有实例的 `index-concurrency` + `table-concurrency` 总和。请注意，引擎文件会消耗磁盘空间，需要根据 tikv-importer 节点的磁盘容量合理配置本参数。一个数据引擎文件的磁盘空间占用等于 tidb-lightning 中 `batch-size` 大小；索引引擎文件的大小可参考后面“一节提供的思路进行估算”。
- `num-import-jobs`：一个 `batch-size` 大小的数据写入到引擎文件后，会有若干个线程负责将其导入 TiKV。这个参数控制同时进行导入的线程数量，通常使用默认配置即可。
- `region-split-size`：一个引擎文件可能会很大（比如 100 GB），很难一次性导入到 TiKV。需要把引擎文件切分成多个较小的 SST 文件，SST 文件不会超过 `region-split-size` 值。通常，不建议低于 96 MB，因为 SST 切分过小会导致 Ingest 的吞吐量小。

## 4. 数据校验



完成数据导入后会自动执行数据校验以确保数据完整性。tidb-lightning 会在每个表完成导入后，对比导入前后的 Checksum 确认二者是否一致。

一个表的 Checksum 是透过计算键值对的哈希值产生的。因为键值对分布在不同的 TiKV 实例上，这个 Checksum 函数应该具备结合性；另外，tidb-lightning 传送键值对之前它们是无序的，所以 Checksum 也不应该考虑顺序，即服从交换律。也就是说，Checksum 计算并不是简单地针对整个 SST 文件计算 SHA-256。

我们的解决办法是这样的：先计算每个键值对的 CRC64，然后用 XOR 结合在一起，得出一个 64 位元的校验数字。为降低 Checksum 值冲突的概率，我们同时会计算键值对的数量和大小。在下面两个地方分别计算来比对表中 3 个指标的和：

- 一次是导入前在 tidb-lightning 编码后。
- 一次是导入后在 TiDB 上执行如下 SQL 命令：

```
ADMIN CHECKSUM TABLE `xxxx`;
```

## 5. 分析与更新自增值

数据校验结束后，tidb-lightning 会重新计算表的统计信息，并更新表的自增值：

```
ANALYZE TABLE `xxxx`;
ALTER TABLE `xxxx` AUTO_INCREMENT=123456;
```

## 6. 使用限制

数据导入开始前须确保以下两点：

- 清空目标表里的数据。
- 不能有其他业务写入数据到目标表。

一个表只能接受一个 tidb-lightning 实例导入，多个 tidb-lightning 实例不能同时导入数据到同一张表。同一个数据文件若需要使用多个 tidb-lightning 实例并行导入，应该修改白名单配置以确保一个表只接受一个 tidb-lightning 实例导入。

## 7. 机器配置要求

- tidb-lightning 和 tikv-importer 均须配备万兆网卡。
- tidb-lightning 对本地磁盘空间大小没有硬性要求。为保证长时间运行数据导入处理，须保证磁盘空间足够保存日志文件。
- tikv-importer 对磁盘空间大小有一定要求。请参考后面“量化指标”一节提供的估算思路。

## 8. 量化指标

下面给出的一些公式和计算方法可以帮助我们计算数据导入过程中的资源使用量。这里，我们假定 tidb-lightning 和 tikv-importer 之间的交互过程不是性能瓶颈所在。

- 假定 tikv-importer 节点使用万兆网卡（理论带宽上限为 10 gbps），则编码速度最高达到每秒 300 MB 时就会耗尽全部带宽。这就是使用 TiDB Lightning 工具导入数据的理论上限速度。

```
max-speed = bandwidth (1.2 GB/s) / replicas (3)
```

- tidb-lightning 的内存占用很低，几乎可以忽略；tikv-importer 的内存占用取决于引擎文件个数和导入线程数。

```
ram-usage = (max-open-engines (8) × write-buffer-size (1 GB) × 2)
           + (num-import-jobs (24) × region-split-size (512 MB) × 2)
```

- tikv-importer 的磁盘空间使用量基本上取决于最大的 N 个表，其中  $N = \max(index-concurrency, table-concurrency)$ 。实际的磁盘空间使用量还和这些表的索引数量以及索引字段构成相关。假定需要导入 20 张表，其中有一张 5 TB 的大表。考虑到该表的索引键值对需要在 tikv-importer 上先进行全量排序后再上传到 TiKV 中，所以需要保证 tikv-importer 的本地磁盘空间至少可以保存对应的索引引擎文件。索引引擎文件在磁盘上的大小主要由表结构里的索引数量决定；当然，如果索引中的字段以整数类型为主，则索引引擎文件会更小一些。这里提供一个经验值：一个包含 5 个索引、体积为 4 TB 的表，对应的索引引擎文件体积约为 2 TB。
- 一般而言，目标 TiKV 集群的磁盘容量可以按照数据文件体积的 4 倍来估算。例如，数据文件体积为 5 TB，则目标 TiKV 集群至少要预留 20 TB 可用空间。如果单表内重复数据较多，最终 TiKV 的压缩比也会较大，则相应地容量要求会降低。

## 2.2.2 Lightning 实操指南

这一节将介绍如何使用 Lightning 导入数据的实操

### 2.2.2.1 TiDB Lightning 快速开始

#### 注意

- TiDB Lightning 运行后，TiDB 集群将无法正常对外提供服务。
- 若 `tidb-lightning` 崩溃，集群会留在“导入模式”。若忘记转回“普通模式”，集群会产生大量未压缩的文件，继而消耗 CPU 并导致延迟。此时，需要使用 `tidb-lightning-ctl` 手动将集群转回“普通模式”：

```
.../tidb-ansible/resource/bin/tidb-lightning-ctl -switch-mode=normal
```

#### 数据库权限要求

TiDB Lightning 需要下游 TiDB 具有如下权限：

| 权限  | SELECT | INSERT | UPDATE | DELETE | CREATE            | DROP              | ALTER  |
|-----|--------|--------|--------|--------|-------------------|-------------------|--------|
| 作用域 | Tables | Tables | Tables | Tables | Databases, tables | Databases, tables | Tables |

如果配置项 `checksum = true`，则 TiDB Lightning 需要有下游 TiDB admin 用户权限。

#### 硬件需求

`tidb-lightning` 和 `tikv-importer` 这两个组件皆为资源密集程序，建议各自单独部署。

为了保证导入效能，建议最低硬件配置如下：

`tidb-lightning`

- 32+ 逻辑核 CPU
- 足够储存整个数据源的 SSD 硬盘，读取速度越快越好
- 使用万兆网卡，带宽需 300 MB/s 以上
- 运行过程默认会占满 CPU，建议单独部署。条件不允许的情况下可以和其他组件（比如 `tidb-server`）部署在同一台机器上，然后通过配置 `region-concurrency` 限制 `tidb-lightning` 使用 CPU 资源。

`tikv-importer`

- 32+ 逻辑核 CPU
- 40 GB+ 内存
- 1 TB+ SSD 硬盘，IOPS 越高越好（要求  $\geq 8000$ ）
- 硬盘必须大于最大的 N 个表的大小总和，其中  $N = \max(\text{index-concurrency}, \text{table-concurrency})$ 。
- 使用万兆网卡，带宽需 300 MB/s 以上
- 运行过程中 CPU、I/O 和网络带宽都可能占满，建议单独部署。

如果机器充裕的话，可以部署多套 `tidb-lightning + tikv-importer`，然后将源数据以表为粒度进行切分，并发导入。

#### 使用 TiDB-Ansible 部署 TiDB Lightning

此方法操作简单，不需要过多的修改配置文件。需完成 TiDB-Ansible 用户的创建、互信、 sudo 操作。

(1) 编辑 inventory.ini，分别配置一个 IP 来部署 tidb-lightning 和 tikv-importer。

```
...
[importer_server]
# import_dir 为转换的中间数据存放路径
IS1 ansible_host=172.16.4.1 deploy_dir=/data/deploy tikv_importer_port=8287 import_dir=/data/import
[lightning_server]
# data_source_dir 为需导入的文件存放路径
LS1 ansible_host=172.16.4.2 deploy_dir=/data/deploy tidb_lightning_pprof_port=8289 data_source_dir=/data/wanted
...
```

(2) 准备需要导入的数据放到配置文件中 data\_source\_dir 指定的路径。数据可以是 mydumper 备份的 sql 文件或者是 csv 文件。如果是 csv 文件，则需要做额外配置。修改 conf/tidb-lightning.yml

```
...
[mydumper]
no-schema: true
[mydumper.csv]
# 字段分隔符，必须为 ASCII 字符。
separator = ','
# 引用定界符，可以为 ASCII 字符或空字符。
delimiter = ""
# CSV 文件是否包含表头。
# 如果为 true，首行将被跳过。
header = true
# CSV 是否包含 NULL。
# 如果为 true，CSV 文件的任何列都不能解析为 NULL。
not-null = false
# 如果 `not-null` 为 false（即 CSV 可以包含 NULL），
# 为以下值的字段将会被解析为 NULL。
null = '\N'
# 是否解析字段内的反斜线转义符。
backslash-escape = true
# 是否移除以分隔符结束的行。
trim-last-separator = false
...
```

(3) 初始化 Lightning 和 Importer

```
$ ansible-playbook bootstrap.yml -l IS1, LS
```

(4) 部署 Lightning 和 Importer

```
$ ansible-playbook deploy.yml -l IS1, LS
或者
$ ansible-playbook deploy.yml --tags=lightning
```

(5) 启动 Importer 以及 Lightning

注意，必须先启动 Importer，再启动 Lightning，顺序不能换。

1. 登录到部署 Importer 以及 Lightning 的服务器
2. 进入部署目录
3. 在 Importer 目录下执行 scripts/start\_importer.sh，启动 Importer
4. 在 Lightning 目录下执行 scripts/start\_lightning.sh，开始导入数据

(6) 查看导入状态

- 使用 grafana 监控查看，后面会详细介绍。
- 使用日志查看

若数据顺利导入完成，lightning 日志会显示["the whole procedure completed"]，["tidb lightning exit"]等关键信息。

## (7) 关闭 Importer

数据导入完成后，在 Importer 目录下执行 `scripts/stop_importer.sh`

## 2.2.2.2 手动部署 TiDB Lightning

从官网下载与 TiDB 版本一致的 Lightning 安装包，并将安装包上传至满足硬件要求的服务器。解压安装包后在 bin 目录下新建 `tikv-importer.toml`, `tidb-lightning.toml`，详细配置说明见后面小节，TiDB Lightning 启停等使用方式与 ansible 部署一致。

## 2.2.2.3 TiDB Lightning TiDB-Backend

### Importer-backend 和 TiDB-backend 的区别

TiDB Lightning 的后端决定 `tidb-lightning` 将如何把将数据导入到目标集群中。目前，TiDB Lightning 支持 Importer-backend（默认）和 TiDB-backend 两种后端，两者导入数据的区别如下：

- Importer-backend：`tidb-lightning` 先将 SQL 或 CSV 数据编码成键值对，由 `tikv-importer` 对写入的键值对进行排序，然后把这些键值对 `Ingest` 到 TiKV 节点中。
- TiDB-backend：`tidb-lightning` 先将数据编码成 `INSERT` 语句，然后直接在 TiDB 节点上运行这些 SQL 语句进行数据导入。

| 后端           | Importer-backend | TiDB-backend  |
|--------------|------------------|---------------|
| 对集群的影响       | 超大               | 小             |
| 速度           | 快 (~300 GB/小时)   | 慢 (~50 GB/小时) |
| 资源使用率        | 高                | 低             |
| 导入时是否满足 ACID | 否                | 是             |
| 目标表          | 必须为空             | 可以不为空         |

TiDB Lightning TiDB-Backend 运行时，TiDB 集群可基本正常对外提供服务。

## 硬件需求

使用 TiDB-backend 时，TiDB Lightning 的速度仅受限于 TiDB 执行 SQL 语句的速度。因此，即使是低配的机器也能够发挥出最佳性能。推荐的硬件配置如下：

- 16 逻辑核 CPU
- 足够储存整个数据源的 SSD 硬盘，读取速度越快越好
- 千兆网卡

## 部署 TiDB-backend

使用 TiDB-backend 时，你无需部署 `tikv-importer`。与之前的部署相比，部署 TiDB-backend 时有如下不同：

- 可以跳过所有涉及 `tikv-importer` 的步骤。
- 必须更改相应配置申明使用的是 TiDB-backend。

(1) 编辑 `inventory.ini`，`[importer_server]` 部分可以留空。

```

...
[importer_server]
# keep empty
[lightning_server]
# data_source_dir 为需导入的文件存放路径
LS1 ansible_host=172.16.4.2 deploy_dir=/data/deploy tidb_lightning_pprof_port=8289 data_source_dir=/data/wante
...

```

(2) 准备需要导入的数据放到配置文件中 data\_source\_dir 指定的路径。同时修改 conf/tidb-lightning.yml 中的 backend 参数

```

tikv_importer:
backend: "tidb" # <-- 改成 "tidb"

```

数据可以是 mydumper 备份的 sql 文件或者是 csv 文件。如果是 csv 文件，则需要做额外配置。修改 conf/tidb-lightning.yml

```

...
[mydumper]
no-schema: true
[mydumper.csv]
# 字段分隔符，必须为 ASCII 字符。
separator = ','
# 引用定界符，可以为 ASCII 字符或空字符。
delimiter = ""
# CSV 文件是否包含表头。
# 如果为 true，首行将会被跳过。
header = true
# CSV 是否包含 NULL。
# 如果为 true，CSV 文件的任何列都不能解析为 NULL。
not-null = false
# 如果 `not-null` 为 false（即 CSV 可以包含 NULL），
# 为以下值的字段将会被解析为 NULL。
null = '\N'
# 是否解析字段内的反斜线转义符。
backslash-escape = true
# 是否移除以分隔符结束的行。
trim-last-separator = false
...

```

(3) 初始化 Lightning

```
$ ansible-playbook bootstrap.yml -l LS
```

(4) 部署 Lightning

```

$ ansible-playbook deploy.yml -l LS
或者
$ ansible-playbook deploy.yml --tags=lightning

```

(5) 启动 Lightning

1. 登录到 Lightning 的服务器
2. 进入部署目录
3. 在 Lightning 目录下执行 `scripts/start_lightning.sh`，开始导入数据

(6) 查看导入状态

使用 gafana 监控查看

使用日志查看

## 2.2.2.4 TiDB Lightning Web 界面

TiDB Lightning 支持在网页上查看导入进度或执行一些简单任务管理，这就是 TiDB Lightning 的服务器模式。本文将介绍服务器模式下的 Web 界面和一些常见操作。

## 注意

服务器模式下，TiDB Lightning 不会立即开始运行，而是通过用户在 web 页面提交（多个）任务来导入数据。

## 启动方式

启用服务器模式的方式有如下几种：

- 在启动 `tidb-lightning` 时加上命令行参数 `--server-mode`。

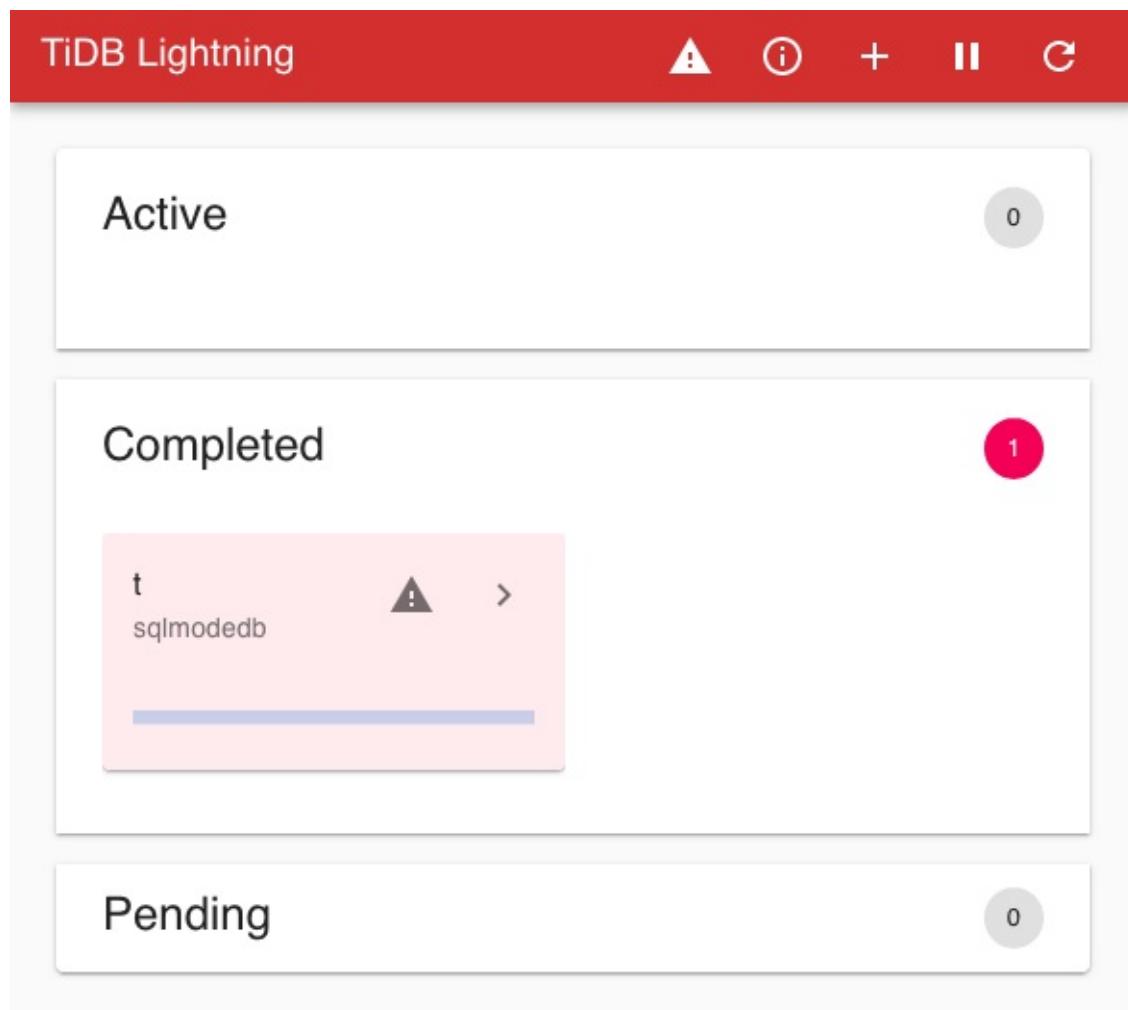
```
./tidb-lightning --server-mode --status-addr :8289
```

- 在 `tidb-lightning.toml` 配置文件中设置 `lightning.server-mode`。

```
[lightning]
server-mode = true
status-addr = ':8289'
```

TiDB Lightning 启动后，可以访问 <http://127.0.0.1:8289> 来管理程序（实际的 URL 取决于你的 `status-addr` 设置）。

## TiDB Lightning Web 首页



标题栏上图标所对应的功能，从左到右依次为：

| 图标               | 功能                              |
|------------------|---------------------------------|
| “TiDB Lightning” | 点击即返回首页                         |
| △                | 显示前一个任务的所有错误信息                  |
| ①                | 列出当前及队列中的任务，可能会出现一个标记提示队列中任务的数量 |
| +                | 提交单个任务                          |
| II/▶             | 暂停/继续当前操作                       |
| ⟳                | 设置网页自动刷新                        |

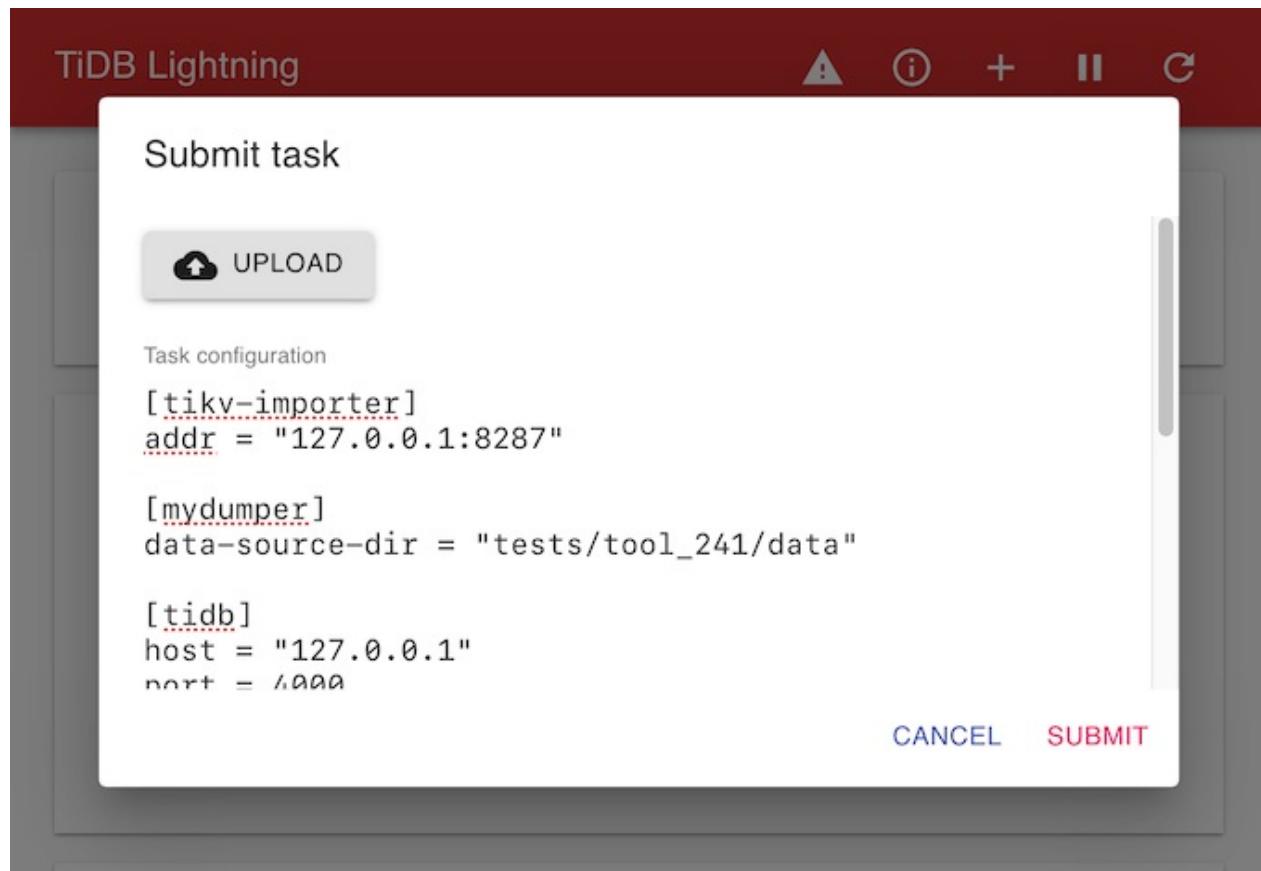
标题栏下方的三个面板显示了不同状态下的所有表：

- Active：当前正在导入这些表
- Completed：这些表导入成功或失败
- Pending：这些表还没有被处理

每个面板都包含用于描述表状态的卡片。

## 提交任务

点击标题栏的 + 图标提交任务。



任务 (task) 为 TOML 格式的文件，具体参考 [TiDB Lightning 任务配置](#)。你也可以点击 UPLOAD 上传一个本地的 TOML 文件。

点击 SUBMIT 运行任务。如果当前有任务正在运行，新增任务会加入队列并在当前任务结束后执行。

## 查看导入进度

点击首页表格卡片上的 > 图标，查看表格导入的详细进度。

| Engines   |               |       |
|-----------|---------------|-------|
| Engine ID | Status        | Files |
| :1        | • • • writing | 0     |
| :0        | • • • writing | 1     |

| Files   |        |                                 |
|---|--------|---------------------------------|
| Chunk   | Engine | Progress                        |
| lightning/mydump/examples/mocker_test.tbl_multi_index.sql:0 | :0     | <div style="width: 10%;"></div> |

该页显示每张表的引擎文件的导入过程。

点击标题栏上的 TiDB Lightning 返回首页。

## 管理任务

单击标题栏上的 ① 图标来管理当前及队列中的任务。

```
{
  "id": 1565549873900573000,
  "lightning": {
    "table-concurrency": 1,
    "index-concurrency": 2,
    "region-concurrency": 4,
    "io-concurrency": 5,
    "check-requirements": false
  },
  "tidb": {
    "host": "127.0.0.1",
    "port": 4000,
    "user": "root",
    "status-port": 10080,
    "pd-addr": "127.0.0.1:2379",
    "sql-mode": "ONLY_FULL_GROUP_BY_STRICT_TRANS_TABLES"
  }
}
```

每个任务都是依据提交时间来标记。点击该任务将显示 JSON 格式的配置文件。

点击任务上的 : 可以对该任务进行管理。你可以立即停止任务，或重新排序队列中的任务。

### 2.2.2.5 库表过滤

TiDB Lightning 通过配置黑白名单的方式来过滤掉某些数据库和表。

#### 库过滤

```
[black-white-list]
do-dbs = ["pattern1", "pattern2", "pattern3"]
ignore-dbs = ["pattern4", "pattern5"]
```

- 如果 [black-white-list] 下的 do-dbs 列表不为空，数据库名称匹配 do-dbs 列表中任何一项，则数据库会被导入。否则，数据库会被略过。
- 如果 do-dbs 列表为空，数据库名称匹配 ignore-dbs 列表中任何一项，数据库会被略过。
- 如果数据库名称同时匹配 do-dbs 和 ignore-dbs 列表，数据库会被导入。

#### 表过滤

```
[[black-white-list.do-tables]]
db-name = "db-pattern-1"
tbl-name = "table-pattern-1"
#可定义多种匹配规则
[[black-white-list.do-tables]]
db-name = "db-pattern-2"
tbl-name = "table-pattern-2"
[[black-white-list.ignore-tables]]
db-name = "db-pattern-3"
tbl-name = "table-pattern-3"
[[black-white-list.ignore-tables]]
db-name = "db-pattern-4"
tbl-name = "table-pattern-4"
```

- 如果 do-tables 列表不为空，表的限定名称匹配 do-tables 列表中任何一对，则表会被导入，否则，表会被略过。
- 如果 do-tables 列表不空，表的限定名称匹配 ignore-tables 列表中任何一对，表会被略过。
- 如果表的限定名称同时匹配 do-tables 和 ignore-tables 列表，表会被导入。

例子：数据源存在如下库表信息：

| 库名                | 表名  |
|-------------------|---|
| logs              | messages_2016 , messages_2017 , messages_2018 |
| forum             | messages                                      |
| forum_backup_2016 | messages                                      |
| forum_backup_2017 | messages                                      |
| forum_backup_2018 | messages                                      |
| admin             | secrets                                       |

配置文件内容如下：

```
[black-white-list]
do-dbs = [
    "forum_backup_2018",    # 规则 A
    "~^(logs|forum)$",      # 规则 B, 首字符 ~ 会被解析为 Go 语言的正则表达式。
]
ignore-dbs = [
    "~^forum_backup_",
    # 规则 C
]
[[black-white-list.do-tables]]      # 规则 D
db-name = "logs"
tbl-name = "~_2018$"

[[black-white-list.ignore-tables]]  # 规则 E
db-name = ".*"
tbl-name = "~^messages.*"

[[black-white-list.do-tables]]      # 规则 F
db-name = "~^forum.*"
tbl-name = "messages"
```

首先进行库过滤：|数据库|结果||----:----:| | logs | 导入 (规则 B) || forum | 导入 (规则 B) || forum\_backup\_2016 | 略过 (规则 C) || forum\_backup\_2017 | 略过 (规则 C) | | forum\_backup\_2018 | 导入 (规则 A) (不会考虑规则 C) || admin | 略过 (do-dbs 不为空，且没有匹配的项目 |

再进行表过滤：

| 表   | 结果                           |
|---|------------------------------|
| logs . messages_2016 , logs . messages_2017               | 略过 (规则 E)                    |
| logs . messages_2018                                      | 导入 (规则 D) (不会考虑规则 E)         |
| forum . users   | 略过 (do-tables 不为空, 且没有匹配的项目) |
| forum . messages  | 导入 (规则 F) (不会考虑规则 E)         |
| forum_backup_2016 . messages forum_backup_2017 . messages | 略过 (数据库已被剔除)                 |
| forum_backup_2018 . message                               | 导入 (规则 F) (不会考虑规则 E)         |
| admin . secrets   | 略过 (数据库已被剔除)                 |

## 2.2.2.6 断点续传

在数据量很大的迁移时，数据导入时间较长，长时间运行的进程有可能异常中断。若此时每次重启都从头开始导入，会浪费大量时间。Lightning断点续传功能可以实现重启时仍然接着之前的进度继续工作。

### 断点续传的启用

```
[checkpoint]
# 启用断点续传。
# 导入时, Lightning 会记录当前进度。
enable = true

# 存储断点的方式
# - file: 存放在本地文件系统 (要求 v2.1.1 或以上)
# - mysql: 存放在兼容 MySQL 的数据库服务器
driver = "file"

# 存储断点的架构名称 (数据库名称)
# 仅在 driver = "mysql" 时生效
# schema = "tidb_lightning_checkpoint"

# 断点的存放位置
# 如果不设置改参数则默认为 `/tmp/CHECKPOINT_SCHEMA.pb`。
# 若driver = "mysql", dsn为数据库连接参数, 格式为"用户:密码@tcp(地址:端口)/"。
# dsn = "/tmp/tidb_lightning_checkpoint.pb"
# 导入成功后是否保留断点。默认为删除。
# keep-after-success = false
```

TiDB Lightning 支持两种存储方式：本地文件或 MySQL 数据库。

- 若 driver = "file"，断点会存放在一个本地文件，其路径由 dsn 参数指定。由于断点会频繁更新，建议将这个文件放到写入次数不受限制的盘上，例如 RAM disk。
- 若 driver = "mysql"，断点可以存放在任何兼容 MySQL 5.7 或以上的数据库中，包括 MariaDB 和 TiDB。在没有选择的情况下，默认会存在目标数据库里(不建议)，建议另外部署一台兼容 MySQL 的临时数据库服务器。此数据库也可以安装在 tidb-lightning 的主机上。导入完毕后可以删除。

### 断点续传的控制

- 若 tidb-lightning 因不可恢复的错误而退出（例如数据出错），重启时不会使用断点，而是直接报错离开。为保证已导入的数据安全，这些错误必须先解决掉才能继续。使用 tidb-lightning-ctl 工具可以标示已经恢复。传入指定表清除错误数据

```
# 若导入 `schema`.`table` 这个表曾经出错，此命令会从目标数据库DROP这个表，清除已导入的数据，将断点重设到“未开始”的状态
# 如果 `schema`.`table` 没有出错，则无操作
$ tidb-lightning-ctl --checkpoint-error-destroy='`schema`.`table`'
```

- 传入 “all” 会对所有表进行上述操作。这是最方便、安全但保守的断点错误解决方法。

```
$ tidb-lightning-ctl --checkpoint-error-destroy=all
```

- 如果导入 `schema . table` 这个表导入出错，发现此错误可以忽略，这条命令会清除出错状态，传入 “all” 会对所有表进行上述操作。

```
$ tidb-lightning-ctl --checkpoint-error-ignore='`schema`.`table`' &&
$ tidb-lightning-ctl --checkpoint-error-ignore=all
```

注意：除非确定错误可以忽略，否则不要使用这个选项。如果错误是真实的话，可能会导致数据不完全。启用校验和 (CHECKSUM) 可以防止数据出错被忽略。

- 无论是否出错，把断点清除，使用`--checkpoint-remove`。

```
$ tidb-lightning-ctl --checkpoint-remove='`schema`.`table`' &&
$ tidb-lightning-ctl --checkpoint-remove=all
```

- 将所有断点备份到传入的文件夹，使用`--checkpoint-dump`

```
#主要用于技术支持。此选项仅于 driver = "mysql" 时有效
$ tidb-lightning-ctl --checkpoint-dump=output/directory
```

## 2.2.2.7 CSV 支持

TiDB Lightning 支持读取 CSV（逗号分隔值）的数据源，以及其他定界符格式如 TSV（制表符分隔值）。

### 文件名

包含整张表的 CSV 文件需命名为 `db_name.table_name.csv`，该文件会被解析为数据库 `db_name` 里名为 `table_name` 的表。

如果一个表分布于多个 CSV 文件，这些 CSV 文件命名需加上文件编号的后缀，如 `db_name.table_name.003.csv`。

文件扩展名必须为 `*.csv`，即使文件的内容并非逗号分隔。

### 表结构

CSV 文件是没有表结构的。要导入 TiDB，就必须为其提供表结构。可以通过以下任一方法实现：

- 创建包含 DDL 语句 CREATE TABLE 的文件 `db_name.table_name-schema.sql`。
- 首先在 TiDB 中直接创建空表，然后在 `tidb-lightning.toml` 中设置 `[mydumper] no-schema = true`。

### 配置

CSV 格式可在 `tidb-lightning.toml` 文件中 `[mydumper.csv]` 下配置。大部分设置项在 MySQL `LOAD DATA` 语句中都有对应的项目。

```
[mydumper.csv]
# 字段分隔符，必须为 ASCII 字符。
separator = ','
# 引用定界符，可以为 ASCII 字符或空字符。
delimiter = ""
# CSV 文件是否包含表头。
# 如果为 true，首行将会被跳过。
header = true
# CSV 是否包含 NULL。
# 如果为 true, CSV 文件的任何列都不能解析为 NULL。
not-null = false
# 如果 `not-null` 为 false (即 CSV 可以包含 NULL) ,
# 为以下值的字段将会被解析为 NULL。
null = '\N'
# 是否解析字段内的反斜线转义符。
backslash-escape = true
# 是否移除以分隔符结束的行。
trim-last-separator = false
```

## separator

- 指定字段分隔符。
- 必须为单个 ASCII 字符。
- 常用值：
  - CSV 用 ';'
  - TSV 用 "\t"
- 对应 LOAD DATA 语句中的 FIELDS TERMINATED BY 项。

## delimiter

- 指定引用定界符。
- 如果 delimiter 为空，所有字段都会被取消引用。
- 常用值：
  - "" 使用双引号引用字段，和 [RFC 4180](#) 一致。
  - " 不引用
- 对应 LOAD DATA 语句中的 FIELDS ENCLOSED BY 项。

## header

- 是否所有 CSV 文件都包含表头行。
- 如为 true，第一行会被用作列名。如为 false，第一行并无特殊性，按普通的数据行处理。

## not-null 和 null

- not-null 决定是否所有字段不能为空。
- 如果 not-null 为 false，设定了 null 的字符串会被转换为 SQL NULL 而非具体数值。

引用不影响字段是否为空。

例如有如下 CSV 文件：

```
A,B,C
\N,"\"N",
```

- 在默认设置 (not-null = false; null = '\N') 下，列 A and B 导入 TiDB 后都将会转换为 NULL。列 C 是空字符串 "，但并不会解析为 NULL。

## backslash-escape

是否解析字段内的反斜线转义符。

- 如果 `backslash-escape` 为 `true`，下列转义符会被识别并转换。
   
| 转义符 | 转换为 | |:----|:----| \0 | 空字符 (U+0000) || \b | 退格 (U+0008) || \n | 换行 (U+000A) || \r | 回车 (U+000D) || \t | 制表符 (U+0009) || \Z | Windows EOF (U+001A) |
- 其他情况下（如 `\\"`）反斜线会被移除，仅在字段中保留其后面的字符（`"`）。引用不会影响反斜线转义符的解析与否。
- 对应 `LOAD DATA` 语句中的 `FIELDS ESCAPED BY \'` 项。

## trim-last-separator

将 `separator` 字段当作终止符，并移除尾部所有分隔符。

例如有如下 CSV 文件：

- `A,,B,,`
- 当 `trim-last-separator = false`，该文件会被解析为包含 5 个字段的行 (`'A', '', 'B', '', ''`)。
- 当 `trim-last-separator = true`，该文件会被解析为包含 3 个字段的行 (`'A', '', 'B'`)。

## 不可配置项

TiDB Lightning 并不完全支持 `LOAD DATA` 语句中的所有配置项。例如：

- 行终止符只能是 CR (`\r`)，LF (`\n`) 或 CRLF (`\r\n`)，也就是说，无法自定义 `LINES TERMINATED BY`。
- 不可使用行前缀 (`LINES STARTING BY`)。
- 不可跳过表头 (`IGNORE n LINES`)。如有表头，必须是有效的列名。
- 定界符和分隔符只能为单个 ASCII 字符。

## 通用配置

### CSV

默认设置已按照 RFC 4180 调整。

```
[mydumper.csv]
separator = ','
delimiter = ""
header = true
not-null = false
null = '\N'
backslash-escape = true
trim-last-separator = false
```

#### 示例内容

```
ID,Region,Count
1,"East",32
2,"South",\N
3,"West",10
4,"Nor
```

### TSV

```
[mydumper.csv]
separator = "\t"
delimiter = ''
header = true
not-null = false
null = 'NULL'
backslash-escape = false
trim-last-separator = false
```

示例内容：

```
ID Region Count
1 East 32
2 South NULL
3 West 10
4 North 39
```

## TPC-H DBGEN

```
[mydumper.csv]
separator = '|'
delimiter = ''
header = false
not-null = true
backslash-escape = false
trim-last-separator = true
```

示例内容：

```
1|East|32|
2|South|0|
3|West|10|
4|North|39|
```

## 2.2.2.8 TiDB Lightning 配置参数

你可以使用配置文件或命令行配置 TiDB Lightning。本文主要介绍 TiDB Lightning 的全局配置、任务配置和 TiKV Importer 的配置，以及如何使用命令行进行参数配置。

### 配置文件

TiDB Lightning 的配置文件分为“全局”和“任务”两种类别，二者在结构上兼容。只有当[服务器模式](#)开启时，全局配置和任务配置才会有区别；默认情况下，服务器模式为禁用状态，此时 TiDB Lightning 只会执行一个任务，且全局和任务配置使用同一配置文件。

#### TiDB Lightning 全局配置

```
### tidb-lightning 全局配置

[lightning]
# 用于拉取 web 界面和 Prometheus 监控项的 HTTP 端口。设置为 0 时为禁用状态。
status-addr = ':8289'

# 切换为服务器模式并使用 web 界面
# 详情参见“TiDB Lightning web 界面”文档
server-mode = false

# 日志
level = "info"
file = "tidb-lightning.log"
max-size = 128 # MB
max-days = 28
max-backups = 14
```

## TiDB Lightning 任务配置

```
### tidb-lightning 任务配置

[lightning]
# 启动之前检查集群是否满足最低需求。
# check-requirements = true

# 引擎文件的最大并发数。
# 每张表被切分成一个用于存储索引的“索引引擎”和若干存储行数据的“数据引擎”。
# 这两项设置控制两种引擎文件的最大并发数。
# 这两项设置的值会影响 tikv-importer 的内存和磁盘用量。
# 两项数值之和不能超过 tikv-importer 的 max-open-engines 的设定。
index-concurrency = 2
table-concurrency = 6

# 数据的并发数。默认与逻辑 CPU 的数量相同。
# 混合部署的情况下可以将其大小配置为逻辑 CPU 数的 75%，以限制 CPU 的使用。
# region-concurrency =

# I/O 最大并发数。I/O 并发量太高时，会因硬盘内部缓存频繁被刷新
# 而增加 I/O 等待时间，导致缓存未命中和读取速度降低。
# 对于不同的存储介质，此参数可能需要调整以达到最佳效率。
io-concurrency = 5

[checkpoint]
# 是否启用断点续传。
# 导入数据时，TiDB Lightning 会记录当前表导入的进度。
# 所以即使 Lightning 或其他组件异常退出，在重启时也可以避免重复再导入已完成的数据。
enable = true
# 存储断点的数据库名称。
schema = "tidb_lightning_checkpoint"
# 存储断点的方式。
# - file：存放在本地文件系统。
# - mysql：存放在兼容 MySQL 的数据库服务器。
driver = "file"

# dsn 是数据源名称 (data source name)，表示断点的存放位置。
# 若 driver = "file"，则 dsn 为断点信息存放的文件路径。
# 若不设置该路径，则默认存储路径为"/tmp/CHECKPOINT_SCHEMA.pb"。
# 若 driver = "mysql"，则 dsn 为“用户:密码@tcp(地址:端口)/*”格式的 URL。
# 若不设置该 URL，则默认会使用 [tidb] 部分指定的 TiDB 服务器来存储断点。
# 为减少目标 TiDB 集群的压力，建议指定另一台兼容 MySQL 的数据库服务器来存储断点。
# dsn = "/tmp/tidb_lightning_checkpoint.pb"

# 所有数据导入成功后是否保留断点。设置为 false 时为删除断点。
# 保留断点有利于进行调试，但会泄漏关于数据源的元数据。
# keep-after-success = false

[tikv-importer]
# 选择后端：“importer” 或 “tidb”
# backend = "importer"
```

```

# 当后端是 "importer" 时, tikv-importer 的监听地址 (需改为实际地址)。
addr = "172.16.31.10:8287"
# 当后端是 "tidb" 时, 插入重复数据时执行的操作。
# - replace : 新数据替代已有数据
# - ignore : 保留已有数据, 忽略新数据
# - error : 中止导入并报错
# on-duplicate = "replace"

[mydumper]
# 设置文件读取的区块大小, 确保该值比数据源的最长字符串长。
read-block-size = 65536 # Byte (默认为 64 KB)

# (源数据文件) 单个导入区块大小的最小值。
# Lightning 根据该值将一张大表分割为多个数据引擎文件。
batch-size = 107_374_182_400 # Byte (默认为 100 GB)

# 引擎文件需按顺序导入。由于并行处理, 多个数据引擎几乎在同时被导入,
# 这样形成的处理队列会造成资源浪费。因此, 为了合理分配资源, Lightning
# 稍微增大了前几个区块的大小。该参数也决定了比例系数, 即在完全并发下
# “导入”和“写入”过程的持续时间比。这个值可以通过计算 1 GB 大小的
# 单张表的 (导入时长/写入时长) 得到。在日志文件中可以看到精确的时间。
# 如果“导入”更快, 区块大小的差异就会更小; 比值为 0 时则说明区块大小一致。
# 取值范围为 (0 <= batch-import-ratio < 1)。
batch-import-ratio = 0.75

# mydumper 本地源数据目录。
data-source-dir = "/data/my_database"
# 如果 no-schema = false, 那么 TiDB Lightning 假设目标 TiDB 集群上
# 已有表结构, 并且不会执行 `CREATE TABLE` 语句。
no-schema = false
# 指定包含 `CREATE TABLE` 语句的表结构文件的字符集。只支持下列选项 :
# - utf8mb4 : 表结构文件必须使用 UTF-8 编码, 否则 Lightning 会报错。
# - gb18030 : 表结构文件必须使用 GB-18030 编码, 否则 Lightning 会报错。
# - auto : 自动判断文件编码是 UTF-8 还是 GB-18030, 两者皆非则会报错 (默认)。
# - binary : 不尝试转换编码。
# 注意: **数据** 文件始终解析为 binary 文件。
character-set = "auto"

# 配置 CSV 文件的解析方式。
[mydumper.csv]
# 字段分隔符, 应为单个 ASCII 字符。
separator = ','
# 引用定界符, 可为单个 ASCII 字符或空字符串。
delimiter = ''
# CSV 文件是否包含表头。
# 如果 header = true, 将跳过首行。
# CSV 文件是否包含 NULL。
# 如果 not-null = true, CSV 所有列都不能解析为 NULL。
not-null = false
# 如果 not-null = false (即 CSV 可以包含 NULL),
# 为以下值的字段将会被解析为 NULL。
null = '\N'
# 是否对字段内“\”进行转义
backslash-escape = true
# 如果有行以分隔符结尾, 删除尾部分隔符。
trim-last-separator = false

[tidb]
# 目标集群的信息。tidb-server 的地址, 填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
password = ""
# 表结构信息从 TiDB 的“status-port”获取。
status-port = 10080
# pd-server 的地址, 填一个即可。
pd-addr = "172.16.31.4:2379"
# tidb-lightning 引用了 TiDB 库, 并生成产生一些日志。
# 设置 TiDB 库的日志等级。
log-level = "error"

# 设置 TiDB 会话变量, 提升 Checksum 和 Analyze 的速度。

```

```

# 各参数定义可参阅“控制 Analyze 并发度”文档
build-stats-concurrency = 20
distsql-scan-concurrency = 100
index-serial-scan-concurrency = 20
checksum-table-concurrency = 16

# 解析和执行 SQL 语句的默认 SQL 模式。
sql-mode = "STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"
# `max-allowed-packet` 设置数据库连接允许的最大数据包大小,
# 对应于系统参数中的 `max_allowed_packet`。如果设置为 0,
# 会使用下游数据库 global 级别的 `max_allowed_packet`。
max-allowed-packet = 67_108_864

# 数据导入完成后, tidb-lightning 可以自动执行 Checksum、Compact 和 Analyze 操作。
# 在生产环境中, 建议将这些参数都设为 true。
# 执行的顺序为: Checksum -> Compact -> Analyze。
[post-restore]
# 如果设置为 true, 会对所有表逐个执行 `ADMIN CHECKSUM TABLE <table>` 操作
# 来验证数据的完整性。
checksum = true
# 如果设置为 true, 会在导入每张表后执行一次 level-1 Compact。
# 默认值为 false。
level-1-compact = false
# 如果设置为 true, 会在导入过程结束时对整个 TiKV 集群执行一次 full Compact。
# 默认值为 false。
compact = false
# 如果设置为 true, 会对所有表逐个执行 `ANALYZE TABLE <table>` 操作。
analyze = true

# 设置周期性后台操作。
# 支持的单位:h(时)、m(分)、s(秒)。
[cron]
# Lightning 自动刷新导入模式状态的持续时间, 该值应小于 TiKV 对应的设定值。
switch-mode = "5m"
# 在日志中打印导入进度的持续时间。
log-progress = "5m"

# 设置表库过滤。详情参见“TiDB Lightning 表库过滤”文档。
# [black-white-list]
# ...

```

## TiKV Importer 配置参数

```

# TiKV Importer 配置文件模版

# 日志文件
log-file = "tikv-importer.log"
# 日志等级：trace, debug, info, warn, error 和 off
log-level = "info"

[server]
# tikv-importer 的监听地址, tidb-lightning 需要连到这个地址进行数据写入。
addr = "192.168.20.10:8287"
# gRPC 服务器的线程池大小。
grpc-concurrency = 16

[metric]
# 给 Prometheus 客户端推送的 job 名称。
job = "tikv-importer"
# 给 Prometheus 客户端推送的间隔。
interval = "15s"
# Prometheus Pushgateway 的地址。
address = ""

[rocksdb]
# background job 的最大并发数。
max-background-jobs = 32

[rocksdb.defaultcf]
# 数据在刷新到硬盘前能存于内存的容量上限。
write-buffer-size = "1GB"
# 内存中写缓冲器的最大数量。
max-write-buffer-number = 8

# 各个压缩层级使用的算法。
# 第 0 层的算法用于压缩 KV 数据。
# 第 6 层的算法用于压缩 SST 文件。
# 第 1 至 5 层的算法目前尚未使用。
compression-per-level = ["lz4", "no", "no", "no", "no", "no", "no", "lz4"]

[rocksdb.writecf]
# 同上
compression-per-level = ["lz4", "no", "no", "no", "no", "no", "no", "lz4"]

[import]
# 存储引擎文件的文件夹路径
import-dir = "/mnt/ssd/data.import/"
# 处理 RPC 请求的线程数
num-threads = 16
# 导入 job 的并发数。
num-import-jobs = 24
# 预处理 Region 最长时间。
# max-prepare-duration = "5m"
# 把要导入的数据切分为这个大小的 Region。
#region-split-size = "512MB"
# 设置 stream-channel-window 的大小。
# channel 满了之后 stream 会处于阻塞状态。
# stream-channel-window = 128
# 同时打开引擎文档的最大数量。
max-open-engines = 8
# Importer 上传至 TiKV 的最大速度（字节/秒）。
# upload-speed-limit = "512MB"
# 目标存储可用空间比率 (store_available_space/store_capacity) 的最小值。
# 如果目标存储空间的可用比率低于该值, Importer 将会暂停上传 SST
# 来为 PD 提供足够时间进行 Regions 负载均衡。
min-available-ratio = 0.05

```

## 命令行参数

### tidb-lightning

使用 tidb-lightning 可以对下列参数进行配置：

| 参数                                   | 描述   | 对应配置项                                 |
|--------------------------------------|--|---------------------------------------|
| <code>-config file</code>            | 从 <code>file</code> 读取全局设置。如果没有指定则使用默认设置。      |                                       |
| <code>-V</code>                      | 输出程序的版本  |                                       |
| <code>-d directory</code>            | 读取数据的目录  | <code>mydumper.data-source-dir</code> |
| <code>-L level</code>                | 日志的等级：debug、info、warn、error 或 fatal (默认为 info) | <code>lightning.log-level</code>      |
| <code>-backend backend</code>        | 选择后端的模式：importer 或 tidb                        | <code>tikv-importer.backend</code>    |
| <code>-log-file file</code>          | 日志文件路径   | <code>lightning.log-file</code>       |
| <code>-status-addr ip:port</code>    | TiDB Lightning 服务器的监听地址                        | <code>lightning.status-port</code>    |
| <code>-importer host:port</code>     | TiKV Importer 的地址                              | <code>tikv-importer.addr</code>       |
| <code>-pd-urls host:port</code>      | PD endpoint 的地址                                | <code>tidb.pd-addr</code>             |
| <code>-tidb-host host</code>         | TiDB Server 的 host                             | <code>tidb.host</code>                |
| <code>-tidb-port port</code>         | TiDB Server 的端口 (默认为 4000)                     | <code>tidb.port</code>                |
| <code>-tidb-status port</code>       | TiDB Server 的状态端口的 (默认为 10080)                 | <code>tidb.status-port</code>         |
| <code>-tidb-user user</code>         | 连接到 TiDB 的用户名                                  | <code>tidb.user</code>                |
| <code>-tidb-password password</code> | 连接到 TiDB 的密码                                   | <code>tidb.password</code>            |

如果同时对命令行参数和配置文件中的对应参数进行更改，命令行参数将优先生效。例如，在 cfg.toml 文件中，不管对日志等级做出什么修改，运行 `./tidb-lightning -L debug --config cfg.toml` 命令总是将日志级别设置为“debug”。

## tidb-lightning-ctl

使用 tidb-lightning-ctl 可以对下列参数进行配置：

| 参数   | 描述                                       |
|--|--|
| <code>-compact</code>                            | 执行 full compact                          |
| <code>-switch-mode mode</code>                   | 将每个 TiKV Store 切换到指定模式 (normal 或 import) |
| <code>-import-engine uuid</code>                 | 将 TiKV Importer 上关闭的引擎文件导入到 TiKV 集群      |
| <code>-cleanup-engine uuid</code>                | 删除 TiKV Importer 上的引擎文件                  |
| <code>-checkpoint-dump folder</code>             | 将当前的断点以 CSV 格式存储到文件夹中                    |
| <code>-checkpoint-error-destroy tablename</code> | 删除断点，如果报错则删除该表                           |
| <code>-checkpoint-error-ignore tablename</code>  | 忽略指定表中断点的报错                              |
| <code>-checkpoint-remove tablename</code>        | 无条件删除表的断点                                |

`tablename` 必须是 `db . tbl` 中的限定表名（包括反引号），或关键词 all。

此外，上表中所有 tidb-lightning 的参数也适用于 tidb-lightning-ctl。

## tikv-importer

使用 tikv-importer 可以对下列参数进行配置：

| 参数                 | 描述                                      | 对应配置项             |
|--------------------|---|-------------------|
| -C, --config file  | 从 file 读取配置。如果没有指定，则使用默认设置              |                   |
| -V, --version      | 输出程序的版本                                 |                   |
| -A, --addr ip:port | TiKV Importer 服务器的监听地址                  | server.addr       |
| -import-dir dir    | 引擎文件的存储目录                               | import.import-dir |
| -log-level level   | 日志的等级：trace、debug、info、warn、error 或 off | log-level         |
| -log-file file     | 日志文件路径                                  | log-file          |

## 2.2.2.9 TiDB Lightning 监控告警

tidb-lightning 和 tikv-importer 都支持使用 [Prometheus](#) 采集监控指标 (metrics)。本文主要介绍 TiDB Lightning 的监控配置与监控指标。

### 监控配置

- 如果是使用 TiDB Ansible 部署 Lightning，只要将服务器地址加到 inventory.ini 文件里的 [monitored\_servers] 部分即可。
- 如果是手动部署 Lightning，则参照以下步骤进行配置。

#### tikv-importer

tikv-importer v2.1 使用 [Pushgateway](#) 来推送监控指标。需要配置 tikv-importer.toml 来连接 Pushgateway：

```
[metric]
# 给 Prometheus 客户端的推送任务名称。
job = "tikv-importer"

# 给 Prometheus 客户端的推送间隔。
interval = "15s"

# Prometheus Pushgateway 地址。
address = ""
```

只要 Prometheus 能发现 tidb-lightning 的监控地址，就能收集监控指标。监控的端口可在 tidb-lightning.toml 中配置：

```
[lightning]
# 用于调试和 Prometheus 监控的 HTTP 端口。输入 0 关闭。
pprof-port = 8289
...
```

要让 Prometheus 发现 Lightning，可以将地址直接写入其配置文件，例如：

```
...
scrape_configs:
- job_name: 'tidb-lightning'
  static_configs:
    - targets: ['192.168.20.10:8289']
```

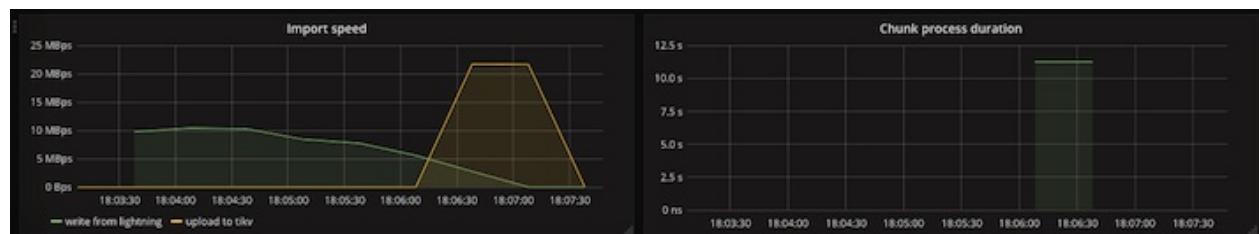
### Grafana 面板

Grafana 的可视化面板可以让你在网页上监控 Prometheus 指标。

使用 TiDB Ansible 部署 TiDB 集群时，会同时部署一套 Grafana + Prometheus 的监控系统。

如果使用其他方式部署 TiDB Lightning，需先导入[面板的 JSON 文件](#)。

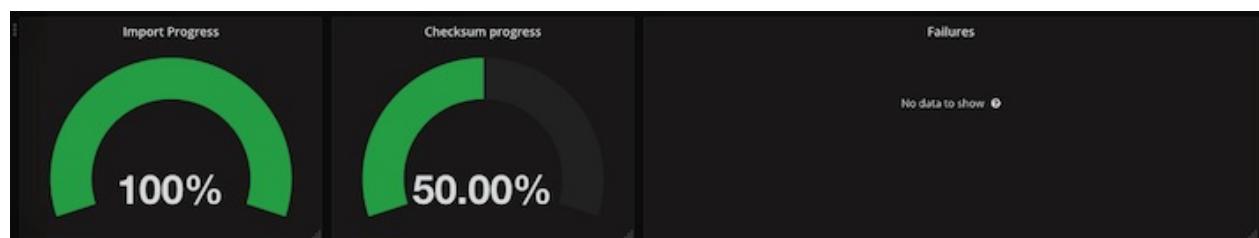
## 第一行：速度面板



| 面板名称                   | 序列                   | 描述   |
|------------------------|----------------------|--|
| Import speed           | write from lightning | 从 TiDB Lightning 向 TiKV Importer 发送键值对的速度，取决于每个表的复杂性 |
| Import speed           | upload to tikv       | 从 TiKV Importer 上传 SST 文件到所有 TiKV 副本的总体速度            |
| Chunk process duration |                      | 完全编码单个数据文件所需的平均时间                                    |

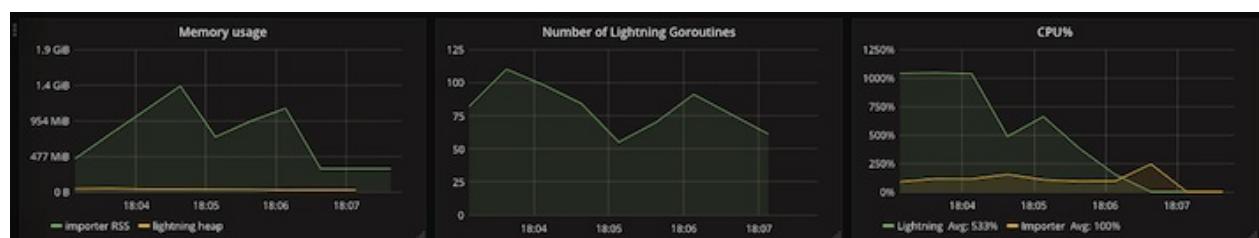
有时导入速度会降到 0，这是为了平衡其他部分的速度，属于正常现象。

## 第二行：进度面板



| 面板名称              | 描述                  |
|-------------------|---------------------|
| Import progress   | 已编码的文件所占百分比         |
| Checksum progress | 已导入的表所占百分比          |
| Failures          | 导入失败的表的数量以及故障点，通常为空 |

## 第三行：资源使用面板



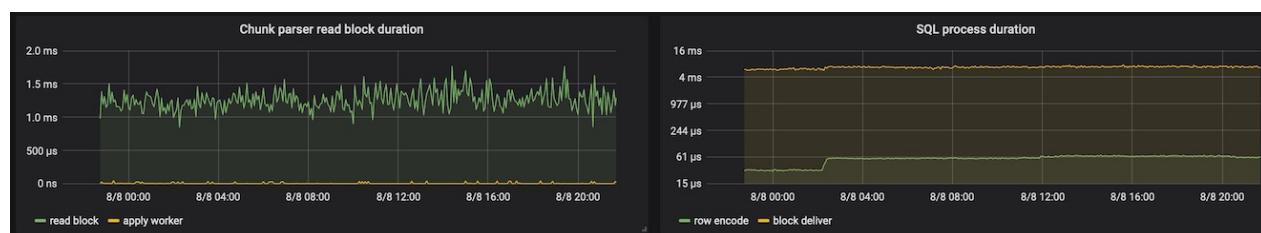
| 面板名称                           | 描述                                   |
|--------------------------------|--------------------------------------|
| Memory usage                   | 每个服务占用的内存                            |
| Number of Lightning Goroutines | TiDB Lightning 使用的运行中的 goroutines 数量 |
| CPU%                           | 每个服务使用的逻辑 CPU 数量                     |

## 第四行：配额使用面板



| 面板名称               | 序列               | 描述   |
|--------------------|------------------|--|
| Idle workers       | io               | 未使用的 io-concurrency 的数量，通常接近配置值（默认为 5），接近 0 时表示磁盘运行太慢  |
| Idle workers       | closed-engine    | 已关闭但未清理的引擎数量，通常接近 index-concurrency 与 table-concurrency 的和（默认为 8），接近 0 时表示 TiDB Lightning 比 TiKV Importer 快，导致 TiDB Lightning 延迟 |
| Idle workers       | table            | 未使用的 table-concurrency 的数量，通常为 0，直到进程结束  |
| Idle workers       | index            | 未使用的 index-concurrency 的数量，通常为 0，直到进程结束  |
| Idle workers       | region           | 未使用的 region-concurrency 的数量，通常为 0，直到进程结束   |
| External resources | KV Encoder       | 已激活的 KV encoder 的数量，通常与 region-concurrency 的数量相同，直到进程结束  |
| External resources | Importer Engines | 打开的引擎文件数量，不应超过 max-open-engines 的设置  |

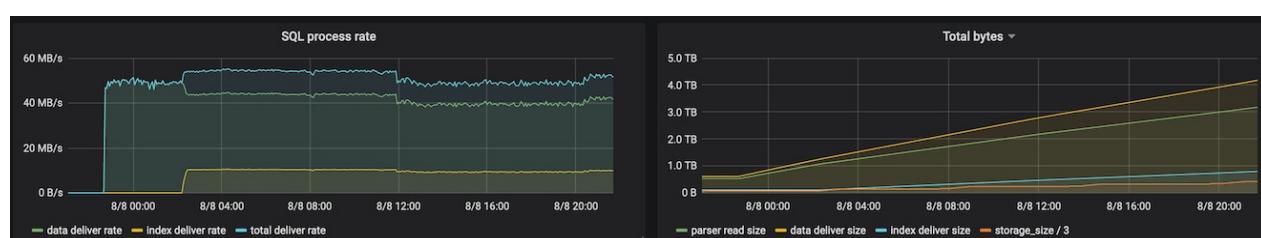
## 第五行：读取速度面板



| 面板名称                             | 序列            | 描述                             |
|----------------------------------|---------------|--------------------------------|
| Chunk parser read block duration | read block    | 读取一个字节块来准备解析时所消耗的时间            |
| Chunk parser read block duration | apply worker  | 等待 io-concurrency 空闲所消耗的时间     |
| SQL process duration             | row encode    | 解析和编码单行所消耗的时间                  |
| SQL process duration             | block deliver | 将一组键值对发送到 TiKV Importer 所消耗的时间 |

如果上述项的持续时间过长，则表示 TiDB Lightning 使用的磁盘运行太慢或 I/O 太忙。

## 第六行：存储空间面板



| 面板名称             | 序列                 | 描述                                 |
|------------------|--------------------|------------------------------------|
| SQL process rate | data deliver rate  | 向 TiKV Importer 发送数据键值对的速度         |
| SQL process rate | index deliver rate | 向 TiKV Importer 发送索引键值对的速度         |
| SQL process rate | total deliver rate | 发送数据键值对及索引键值对的速度之和                 |
| Total bytes      | parser read size   | TiDB Lightning 正在读取的字节数            |
| Total bytes      | data deliver size  | 已发送到 TiKV Importer 的数据键值对的字节数      |
| Total bytes      | index deliver size | 已发送到 TiKV Importer 的索引键值对的字节数      |
| Total bytes      | storage_size/3     | TiKV 集群占用的存储空间大小的 1/3 (3 为默认的副本数量) |

## 第七行：导入速度面板



| 面板名称                 | 序列             | 描述                              |
|----------------------|----------------|---------------------------------|
| Delivery duration    | Range delivery | 将一个 range 的键值对上传到 TiKV 集群所消耗的时间 |
| Delivery duration    | SST delivery   | 将单个 SST 文件上传到 TiKV 集群所消耗的时间     |
| SST process duration | Split SST      | 将键值对流切分成若干 SST 文件所消耗的时间         |
| SST process duration | SST upload     | 上传单个 SST 文件所消耗的时间               |
| SST process duration | SST ingest     | ingest 单个 SST 文件所消耗的时间          |
| SST process duration | SST size       | 单个 SST 文件的大小                    |

## 监控指标

本节将详细描述 tikv-importer 和 tidb-lightning 的监控指标。

### tikv-importer

tikv-importer 的监控指标皆以 tikvimport\* 为前缀。

tikv\_import\_rpc\_duration (直方图)

- 完成一次 RPC 用时直方图。标签：
  - request : 所执行 RPC 请求的类型
    - switch\_mode — 将一个 TiKV 节点切换为 import/normal 模式
    - open\_engine — 打开引擎文件
    - write\_engine — 接收数据并写入引擎文件
    - close\_engine — 关闭一个引擎文件
    - import\_engine — 导入一个引擎文件到 TiKV 集群中
    - cleanup\_engine — 删除一个引擎文件
    - compact\_cluster — 显式压缩 TiKV 集群
    - upload — 上传一个 SST 文件
    - ingest — Ingest 一个 SST 文件
    - compact — 显式压缩一个 TiKV 节点

- result : RPC 请求的执行结果
  - ok
  - error

tikv\_import\_write\_chunk\_bytes (直方图)

- 从 Lightning 接收的键值对区块大小 (未压缩) 的直方图。

tikv\_import\_write\_chunk\_duration (直方图)

- 从 tidb-lightning 接收每个键值对区块所需时间的直方图。

tikv\_import\_upload\_chunk\_bytes (直方图)

- 上传到 TiKV 的每个 SST 文件区块大小 (压缩) 的直方图。

tikv\_import\_range\_delivery\_duration (直方图)

- 将一个 range 的键值对发送至 dispatch-job 任务所需时间的直方图。

tikv\_import\_split\_sst\_duration (直方图)

- 将 range 从引擎文件中分离到单个 SST 文件中所需时间的直方图。

tikv\_import\_sst\_delivery\_duration (直方图)

- 将 SST 文件从 dispatch-job 任务发送到 ImportSSTJob 任务所需时间的直方图

tikv\_import\_sst\_recv\_duration (直方图)

- ImportSSTJob 任务接收从 dispatch-job 任务发送过来的 SST 文件所需时间的直方图。

tikv\_import\_sst\_upload\_duration (直方图)

- 从 ImportSSTJob 任务上传 SST 文件到 TiKV 节点所需时间的直方图。

tikv\_import\_sst\_chunk\_bytes (直方图)

- 上传到 TiKV 节点的 SST 文件 (压缩) 大小的直方图。

tikv\_import\_sst\_ingest\_duration (直方图)

- 将 SST 文件传入至 TiKV 所需时间的直方图。

tikv\_import\_each\_phase (测量仪)

- 表示运行阶段。值为 1 时表示在阶段内运行，值为 0 时表示在阶段外运行。标签：
  - phase : prepare / import

tikv\_import\_wait\_store\_available\_count (计数器)

- 计算出现 TiKV 节点没有充足空间上传 SST 文件现象的次数。标签：
  - store\_id : TiKV 存储 ID。

tikv\_import\_upload\_chunk\_duration (直方图)

- 上传到 TiKV 的每个区块所需时间的直方图。

## tidb-lightning

tidb-lightning 的监控指标皆以 lightning\_\* 为前缀。

lightning\_importer\_engine (计数器)

- 计算已开启及关闭的引擎文件数量。标签：
  - type:
    - open

- closed

#### lightning\_idle\_workers (计量表盘)

- 计算闲置的 worker。标签：

- name :
  - table — 未使用的 table-concurrency 的数量，通常为 0，直到进程结束
  - index — 未使用的 index-concurrency 的数量，通常为 0，直到进程结束
  - region — 未使用的 region-concurrency 的数量，通常为 0，直到进程结束
  - io — 未使用的 io-concurrency 的数量，通常接近配置值（默认为 5），接近 0 时表示磁盘运行太慢
  - closed-engine — 已关闭但未清理的引擎数量，通常接近 index-concurrency 与 table-concurrency 的和（默认为 8），接近 0 时表示 TiDB Lightning 比 TiKV Importer 快，导致 TiDB Lightning 延迟

#### lightning\_kv\_encoder (计数器)

- 计算已开启及关闭的 KV 编码器。KV 编码器是运行于内存的 TiDB 实例，用于将 SQL 的 INSERT 语句转换成键值对。此度量的净值（开启减掉关闭）在正常情况下不应持续增长。标签：

- type:
  - open
  - closed

#### lightning\_tables (计数器)

- 计算处理过的表及其状态。标签：

- state : 表的状态，表明当前应执行的操作
  - pending — 等待处理
  - written — 所有数据已编码和传输
  - closed — 所有对应的引擎文件已关闭
  - imported — 所有引擎文件已上传到目标集群
  - altered\_auto\_inc — 自增 ID 已改
  - checksum — 已计算校验和
  - analyzed — 已进行统计信息分析
  - completed — 表格已完全导入并通过验证
- result : 当前操作的执行结果
  - success — 成功
  - failure — 失败（未完成）

#### lightning\_engines (计数器)

- 计算处理后引擎文件的数量以及其状态。标签：

- state : 引擎文件的状态，表明当前应执行的操作
  - pending — 等待处理
  - written — 所有数据已编码和传输
  - closed — 引擎文件已关闭
  - imported — 当前引擎文件已上传到目标集群
  - completed — 当前引擎文件已完全导入
- result : 当前操作的执行结果
  - success — 成功
  - failure — 失败（未完成）

#### lightning\_chunks (计数器)

- 计算处理过的 Chunks 及其状态。标签：

- state: 单个 Chunk 的状态，表明该 Chunk 当前所处的阶段
  - estimated — (非状态) 当前任务中 Chunk 的数量
  - pending — 已载入但未执行
  - running — 正在编码和发送数据
  - finished — 该 Chunk 已处理完毕

- failed — 处理过程中发生错误

lightning\_import\_seconds (直方图)

- 导入每个表所需时间的直方图。

lightning\_row\_read\_bytes (直方图)

- 单行 SQL 数据大小的直方图。

lightning\_row\_encode\_seconds (直方图)

- 解码单行 SQL 数据到键值对所需时间的直方图。

lightning\_row\_kv\_deliver\_seconds (直方图)

- 发送一组与单行 SQL 数据对应的键值对所需时间的直方图。

lightning\_block\_deliver\_seconds (直方图)

- 每个键值对中的区块传送到 tikv-importer 所需时间的直方图。

lightning\_block\_deliver\_bytes (直方图)

- 发送到 Importer 的键值对中区块（未压缩）的大小的直方图。

lightning\_chunk\_parser\_read\_block\_seconds (直方图)

- 数据文件解析每个 SQL 区块所需时间的直方图。

lightning\_checksum\_seconds (直方图)

- 计算表中 Checksum 所需时间的直方图。

lightning\_apply\_worker\_seconds (直方图)

- 获取闲置 worker 等待时间的直方图（参见 lightning\_idle\_workers 计量表盘）。标签：

- name：
  - table
  - index
  - region
  - io
  - closed-engine



## 2.3 4.0 分布式备份恢复工具 BR

BR(<https://github.com/pingcap/br>) 是分布式备份恢复工具，分布式意味着可以通过 BR 可以驱动集群中所有 TiKV 节点进行备份恢复工作，相比起 SQL dump 形式，这种分布式备份恢复的效率会有极大的提升。本章节会详细介绍下 BR 的工作原理，会分别介绍备份和恢复的具体实现。

### 2.3.1 BR 工作原理

#### 2.3.1.1 备份流程

备份过程主要由两个组件参与（BR 和 TiKV），整体的基本流程如下：

- \* BR 根据用户指定备份范围下发备份命令到 TiKV。
- \* TiKV 接受备份请求后交由 region leader 处理，并使用一个特殊的 iterator 读取数据。
- \* TiKV 读取的数据会被组织成 SST 文件。这里需要控制 IO 和内存使用。
- \* TiKV 会把 SST 文件写到外部存储，比如 http, s3 等。
- \* TiKV 把执行信息汇报给 BR。

BR 侧备份详细流程：

- \* 下推备份请求到所有 TiKV。
- \* 接受 TiKV Streaming 发过来的备份结果。
- \* 聚合检查是否有范围没有备份到或者发生错误。
- \* 如果有范围错误或者没有备份到则重试。
- \* 重试需要精细化，查询 PD 对应的 leader 然后下发。
- \* 除了备份的数据外，还需要备份 schema。

TiKV 侧备份原理：由于 TiDB/TiKV 的事务模型是 percolator，数据的存储需要使用 3 个 CF，分别是 default，lock，write，所以如果 TiKV 想把数据保存在 SST 中，那么起码 TiKV 需要扫出 write 和 default 的数据。

在全量备份中，TiKV 需要根据一个 ts（称为 backup\_ts）扫出 write 和 default 的 key-value。

在增量备份中，TiKV 需要扫出一段时间的增量数据，一段时间具体来说：

`(backup_ts, current_ts]`

由于备份的保证是 SI，所有增量数据可以直接通过扫 write CF 中的记录即可得到。需要备份的记录有：

- Put，新增的数据。
- Delete，删除的数据。

不需要备份的记录有：

- Lock，select for update 写的记录，实际上没有任何数据变更。
- Rollback，清理由提交失败造成的垃圾数据。

通过以上信息我们可以总结出备份方案的特点：

1. 分布式导出数据，数据由各个 region 的 leader 生成，理论上备份的吞吐能达到集群极限。
2. 数据是形式是 SST，SST 格式有优点在于能快速恢复，同时可以直接用 rocksdb 自带一些功能比如，数据压缩/加密。
3. 数据直接保存第三方存储，比如 S3，HTTP 等。
4. 备份的一致性保证：SI，需要保证能恢复到 point-in-time 的状态。

#### 2.3.1.1 备份中注意事项

1. 性能 性能是 KV Scan 方案最头疼的问题之一，因为全表扫描势必会对线上业务造成影响。增量的实现也是全表扫，同样有性能问题，这个和传统数据库的增量不一样。

2. 外部存储 备份存储设计了一个接口，下层可以有不同的实现，比如本地磁盘，s3 等。由于外部存储不一定是真正的磁盘，所以 TiKV 在生成 SST 的时候，不会直接把数据写到本地磁盘上，而是先缓存在内存中，生成完毕后直接写到外部存储中，这样能避免 IO，提高整个备份的吞吐，当然这里需要内存控制。
3. 异常处理 备份期间的异常和普通查询一样，分为可恢复异常和不可恢复异常，所有的异常都可以直接复用 TiDB 现有的机制。

可恢复异常一般包含：

- RegionError，一般由 region split/merge，not leader 造成。
- KeyLocked，一般由事务冲突造成。
- Server is busy，一般由于 TiKV 太忙造成。

当发生这些异常时，备份的进度不会被打断。

除了以上的其他错误都是不可恢复异常，发生后，它们会打断备份的进度。

1. 超出 GC 时间 超出 GC 时间是说，需要备份的数据已经被 GC，这情况一般发生在增量备份上，会导致增量不完整。在发生这个错的时候，BR 需要重新来一个全量备份。所以我们推荐在 BR 启动前手动调整 GC 时间。由现在默认的 GC 时间是 10 分钟，根据适当场景延长时间。

### 2.3.1.2 恢复

恢复所需的工作有以下几个：

- 创建需要恢复的 database 和 table
- 根据 table 和 SST 文件元信息，进行 Split & Scatter Region
- 将备份下来的 SST 文件按需读取到 TiKV 各节点
- 根据新 table 的 ID 对 SST 进行 Key Rewrite
- 将处理好的 SST 文件 Ingest 到 TiKV

恢复原理相对备份原理理解起来复杂一些，要想理解恢复原理需要先理解三个核心处理，Key Rewrite，Split & Scatter Region，Ingest SST，下面将逐一介绍

1. Key Rewrite 由于 TiDB 的数据在 TiKV 那保存的形式是

```
Key: tablePrefix{tableID}_recordPrefixSep{rowID} Value: [col1, col2, col3, col4]
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue Value: rowID
```

在 Key 中编码了 tableID，所以我们不能直接把备份下来的 SST 文件不经任何处理恢复到 TiKV 中，否则就有可能因为 tableID 对不上而导致数据错乱。

为了解决该问题，我们必须在恢复前对 SST 文件进行 key 的改写，将原有的 tableID 替换为新创建的 tableID，同样的 indexID 也需要相同的处理。

1. Split & Scatter TiKV 对 Region 的大小是有限制的，默认为 96MB，超出该阈值则需要分裂（Split）。集群刚启动时，只有少量的 Region 的，我们在恢复的时候不能把所有数据都恢复到这些少量的 Region 中，所以需要提前将 Region 分裂并打散（Scatter）。

由于备份 SST 文件是按照 Region 生成的，天然就有合适的大小和对应的数据范围，所以我们可以根据各个 SST 文件中的范围对集群中的 Region 进行分裂。分裂完成后还需要打散新分裂出来的 Region，防止发生数据倾斜。

1. Ingest SST Ingest SST 复用了现有的 sst\_importer 模块，可以将处理好的 SST 文件通过 Raft 命令安全地在所有副本上 Ingest，从而保证副本间数据一致性。

我们对比备份原理发现，恢复原理有几个特殊之处：

\* 备份只在 region leader 上执行，恢复时会把 leader 的数据恢复到包含这个 region 副本的所有 TiKV 节点上。所以恢复的数据量 = 备份数据量 \* 副本数。  
 \* 恢复时在 Key Rewrite 情况下，会多出一些 IO 操作，使得恢复总时间增加。

所以恢复耗时要高于备份耗时。

了解以上原理后，可以阅读下一章节，亲手实践 BR。

## 2.3.2 BR 实操指南

### 1. 集群部署

为了测试 BR 备份恢复的完整流程，需要提前部署好一套完整的集群，部署方案主要参考[使用Ansible部署这篇文章](#)，简约流程如下：

1. 下载对应 ansible 版本，本次采用[v4.0.0-beta.1](#)。
2. 执行 pip install -r ./requirements.txt 安装依赖。
3. 设置部署结构：
  - i. 本次部署一共使用6台机器，IP 从 101-106。
  - ii. 101-106 为 TiKV 节点，101-103 为 PD 节点，104-106 为 TiDB 节点。

**注意:** 仅做备份恢复相关的测试，这里可以考虑不去做操作系统、文件系统等相关的参数调整

关于备份恢复中需要用到的br工具，已经附带在官方 tidb-toolkit-v4.0.0-beta.1.tar.gz 包中，如果是使用 ansible 安装，可以直接在 tidb-ansible 目录下的 downloads 中找到。

### 2. 构建测试数据

在 TiDB 中创建对应库表

```
MySQL [(none)]> create database br_test;
Query OK, 0 rows affected (0.11 sec)

MySQL [(none)]> use br_test;
Database changed

MySQL [br_test]> create table br_table(id int primary key,c varchar(128),ctime timestamp);
Query OK, 0 rows affected (0.12 sec)
```

使用任意方式构造数据，比如使用 python 脚本：

```
import mysql.connector
import time
mydb = mysql.connector.connect(
    host="xxxx.104", # 这里需要替换成 tidb-server 的 ip
    user='root',
    port=4000,
    database='br_test'
)
mycursor = mydb.cursor()

for i in range (100000):
    mycursor.execute('insert into br_table values(%s,%s,now())',(i,str(i)+'xxxx'))
    if i%1000==0:
        mycursor.execute('commit')

mycursor.execute('commit')
mycursor.close()
mydb.close()
```

接下来在 TiDB 中查看数据已经生成

```
MySQL [br_test]> select count(1) from br_table;
+-----+
| count(1) |
+-----+
| 100000 |
+-----+
1 row in set (0.04 sec)
```

最后总共生成的数据文件分布在 6 个 TiKV 节点上。

### 3. 备份准备

在进行备份前，有一些需要调整的配置项

#### 1. tikv\_gc\_life\_time 参数

```
# 设置 gc 时间，避免备份时间过长导致数据被回收，需要注意的是，备份完成后，需要改回来参数。
#默认值
SELECT VARIABLE_VALUE FROM mysql.tidb WHERE VARIABLE_NAME = 'tikv_gc_life_time';

10m0s

# 设置为720h
UPDATE mysql.tidb SET VARIABLE_VALUE = '720h' WHERE VARIABLE_NAME = 'tikv_gc_life_time';

# 验证修改确实成功
SELECT * FROM mysql.tidb WHERE VARIABLE_NAME = 'tikv_gc_life_time';

720h
```

#### 1. 备份存储位置

当前备份是备份到文件系统，也就是可以通过SMB/NFS之类的挂载，备份到远程备份中心。

**注：**后续会增加S3，GCS云存储

需要注意的是，下文中执行挂载操作的，是所有的 TiKV, BR 节点，而非 TiDB, PD 所在节点。

挂载 NFS：

```
mount -t nfs //nfs_address/:/data /data_nfs1
```

### 4. 备份执行

BR 命令包括备份，恢复两个操作，而备份，恢复又单独针对全库，单库，单表各有操作，因此单独讨论。

而在这些操作之前，其他共用参数单独讨论。

另外需要注意的一点是，因为备份通过 gRPC 发送相关到 TiKV，因此 BR 执行的位置，最好是 PD 节点，避免额外的麻烦。

### 5. 通用参数

--ca, --cert, --key 如果设置了TLS类连接安全认证，这些参数指定相关安全证书等。

--concurrency 每个节点执行任务的并行度，默认4。

--log-file,--log-level设置日志输出位置以及级别。

-u, --pd 链接 PD 地址，默认127.0.0.1:2379

--ratelimit 限制每个节点的速度，单位是MB

-s, --storage 指定存储位置，比方"local:///data\_nfs1"

## 6. 全库备份与恢复

参考命令：

```
bin/br backup full --pd "192.168.122.101:2379" --storage "local:///data_nfs1/backup"
```

这个命令会备份全库到各个 TiKV 节点下的 /data\_nfs1/backup 目录。

简单从日志看一下整个备份流程：

```
* 从 PD 连接获取到所有 TiKV 节点。
* 查询 infoSchema 获取元数据。
* 发送备份请求：{"cluster_id":6801677637806839235,"start_key":"dIAAAAAAAAAAvX3IAAAAAAAA==","end_key":"dIAAAAAAAAAAvX3L
//////////wA=","end_version":415142848617512967,"concurrency":4,"storage_backend":{"Backend":{"Local":{"path":"/data_n
fs1/backup"}}}}
* 各个 TiKV 节点开始执行备份，执行命令完成后，返回到 BR 进行统计。
* 执行表的checksum [table=`br_test`.`br_table`] [Crc64Xor=12896770389982935753] [TotalKvs=100000] [TotalBytes=4788890]

* 保存备份的元数据。
* 完成备份。
```

备份过程中，提到的元数据，最终会保存到备份目录下，其主要包含的是校验和，以及备份集的相关描述信息，包括备份集合中，每个库，表，列的逻辑排列，字符集信息等（对应的是一个 protobuf 格式的描述文件）。

备份完成后，在指定的备份目录，会最终出现命名类似

5\_2\_23\_80992061af3e5194c3f28a5b79d486c5e9db2feda1afb3f84b4ca229ddce9932\_write.sst的备份集合，也就是最终的备份文件。

## 7. 恢复数据

为了简化操作，这里在原有集群上进行恢复，往往实际中是要恢复到一个全新的集群上。

首先执行以下语句删除数据：

```
MySQL [br_test]> drop table br_table;
```

注意：恢复时候，每个 TiKV 节点都需要访问到所有备份文件，如果不是共享存储，需要手动复制所有备份文件到所有节点

```
bin/br restore full --pd "192.168.122.101:2379" --storage "local:///data_nfs1/backup"
```

从日志看：

1. 寻找并确认 PD 节点。
2. 执行DDL语句：CREATE DATABASE !/32312 IF NOT EXISTS/ 以及 CREATE TABLE IF NOT EXISTS。
3. 执行必要的alter auto incrementID 语句，防止恢复后从之前的 id 分配，导致数据覆盖。
4. 切割sst为Region负责的小数据集合，分别进行数据写入。
5. 完成操作后，输出统计信息 ["Full restore summary: total restore tables: 1, total success: 1, total failed: 0, total take(s): 0.25, total size(MB): 2.28, avg speed(MB/s): 9.08, total kv: 50001"] ["restore files"=1] ["restore ranges"=1] ["split region"=6.373065871s] ["restore checksum"=45.843202ms]

执行完成后，可以看到数据已经恢复完成：

```
MySQL [br_test]> select count(1) from br_table;
+-----+
| count(1) |
+-----+
|    100000|
+-----+
1 row in set (0.12 sec)
```

## 8. 单库备份与恢复

单库的备份恢复参考命令如下：

备份：

```
bin/br backup db --db "br_test" --pd "192.168.122.101:2379" --storage "local:///data_nfs1/backup"
```

恢复：

```
bin/br restore db --db "br_test" --pd "192.168.122.101:2379" --storage "local:///data_nfs1/backup"
```

## 9. 单表备份与恢复

单库的备份恢复参考命令如下：

备份：

```
bin/br backup table --db "br_test" --table "br_table" --pd "192.168.122.101:2379" --storage "local:///data_nfs1/backup"
```

恢复：

```
bin/br restore table --db "br_test" --table "br_table" --pd "192.168.122.101:2379" --storage "local:///data_nfs1/backup"
```

通过上述实践，我们了解了 BR 基本用法，想要了解具体代码实现可以登陆 BR 项目主页(<https://github.com/pingcap/br>)，欢迎提供更多的使用建议，帮助我们改进。



## 2.4.1 Dumpling 工作原理

### 1. 背景介绍

首先介绍一下 Dumpling 的诞生背景，在 Dumpling 诞生之前 PingCAP 提供了 Mydumper 的 fork 版本作为 TiDB 逻辑备份工具。随着 TiDB 生态的发展，Mydumper 由于种种先天不足，无法进一步满足实际应用中的需要。Mydumper 的不足主要有以下几点：

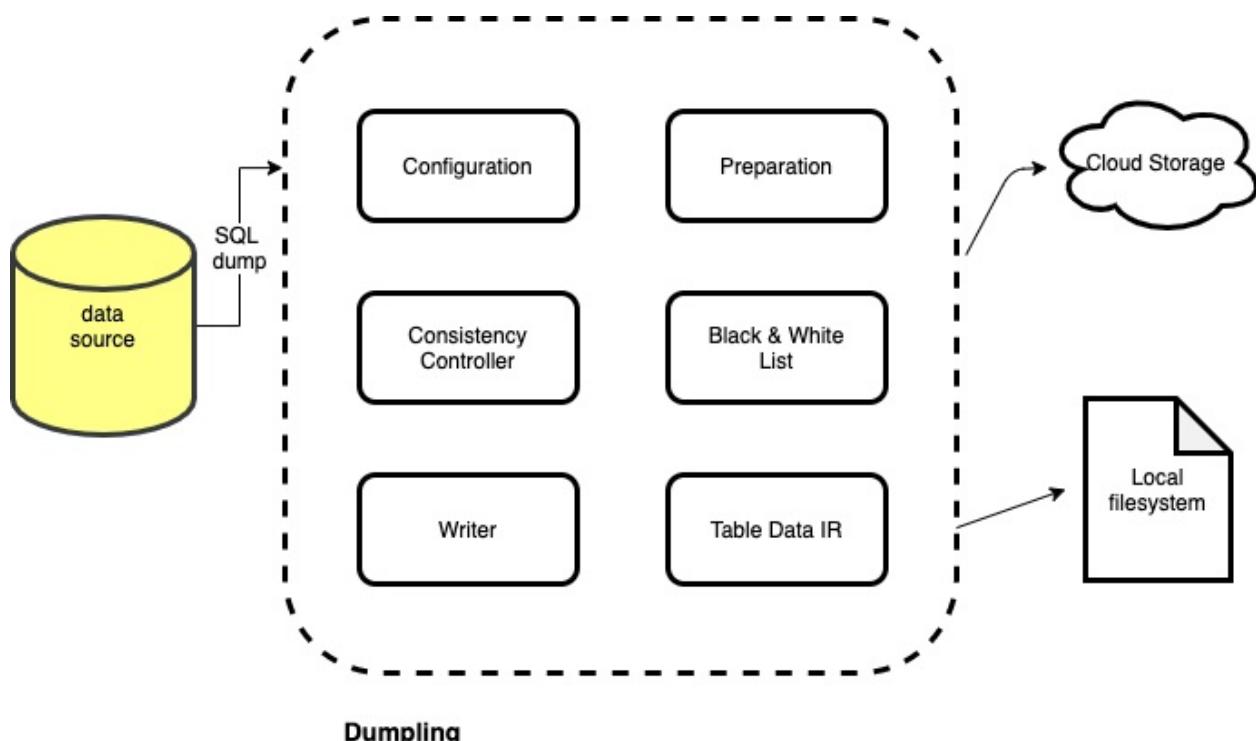
- 导出的数据格式无法满足 TiDB Lighting 数据导入工具的快速解析需要
- Mydumper 官方仓库年久失修，TiDB 并不倾向于在 fork 版本中继续提供新特性
- Mydumper 使用 C 语言及 GLib 开发，难以集成到与 Go 语言为主的 TiDB 生态工具，例如 DM 等
- Mydumper 本身采用的开源协议与 TiDB 所使用的开源协议不兼容

基于这些原因考虑，以 Dumpling 取而代之成为了更好的选择。Dumpling 将提供以下几个特性：

- 完全采用 Golang，与 TiDB 生态集成度高
- 能够提供 Mydumper 类似功能，且支持并发高速导出 MySQL 协议兼容数据库数据
- 提供 SQL、CSV 等多种数据输出格式，以便于快速导出及导入
- 支持直接导出数据到云存储系统，比如 S3

### 2. 工作原理

一图胜千言，下面是 Dumpling 导出流程示意图：



如图所示，Dumpling 分为六个比较重要的部分，分别负责配置解析、数据库信息预处理、一致性控制器、Black & White 列表、写控制器及表数据中间层表示。下面详细介绍各个部分的工作内容：

- 配置解析：处理用户通过命令行传入的参数
- 数据库信息预处理：在数据导出任务进行之前，获取数据库服务器版本、数据表、数据库视图等相关信息并进行预处理
- 一致性控制器：通过用户传入的一致性规则，在数据导出过程中保障数据一致性
- Black & White 列表：根据设置的规则过滤不需要导出的数据表
- 写控制器：负责将导出的数据写入到本地文件或云端存储系统
- 表数据中间表示层：中间表示层将数据表进行封装，提供一套 API 以供写控制器进行数据迭代写入



## 2.4.2 Dumpling 实操指南

### 1. 需要的权限

需要为用户分配的权限：

- SELECT
- RELOAD
- LOCK TABLES
- REPLICATION CLIENT

关于 TiDB 权限管理请参考 [TiDB 数据库权限管理](#)。

### 2. 使用举例

导出命令：

```
dumpling -B tidb -F 2048 -H 127.0.0.1 -u root -P 4000 --loglevel debug
```

输出样例：

```
Release version:  
Git commit hash: a35708fb6a9ca19294b92598b88d7894cf130ca6  
Git branch: master  
Build timestamp: 2020-03-08 03:15:54Z  
Go version: go version go1.13.1 darwin/amd64  
  
[2020/03/08 18:09:43.780 +08:00] [DEBUG] [config.go:72] ["parse server info"] ["server info string"=5.7.25-TiDB-v4.0.0-beta-313-g2d5d2fde27]  
[2020/03/08 18:09:43.780 +08:00] [INFO] [config.go:85] ["detect server type"] [type=TiDB]  
[2020/03/08 18:09:43.780 +08:00] [INFO] [config.go:103] ["detect server version"] [version=4.0.0-beta-313-g2d5d2fde27]  
[2020/03/08 18:09:43.781 +08:00] [DEBUG] [prepare.go:27] ["list all the tables"]  
[2020/03/08 18:09:43.788 +08:00] [DEBUG] [black_white_list.go:78] ["filter tables"]  
[2020/03/08 18:09:43.788 +08:00] [WARN] [black_white_list.go:70] ["unsupported dump schema in TiDB now"] [schema=mysql]
```

### 3. 命令参数说明

使用 Dumpling 可以对下列参数进行配置：

| 参数                        | 描述   |
|---------------------------|--|
| --consistency <level>     | 一致性级别: auto/none/flush/lock/snapshot , 默认为 auto            |
| -B, --database <database> | 需要导出数据的数据库   |
| -F, --filesize <size>     | 输出文件的最大尺寸 , 单位为 bytes                                      |
| -H, --host <hostname>     | 主机名 , 默认为 127.0.0.1  |
| --loglevel <level>        | 日志级别 : debug/info/warn/error/dpanic/panic/fatal , 默认为 info |
| -W, --no-views            | 是否导出视图 , 默认为 true  |
| -o, --output <dir>        | 输出文件目录 , 默认格式为 ./export-2020-03-08T11:37:05+08:00          |
| -p, --password <password> | 数据库连接密码  |
| -P, --port <port>         | 数据库连接端口 , 默认为 4000   |
| --snapshot <position>     | 快照起始位置 , 仅在一致性级别为 snapshot 时有效                             |
| -t, --threads <num>       | 并发线程数 , 默认为 4  |
| -u, --user <user>         | 数据库连接用户名 , 默认为 root  |

# 第1章 SQL 调优原理

如果想要进行 SQL 调优，第一步就是能够读懂一条 SQL 语句的执行计划，以了解其在 TiDB 分布式数据库中是如何被执行的。在读懂执行计划后，本章将讲解一些 SQL 优化器原理的内容，并且将和大家介绍执行计划是如何生成的。优化器是数据库系统中一个非常复杂的模块，很多数据库都避免不了索引选错、Join 顺序不优等问题，因此优化器的表现对数据库系统而言至关重要。SQL Hint 是一种用于 SQL 调优的技术，它可以指导优化器的行为，让其在复杂的情况下能够生成出符合开发者意图的执行计划，本章第三节将介绍基于 SQL Hint 的 SQL Plan Management 技术。本章的最后一节将为大家介绍一些 TiDB 系统参数，方便大家对特定场景的 SQL 进行调优。这些参数有些用于控制优化器的优化规则，有些用于控制执行引擎并发数，或 Batch 大小，以使得 SQL 执行的更快。

希望通过本章的描述，能够让大家对于如何进行 SQL 调优有所认识。

## 1.1 执行计划概览

阅读和理解执行计划对于 `SQL` 调优非常重要。本小节将向大家介绍如何查看 `SQL` 的执行计划。

### 1.1.1 使用 EXPLAIN 语句查看执行计划

执行计划由一系列的算子构成。和其他数据库一样，在 TiDB 中可通过 `EXPLAIN` 语句返回的结果查看某条 `SQL` 的执行计划。

目前 TiDB 的 `EXPLAIN` 会输出 5 列，分别是：`id`，`estRows`，`task`，`access object`，`operator info`。执行计划中每个算子都由这 5 列属性来描述，`EXPLAIN` 结果中每一行描述一个算子。每个属性的具体含义如下：

| 属性名                        | 含义   |
|----------------------------|--|
| <code>id</code>            | 算子的 ID，在整个执行计划中唯一的标识一个算子。在 TiDB 2.1 中，ID 会格式化的显示算子的树状结构。数据从孩子结点流向父亲结点，每个算子的父亲结点有且仅有一个。   |
| <code>estRows</code>       | 算子预计将会输出的数据条数，基于统计信息以及算子的执行逻辑估算而来。在 4.0 之前叫 <code>count</code> 。   |
| <code>task</code>          | 算子属于的 task 种类。目前的执行计划分成为两种 task，一种叫 <code>root task</code> ，在 <code>tidb-server</code> 上执行，一种叫 <code>cop task</code> ，并行的在 <code>TiKV</code> 或者 <code>TiFlash</code> 上执行。当前的执行计划在 task 级别的拓扑关系是一个 root task 后面可以跟许多 cop task，root task 使用 cop task 的输出结果作为输入。cop task 中执行的也即是 TiDB 下推到 <code>TiKV</code> 或者 <code>TiFlash</code> 上的任务，每个 cop task 分散在 <code>TiKV</code> 或者 <code>TiFlash</code> 集群中，由多个进程共同执行。 |
| <code>access object</code> | 算子所访问的数据项信息。包括表 <code>table</code> ，表分区 <code>partition</code> 以及使用的索引 <code>index</code> （如果有）。只有直接访问数据的算子才拥有这些信息。  |
| <code>operator info</code> | 算子的其它信息。各个算子的 operator info 各有不同，可参考下面的示例解读。   |

### 1.1.2 EXPLAIN ANALYZE 输出格式

和 `EXPLAIN` 不同，`EXPLAIN ANALYZE` 会执行对应的 `SQL` 语句，记录其运行时信息，和执行计划一并返回出来，可以视为 `EXPLAIN` 语句的扩展。`EXPLAIN ANALYZE` 语句的返回结果中增加了 `actRows`，`execution info`，`memory`，`disk` 这几列信息：

| 属性名                         | 含义   |
|-----------------------------|--|
| <code>actRows</code>        | 当前算子实际输出的数据条数。   |
| <code>execution info</code> | <code>time</code> 显示从进入算子到离开算子的全部时间，包括所有子算子操作的全部执行时间。如果该算子被父算子多次调用 (loops)，这个时间就是累积的时间。 <code>loops</code> 是当前算子被父算子调用的次数。 <code>rows</code> 是当前算子返回的行的总数。 |
| <code>memory</code>         | 当前算子占用内存大小   |
| <code>disk</code>           | 当前算子占用磁盘大小   |

一个例子：

```
mysql> explain analyze select * from t where a < 10;
+-----+-----+-----+-----+
| id      | estRows | actRows | task      | access object      | execution info
|          |          |          | operator info    |                   | memory           |
| disk   |          |          |          |          |                   |
+-----+-----+-----+-----+
| IndexLookUp_10      | 9.00    | 9       | root      |                   | time:641.245µs, loops:2,
| rpc num: 1, rpc time:242.648µs, proc keys:0 |                   | 9.23046875 KB |
| N/A   |          |          |          |          |                   |
| |-IndexRangeScan_8(Build) | 9.00    | 9       | cop[tikv] | table:t, index:idx_a(a) | time:142.94µs, loops:10,
|                           |          |          |          | range:[-inf,10), keep order:false | N/A           |
| N/A   |          |          |          |          |                   |
| \-TableRowIDScan_9(Probe) | 9.00    | 9       | cop[tikv] | table:t      | time:141.128µs, loops:10
|                           |          |          |          | keep order:false | N/A           |
| N/A   |          |          |          |          |                   |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

从上述例子中可以看出，优化器估算的 `estRows` 和实际执行中统计得到的 `actRows` 几乎是相等的，说明优化器估算的行数与实际行数的误差很小。同时 `IndexLookUp_10` 算子在实际执行过程中使用了约 `9 KB` 的内存，该 `SQL` 在执行过程中，没有触发过任何算子的落盘操作。

### 1.1.3 如何阅读算子的执行顺序

TiDB 的执行计划是一个树形结构，树中每个节点即是算子。考虑到每个算子内多线程并发执行的情况，在一条 `SQL` 执行的过程中，如果能够有一个手术刀把这棵树切开看看，大家可能会发现所有的算子都正在消耗 `CPU` 和 `内存` 处理数据，从这个角度来看，算子是没有执行顺序的。

但是如果从一行数据先后被哪些算子处理的角度来看，一条数据在算子上的执行是有顺序的。这个顺序可以通过下面这个规则简单总结出来：

`Build` 总是先于 `Probe` 执行，并且 `Build` 总是出现 `Probe` 前面

这个原则的前半句是说：如果一个算子有多个孩子节点，孩子节点 ID 后面有 `Build` 关键字的算子总是先于有 `Probe` 关键字的算子执行。后半句是说：TiDB 在展现执行计划的时候，`Build` 端总是第一个出现，接着才是 `Probe` 端。

一些例子：

```
TiDB(root@127.0.0.1:test) > explain select * from t use index(idx_a) where a = 1;
+-----+-----+-----+-----+
| id      | estRows | task      | access object      | operator info
|          |          |          | operator info    |                   |
|          |          |          |          |                   |
+-----+-----+-----+-----+
| IndexLookUp_7      | 10.00   | root      |                   |
|                   |          |          |          |                   |
| |-IndexRangeScan_5(Build) | 10.00   | cop[tikv] | table:t, index:idx_a(a) | range:[1,1], keep order:false, stat
|                           |          |          |          | s:pseudo |
| \-TableRowIDScan_6(Probe) | 10.00   | cop[tikv] | table:t      | keep order:false, stats:pseudo
|                           |          |          |          |                   |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

这里 `IndexLookUp_7` 算子有两个孩子结点：`IndexRangeScan_5(Build)` 和 `TableRowIDScan_6(Probe)`。可以看到，`IndexRangeScan_5(Build)` 是第一个出现的，并且基于上面这条规则，要得到一条数据，需要先执行它得到一个 `RowID` 以后，再由 `TableRowIDScan_6(Probe)` 根据前者读上来的 `RowID` 去获取完整的一行数据。

这种规则隐含的另一个信息是：在同一层级的节点中，出现在最前面的算子可能是最先被执行的，而出现在最末尾的算子可能是最后被执行的。比如下面这个例子：

```
TiDB(root@127.0.0.1:test) > explain select * from t t1 use index(idx_a) join t t2 use index() where t1.a = t2.a;
+-----+-----+-----+-----+
| id      | estRows | task      | access object           | operator info
|-----+-----+-----+-----+
| HashJoin_22          | 12487.50 | root      |                   | inner join, inner:TableReader_
26, equal:[eq(test.t.a, test.t.a)] |
| |-TableReader_26(Build) | 9990.00  | root      |                   | data:Selection_25
| |
| | \Selection_25        | 9990.00  | cop[tikv] |                   | not(isnull(test.t.a))
| |
| | \TableFullScan_24    | 10000.00 | cop[tikv] | table:t2           | keep order:false, stats:pseudo
| |
| \IndexLookUp_29(Probe) | 9990.00  | root      |                   |
| |
| |-IndexFullScan_27(Build) | 9990.00  | cop[tikv] | table:t1, index:idx_a(a) | keep order:false, stats:pseudo
| |
| \TableRowIDScan_28(Probe) | 9990.00  | cop[tikv] | table:t1           | keep order:false, stats:pseudo
| |
+-----+-----+-----+-----+
-----+
7 rows in set (0.00 sec)
```

要完成 `HashJoin_22`，需要先执行 `TableReader_26(Build)` 再执行 `IndexLookUp_29(Probe)`。而在执行 `IndexLookUp_29(Probe)` 的时候，又需要先执行 `IndexFullScan_27(Build)` 再执行 `TableRowIDScan_28(Probe)`。所以从整条执行链路来看，`TableRowIDScan_28(Probe)` 是最后被唤起执行的。

## 1.1.4 如何阅读扫表的执行计划

真正执行扫表（读盘或者读 TiKV Block Cache）操作的算子有如下几类：

- **TableFullScan**：这是大家所熟知的“全表扫”操作
- **TableRangeScan**：带有范围的表数据扫描操作，通常扫描的数据量不大
- **TableRowIDScan**：根据上层传递下来的 `RowID` 精确的扫描表数据的算子
- **IndexFullScan**：另一种“全表扫”，只不过这里扫的是索引数据，不是表数据
- **IndexRangeScan**：带有范围的索引数据扫描操作，通常扫描的数据量不大

TiDB 会汇聚 TiKV/TiFlash 上扫描的数据或者计算结果，这种“数据汇聚”算子目前有如下几类：

- **TableReader**：汇总 TiKV 上底层扫表算子是 `TableFullScan` 或 `TableRangeScan` 的算子。
- **IndexReader**：汇总 TiKV 上底层扫表算子是 `IndexFullScan` 或 `IndexRangeScan` 的算子。
- **IndexLookUp**：先汇总 Build 端 TiKV 扫描上来的 RowID，再去 Probe 端上根据这些 RowID 精确的读取 TiKV 上的数据。Build 端是 `IndexFullScan` 或 `IndexRangeScan`，Probe 端是 `TableRowIDScan`。
- **IndexMerge**：和 `IndexLookupReader` 类似，可以看做是它的扩展，可以同时读取多个索引的数据，有多个 Build 端，一个 Probe 端。执行过程也很类似，先汇总所有 Build 端 TiKV 扫描上来的 RowID，再去 Probe 端上根据这些 RowID 精确的读取 TiKV 上的数据。Build 端是 `IndexFullScan` 或 `IndexRangeScan`，Probe 端是 `TableRowIDScan`。

**IndexLookUp** 示例：

```
mysql> explain select * from t use index(idx_a);
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info |
+-----+-----+-----+-----+
IndexLookUp_6	10000.00	root			
	-IndexFullScan_4(Build)	10000.00	cop[tikv]	table:t, index:idx_a(a)	keep order:false, stats:pseudo
	TableRowIDScan_5(Probe)	10000.00	cop[tikv]	table:t	keep order:false, stats:pseudo
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

这里 `IndexLookUp_6` 算子有两个孩子节点：`IndexFullScan_4(Build)` 和 `TableRowIDScan_5(Probe)`。可以看到，`IndexFullScan_4(Build)` 执行索引全表扫，扫描索引 `a` 的所有数据，因为是全范围扫，这个操作将获得表中所有数据的 RowID，之后再由 `TableRowIDScan_5(Probe)` 去根据这些 RowID 去扫描所有的表数据。可以预见的是，这个执行计划不如直接使用 TableReader 进行全表扫，因为同样都是全表扫，这里的 `IndexLookUp` 多扫了一次索引，带来了额外的开销。

#### TableReader 示例：

```
mysql> explain select * from t where a > 1 or b >100;
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info |
+-----+-----+-----+-----+
TableReader_7	8000.00	root		data:Selection_6	
	-Selection_6	8000.00	cop[tikv]		or(gt(test.t.a, 1), gt(test.t.b, 100))
	TableFullScan_5	10000.00	cop[tikv]	table:t	keep order:false, stats:pseudo
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

在上面例子中 `TableReader_7` 算子的孩子节点是 `Selection_6`。以这个孩子节点为根的子树被当做一个 `Cop Task` 下发给了相应的 TiKV，这个 `Cop Task` 使用 `TableFullScan_5` 算子执行扫表操作。`Selection` 表示 SQL 语句中的选择条件，可能来自 SQL 语句中的 `WHERE / HAVING / ON` 子句。由 `TableFullScan_5` 可以看到，这个执行计划使用了一个全表扫描的操作，集群的负载将因此而上升，可能会影响到集群中正在运行的其他查询。这时候如果能够建立合适的索引，并且使用 `IndexMerge` 算子，将能够极大的提升查询的性能，降低集群的负载。

#### IndexMerge 示例：

注意：

目前 TiDB 的 `Index Merge` 特性在 4.0 RC 版本中默认关闭，同时 4.0 中的 `Index Merge` 目前支持的场景仅限于析取范式（`or` 连接的表达式），对合取范式（`and` 连接的表达式）将在之后的版本中支持。开启 `Index Merge` 特性，可通过在客户端中设置 session 或者 global 变量完成：`set @@tidb_enable_index_merge = 1;`

```
mysql> set @@tidb_enable_index_merge = 1;
mysql> explain select * from t use index(idx_a, idx_b) where a > 1 or b > 1;
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info |
+-----+-----+-----+-----+
IndexMerge_16	6666.67	root			
	-IndexRangeScan_13(Build)	3333.33	cop[tikv]	table:t, index:idx_a(a)	range:(1,+inf], keep order:false, stats:pseudo
	-IndexRangeScan_14(Build)	3333.33	cop[tikv]	table:t, index:idx_b(b)	range:(1,+inf], keep order:false, stats:pseudo
	TableRowIDScan_15(Probe)	6666.67	cop[tikv]	table:t	keep order:false, stats:pseudo
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

`IndexMerge` 使得数据库在扫描表数据时可以使用多个索引。这里 `IndexMerge_16` 算子有三个孩子节点，其中 `IndexRangeScan_13` 和 `IndexRangeScan_14` 根据范围扫描得到符合条件的所有 `RowID`，再由 `TableRowIDScan_15` 算子根据这些 `RowID` 精确的读取所有满足条件的数据。

## 1.1.5 如何阅读聚合的执行计划

**Hash Aggregate** 示例：

TiDB 上的 `Hash Aggregation` 算子采用多线程并发优化，执行速度快，但会消耗较多内存。下面是一个 `Hash Aggregate` 的例子：

```
TiDB(root@127.0.0.1:test) > explain select /*+ HASH_AGG() */ count(*) from t;
+-----+-----+-----+-----+
| id      | estRows | task      | access object | operator info
+-----+-----+-----+-----+
| HashAgg_11   | 1.00    | root      |               | funcs:count(Column#7)->Column#4
| └TableReader_12 | 1.00    | root      |               | data:HashAgg_5
|   └HashAgg_5   | 1.00    | cop[tikv] |               | funcs:count(1)->Column#7
|     └TableFullScan_8 | 10000.00 | cop[tikv] | table:t      | keep order:false, stats:pseudo
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

一般而言 TiDB 的 `Hash Aggregate` 会分成两个阶段执行，一个在 TiKV/TiFlash 的 `coprocessor` 上，计算聚合函数的中间结果。另一个在 TiDB 层，汇总所有 `Coprocessor Task` 的中间结果后，得到最终结果。

**Stream Aggregate** 示例：

TiDB `Stream Aggregation` 算子通常会比 `Hash Aggregate` 占用更少的内存，有些场景中也会比 `Hash Aggregate` 执行的更快。当数据量太大或者系统内存不足时，可以试试 `Stream Aggregate` 算子。一个 `Stream Aggregate` 的例子如下：

```
TiDB(root@127.0.0.1:test) > explain select /*+ STREAM_AGG() */ count(*) from t;
+-----+-----+-----+-----+
| id      | estRows | task      | access object | operator info
+-----+-----+-----+-----+
| StreamAgg_16   | 1.00    | root      |               | funcs:count(Column#7)->Column#4
| └TableReader_17 | 1.00    | root      |               | data:StreamAgg_8
|   └StreamAgg_8   | 1.00    | cop[tikv] |               | funcs:count(1)->Column#7
|     └TableFullScan_13 | 10000.00 | cop[tikv] | table:t      | keep order:false, stats:pseudo
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

和 `Hash Aggregate` 类似，一般而言 TiDB 的 `Stream Aggregate` 也会分成两个阶段执行，一个在 TiKV/TiFlash 的 `Coprocessor` 上，计算聚合函数的中间结果。另一个在 TiDB 层，汇总所有 `Coprocessor Task` 的中间结果后，得到最终结果。

## 1.1.6 如何阅读 Join 的执行计划

TiDB 的 Join 算法包括如下几类：

- Hash Join
- Merge Join
- Index Hash Join
- Index Merge Join
- Apply

下面分别通过一些例子来解释这些 Join 算法的执行过程

**Hash Join** 示例：

TiDB 的 `Hash Join` 算子采用了多线程优化，执行速度较快，但会消耗较多内存。一个 `Hash Join` 的例子如下：

```
mysql> explain select /*+ HASH_JOIN(t1, t2) */ * from t t1 join t2 on t1.a = t2.a;
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info
+-----+-----+-----+-----+
HashJoin_33	10000.00	root	inner join, inner:TableReader_43, equal:[eq(test.t.a, test.t2.a)]		
+--TableReader_43(Build)	10000.00	root	data:Selection_42		
	+--Selection_42	10000.00	cop[tikv]	not(isnull(test.t2.a))	
	+--TableFullScan_41	10000.00	cop[tikv]	table:t2	keep order:false
	+--TableReader_37(Probe)	10000.00	root	data:Selection_36	
	+--Selection_36	10000.00	cop[tikv]	not(isnull(test.t.a))	
	+--TableFullScan_35	10000.00	cop[tikv]	table:t1	keep order:false
+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

Hash Join 会将 Build 端的数据缓存在内存中，根据这些数据构造出一个 Hash Table，然后读取 Probe 端的数据，用 Probe 端的数据去探测（Probe）Build 端构造出来的 Hash Table，将符合条件的数据返回给用户。

#### Merge Join 示例：

TiDB 的 Merge Join 算子相比于 Hash Join 通常会占用更少的内存，但可能执行时间会更久。当数据量太大，或系统内存不足时，建议尝试使用。下面是一个 Merge Join 的例子：

```
mysql> explain select /*+ SM_JOIN(t1) */ * from t t1 join t t2 on t1.a = t2.a;
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info
+-----+-----+-----+-----+
MergeJoin_6	10000.00	root	inner join, left key:test.t.a, right key:test.t.a		
+--IndexLookUp_13(Build)	10000.00	root			
	+--IndexFullScan_11(Build)	10000.00	cop[tikv]	table:t2, index:idx_a(a)	keep order:true
	+--TableRowIDScan_12(Probe)	10000.00	cop[tikv]	table:t2	keep order:false
	+--IndexLookUp_10(Probe)	10000.00	root		
	+--IndexFullScan_8(Build)	10000.00	cop[tikv]	table:t1, index:idx_a(a)	keep order:true
	+--TableRowIDScan_9(Probe)	10000.00	cop[tikv]	table:t1	keep order:false
+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

Merge Join 算子在执行时，会从 Build 端把一个 Join Group 的数据全部读取到内存中，接着再去读 Probe 端的数据，用 Probe 端的每行数据去和 Build 端的完整的一个 Join Group 依次去看是否匹配（除了满足等值条件以外，还有其他非等值条件，这里的“匹配”主要是指查看是否满足非等值条件）。Join Group 指的是所有 Join Key 上值相同的数据。

#### Index Hash Join 示例：

INL\_HASH\_JOIN(t1\_name [, tl\_name]) 提示优化器使用 Index Nested Loop Hash Join 算法。该算法与 Index Nested Loop Join 使用条件完全一样，但在某些场景下会更为节省内存资源。

```
mysql> explain select /*+ INL_HASH_JOIN(t1) */ * from t t1 join t t2 on t1.a = t2.a;
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info
+-----+-----+-----+-----+
| IndexHashJoin_32 | 10000.00 | root | | inner join, inner:IndexLookUp_
23, outer key:test.t.a, inner key:test.t.a |
| TableReader_35(Build) | 10000.00 | root | | data:Selection_34
| | Selection_34 | 10000.00 | cop[tikv] | not(isnull(test.t.a))
| | TableFullScan_33 | 10000.00 | cop[tikv] | table:t2 | keep order:false
| | IndexLookUp_23(Probe) | 1.00 | root | |
| | Selection_22(Build) | 1.00 | cop[tikv] | not(isnull(test.t.a))
| | | IndexRangeScan_20 | 1.00 | cop[tikv] | table:t1, index:idx_a(a) | range: decided by [eq(test.t.a
, test.t.a)], keep order:false
| | TableRowIDScan_21(Probe) | 1.00 | cop[tikv] | table:t1 | keep order:false
+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

**Index Merge Join** 示例：

INL\_MERGE\_JOIN(t1\_name [, tl\_name]) 提示优化器使用 Index Nested Loop Merge Join 算法。该算法相比于 INL\_JOIN 会更节省内存。该算法使用条件包含 INL\_JOIN 的所有使用条件，但还需要添加一条：join keys 中的内表列集合是内表使用的 index 的前缀，或内表使用的 index 是 join keys 中的内表列集合的前缀。

```
mysql> explain select /*+ INL_MERGE_JOIN(t2@sel_2) */ * from t t1 where t1.a in ( select t2.a from t t2 where t2.b
< t1.b);
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info
+-----+-----+-----+-----+
| IndexMergeJoin_23 | 6.39 | root | | semi join, inner:Projection_21,
outer key:test.t.a, inner key:test.t.a, other cond:lt(test.t.b, test.t.b) |
| TableReader_28(Build) | 7.98 | root | | data:Selection_27
| | Selection_27 | 7.98 | cop[tikv] | not(isnull(test.t.a)), not(isnul
l(test.t.b))
| | TableFullScan_26 | 8.00 | cop[tikv] | table:t1 | keep order:false, stats:pseudo
| | Projection_21(Probe) | 1.25 | root | test.t.a, test.t.b
| | IndexLookUp_20 | 1.25 | root | |
| | Selection_18(Build) | 1.25 | cop[tikv] | not(isnull(test.t.a))
| | | IndexRangeScan_16 | 1.25 | cop[tikv] | table:t2, index:idx_a(a) | range: decided by [eq(test.t.a,
test.t.a)], keep order:true, stats:pseudo
| | | Selection_19(Probe) | 1.25 | cop[tikv] | not(isnull(test.t.b))
| | TableRowIDScan_17 | 1.25 | cop[tikv] | table:t2 | keep order:false, stats:pseudo
+-----+-----+-----+-----+
10 rows in set (0.01 sec)
```

**Apply** 示例：

```
mysql> explain select * from t t1 where t1.a in ( select avg(t2.a) from t2 where t2.b < t1.b);
+-----+-----+-----+-----+
| id      | estRows | task      | access object | operator info
|         |          |          |               |
+-----+-----+-----+-----+
| Projection_10    | 10000.00 | root      |              | test.t.id, test.t.a, test.t.b
|         |          |          |               |
| └─Apply_12       | 10000.00 | root      | semi join, inner:StreamAgg_30, equal:[eq(Column#8, Column#7)] | test.t.id, test.t.a, test.t.b, cast(test.t.a, decimal(20,0) BINARY)->Column#8
|   └─Projection_13(Build) | 10000.00 | root      |              | test.t.id, test.t.a, test.t.b, cast(test.t.a, decimal(20,0) BINARY)->Column#8
|     └─TableReader_15  | 10000.00 | root      | data:TableFullScan_14 | data:TableFullScan_14
|         |          |          |               |
|         └─TableFullScan_14 | 10000.00 | cop[tikv] | table:t1    | keep order:false
|             |          |          |               |
|             └─StreamAgg_30(Probe) | 1.00    | root      | funcs:avg(Column#12, Column#13)->Column#7 | funcs:avg(Column#12, Column#13)->Column#7
|                 |          |          |               |
|                 └─TableReader_31  | 1.00    | root      | data:StreamAgg_19 | data:StreamAgg_19
|                     |          |          |               |
|                     └─StreamAgg_19  | 1.00    | cop[tikv] |              | funcs:count(test.t2.a)->Column#12, funcs:sum(test.t2.a)->Column#13 | funcs:count(test.t2.a)->Column#12, funcs:sum(test.t2.a)->Column#13
|                         |          |          |               |
|                         └─Selection_29  | 8000.00 | cop[tikv] |              | lt(test.t2.b, test.t.b) | lt(test.t2.b, test.t.b)
|                             |          |          |               |
|                             └─TableFullScan_28 | 10000.00 | cop[tikv] | table:t2    | keep order:false
|                                 |          |          |               |
+-----+-----+-----+-----+
-----+
10 rows in set, 1 warning (0.00 sec)
```

## 1.1.7 EXPLAIN FOR CONNECTION

`EXPLAIN FOR CONNECTION` 用于获得一个连接中最后执行的查询的执行计划，其输出格式与 `EXPLAIN` 完全一致。但 TiDB 中的实现与 MySQL 不同，除了输出格式之外，还有以下区别：

- MySQL 返回的是正在执行的查询计划，而 TiDB 返回的是最后执行的查询计划。
- MySQL 的文档中指出，MySQL 要求登录用户与被查询的连接相同，或者拥有 `PROCESS` 权限，而 TiDB 则要求登录用户与被查询的连接相同，或者拥有 `SUPER` 权限。

## 1.2 优化器模型

优化器的作用是在合理的时间内找到合理的执行计划。TiDB 采用了 System R 的优化器模型，优化过程分为逻辑优化和物理优化两个阶段。在逻辑优化过程中，依次遍历内部定义实现的优化规则，不断地调整 SQL 的逻辑执行计划。物理优化则是将改写后逻辑执行计划变成可以执行的物理执行计划，这一过程会决定执行操作的具体方法，比如用什么索引读表，用什么算法做 Join 操作等。

### 1.2.1 常见逻辑优化规则简介

#### 1 基本逻辑算子介绍

TiDB 中的逻辑算子主要有以下几个：

- DataSource：数据源，表示一个源表，如 `select * from t` 中的 `t`。
- Selection：代表了相应的过滤条件，`select * from t where a = 5` 中的 `where a = 5`。
- Projection：投影操作，也用于表达式计算，`select c, a + b from t` 里面的 `c` 和 `a + b` 就是投影和表达式计算操作。
- Join：两个表的连接操作，`select t1.b, t2.c from t1 join t2 on t1.a = t2.a` 中的 `t1 join t2 on t1.a = t2.a` 就是两个表 `t1` 和 `t2` 的连接操作。Join 有内连接，左连接，右连接等多种连接方式。

`Selection`, `Projection`, `Join` (简称 SPJ) 是 3 种最基本的算子。

#### 2 常见逻辑优化规则

逻辑优化是基于规则的优化，通过对输入的逻辑执行计划按顺序应用优化规则，使整个逻辑执行计划变得更加高效。这些常用逻辑优化规则包括：

| 列表 1           | 列表 2               |
|----------------|--------------------|
| 1、列裁剪          | 6、外连接转内连接          |
| 2、分区剪裁         | 7、子查询去关联           |
| 3、聚合消除         | 8、谓词下推             |
| 4、Max / Min 优化 | 9、聚合下推             |
| 5、外连接消除        | 10、TopN / Limit 下推 |

逻辑优化部分示例如下：

#### 例子 1：外连接消除

外连接消除指的是将整个连接操作从查询中移除。外连接消除需要满足一定条件：

- 条件 1：LogicalJoin 的父亲算子只会用到 LogicalJoin 的 outer plan 所输出的列
- 条件 2：
  - 条件 2.1：LogicalJoin 中的 join key 在 inner plan 的输出结果中满足唯一性
  - 条件 2.2：LogicalJoin 的父亲算子会对输入的记录去重

条件 1 和条件 2 必须同时满足，但条件 2.1 和条件 2.2 只需满足一条即可。

满足条件 1 和 条件 2.1 的一个例子：

```
select t1.a from t1 left join t2 on t1.b = t2.b;
```

可以被改写成：

```
select t1.a from t1;
```

## 例子 2：Max / Min 优化

Max / Min 优化，会对 Max / Min 语句进行改写。如下面的语句：

```
select min(id) from t;
```

改成下面的写法，可以实现类似的效果：

```
select id from t order by id desc limit 1;
```

前一个语句生成的执行计划，是一个 TableScan 上面接一个 Aggregation，这是一个全表扫描的操作。后一个语句，生成执行计划是 TableScan + Sort + Limit。通常数据表中的 id 列是主键或者存在索引，数据本身有序，这样 Sort 就可以消除，最终变成 TableScan/IndexLookUp + Limit，这样就避免了全表扫描的操作，只需要读到第一条数据就能返回结果。

最大最小消除由优化器“自动”地做这个变换。

## 1.2.2 物理优化原理

物理优化是基于代价的优化，这一阶段中，优化器会为逻辑执行计划中的每个算子选择具体的物理实现，以将逻辑优化阶段产生的逻辑执行计划转换成物理执行计划。逻辑算子的不同物理实现有着不同的时间复杂度、资源消耗和物理属性等。在这个过程中，优化器会根据数据的统计信息来确定不同物理实现的代价，并选择整体代价最小的物理执行计划。

物理优化需要做的决策有很多，比如说：

- 读取数据的方式：使用索引扫描或全表扫描读取数据。
- 与此同时，如果存在多个索引，索引之间的选择，也同步完成。
- 逻辑算子的物理实现，即实际使用的算法。
- 是否可以将算子下推到存储层执行，以提升执行效率。

## 1.2.3 统计信息的收集与维护

TiDB 优化器会根据统计信息来选择最优的执行计划。统计信息收集了表级别和列级别的信息，表的统计信息包括总行数和修改的行数。列的统计信息包括不同值的数量、NULL 的数量、直方图、列上出现次数最多的值 TOPN 等信息。

### 1 手动搜集

通过执行 ANALYZE 语句来收集统计信息。如需更快的分析速度，可将 `tidb_enable_fast_analyze`（默认值为 0）设置为 1 来打开快速分析功能，此时将采取采样的方式收集统计信息。以数据库中 person 表为例，使用 fast analyze 的执行语句如下：

```
set @@tidb_enable_fast_analyze = 1;
analyze table person;
```

收集统计信息过程中，可以通过 `show analyze status` 语句查询执行状态，该语句也可以通过 `where` 子句对输出结果进行过滤，显示输出结果如下：

```
mysql> show analyze status where job_info = 'analyze columns';
+-----+-----+-----+-----+
| Table_schema | Table_name | Job_info      | Start_time      | State   |
+-----+-----+-----+-----+
test	person	analyze columns	2020-03-07 06:22:34	finished
test	customer	analyze columns	2020-03-07 06:32:19	finished
test	person	analyze columns	2020-03-07 06:35:27	finished
+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

## 2 自动更新

在执行 DML 语句时，TiDB 会自动更新表的总行数以及修改的行数。这些信息会定期自动持久化，更新周期默认是 1 分钟（`20 * stats-lease`）

注意：`stats-lease` 的默认值是 3s，如果将其设定为 0，则关闭统计信息自动更新。

## 3 统计信息查看

查看表的统计信息 meta 信息：

```
mysql> show stats_meta where table_name = 'person';
+-----+-----+-----+-----+
| Db_name | Table_name | Partition_name | Update_time      | Modify_count | Row_count |
+-----+-----+-----+-----+
| test    | person     |                 | 2020-03-07 07:20:54 |          0 |       4 |
+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

查看表的健康度信息：

```
mysql> show stats_healthy where table_name = 'person';
+-----+-----+-----+
| Db_name | Table_name | Partition_name | Healthy |
+-----+-----+-----+
| test    | person     |                 | 100 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

可通过 `SHOW STATS_HISTOGRAMS` 来查看列的不同值数量以及 NULL 值数量等信息：

```
mysql> show stats_histograms where table_name = 'person';
+-----+-----+-----+-----+-----+-----+-----+
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time      | Distinct_count | Null_count
| Avg_col_size | Correlation |
+-----+-----+-----+-----+-----+-----+-----+
| test    | person     |                 | name       | 0 | 2020-03-07 07:20:54 |          4 |       0
|       6.25 |      -0.2 |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

可通过 `SHOW STATS_BUCKETS` 来查看直方图每个桶的信息：

```
mysql> show stats_buckets;
+-----+-----+-----+-----+-----+-----+-----+-----+
| Db_name | Table_name | Partition_name | Column_name | Is_index | Bucket_id | Count | Repeats | Lower_Bound | Upper_Bound |
+-----+-----+-----+-----+-----+-----+-----+-----+
| test   | person    |           | name      | 0 | 0 | 1 | 1 | jack     | jack
|       |           |           |           |   |   |   |   |           |
| test   | person    |           | name      | 0 | 1 | 2 | 1 | peter    | peter
|       |           |           |           |   |   |   |   |           |
| test   | person    |           | name      | 0 | 2 | 3 | 1 | smith    | smith
|       |           |           |           |   |   |   |   |           |
| test   | person    |           | name      | 0 | 3 | 4 | 1 | tom     | tom
|       |           |           |           |   |   |   |   |           |
+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)
```

## 4 删除统计信息

可通过执行 `DROP STATS` 语句来删除统计信息。语句如下：

```
mysql> DROP STATS person;
```

## 5 统计信息导入导出

### 统计信息导出

通过以下接口可以获取数据库  `${db_name}` 中的表  `${table_name}` 的 json 格式的统计信息：

```
http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}
```

示例：获取本机上 test 数据库中 person 表的统计信息：

```
curl -G "http://127.0.0.1:10080/stats/dump/test/person" > person.json
```

### 统计信息导入

将统计信息导出接口得到的 json 文件导入数据库中：

```
mysql> LOAD STATS 'file_name';
```

`file_name` 为被导入的统计信息文件名。

## 1.3 SQL Plan Management

### 1.3.1 背景

执行计划是影响 SQL 执行性能的一个非常关键的因素，SQL 执行计划的稳定性也对整个集群的效率有着非常大的影响。然而，当出现类似统计信息过时、添加或者删除了索引等情况时，优化器并不能确保一定生成一个很好的执行计划。此时执行计划可能发生预期外的改变，导致执行时间过长。因此 TiDB 提供了 SQL Plan Management 功能，用于为某些类型的 SQL 绑定执行计划，并且被绑定的执行计划会根据数据的变化而不断地演进。

### 1.3.2 SQL Bind

SQL Bind 是 SQL Plan Management 的第一步，在 TiDB 3.0 版本中 GA。使用它，用户可以为某一类型的 SQL 绑定执行计划。当出现执行计划不优时，可以使用 SQL Bind 在不更改业务的情况下快速地对执行计划进行修复。

创建绑定可以使用如下的 SQL：

```
CREATE [GLOBAL | SESSION] BINDING FOR SelectStmt USING SelectStmt;
```

该语句可以在 GLOBAL 或者 SESSION 作用域内为 SQL 绑定执行计划。在不指定作用域时，默认作用域为 SESSION。被绑定的 SQL 会被参数化，然后存储到系统表中。在处理 SQL 查询时，只要参数化后的 SQL 和系统表中某个被绑定的 SQL 匹配即可使用相应的优化器 Hint。

“参数化”指的是把 SQL 中的常量用 “?” 替代，统一语句中的大小写，清理掉多余的空格、换行符等操作。

创建一个绑定的例子：

```
TiDB(root@127.0.0.1:test) > create binding for select * from t where a = 1 using select * from t use index(idx_a) where a = 1;
Query OK, 0 rows affected (0.00 sec)
```

查看刚才创建的 binding，下面输出结果中 `original_sql` 即为参数化后的 SQL：

```
TiDB(root@127.0.0.1:test) > show bindings;
+-----+-----+-----+-----+-----+
| original_sql          | Bind_sql           | Default_db | Status | Create_time
| Update_time            | Bind_id | Charset | Collation |           |
+-----+-----+-----+-----+-----+
| select * from t where a = ? | select * from t use index(idx_a) where a = 1 | test      | using   | 2020-03-08 14:00:28.819 |
| 2020-03-08 14:00:28.819 | 1 | utf8    | utf8_general_ci |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

如果要删除创建的 binding 可通过如下语句：

```
TiDB(root@127.0.0.1:test) > drop binding for select * from t where a = 1;
Query OK, 0 rows affected (0.00 sec)

TiDB(root@127.0.0.1:test) > show bindings;
Empty set (0.00 sec)
```

### 1.3.3 Baseline Evolution

为了解决只能手动创建 Binding 的问题，4.0 版本中 TiDB 提供了自动创建 Binding 功能，通过将 `tidb_capture_plan_baselines` 变量的值设置为 `on`，就可以自动为某一段时间内出现多次的 SQL 去创建绑定。TiDB 会为那些出现了至少两次的 SQL 创建绑定，统计 SQL 的出现次数依赖 TiDB 4.0 版本中提供的 Statements Summary 功能。可通过如下方法打开自动为出现了两次以上的 SQL 创建绑定的开关：

```
set tidb_enable_stmt_summary = 1;      -- 开启 statement summary
set tidb_capture_plan_baselines = 1;    -- 开启自动绑定功能
```

接着连续跑两遍如下查询即可自动为其创建一条绑定：

```
TiDB(root@127.0.0.1:test) > select * from t;
Empty set (0.01 sec)

TiDB(root@127.0.0.1:test) > select * from t;
Empty set (0.00 sec)
```

再查看 global bindings 即可发现自动创建的 binding：

```
TiDB(root@127.0.0.1:test) > show global bindings;
+-----+-----+-----+-----+-----+-----+
| Original_sql | Bind_sql | Default_db | Status | Create_time
| Update_time   | Charset | Collation |
+-----+-----+-----+-----+-----+
| select * from t | SELECT /*+ USE_INDEX(@`sel_1` `test`.`t` )*/ * FROM `t` | test | using | 2020-03-08 14:09:30.129 | 2020-03-08 14:09:30.129 |           |           |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

随着数据的变更，或者表结构定义的变化，可能原先绑定的执行计划已经不是最优的了，在 4.0 版本中可以通过 `set global tidb_evolve_plan_baselines = 1` 开启自动演进功能，来适应新的变化。对于自动演进来说，需要解决两个主要问题：一个是演进哪些 SQL，一个是如何演进 SQL。

对于第一个问题，直观的想法是对于那些已绑定，且仍然是最优的执行计划，是不需要被演进的。如果某条 SQL 的最优执行计划没有在绑定的执行计划中，优化器会去演进绑定的执行计划，去看看这个新生成的执行计划究竟是不是最优的。在遇到这种情况后，TiDB 会将新生成的执行计划标记为待验证，并交由后台线程去验证执行计划，也就是上面所说的第二个问题。

对于第二个问题，关键点在于如何确定新生成的执行计划是比之前更好。最可靠的办法便是真实地执行一遍，并将其与优化器在多个绑定中选出的最优执行计划进行对比，只有当待验证的比已经被绑定的执行计划要好一定程度，才将其标记为可用。当然，实际执行带来的问题便是对系统集群的影响。为了减少自动演进对集群的影响，可以通过

`tidb_evolve_plan_task_max_time` 来限制每个执行计划运行的最长时间，其默认值为十分钟；通过 `tidb_evolve_plan_task_start_time` 和 `tidb_evolve_plan_task_end_time` 可以限制运行演进任务的时间窗口，默认的时间窗口为全天。

## 1.4 参数调优指南

### 1.4.1 优化器参数调优

TiDB 中所有的优化器参数可通过如下语句查看：

```
TiDB(root@127.0.0.1:test) > show variables like "%tidb%opt%";
+-----+-----+
| Variable_name | Value |
+-----+-----+
tidb_opt_agg_push_down	0
tidb_opt_concurrency_factor	3
tidb_opt_copcpu_factor	3
tidb_opt_correlation_exp_factor	1
tidb_opt_correlation_threshold	0.9
tidb_opt_cpu_factor	3
tidb_opt_desc_factor	3
tidb_opt_disk_factor	1.5
tidb_opt_insubq_to_join_and_agg	1
tidb_opt_join_reorder_threshold	0
tidb_opt_memory_factor	0.001
tidb_opt_network_factor	1
tidb_opt_scan_factor	1.5
tidb_opt_seek_factor	20
tidb_opt_write_row_id	0
tidb_optimizer_selectivity_level	0
+-----+-----+
16 rows in set (0.01 sec)
```

接下来的小节将描述如何调整这些参数来控制优化器的行为。

### 1.4.2 控制优化器代价模型

以下 10 个参数用于控制优化器的代价模型：

```
TiDB(root@127.0.0.1:test) > show variables like "%tidb%factor%";
+-----+-----+
| Variable_name | Value |
+-----+-----+
tidb_opt_concurrency_factor	3
tidb_opt_copcpu_factor	3
tidb_opt_correlation_exp_factor	1
tidb_opt_cpu_factor	3
tidb_opt_desc_factor	3
tidb_opt_disk_factor	1.5
tidb_opt_memory_factor	0.001
tidb_opt_network_factor	1
tidb_opt_scan_factor	1.5
tidb_opt_seek_factor	20
+-----+-----+
10 rows in set (0.01 sec)
```

假设要让优化器更加偏向先读再按照逆序排序而不是使用 TiKV 的逆序扫，可以调高 `tidb_opt_desc_factor`：

默认情况下按照索引逆序排序的执行计划：

```
TiDB(root@127.0.0.1:test) > desc select * from t order by a desc;
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info |
+-----+-----+-----+-----+
| Projection_13 | 10000.00 | root | | test.t.a, test.t.b
| IndexLookUp_12 | 10000.00 | root | | 
| IndexFullScan_10(Build) | 10000.00 | cop[tikv] | table:t, index:idx_a(a) | keep order:true, desc, stats:ps
eudo |
| TableRowIDScan_11(Probe) | 10000.00 | cop[tikv] | table:t | keep order:false, stats:pseudo
|
+-----+-----+-----+-----+
-----+
4 rows in set (0.00 sec)
```

假设因为某种原因 TiKV 逆序扫的速度非常慢，可以通过调高该参数来摆脱逆序扫的性能问题：

```
TiDB(root@127.0.0.1:test) > set @@tidb_opt_desc_factor = 10;
Query OK, 0 rows affected (0.00 sec)

TiDB(root@127.0.0.1:test) > desc select * from t order by a desc;
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info |
+-----+-----+-----+-----+
| Sort_4 | 10000.00 | root | | test.t.a:desc
| TableReader_8 | 10000.00 | root | | data:TableFullScan_7
| TableFullScan_7 | 10000.00 | cop[tikv] | table:t | keep order:false, stats:pseudo
|
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

### 1.4.3 优化规则开关和黑名单

TiDB 有一个聚合下推的优化规则，因为不能确保所有场景下该优化规则都是合适的，所以目前默认关闭。这个优化规则会尽可能的把聚合算子下推到 Join 算子的下面，如果下推后能够大大减少 Join 的计算量，可以通过打开这个下推开关来提速 SQL 的执行。

一个默认情况下聚合没有下推到 Join 下面的例子：

```
TiDB(root@127.0.0.1:test) > desc select count(*) from t t1 join t t2 on t1.a = t2.a group by t1.a;
+-----+-----+-----+-----+
| id | estRows | task | access object | operator info |
+-----+-----+-----+-----+
| HashAgg_10 | 7992.00 | root | | group by:test.t.a, funcs:count(1)
->Column#7 |
| MergeJoin_13 | 12487.50 | root | | inner join, left key:test.t.a, right key:test.t.a |
| IndexReader_41(Build) | 9990.00 | root | | index:IndexFullScan_40
| | IndexFullScan_40 | 9990.00 | cop[tikv] | table:t2, index:idx_a(a) | keep order:true, stats:pseudo
| | IndexReader_39(Probe) | 9990.00 | root | | index:IndexFullScan_38
| | IndexFullScan_38 | 9990.00 | cop[tikv] | table:t1, index:idx_a(a) | keep order:true, stats:pseudo
|
+-----+-----+-----+-----+
-----+
6 rows in set (0.00 sec)
```

接下来可以通过打开开关 `tidb_opt_agg_push_down` 来把聚合下推到 Join 下面：

```
TiDB(root@127.0.0.1:test) > set tidb_opt_agg_push_down = 1;
Query OK, 0 rows affected (0.00 sec)

TiDB(root@127.0.0.1:test) > desc select count(*) from t t1 join t t2 on t1.a = t2.a group by t1.a;
+-----+-----+-----+-----+
| id      | estRows | task      | access object          | operator info
+-----+-----+-----+-----+
| HashAgg_11        | 7992.00 | root      |                      | group by:test.t.a, funcs:count(Co
lumn#8)->Column#7
| └─HashJoin_24      | 9990.00 | root      |                      | inner join, inner:HashAgg_37, equ
al:[eq(test.t.a, test.t.a)]
|   ├─HashAgg_37(Build) | 7992.00 | root      |                      | group by:test.t.a, funcs:count(1)
->Column#8, funcs:firstrow(test.t.a)->test.t.a |
|   | └─IndexReader_44    | 9990.00 | root      |                      | index:IndexFullScan_43
|   |   └─IndexFullScan_43 | 9990.00 | cop[tikv] | table:t2, index:idx_a(a) | keep order:false, stats:pseudo
|   └─IndexReader_48(Probe) | 9990.00 | root      |                      | index:IndexFullScan_47
|     └─IndexFullScan_47 | 9990.00 | cop[tikv] | table:t1, index:idx_a(a) | keep order:false, stats:pseudo
+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

此外，TiDB 优化器使用 `mysql.opt_rule_blacklist` 来禁用出现在这个表中的逻辑优化规则。

```
TiDB(root@127.0.0.1:test) > desc select * from t where a > 10;
+-----+-----+-----+-----+
| id      | estRows | task      | access object | operator info
+-----+-----+-----+-----+
| TableReader_7        | 3333.33 | root      |                      | data:Selection_6
| └─Selection_6        | 3333.33 | cop[tikv] |                      | gt(test.t.a, 10)
|   └─TableFullScan_5    | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:pseudo
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

假设上面的表达式下推导致了性能回退，可以通过把 `predicate_pushdown` 添加到黑名单中来禁用：

```
TiDB(root@127.0.0.1:test) > insert into mysql.opt_rule_blacklist values("predicate_push_down");
Query OK, 1 row affected (0.00 sec)
```

要在当前 session 生效，需要执行 `reload` 语句：

```
TiDB(root@127.0.0.1:test) > admin reload opt_rule_blacklist;
Query OK, 0 rows affected (0.00 sec)
```

接着再查看执行计划，会发现该过滤条件没有下推到 TiKV 上执行了：

```
TiDB(root@127.0.0.1:test) > desc select * from t where a > 10;
+-----+-----+-----+-----+
| id      | estRows | task      | access object | operator info
+-----+-----+-----+-----+
| Selection_5        | 8000.00 | root      |                      | gt(test.t.a, 10)
| └─TableReader_7      | 10000.00 | root      |                      | data:TableFullScan_6
|   └─TableFullScan_6    | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:pseudo
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

## 1.4.4 表达式下推黑名单

当 TiDB 从 TiKV 中读取数据的时候，TiDB 会尽量下推一些表达式运算到 TiKV 中，从而减少数据传输量以及 TiDB 单一节点的计算压力。本文将介绍 TiDB 已支持下推的表达式，以及如何禁止下推特定表达式。

**禁止特定表达式下推：**当函数的计算过程由于下推而出现异常时，可通过黑名单功能禁止其下推来快速恢复业务。具体而言，用户可以将函数或运算符名加入黑名单 `mysql.expr_pushdown_blacklist` 中，以禁止特定表达式下推。

**加入黑名单：**执行以下步骤，可将一个或多个函数或运算符加入黑名单：

- 向 `mysql.expr_pushdown_blacklist` 插入对应的函数名或运算符名。
- 执行 `admin reload expr_pushdown_blacklist`。

**移出黑名单：**执行以下步骤，可将一个或多个函数及运算符移出黑名单：

- 从 `mysql.expr_pushdown_blacklist` 表中删除对应的函数名或运算符名。
- 执行 `admin reload expr_pushdown_blacklist`。

一个例子：

```
TiDB(root@127.0.0.1:test) > explain select * from t where a < 2;
+-----+-----+-----+-----+
| id      | estRows | task      | access object | operator info      |
+-----+-----+-----+-----+
TableReader_7	3323.33	root		data:Selection_6
└Selection_6	3323.33	cop[tikv]		lt(test.t.a, 2)
└TableFullScan_5	10000.00	cop[tikv]	table:t	keep order:false, stats:pseudo
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

将 `<` 加入黑名单：

```
TiDB(root@127.0.0.1:test) > insert into mysql.expr_pushdown_blacklist values('<');
Query OK, 1 row affected (0.00 sec)

TiDB(root@127.0.0.1:test) > admin reload expr_pushdown_blacklist;
Query OK, 0 rows affected (0.00 sec)
```

再次执行，就会发现这个 `<` 就没有被下推到 TiKV 了：

```
TiDB(root@127.0.0.1:test) > explain select * from t where a < 2;
+-----+-----+-----+-----+
| id      | estRows | task      | access object | operator info      |
+-----+-----+-----+-----+
Selection_5	8000.00	root		lt(test.t.a, 2)
└TableReader_7	10000.00	root		data:TableFullScan_6
└TableFullScan_6	10000.00	cop[tikv]	table:t	keep order:false, stats:pseudo
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

# 1.5 限制 SQL 内存使用和执行时间

限制SQL内存使用和执行时间主要是为限制消耗系统资源多的SQL，防止某条SQL造成OOM或影响到集群的整体性能。

## 1.5.1 限制SQL内存使用

限制SQL内存使用有如下三种方式。

1. 通过配置文件参数修改
2. `oom-action`
  - 默认值：“log”
  - 当 TiDB 中单条 SQL 的内存使用超出 `mem-quota-query` 限制且不能再利用临时磁盘时的行为。
  - 目前合法的选项为 [“log”, “cancel”]。
  - 如果配置项使用的是 “log”，那么当一条 SQL 的内存使用超过一定阈值后，TiDB 会在 log 文件中打印一条 LOG，然后这条 SQL 继续执行，之后如果发生了 OOM 可以在 LOG 中找到对应的 SQL。
  - 如果上面的配置项使用的是 “cancel”，那么当一条 SQL 的内存使用超过一定阈值后，TiDB 会立即中断这条 SQL 的执行并给客户端返回一个 error，error 信息中会详细写明这条 SQL 执行过程中各个占用内存比较多的物理执行算子的内存使用情况。
3. `mem-quota-query`
  - 默认值：34359738368 (32GB)
  - 单条 SQL 语句可以占用的最大内存阈值。
  - 超过该值的请求会被 `oom-action` 定义的行为所处理。
4. `oom-use-tmp-storage`
  - 默认值：true
  - 设置是否在单条 SQL 语句的内存使用超出 `mem-quota-query` 限制时为某些算子启用临时磁盘。
5. `tmp-storage-path`
  - 默认值：`<操作系统临时文件夹>/tidb/tmp-storage`
  - 单条 SQL 语句的内存使用超出 `mem-quota-query` 限制时，某些算子的临时磁盘存储位置。
  - 此配置仅在 `oom-use-tmp-storage` 为 true 时有效。
6. 修改 session 级变量
7. `tidb_mem_quota_query`
  - 默认值：32 GB
  - 设置一条查询语句的内存使用阈值。如果一条查询语句执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 `oomAction` 项所指定的行为。
8. `tidb_mem_quota_hashjoin`
  - 默认值：32 GB
  - 设置 HashJoin 算子的内存使用阈值。如果 HashJoin 算子执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 `oomAction` 项所指定的行为。
9. `tidb_mem_quota_mergejoin`
  - 默认值：32 GB
  - 设置 MergeJoin 算子的内存使用阈值。如果 MergeJoin 算子执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 `oomAction` 项所指定的行为。
10. `tidb_mem_quota_sort`
  - 默认值：32 GB
  - 设置 Sort 算子的内存使用阈值。如果 Sort 算子执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 `oomAction` 项所指定的行为。
11. `tidb_mem_quota_topn`
  - 默认值：32 GB
  - 设置 TopN 算子的内存使用阈值。如果 TopN 算子执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 `oomAction` 项所指定的行为。

12. `tidb_mem_quota_indexlookupreader`
  - 默认值：32 GB
  - 设置 `IndexLookupReader` 算子的内存使用阈值。如果 `IndexLookupReader` 算子执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 `OOMAction` 项所指定的行为。
13. `tidb_mem_quota_indexlookupjoin`
  - 默认值：32 GB
  - 设置 `IndexLookupJoin` 算子的内存使用阈值。如果 `IndexLookupJoin` 算子执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 `OOMAction` 项所指定的行为。
14. `tidb_mem_quota_nestedloopapply`
  - 默认值：32 GB
  - 设置 `NestedLoopApply` 算子的内存使用阈值。如果 `NestedLoopApply` 算子执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 `OOMAction` 项所指定的行为。

示例：

配置整条 SQL 的内存使用阈值为 8GB：

```
set @@tidb_mem_quota_query = 8 << 30;
```

配置整条 SQL 的内存使用阈值为 8MB：

```
set @@tidb_mem_quota_query = 8 << 20;
```

### 1. 使用 Optimizer Hints

#### 2. `memory_quota`

- Hint 支持 MB 和 GB 两种单位。
- 限制查询执行时的内存使用，内存使用超过该限制时会根据当前设置的内存超限行为来打出一条 log 或者终止语句的执行。

示例：

限制SQL执行的内存为1024 MB：

```
select /*+ MEMORY_QUOTA(1024 MB) */ * from t;
```

## 1.5.2 限制SQL执行时间

### 1. 修改session/global变量

#### 2. `max_execution_time`

- 单位为：ms
- 目前对所有类型的 statement 生效，并非只对 SELECT 语句生效。实际精度在 100ms 级别，而非更准确的毫秒级别。

示例：

设置最大执行时间为10秒。

```
set @@global.MAX_EXECUTION_TIME=10000
```

### 1. 使用 Optimizer Hints

#### 2. `max_execution_time`

- 单位为：ms
- 把查询的执行时间限制在指定的 N 毫秒以内，超时后服务器会终止这条语句的执行。

示例：

设置SQL执行超时时间为1000 毫秒（即 1 秒）。

```
select /*+ MAX_EXECUTION_TIME(1000) */ * from t1 inner join t2 where t1.id = t2.id;
```

### 1.5.3 定位消耗系统资源多的查询语句

上面介绍了如何限制SQL的内存使用和执行时间，如果一条语句在执行过程中达到或超过资源使用阈值时（执行时间/使用内存量）则会即时将这条语句写入到日志文件（默认文件为：`tidb.log`），用于在语句执行结束前定位消耗系统资源多的查询语句，帮助用户分析和解决语句执行的性能问题。以下是一条 Expensive query 日志示例：

```
[2020/02/05 15:32:25.096 +08:00] [WARN] [expensivequery.go:167] [expensive_query] [cost_time=60.008338935s] [wait_time=0s] [request_count=1] [total_keys=70] [process_keys=65] [num_cop_tasks=1] [process_avg_time=0s] [process_p90_time=0s] [process_max_time=0s] [process_max_addr=10.0.1.9:20160] [wait_avg_time=0.002s] [wait_p90_time=0.002s] [wait_max_time=0.002s] [wait_max_addr=10.0.1.9:20160] [stats=t:pseudo] [conn_id=60026] [user=root] [database=test] [table_ids="[1 22]"] [txn_start_ts=414420273735139329] [mem_max="1035 Bytes (1.0107421875 KB)"] [sql="insert into t select sleep(1) from t"]
```

各字段的含义如下：

#### 基本字段

- `cost_time`：日志打印时语句已经花费的执行时间。
- `stats`：语句涉及到的表或索引使用的统计信息版本。值为 `pesudo` 时表示无可用统计信息，需要对表或索引进行 `analyze`。
- `table_ids`：语句涉及到的表的 ID。
- `txn_start_ts`：事务的开始时间戳，也是事务的唯一 ID，可以用这个值在 TiDB 日志中查找事务相关的其他日志。
- `sql`：SQL 语句。

#### 内存使用相关字段

- `mem_max`：日志打印时语句已经使用的内存空间。该项使用两种单位标识内存使用量，分别为 Bytes 以及易于阅读的自适应单位（比如 MB、GB 等）。

#### 用户相关字段

- `user`：执行语句的用户名。
- `conn_id`：用户的连接 ID，可以用类似 `con:60026` 的关键字在 TiDB 日志中查找该连接相关的其他日志。
- `database`：执行语句时使用的 database。

#### TiKV Coprocessor Task 相关字段

- `wait_time`：该语句在 TiKV 的等待时间之和，因为 TiKV 的 Coprocessor 线程数是有限的，当所有的 Coprocessor 线程都在工作的时候，请求会排队；当队列中有某些请求耗时很长的时候，后面的请求的等待时间都会增加。
- `request_count`：该语句发送的 Coprocessor 请求的数量。
- `total_keys`：Coprocessor 扫过的 key 的数量。
- `processed_keys`：Coprocessor 处理的 key 的数量。与 `total_keys` 相比，`processed_keys` 不包含 MVCC 的旧版本。如果 `processed_keys` 和 `total_keys` 相差很大，说明旧版本比较多。
- `num_cop_tasks`：该语句发送的 Coprocessor 请求的数量。
- `process_avg_time`：Coprocessor 执行 task 的平均执行时间。
- `process_p90_time`：Coprocessor 执行 task 的 P90 分位执行时间。
- `process_max_time`：Coprocessor 执行 task 的最长执行时间。
- `process_max_addr`：task 执行时间最长的 Coprocessor 所在地址。
- `wait_avg_time`：Coprocessor 上 task 的等待时间。
- `wait_p90_time`：Coprocessor 上 task 的 P90 分位等待时间。
- `wait_max_time`：Coprocessor 上 task 的最长等待时间。
- `wait_max_addr`：task 等待时间最长的 Coprocessor 所在地址。



## 第 2 章 TiDB Dashboard 介绍

从4.0版本开始，TiDB 提供了一个新的 Dashboard 运维管理工具，集成在 PD 组件上，默认地址为 `http://pd-url:pd_port/dashboard`。不同于 Grafana 监控是从数据库的监控视角出发，TiDB Dashboard 从 DBA 管理员角度出发，最大限度的简化管理员对 TiDB 数据库的运维，可在在一个界面查看到整个分布式数据库集群的运行状况，包括数据热点、SQL 运行情况、集群信息、日志搜索、实时性能分析等。

- [识别集群热点和业务访问模](#)
- [分析 SQL 执行性](#)
- [生成集群诊断报告](#)
- [日志搜索和导](#)
- [分析组件 CPU 消耗情](#)

## 2.1 识别集群热点和业务访问模式

TiDB Dashboard 通过 Key Visualizer (KeyVis) 工具提供了一个简单，直观，符合直觉的方式来帮助用户观测 TiDB 的数据访问模式以及数据热点，第一次让 DBA 或者开发者可以从数据库的角度去理解业务的负载，看到业务负载大概的长相，有点像医学上的 CT 成像，从而能根据负载的模式给业务的开发人员提供很好的改进建议，改进业务系统的设计。

### 1. 概念

- Bucket

一个 TiDB 集群中，Region（即一个数据块）的数量可能多达数十万。一般难以将这么多的 Region 信息显示在一个屏幕上。因此，在一张热力图中，Region 会被压缩到约 1500 个连续范围，每个范围称为「Bucket」。Region 压缩总是倾向于将流量较小的大量 Region 压缩为一个 Bucket，而尽量让高流量的 Region 独立成 Bucket，以便突出热点情况。

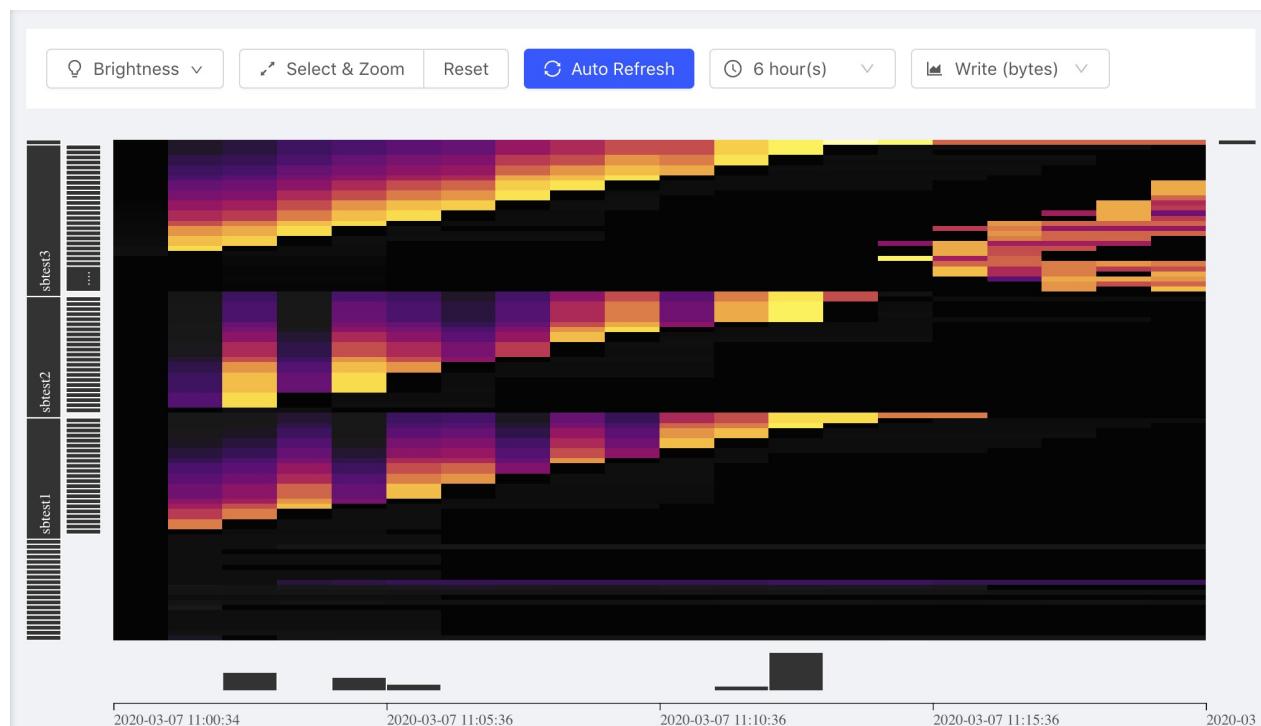
- 热力图

热力图是热点可视化的核心，它显示了一个指标随时间的变化。热力图的横轴 X 是时间，纵轴 Y 则表示集群里面的 Region，横跨 TiDB 集群上所有数据库和数据表。颜色越暗（cold）表示该区域的 Region 在这个时间段上读写流量较低，颜色越亮（hot）表示读写流量越高，即越热。

### 2. 界面图示

借用如下的热点示例图，可以观察到以下信息：

- 一个大型热力图，显示访问流量随时间的变化情况
- 下方和右侧为沿热力图的每个轴的平均值
- 左侧为表名、索引名等信息



### 3. 观察某一段时间或者 Region 范围

热点可视化默认会显示最近 6 小时整个数据库内的热力图。其中，越靠近右侧（当前时间）时，每个 Bucket 对应的时间间隔会越小。如果您想观察某个特定时间段或者特定的 Region 范围，则可以通过放大来获得更多细节。

- 在热图中向上或向下滚动
- 点击 Select & Zoom 按钮，然后点击并拖动以选择要放大的区域
- 点击 Reset 按钮，将 Region 范围重置为整个数据库
- 点击 时间选择框，重新选择观察时间段

注：使用后几种方法，将引起热图的重新绘制，您可能观察到热图与放大前有较大差异，这是一个正常的现象。它可能是由于在进行局部观察时，Region 压缩的粒度发生了变化，或者是局部范围内，「热」的基准发生了改变。

## 4. 调整亮度

热图使用颜色的明暗来表达一个 Bucket 的流量高低，颜色越暗（cold）表示该区域的 Region 在这个时间段上读写流量较低，颜色越亮（hot）表示读写流量越高，即越热。如果热图中的颜色太亮或太暗，则可能很难观察到细节。此时，我们可以点击 Brightness 按钮，然后通过滑块来调节页面的亮度。注：显示一个区域内的热图时，会根据区域内的流量情况来界定冷热。当整个区域流量不够均匀时，即使整体流量在数值上很低，您依然有可能观察到较大的亮色区域。请注意一定要结合数值一起分析。

## 5. 自动刷新

当我们需要实时观察数据库的流量分布情况时，可以点击 Auto Refresh 按钮来让热图每分钟自动刷新。请注意，如果您进行了时间范围或者 Region 范围的调整，自动刷新会被关闭。

## 6. 选择不同的指标

可以通过「指标选择框」来查看您关心的指标，目前支持如下指标：

- Read (bytes) 读流量
- Write (bytes) 写流量
- Read (keys) 读取行数
- Write (keys) 写入行数
- All 读写流量的总和

## 7. 关于热点引发的问题

热点的本质是大多数读写流量都只涉及个别 Region，进而导致集群中只有个别 TiKV 节点承载了大部分操作。KeyVis 将所有 Region 的读写流量按时间依次展示出来，使用颜色明暗表示读写流量的多少，以热力图的方式呈现。通常业务在访问数据库的时候都有特定的模式，而数据库的监控系统往往难以识别这些模式，需要数据库专家或者资深的 DBA 靠自己多年丰富的经验仔细的推测。所有业务开发人员可能会经常被问到一些问题，比如：

- 读多写少还是写多读少？
- 读写比例是怎样的？
- 有没有热点？
- 追加写还是随机写？
- 一直均匀的读写还是有周期性流量（比如每小时一次促销）？
- 有没有突发性流量？
- 如果业务发展了，是否存在设计瓶颈，能不能 scale 10 倍？
- 有没有扫表（其实就是顺序读）？

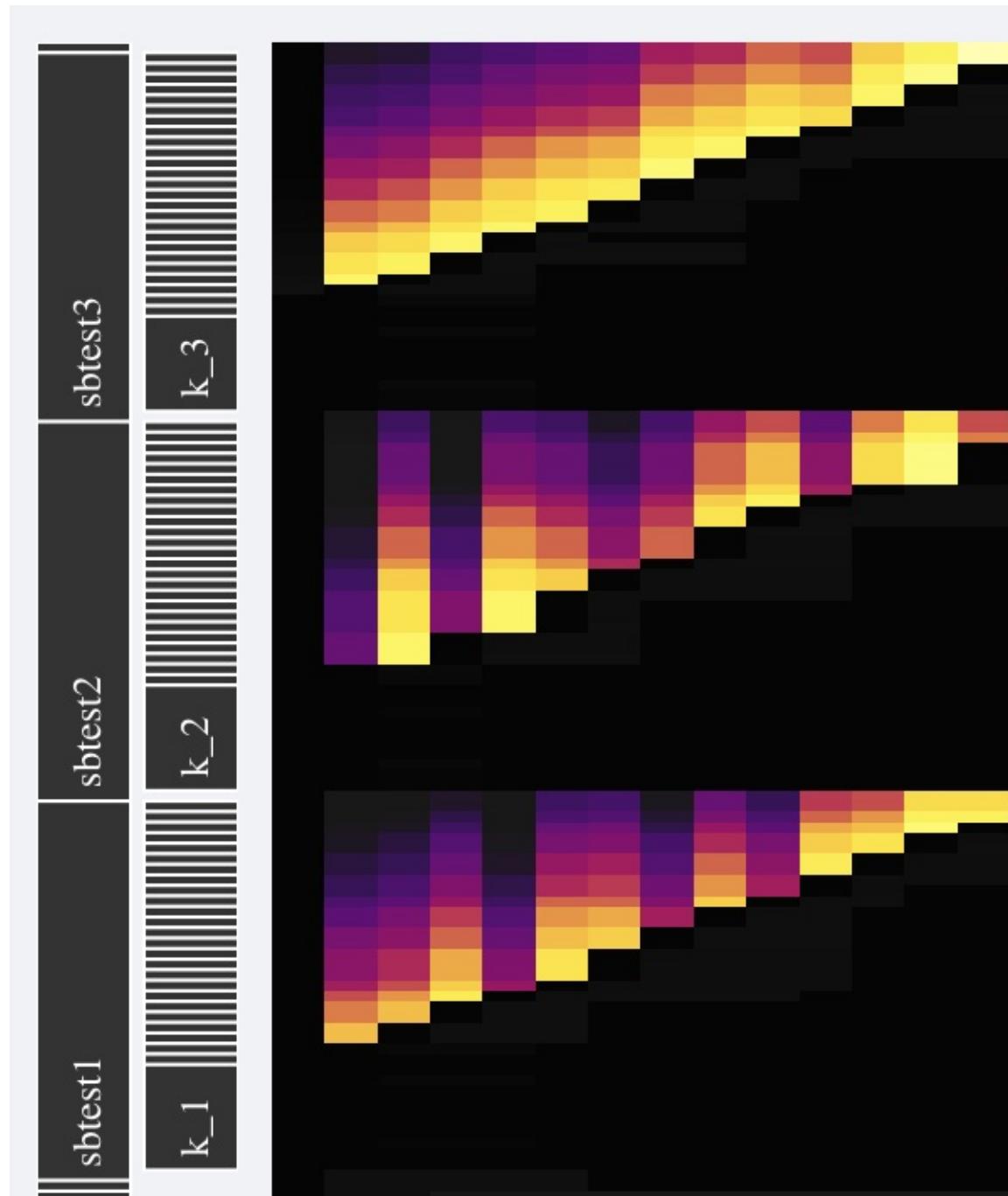
业务开发人员面对这些问题有时候一脸问号，脑子里面快速过了一遍自己的业务，大概有一百多张表，每个都有增删改查（CRUD），还有各种统计汇总的业务，有很多请求还是用户触发的，这些问题真的是很难回答。但是现在通过 TiDB Dashboard 很多业务的访问模式都可以得到比较直观的观测，DBA 和开发者能够一眼发现系统的读写比例，哪里有热点，甚

至能够精确到哪个数据块，哪些索引，也能预测如果流量持续增长，数据库系统是否能够轻松扩展。

## 8. 几种常见的访问模式

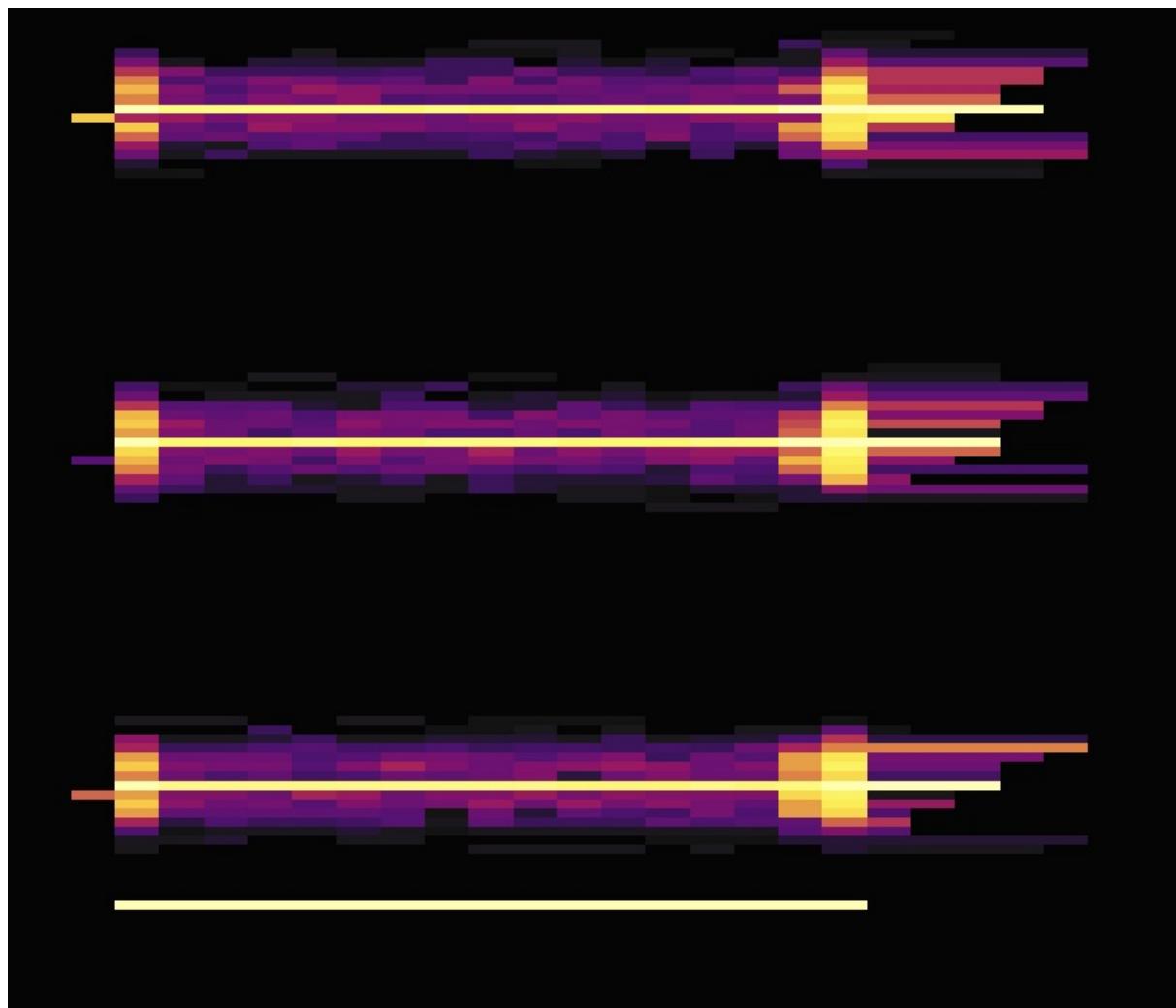
- 顺序读写

这里是一个三张表的顺序写入的截图，用的(sysbench prepare 命令)最左边我们可以看到表名为 sbtest1, sbtest2, sbtest3, 注意观察黄色（最亮的）的区块，大体上是呈现一个向上的斜线，这里的颜色亮度越高代表数据写得越多

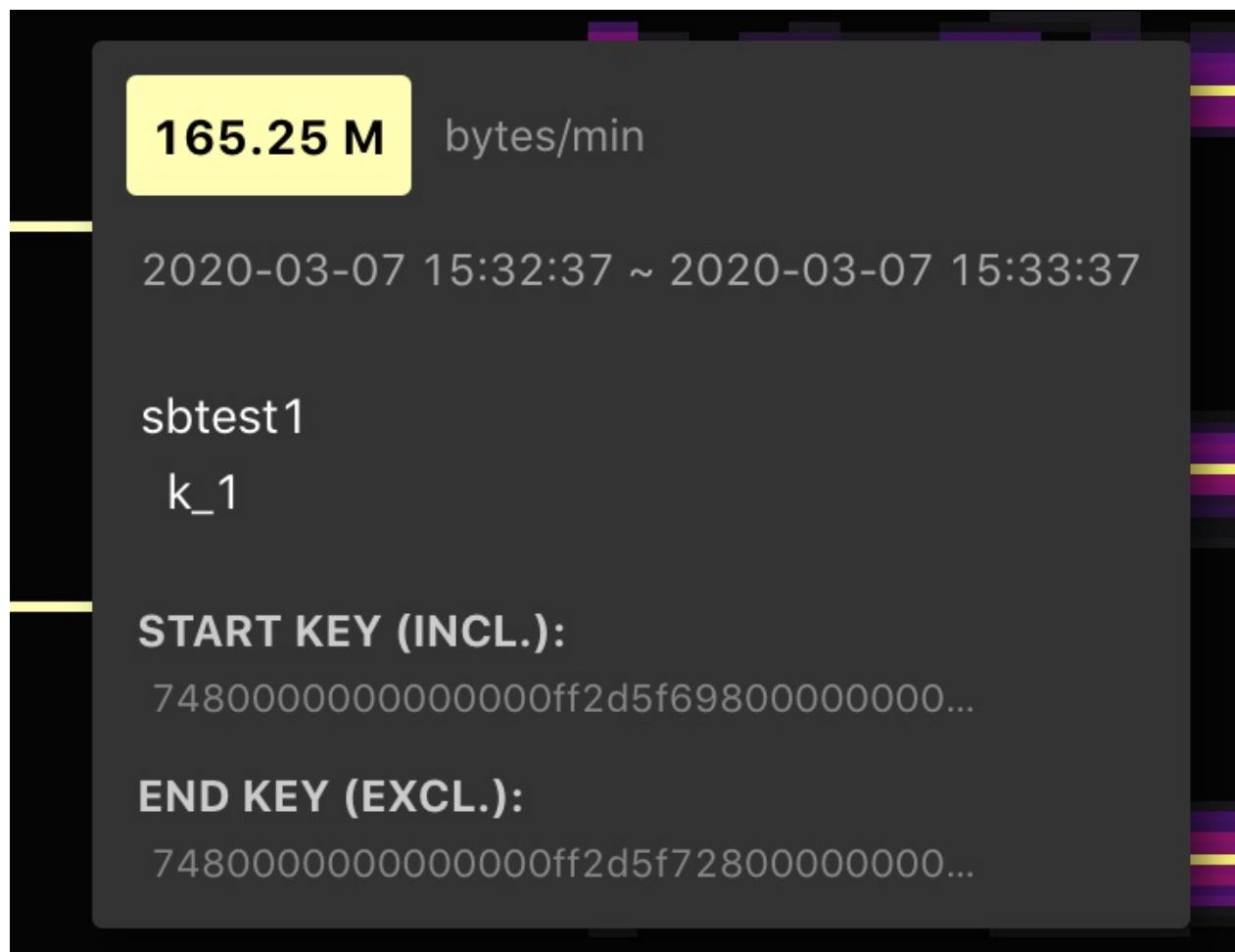


- 持续热点

通常如果业务有持续的热点，在KeyVis上也是一眼就看出来了，如下图，可以看到连续的金黄色光条，每个光条代表一段持续的热点块

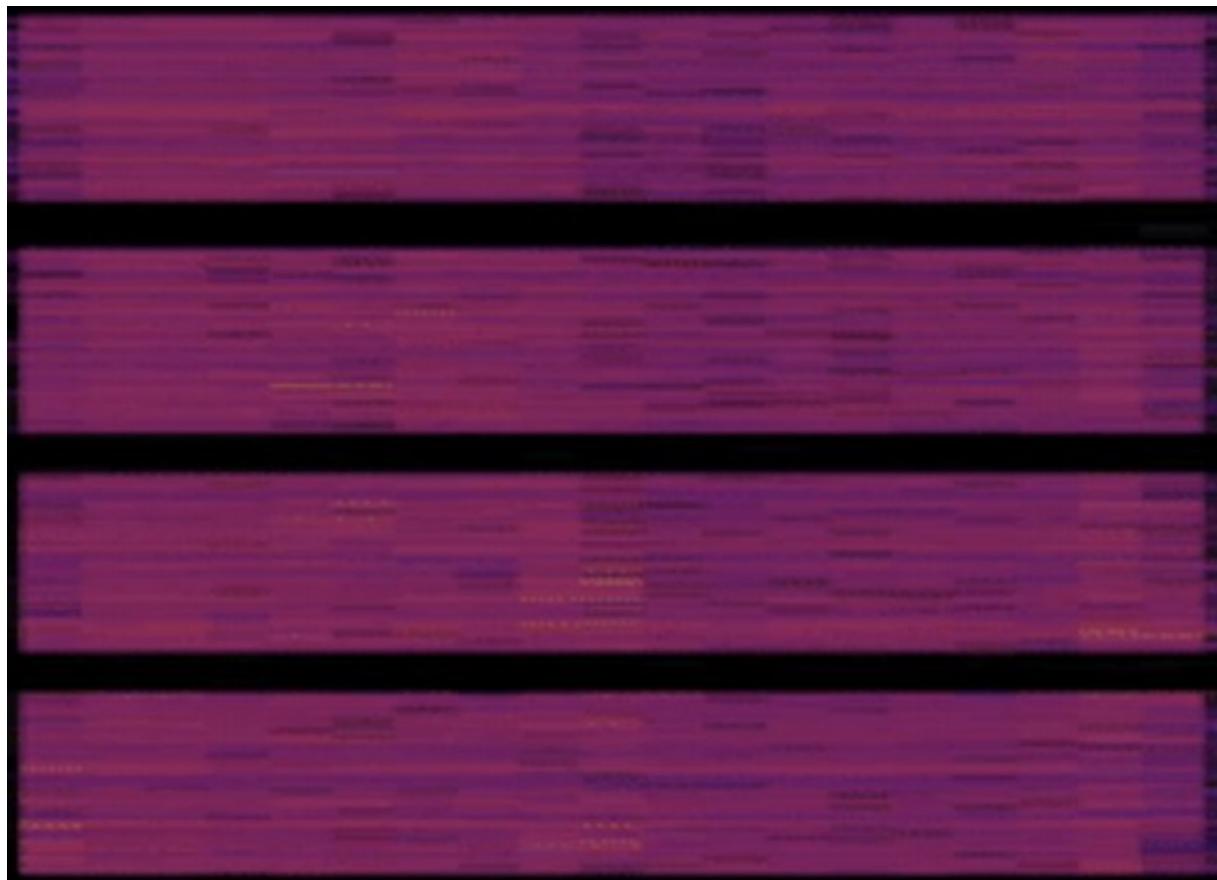


如果我们把光标移动到金黄色的热点上还能看到更具体的提示，如下图所示，可以看到每分钟的流量为 165.25 兆字节, 访问的表名是 sbtest1，访问的对象是表的一个索引，索引的名字是 k\_1, 我们也能看到这个块在存储层对应的 key 范围，即图中展示的 start key 和 end key。



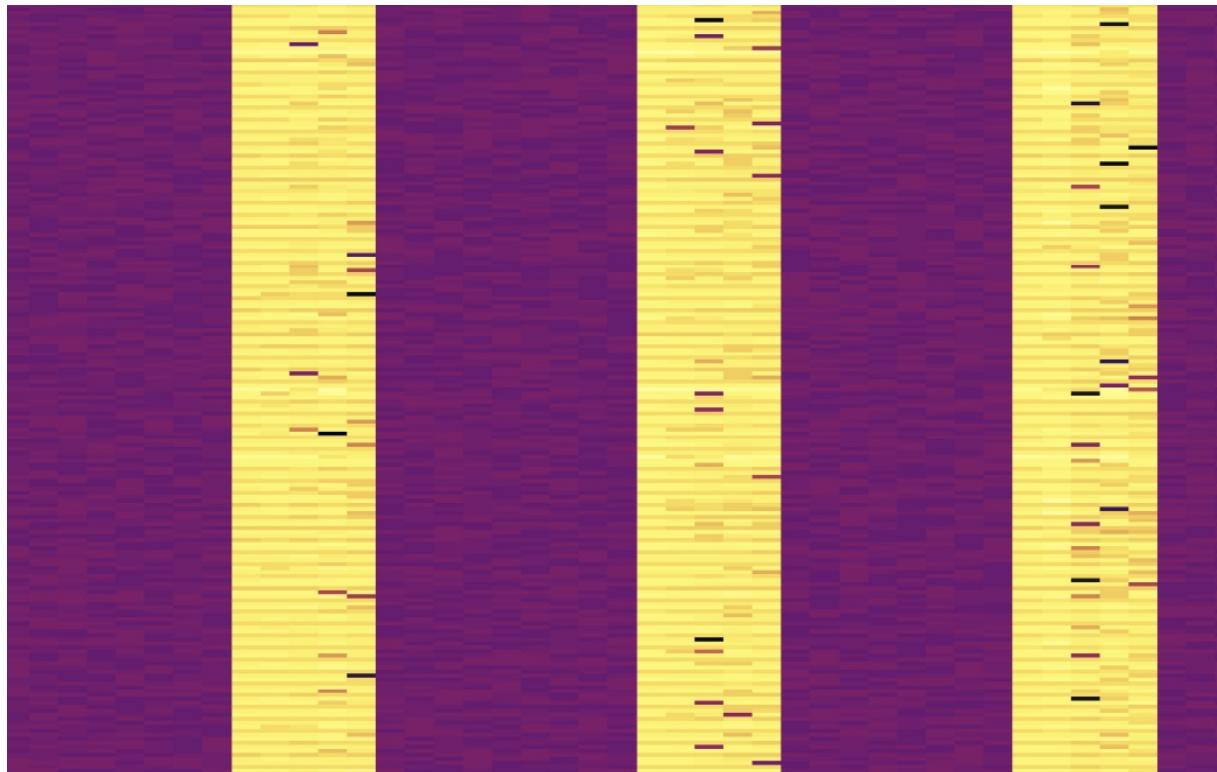
- 均匀分布

相信聪明的同学已经猜出来了，没有明显热点就是均匀负载，有兴趣的同学可以自己动手尝试构造一个类似的负载。这种负载下数据库有完美的扩展性，随着业务的流量上升做好扩容操作即可，流量下来了直接缩容。



- 周期性负载

系统的负载每隔一段时间会上来一次，随后又下降，如此反复，比如定时任务，整点促销，这些从图上看一目了然



如果大家用的是 TiDB 4.0 或者以上的版本，系统会自动根据负载情况来弹性伸缩，不再需要人工干预。弹性伸缩相关的细节请参考弹性调度章节。



## 2.2 分析 SQL 执行性能

上一节介绍通过 KeyVis 识别 TiDB 的业务模式。

本节将带领读者体会如何借助 TiDB Dashboard 的 Statements 信息，分析 SQL 执行情况，快速定位 SQL 性能问题。

### 1. Statements 是什么

Statement，即 SQL 语句。

针对 SQL 性能相关的问题，TiDB Dashboard 提供了 Statements 用来监控和统计 SQL。

例如页面上提供了丰富的列表信息，包括延迟、执行次数、扫描行数、全表扫描次数等，用来分析哪些类别的 SQL 语句耗时过长、消耗内存过多等情况，帮助用户定位性能问题。

### 2. 为什么要用可视化 Statements

TiDB 已支持多种性能排查工具。但在多种应用场景需求下，仍有不足，例如：

1. Grafana 不能排查单条 SQL 的性能问题
2. Slow log 只记录超过慢日志阀值的 SQL
3. General log 本身对性能有一定影响
4. Explain analyze 只能查看可以复现的问题
5. Profile 只能查看整个实例的瓶颈

因此推出可视化 Statements，可以直接在页面观察 SQL 执行情况，不需要查询系统表，便于用户定位性能问题。

### 3. 查看 Statements 整体情况

登录后，在左侧点击「SQL 语句分析」即可进入此功能页面。

在时间区间选项框中选择要分析的时间段，即可得到该时段所有数据库的 SQL 语句执行统计情况。

如果只关心某些数据库，则可以在第二个选项框中选择相应的数据库对结果进行过滤，支持多选。

结果以表格的形式展示，并支持按不同的列对结果进行排序，如下图所示。

1. 选择需要分析的时间段
2. 支持按数据库过滤
3. 支持按不同的指标排序

注意：

这里所指的 SQL 语句实际指的是某一类 SQL 语句。语法一致的 SQL 语句会规一化为一类相同的 SQL 语句。

例如：

```
SELECT * FROM employee WHERE id IN (1, 2, 3);
select * from EMPLOYEE where ID in (4, 5);
```

规一化为

```
select * from employee where id in (...);
```



## 4. 查看 Statements 详情

在 SQL 类别列，点击某类 SQL 语句，可以进入该 SQL 语句的详情页查看更详细的信息，以及该 SQL 语句在不同节点上执行的统计情况。

单个 Statements 详情页关键信息如下图所示。



## 5. Statements 参数配置

- `tidb_enable_stmt_summary`

Statements 功能默认开启，也可以通过设置系统变量打开，例如：

```
set global tidb_enable_stmt_summary = true;
```

- `tidb_stmt_summary_refresh_interval`

设置 `performance_schema.events_statements_summary_by_digest` 表的清空周期，单位是秒(s)，默认值是 1800，例如：

```
set global tidb_stmt_summary_refresh_interval = 1800;
```

- `tidb_stmt_summary_history_size`

设置 `performance_schema.events_statements_summary_by_digest_history` 表保存每种 SQL 的历史的数量，默认值是 24，例如：

```
set global tidb_stmt_summary_history_size = 24;
```

由于 Statements 信息是存储在内存表中，为了防止内存溢出等问题，需要限制保存的 SQL 条数和 SQL 的最大显示长度。这两个参数需要在 config.toml 的 `[stmt-summary]` 类别下配置：

- 通过 `max-stmt-count` 更改保存的 SQL 种类数量，默认 200 条。当 SQL 种类超过 `max-stmt-count` 时，会移除最近没有使用的 SQL
- 通过 `max-sql-length` 更改 `DIGEST_TEXT` 和 `QUERY_SAMPLE_TEXT` 的最大显示长度，默认是 4096

注意：

`tidb_stmt_summary_history_size`、`max-stmt-count`、`max-sql-length` 几项配置影响内存占用，建议根据实际情况调整，不宜设置得过大。

综上所述，可视化 Statements 可以快速定位某个 SQL 性能问题，也可以配合前一小节介绍的 KeyVis 进行分析。

## 2.3 生成集群诊断报告

诊断报告全称是 诊断报告 + 系统报告。

诊断报告是指 TiDB 内置的诊断，对系统在某一时间范围内的状态做出的系统诊断以及自动系统巡检，将诊断结果汇总一个诊断报告表，帮助 DBA 发现集群潜在的问题。

系统报告包括集群的拓扑信息，机器的硬件信息，服务器以及各个组件的负载信息，各个组件的监控信息以及配置信息。

最终将诊断报告和系统报告生成一份 html 格式的报告，并支持下载离线浏览及传阅。

### 2.3.1 如何使用

#### 1. 依赖

诊断报告需要读取监控数据，目前需要设置 PD 的 `pd-server.metric-storage` 值为 prometheus 的地址。可以通过下面命令动态设置改配置项。

```
curl -X POST -d '{"metric-storage":"http://{{PROMETHEUS_ADDRESS}}"}' http://{{PD_ADDRESS}}/pd/api/v1/config
```

示例：

```
curl -X POST -d '{"metric-storage":"http://127.0.0.1:9090"}' http://127.0.0.1:2379/pd/api/v1/config
```

#### 2. 关于时间范围的选择

建议生成诊断报告的时间范围在 2 min ~ 60 min 内，目前不建议生成时间范围超过 1 小时的诊断报告。

#### 3. 生成诊断报告并查看

点击完整报告的链接即可查看报告内容。





## 4. 生成对比诊断报告

对比两个诊断时间段的报告，要求 2 个时间段的跨度一样，比如 start 和 end 的时间差均为 10 分钟。



## 5. 提示

- i 图标：鼠标移动到 i 图标会显示每一行的说明注释。
- expand：点击 expand 展开会看到这项监控更加详细的信息，例如是哪个 instance, 哪个 label 等等。

| Load                        | the disk write latency in each node | instance          | Avg    | Max    | Min      |
|-----------------------------|-------------------------------------|-------------------|--------|--------|----------|
| Metric Name                 |                                     |                   |        |        |          |
| node_disk_write_latency     | <i>fold</i>                         |                   | 0.0002 | 0.0003 | 0.000007 |
| l-- node_disk_write_latency |                                     | 172.16.5.40:19110 | 0.0002 | 0.0003 | 0.000007 |
| node_disk_read_latency      | <i>expand</i>                       |                   |        |        |          |
| node_cpu_usage              | <i>expand</i>                       |                   | 80.56% | 81.55% | 79.4%    |
| node_mem_usage              | <i>expand</i>                       |                   | 27.9%  | 28.71% | 26.89%   |

## 2.3.2 自动诊断报告

自动诊断是 TiDB 的 4.0 引入的新特性，根据一些内置的诊断规则对系统做出诊断，这里具体可以参考下一章诊断系统表中的 `INSPECTION_RESULT` 诊断结果表，查看更详细的内容。

## 2.3.3 系统报告

### 1. Header

报告头，包括报告的时间范围，集群机器的硬件信息，集群拓扑。

## (1) Report Time Range

生成报告的时间范围。

| START_TIME          | END_TIME            |
|---------------------|---------------------|
| 2020-03-05 16:30:00 | 2020-03-05 16:50:00 |

## (2) cluster hardware

集群中服务器的硬件信息。

| HOST        | INSTANCE              | CPU_CORES | MEMORY (GB) | DISK (GB)  | UPTIME (DAY)       |
|-------------|-----------------------|-----------|-------------|--|--------------------|
| 172.16.5.40 | tidb*1<br>tikv*4 pd*1 | 20/40     | 122.696     | sda3: 1465.187500<br>nvme0n1: 343.715603<br>sda1: 0.185237 | 253.14520396237572 |

- HOST : 服务器地址
- INSTANCE : 表示服务器上运行的组件。 `tidb*1 tikv*4 pd*1` 表示这台服务器上运行着 1 个 TiDB 实例，4 个 TiKV 实例以及 1 个 PD 实例
- CPU\_CORES : CPU 核心数，逻辑核心 / 物理核心
- MEMORY : 内存容量，单位是 GB
- DISK : 磁盘容量，单位是 GB
- UPTIME: 服务器的启动时间，单位是 Day

## (3) cluster info

集群拓扑信息，信息来自 TiDB 的 `information_schema.cluster_info` 系统表。

| TYPE | INSTANCE          | STATUS_ADDRESS    | VERSION                                | GIT_HASH                                 |
|------|-------------------|-------------------|--|--|
| tidb | 172.16.5.40:4009  | 172.16.5.40:10089 | 5.7.25-TiDB-v4.0.0-beta-311-gd93c06149 | d93c061491d8094751a53c510bd8de886722952a |
| pd   | 172.16.5.40:24799 | 172.16.5.40:24799 | 4.1.0-alpha                            | 61d9f9cc35d3f191eb5e7ea1eb4f8e29eb73eda0 |
| tikv | 172.16.5.40:21150 | 172.16.5.40:21151 | 4.1.0-alpha                            | 8fa0e059e14c3a1433fcb581452f9ea0a14a72ce |
| tikv | 172.16.5.40:22150 | 172.16.5.40:22151 | 4.1.0-alpha                            | 8fa0e059e14c3a1433fcb581452f9ea0a14a72ce |
| tikv | 172.16.5.40:23150 | 172.16.5.40:23151 | 4.1.0-alpha                            | 8fa0e059e14c3a1433fcb581452f9ea0a14a72ce |
| tikv | 172.16.5.40:20150 | 172.16.5.40:20151 | 4.1.0-alpha                            | 8fa0e059e14c3a1433fcb581452f9ea0a14a72ce |

## 2. Load info

### (1) node load

服务器节点的负载信息，包括磁盘的读 / 写延迟，CPU、Memory 的 AVG、MAX、MIN 使用率。

| METRIC_NAME             | instance | AVG    | MAX    | MIN      |
|-------------------------|----------|--------|--------|----------|
| node_disk_write_latency |          | 0.0002 | 0.0003 | 0.000007 |
| node_disk_read_latency  |          |        |        |          |
| node_cpu_usage          |          | 80.56% | 81.55% | 79.4%    |
| node_mem_usage          |          | 27.9%  | 28.71% | 26.89%   |

## (2) process cpu usage

各个 TiDB / PD / TiKV 组件的 CPU 的 AVG、 MAX、 MIN 使用率。

| instance          | job  | AVG   | MAX   | MIN   |
|-------------------|------|-------|-------|-------|
| 172.16.5.40:10089 | tidb | 18.38 | 19.16 | 17.76 |
| 172.16.5.40:20151 | tikv | 9.68  | 10.03 | 9.42  |
| 172.16.5.40:24799 | pd   | 0.6   | 0.63  | 0.56  |
| 172.16.5.40:22151 | tikv | 0.36  | 1.08  | 0.04  |
| 172.16.5.40:23151 | tikv | 0.3   | 1.07  | 0.04  |
| 172.16.5.40:21151 | tikv | 0.27  | 0.71  | 0.04  |

## (3) TiKV Thread CPU Usage

查看 TiKV 各个线程的 CPU 使用率。

| METRIC_NAME                            | instance     | AVG               | MAX     | MIN    |
|--|--------------|-------------------|---------|--------|
| raftstore <a href="#">fold</a> >)      |              | 0.01              | 0.06    | 0      |
| \                                      | -- raftstore | 172.16.5.40:20151 | 0.06    | 0.06   |
| \                                      | -- raftstore | 172.16.5.40:22151 | 0.00003 | 0.0002 |
| \                                      | -- raftstore | 172.16.5.40:23151 | 0       | 0      |
| \                                      | -- raftstore | 172.16.5.40:21151 | 0       | 0      |
| apply <a href="#">expand</a> >)        |              | 0                 | 0       | 0      |
| sched_worker <a href="#">expand</a> >) |              | 0                 | 0       | 0      |
| snap <a href="#">expand</a> >)         |              | 0.000002          | 0.0002  | 0      |
| cop <a href="#">expand</a> >)          |              | 0.01              | 0.18    | 0      |
| grpc <a href="#">expand</a> >)         |              | 0.16              | 0.85    | 0      |

## (4) goroutines count

TiDB / PD 组件的 goroutines 的 AVG、 MAX、 MIN 数量信息。

| instance          | job  | AVG | MAX | MIN |
|-------------------|------|-----|-----|-----|
| 172.16.5.40:10089 | tidb | 899 | 996 | 651 |
| 172.16.5.40:24799 | pd   | 114 | 115 | 113 |

## 3. Overview

## (1) Time Consume

包括集群总体各项监控耗时以及各项耗时的占比。

| METRIC_NAME                                     | LABEL | TIME_RATIO | TOTAL_TIME | TOTAL_COUNT | P999   | P99      |
|---|-------|------------|------------|-------------|--------|----------|
| tidb_query <a href="#">expand</a> )             |       | 1          | 184598.05  | 9467997     | 0.4    | 0.12     |
| tidb_get_token <a href="#">expand</a> )         |       | 0.00007    | 12.1       | 9467600     | 0.0002 | 0.000001 |
| tidb_parse <a href="#">expand</a> )             |       | 0.000007   | 1.23       | 3593        | 0.04   | 0.008    |
| tidb_compile <a href="#">expand</a> )           |       | 0.00001    | 1.82       | 3593        | 0.04   | 0.03     |
| tidb_execute <a href="#">expand</a> )           |       | 0.000007   | 1.26       | 3593        | 0.2    | 0.17     |
| tidb_distsql_execution <a href="#">expand</a> ) |       | 0.0003     | 56.72      | 1953        | 1.41   | 0.5      |
| tidb_cop <a href="#">expand</a> )               |       | 0.001      | 206.21     | 2756        | 2.01   | 1.66     |
| tidb_transaction <a href="#">expand</a> )       |       | 0.09       | 15920.8    | 9456815     | 0.49   | 0.06     |

- METRIC\_NAME : 该项监控项的名称
- LABEL : 该项监控的 LABEL 信息，点击 expand 后可以查看该项监控更加详细的各项 LABEL 的监控信息。
- TIME\_RATIO : 该项监控消耗的总时间和 TIME\_RATIO 为 1 的监控行总时间比例

例如 tidb\_cop 的总耗时占 tidb\_query 总耗时的 0.001 秒

- TOTAL\_TIME : 该项监控的总耗时
- TOTAL\_COUNT : 该项监控执行的总次数
- P999 : 该项监控的 P999 最大执行时间
- P99 : 该项监控的 P99 最大执行时间
- P90 : 该项监控的 P90 最大执行时间
- P80 : 该项监控的 P80 最大执行时间

## (2) Error

集群内各种 error 出现的总次数。

| METRIC_NAME  | LABEL   | TOTAL_COUNT       |
|--|---|-------------------|
| tidb_binlog_error_total_count                                      |   | 0                 |
| tidb_handshake_error_total_count                                   |   | 0                 |
| tidb_transaction_retry_error_total_count                           |   |                   |
| tidb_kv_region_error_total_count <code>fold&gt;</code> )           |   | 739               |
| \  | -- tidb_kv_region_error_total_count           | not_leader 481    |
| \  | -- tidb_kv_region_error_total_count           | stale_command 256 |
| \  | -- tidb_kv_region_error_total_count           | epoch_not_match 1 |
| tidb_schema_lease_error_total_count                                |   |                   |
| tikv_grpc_error_total_count  |   |                   |
| tikv_critical_error_total_count                                    |   |                   |
| tikv_scheduler_is_busy_total_count                                 |   |                   |
| tikv_channel_full_total_count                                      |   |                   |
| tikv_coprocessor_request_error_total_count <code>fold&gt;</code> ) |   | 8                 |
| \  | -- tikv_coprocessor_request_error_total_count | meet_lock 7       |
| \  | -- tikv_coprocessor_request_error_total_count | not_leader 1      |
| tikv_engine_write_stall  |   | 0                 |
| tikv_server_report_failures_total_count                            |   | 0                 |
| tikv_storage_async_request_error <code>fold&gt;</code> )           |   | 793               |
| \  | -- tikv_storage_async_request_error           | snapshot 793      |
| tikv_lock_manager_detect_error_total_count                         |   | 0                 |
| tikv_backup_errors_total_count                                     |   |                   |
| node_network_in_errors_total_count                                 |   | 0                 |
| node_network_out_errors_total_count                                |   | 0                 |

## 4. TiDB

TiDB 组件相关的监控信息。

### (1) Time Consume

TiDB 的各项监控耗时以及各项耗时的占比。与 Overview 中类似，并且 label 信息会更丰富，可以看到更多细节。

### (2) Transaction

TiDB 事务相关的监控项。

| METRIC_NAME   | LABEL | TOTAL_VALUE | TOTAL_COUNT | P999    | P99    | P90  |
|---|-------|-------------|-------------|---------|--------|------|
| tidb_transaction_retry_num                                      |       | 0           | 0           | 0       | 0      | 0    |
| tidb_transaction_statement_num<br><a href="#">expand &gt;</a> ) |       | 17524099    | 17500676    | 511     | 501    | 398  |
| tidb_txn_region_num<br><a href="#">expand &gt;</a> )            |       | 900385      | 900095      | 4       | 1      | 1    |
| tidb_txn_kv_write_num<br><a href="#">expand &gt;</a> )          |       | 658293      | 1379        | 1023    | 1011   | 896  |
| tidb_txn_kv_write_size<br><a href="#">expand &gt;</a> )         |       | 216307980   | 1379        | 1043333 | 996147 | 2487 |
| tidb_load_safepoint_total_num<br><a href="#">expand &gt;</a> )  |       | 67          |             |         |        |      |
| tidb_lock_resolver_total_num<br><a href="#">expand &gt;</a> )   |       | 40          |             |         |        |      |

- TOTAL\_VALUE：该项监控在报告时间段内所有值的和
- TOTAL\_COUNT：该项监控出现的总次数
- P999：该项监控的 P999 最大值
- P99：该项监控的 P99 最大值
- P90：该项监控的 P90 最大值
- P80：该项监控的 P80 最大值

示例：

在报告时间范围内，`tidb_txn_kv_write_size`：一共有 1379 次事务的 kv 写入，总 kv 写入大小是 216307980，其中 P999、P99、P90、P80 的最大值分别为 1043333、996147、248705、235266，单位是 byte。

### (3) DDL-owner

TiDB DDL 的 owner 信息。

| MIN_TIME            | DDL OWNER         |
|---------------------|-------------------|
| 2020-03-05 16:30:00 | 172.16.5.40:10089 |

注意：

如果 owner 信息为空，不代表这个时间段内一定没有 owner。因为 TiDB 依据 `ddl_worker` 的监控信息来判断 DDL owner，所以也可能由于这个时间段内 `ddl_worker` 没有做任何 DDL job，导致 owner 信息为空。

## 5. PD / TiKV

PD 和 TiKV 的监控报告与之前的表结构类似，这里不再重复赘述，点击 i 图标查看相关注释查看更多信息即可。

## 6. Config

### (1) Scheduler Config

PD Scheduler 配置参数的 change history

| MIN_TIME                   | CONFIG_ITEM                     | VALUE  | CHANGE_COUNT |
|----------------------------|---------------------------------|--------|--------------|
| 2020-03-05 16:30:00.000000 | enable-makeup-replica           | 1      | 1            |
| 2020-03-05 16:30:00.000000 | enable-remove-down-replica      | 1      | 1            |
| 2020-03-05 16:30:00.000000 | enable-remove-extra-replica     | 1      | 1            |
| 2020-03-05 16:30:00.000000 | enable-replace-offline-replica  | 1      | 1            |
| 2020-03-05 16:30:00.000000 | high-space-ratio                | 0.6    | 1            |
| 2020-03-05 16:30:00.000000 | hot-region-cache-hits-threshold | 3      | 1            |
| 2020-03-05 16:30:00.000000 | hot-region-schedule-limit       | 4      | 1            |
| 2020-03-05 16:30:00.000000 | leader-schedule-limit           | 4      | 2            |
| 2020-03-05 16:46:00.000000 | leader-schedule-limit           | 8      | 2            |
| 2020-03-05 16:30:00.000000 | low-space-ratio                 | 0.8    | 1            |
| 2020-03-05 16:30:00.000000 | max-merge-region-keys           | 200000 | 1            |
| 2020-03-05 16:30:00.000000 | max-merge-region-size           | 20     | 1            |
| 2020-03-05 16:30:00.000000 | max-pending-peer-count          | 16     | 1            |
| 2020-03-05 16:30:00.000000 | max-replicas                    | 3      | 1            |
| 2020-03-05 16:30:00.000000 | max-snapshot-count              | 3      | 1            |
| 2020-03-05 16:30:00.000000 | merge-schedule-limit            | 8      | 1            |
| 2020-03-05 16:30:00.000000 | region-schedule-limit           | 2048   | 1            |
| 2020-03-05 16:30:00.000000 | replica-schedule-limit          | 64     | 1            |
| 2020-03-05 16:30:00.000000 | store-balance-rate              | 15     | 1            |
| 2020-03-05 16:30:00.000000 | tolerant-size-ratio             | 0      | 1            |

- MIN\_TIME：在报告时间范围内监控项生效的最长时间
- CONFIG\_ITEM：配置项的名称
- VALUE：配置项的值
- CHANGE\_COUNT：配置项在报告时间范围内被修改的次数

示例，上面报告中，`leader-schedule-limit` 配置参数有过修改

- 2020-03-05 16:30:00，`leader-schedule-limit` 的配置值为 4
- 2020-03-05 16:46:00，`leader-schedule-limit` 的配置值为 8，说明该配置的值在 2020-03-05 16:46:00 修改

注意：

由于拉取监控信息的延迟，上面 `leader-schedule-limit` 的修改时间可能要比 2020-03-05 16:46:00 早几秒，但延迟不会超过 prometheus 拉取监控的间隔时间。

## (2) TiDB GC Config

TiDB 的 GC 配置参数的 change history

| MIN_TIME                   | CONFIG_ITEM          | VALUE | CHANGE_COUNT |
|----------------------------|----------------------|-------|--------------|
| 2020-03-03 17:00:00.000000 | tikv_gc_life_time    | 6000  | 1            |
| 2020-03-03 17:00:00.000000 | tikv_gc_run_interval | 60    | 1            |

### (3) TiDB / PD / TiKV Current Config

该表的数据来源是 TiDB 的 `information_schema.cluster_config` 系统表。

注意：

current Config 是指生成报告时集群当前的配置值，不是用户选择的报告时间范围内的值。

## 2.4 日志搜索和导出

上一节介绍如何操作生成集群诊断报告。

本节将介绍如何使用 TiDB Dashboard 分析 TiDB 系统日志。

日志搜索功能允许用户在集群中搜索所有节点上的日志，在页面上预览搜索结果和导出日志。

### 1. 日志搜索功能

日志搜索主要是根据用户设置，对服务器上的日志内容进行筛选。

可设置参数包括：

- 时间范围：限定搜索日志的时间范围，为空的话默认搜索所有日志
- 日志等级：限定 最低日志等级，搜索该日志等级以上的所有日志
- 组件：选择要搜索的集群组件。支持选择多个组件
- 关键词：任何合法的任何字符串。选填，关键词之间以空格分割，支持 regex

点击「日志搜索」按钮后，会进入下面详情页面，如下图：

| 时间                        | 日志等级 | 组件   | 日志  |
|---------------------------|------|------|---|
| 2020-03-07T20:04:39+08:00 | WARN | TiDB | [base_client.go:170] "[pd] failed to get cluster id" [url=http://10.... |
| 2020-03-07T20:04:48+08:00 | WARN | TiKV | [server.rs:790] ["check: kernel"] [err="kernel parameters net.core....  |
| 2020-03-07T20:04:48+08:00 | WARN | TiKV | [server.rs:790] ["check: kernel"] [err="kernel parameters net.ipv4.t... |

### 2. 日志搜索进度

搜索进度在左边栏中，可以展示当前搜索的进度、状态以及每个节点日志搜索状态和统计等，搜索进度中还包括下载日志功能，见下小节图。展示状态示例：

- 运行中：开始搜索后，所有任务会进入「运行中」状态
- 成功：任务完成后自动转到「成功」，此时日志已缓存在 Dashboard 后端所在的本地磁盘中，可以提供给前端下载
- 失败：用户主动取消，或者某种原因报错退出的任务进入「失败」状态。任务失败时会自动清理本地临时文件

包含三个控制按钮：

- 下载选中日志：下载被勾选组件的日志(只有已完成的才能被勾选)
- Cancel：取消所有正在运行的任务，只能在有运行中的任务时才能点击
- Retry：重试所有失败的任务，只有在有失败任务且无正在运行的任务时才能点击

### 3. 日志搜索结果

日志搜索结果展示各个节点的日志信息。

主要包括以下 4 项信息：

- 时间：日志产生的时间，时区与前端用户所处时区相同
- 日志等级：日志 WARN、ERROR 等日志等级
- 组件类型：显示组件名称，目前主要有 TiDB、TiKV、PD
- 日志：每条日志记录的正文部分，不包含日志的时间和日志等级。过长的日志，会自动截断，鼠标悬停可以查看完整的日志，完整日志最长显示 512 个字符。如下图：

The screenshot shows a log search interface with a dark-themed card. On the left, there's a sidebar labeled "日志" (Logs). The main area displays a log entry from a PD component:

```
[base_client.go:170] ["[pd] failed to get cluster id"]
[url=http://10.1.20.41:2379]
[error="error:rpc error: code =
Unknown desc = server not started
target:10.1.20.41:2379
status:READY"]
```

Below the card, a partial log entry is visible:

```
[base_client.go:170] ["[pd] failed to get cluster id"] [url=http://10....]
```

### 4. 日志导出功能

搜索结果可以在右边栏中看到日志下载功能，用户可以选择要下载的组件日志。

点击「下载选中日志」按钮，自动下载日志压缩包到本地，供用户分析查看，如下图：

## 搜索进度

3 成功

[下载选中日志](#)

[Cancel](#)

[Retry](#)

- ▼   TiDB 1 成功
  -  10.1.20.41:4000
- ▼   TiKV 1 成功
  -  10.1.20.41:20160
- ▼   PD 1 成功
  -  10.1.20.41:2379

- 当选择一个组件时会返回 zip 文件，解压得到 log 文本文件
- 当选择多个组件时会返回 tar 文件，解压得到多个 zip 文件，解压 zip 得到 log 文本文件

## 2.5 分析组件 CPU 消耗情况

上一节介绍日志搜索功能。

本节将重点对 TiDB、TiKV、PD 节点在不重启的情况下进行内部性能数据分析。

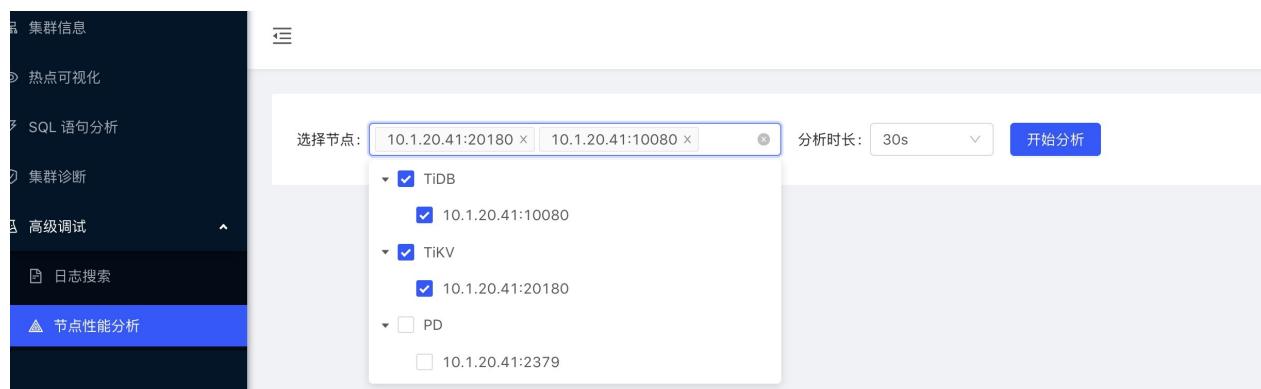
收集的性能数据可显示为火焰图或 go profile 有向无环图，直观展现各节点在性能收集时间段内执行的各种内部操作及比例，快速了解该节点 CPU 资源消耗的主要方向。

### 1. 开始性能分析

登录 Dashboard 后，可在左侧功能导航处点击「高级调试 → 节点性能分析」进入性能分析页面。

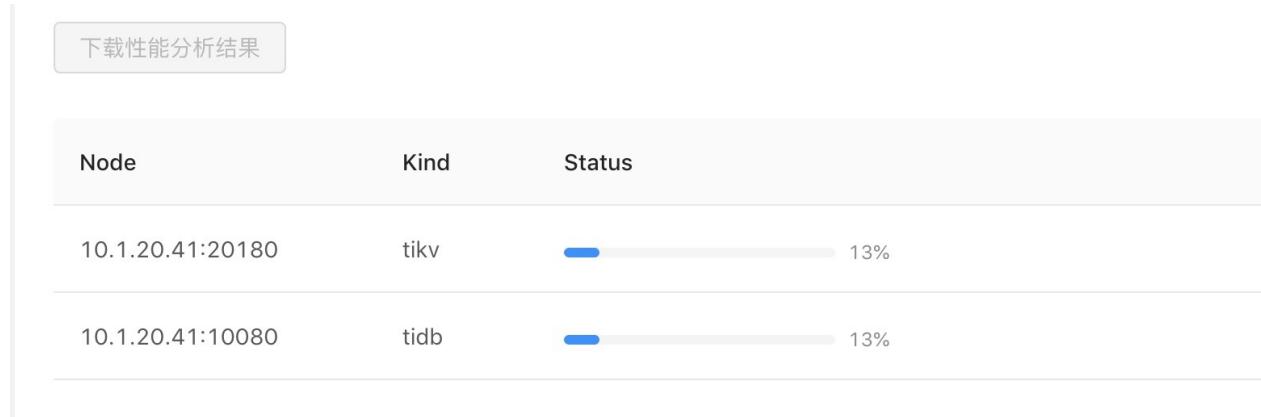
选择一个或多个需要进行性能分析的节点，并选择性能分析时长(默认为 30 秒，最多 120 秒)，点击「开始分析」，即可开始性能分析。

如下图：



### 2. 查看性能分析状态

开始性能分析后，页面将以 1 秒为周期更新显示性能分析的进度，如下图：



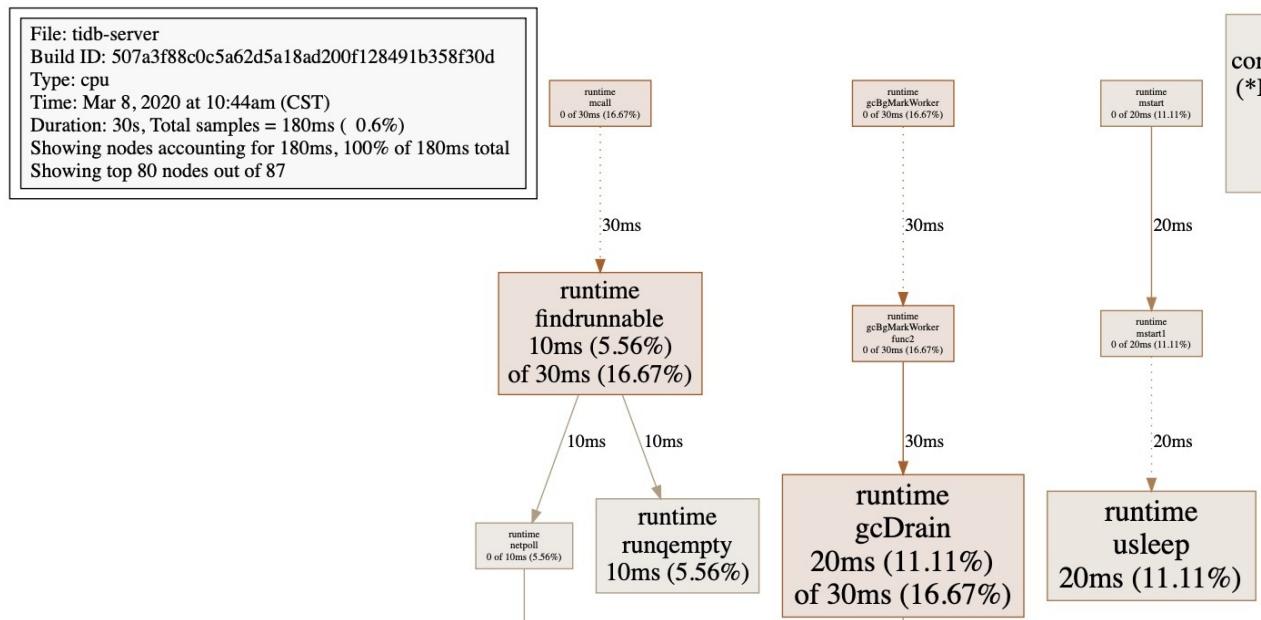
注意：

Dashboard 所在 PD 节点上需安装有 go、go pprof、GraphViz，否则无法对 TiDB 和 PD 节点进行性能分析，TiKV 的性能分析没有依赖要求。

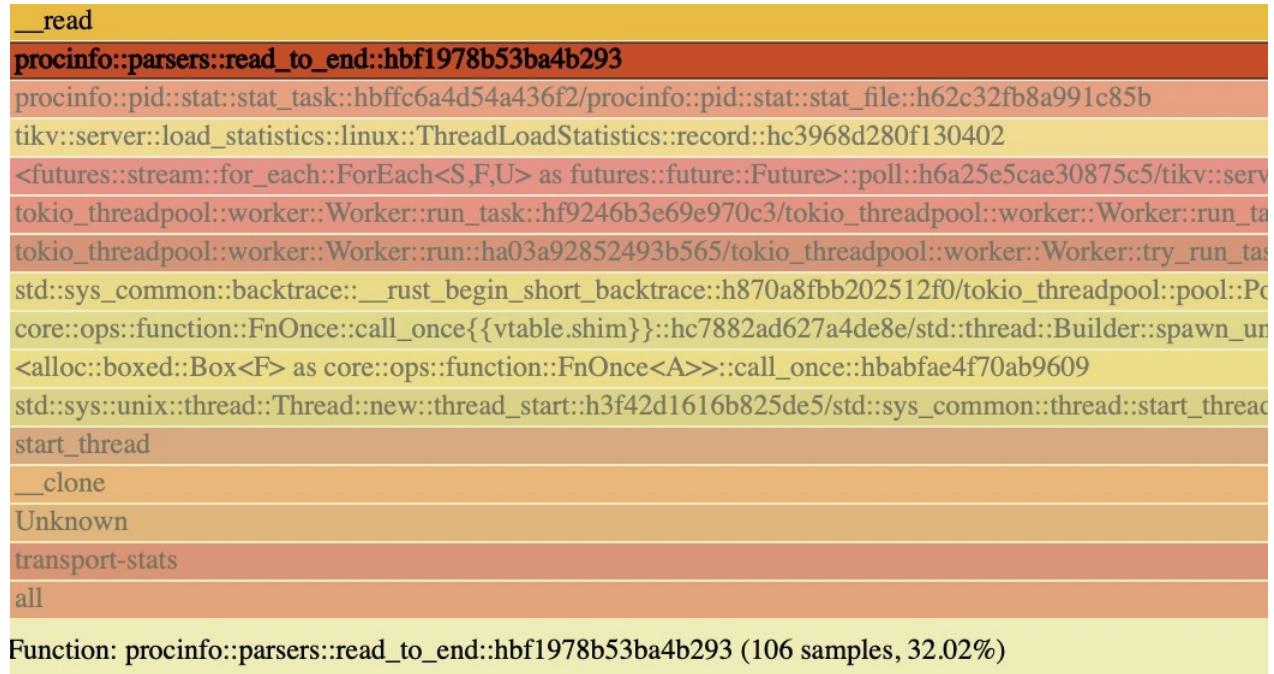
### 3. 下载性能分析结果

所有节点的性能分析完成后，点击「下载性能分析结果」按钮，打包下载性能分析成功节点的分析结果，如下 TiDB、TiKV 图：

- TiDB Profile 图可以清楚的分析每个函数执行的时间



- TiKV 火焰图可以分析出 CPU 资源消耗情况



综上所述，TiDB 4.0 实现了对各个组件的 CPU 性能分析功能，帮助 TiDB 用户直观的了解组件性能情况，及时准备优化方案。

## 第 3 章 诊断系统表

本章主要介绍 TiDB 4.0 引入的新特性——SQL 诊断功能。SQL 诊断主要用于提升 TiDB 问题定位效率。在 4.0 版本之前，不同的诊断信息需要使用不同工具获取。这种异构的信息获取方式既费时，又离散。新的 SQL 诊断对这些离散的信息进行了整体设计，它将系统的各种维度信息以系统表的方式向上层暴露为一致的接口，并在此基础上进行监控汇总与自动诊断，方便了用户对于集群信息的查询。

本章 3.1 节介绍了 SQL 诊断的集群信息表。TiDB 4.0 诊断功能为原先离散的各节点实例信息提供了一致的获取方式，它将整个集群的集群拓扑、硬件信息、软件信息、内核参数、监控、系统信息、慢查询、Statements、日志完全打通，汇聚成单个的集群信息表，让用户能够统一使用 SQL 查询这些信息。

本章 3.2 节和 3.4 节介绍了 SQL 诊断的集群监控表。TiDB 4.0 诊断系统添加了集群监控系统表，所有表都在 `metrics_schema` 中，可以通过 SQL 的方式查询监控。相对于原先的可视化监控，SQL 查询监控的好处在于可以对整个集群的所有监控进行关联查询，并对比不同时间段的结果，迅速找出性能瓶颈。此外，由于 TiDB 集群的监控指标数量较大，SQL 诊断还提供了监控汇总表，从而让用户能以一种更加便捷的方式从众多监控中找出异常的监控项。

本章 3.3 节介绍了 SQL 诊断的自动诊断功能。尽管用户可以通过手动执行 SQL 查询集群信息表和集群监控表与汇总表的方式发现集群问题，但自动挡总是更香的，所以 SQL 诊断在已有的基础信息表基础上，提供了诊断相关的系统表来自动执行诊断。使用 SQL `select * from inspection_result` 会触发所有的诊断规则对系统进行诊断，并会在结果集中展示系统中的故障或风险。

本章 3.5 节和 3.6 节分别介绍了比较重要的两个系统表 `processlist` 和 `slow_query` 以及它们所对应的在 SQL 诊断中新加入的集群表 `cluster_processlist` 和 `cluster_slow_query`。`cluster_processlist` 可以用来查看当前 TiDB 集群的所有会话信息，`cluster_slow_query` 可以查询集群所有的慢日志。

本章 3.7 节介绍了 `Statement Summary` 系统表。`Statement Summary` 把相似的 SQL 和执行计划汇总到一组，然后统计每一组的各项性能指标。这些 SQL 的性能指标通过系统表的形式暴露给用户，用户可以以此定位性能问题、排查原因。

## 3.1 集群信息表

### 3.1.1 背景

为了提升 TiDB 问题定位效率，TiDB 4.0 诊断功能以系统表为统一入口将各种维度的系统信息展现给用户，让用户能够以 SQL 的方式查询各种集群信息。

整个集群的信息拉通之后，信息量会变大，所以需要对信息进行进一步归类和自动化整理和判断。基于此原则，TiDB 4.0 会按照信息的类型将信息组织到不同的系统表。

### 3.1.2 集群信息表总览

TiDB 4.0 新增的集群信息表

- 集群拓扑表 `information_schema.cluster_info` 主要用于获取集群当前的拓扑信息，以及各个节点的版本、版本对应的 Git Hash、启动时间、运行时间信息
- 集群配置表 `information_schema.cluster_config` 用于获取集群当前所有节点的配置。TiDB 4.0 之前的版本必须逐个访问各个节点的 HTTP API
- 集群硬件表 `information_schema.cluster_hardware` 主要用于快速查询集群硬件信息
- 集群负载表 `information_schema.cluster_load` 主要用于查询集群不同节点的不同硬件类型的负载信息
- 集群负载表 `information_schema.cluster_systeminfo` 主要用于查询集群不同节点的内核配置信息，目前支持查询 sysctl 的信息
- 集群日志表 `information_schema.cluster_log` 主要用于集群日志查询。为了降低日志查询对集群的影响，诊断功能会将查询条件下推到各个节点。通过这个优化，日志查询对性能的影响小于等于 grep 命令

TiDB 4.0 之前的以下系统表，只能查看当前节点，TiDB 4.0 实现了对应的集群表，可以在单个 TiDB 节点上拥有整个集群的全局视图。

### 3.1.3 信息表示例

#### 1. 集群拓扑表

可以通过集群拓扑表 `information_schema.cluster_info` 查询当前服务器实例运行时间，具体示例如下：

```
mysql> select type, instance, status_address, uptime from cluster_info;
+-----+-----+-----+-----+
| type | instance      | status_address | uptime      |
+-----+-----+-----+-----+
tidb	127.0.0.1:4000	127.0.0.1:10080	11m6.204302s
tidb	127.0.0.1:4001	127.0.0.1:10081	11m6.204306s
tidb	127.0.0.1:4002	127.0.0.1:10082	11m6.204307s
pd	127.0.0.1:2380	127.0.0.1:2380	11m16.204308s
pd	127.0.0.1:2381	127.0.0.1:2381	11m16.20431s
pd	127.0.0.1:2382	127.0.0.1:2382	11m16.204311s
tikv	127.0.0.1:20161	127.0.0.1:20181	11m11.204312s
tikv	127.0.0.1:20162	127.0.0.1:20182	11m11.204313s
tikv	127.0.0.1:20160	127.0.0.1:20180	11m11.204314s
tikv	127.0.0.1:20163	127.0.0.1:20183	11m11.204315s
+-----+-----+-----+-----+
10 rows in set (0.01 sec)
```

字段解释：

- TYPE：节点类型，目前节点的类型为 pd/tikv/tidb，节点类型始终为小写
- INSTANCE：实例地址，始终为 IP:PORT 格式的字符串
- STATUS\_ADDRESS：HTTP API 服务地址，部分 tikv-ctl / pd-ctl / tidb-ctl 使用到 HTTP API 的命令会使用这个地址，用户也可以通过这个地址获取一些额外的集群信息，具体 HTTP API 参考官方文档

- VERSION：对应节点的语义版本号，TiDB 版本为了兼容 MySQL 的版本号，以 \${mysql-version}-\${tidb-version} 方式展示
- GIT\_HASH：对应节点版本编译时的 git commit hash，主要用于识别两个节点是否是绝对一致的版本
- START\_TIME：对应节点的启动时间
- UPTIME：对应节点已经运行的时间

## 2. 集群配置表

集群配置表 `information_schema.cluster_config` 用于获取集群当前所有节点的配置，TiDB 4.0 之前的版本中，用户必须逐个访问各个节点的 HTTP API 来获取这些信息。通过集群配置表可以查看当前集群任意实例当前生效配置。比如查询 TiKV 与 Coprocessor 相关的配置：

```
mysql> select * from cluster_config where type='tikv' and `key`='coprocessor.batch-split-limit';
+-----+-----+-----+
| TYPE | INSTANCE | KEY | VALUE |
+-----+-----+-----+
tikv	127.0.0.1:20163	coprocessor.batch-split-limit	10
tikv	127.0.0.1:20161	coprocessor.batch-split-limit	10
tikv	127.0.0.1:20162	coprocessor.batch-split-limit	10
tikv	127.0.0.1:20160	coprocessor.batch-split-limit	10
+-----+-----+-----+
4 rows in set (2.98 sec)
```

类似的，可以通过这张表来查询所有 TiDB 的配置是否一致，比如以下 SQL 结果表示

`log.file.filename / port / status.status-port` 的配置在各个实例不一致：

```
mysql> select `key`,count(distinct value) as c from cluster_config where type='tidb' group by `key` having c > 1;
+-----+---+
| key | c |
+-----+---+
log.file.filename	3
port	3
status.status-port	3
+-----+---+
3 rows in set (0.01 sec)
```

字段解释：

- TYPE：对应于节点信息表 `information_schema.cluster_info` 中的 TYPE 字段，可取值为 tidb/pd/tikv，且均为小写
- INSTANCE：对应于节点信息表 `information_schema.cluster_info` 中的 STATUS\_ADDRESS 字段
- KEY：配置项名
- VALUE：配置项值

## 3. 集群硬件表

集群硬件表 `information_schema.cluster_hardware` 主要用于快速查询集群硬件信息。可以通过此表查询集群 CPU、内存、网卡、磁盘信息。下面以查询集群各个节点的逻辑处理器数量为例：

```
mysql> select type, instance, name, value from cluster.hardware where name='cpu-logical-cores';
+-----+-----+-----+
| type | instance | name | value |
+-----+-----+-----+
tidb	127.0.0.1:10080	cpu-logical-cores	8
tidb	127.0.0.1:10081	cpu-logical-cores	8
tidb	127.0.0.1:10082	cpu-logical-cores	8
pd	127.0.0.1:2380	cpu-logical-cores	8
pd	127.0.0.1:2381	cpu-logical-cores	8
pd	127.0.0.1:2382	cpu-logical-cores	8
tikv	127.0.0.1:20160	cpu-logical-cores	8
tikv	127.0.0.1:20163	cpu-logical-cores	8
tikv	127.0.0.1:20161	cpu-logical-cores	8
tikv	127.0.0.1:20162	cpu-logical-cores	8
+-----+-----+-----+
10 rows in set (0.78 sec)
```

字段解释：

- TYPE：对应于节点信息表 `information_schema.cluster_info` 中的 TYPE 字段，可取值为 tidb/pd/tikv，且均为小写
- INSTANCE：对应于节点信息表 `information_schema.cluster_info` 中的 STATUS\_ADDRESS 字段
- DEVICE\_TYPE：硬件类型，目前可以查询的硬件类型有 cpu/memory/disk/net
- DEVICE\_NAME：硬件名，对于不同的 DEVICE\_TYPE，取值不同
  - cpu：硬件名为 cpu
  - disk：磁盘名
  - net：NIC 名
  - memory：硬件名为 memory
- NAME：硬件不同的信息名，比如 cpu 有 `cpu-logical-cores / cpu-physical-cores`，可以通过 `select name from cluster.hardware where device_type='cpu' group by name` 来查询不同硬件类型支持的 NAME
- VALUE：对应硬件信息的值，比如磁盘容量，CPU 核数

## 4. 集群负载表

集群负载表 `information_schema.cluster_load` 主要用于查询集群不同节点的不同硬件类型的当前负载信息。比如以下 SQL 可以查询所有节点最近一分钟的 CPU 平均负载：

```
mysql> select type, instance, name, value from cluster.load where device_type='cpu' and device_name='cpu' and name='load1';
+-----+-----+-----+
| type | instance | name | value |
+-----+-----+-----+
tidb	127.0.0.1:10080	load1	3.10
tidb	127.0.0.1:10081	load1	3.10
tidb	127.0.0.1:10082	load1	3.10
pd	127.0.0.1:2380	load1	3.10
pd	127.0.0.1:2381	load1	3.10
pd	127.0.0.1:2382	load1	3.10
tikv	127.0.0.1:20163	load1	3.1015625
tikv	127.0.0.1:20161	load1	3.1015625
tikv	127.0.0.1:20162	load1	3.1015625
tikv	127.0.0.1:20160	load1	3.1015625
+-----+-----+-----+
10 rows in set (0.55 sec)
```

字段解释：

- TYPE：对应于节点信息表 `information_schema.cluster_info` 中的 TYPE 字段，可取值为 tidb/pd/tikv，且均为小写
- INSTANCE：对应于节点信息表 `information_schema.cluster_info` 中的 STATUS\_ADDRESS 字段
- DEVICE\_TYPE：硬件类型，目前可以查询的硬件类型有 cpu/memory/disk/net
- DEVICE\_NAME：硬件名，对于不同的 DEVICE\_TYPE，取值不同
  - cpu：硬件名为 cpu
  - disk：磁盘名

- net : NIC 名
- memory : 硬件名为 memory
- NAME : 不同负载类型，比如 cpu 有 load1/load5/load15 分别表示 CPU 在 1min/5min/15min 中的平均负载，可以通过  
select name from cluster\_load where device\_type='cpu' group by name 来查询不同硬件类型支持的 NAME
- VALUE : 对应硬件负载的值，比如 CPU 的 1min/5min/15min 平均负载

## 5. 集群内核配置

集群内核配置表 `information_schema.cluster_systeminfo` 主要用于查询集群不同节点的内核配置信息，目前支持查询 sysctl 的信息。下面以查询 TiDB 实例所在机器内核 fd 相关配置为例：

```
mysql> select type, instance, name, value from cluster_systeminfo where type='tidb' and name like '%fd%';
+-----+-----+-----+-----+
| type | instance | name | value |
+-----+-----+-----+-----+
tidb	127.0.0.1:10080	net.inet6.ip6.maxifdefrouters	16
tidb	127.0.0.1:10080	net.necp.client_fd_count	89
tidb	127.0.0.1:10080	net.necp.observer_fd_count	0
tidb	127.0.0.1:10081	net.inet6.ip6.maxifdefrouters	16
tidb	127.0.0.1:10081	net.necp.client_fd_count	89
tidb	127.0.0.1:10081	net.necp.observer_fd_count	0
tidb	127.0.0.1:10082	net.inet6.ip6.maxifdefrouters	16
tidb	127.0.0.1:10082	net.necp.client_fd_count	89
tidb	127.0.0.1:10082	net.necp.observer_fd_count	0
+-----+-----+-----+-----+
9 rows in set (0.04 sec)
```

字段解释：

- TYPE：对应于节点信息表 `information_schema.cluster_info` 中的 TYPE 字段，可取值为 tidb/pd/tikv，且均为小写
- INSTANCE：对应于节点信息表 `information_schema.cluster_info` 中的 STATUS\_ADDRESS 字段
- SYSTEM\_TYPE：系统类型，目前可以查询的系统类型有 system
- SYSTEM\_NAME：目前可以查询的 SYSTEM\_NAME 为 sysctl
- NAME：sysctl 对应的配置名
- VALUE：sysctl 对应配置项的值

## 6. 集群日志表

集群日志表 `information_schema.cluster_log` 表是 TiDB 引入的一张非常重要的系统内存表，主要解决集群日志查询问题。它的实现非常轻量，不需要依赖外部组件，也不会有中央节点保存全局日志，通过将查询条件下推到各个节点，我们降低了日志查询对集群的影响，使其对性能影响小于 grep 命令。下面以通过集群日志表查询集群的 warning 日志为例：

```

mysql> select * from cluster_log where level='warn'\G
***** 1. row ****
TIME: 2020/03/08 12:17:41.329
TYPE: pd
INSTANCE: 127.0.0.1:2382
LEVEL: WARN
MESSAGE: [grpclog.go:60] ["transport: http2Server.HandleStreams failed to read frame: read tcp 127.0.0.1:2382->127.0.0.1:57030: use of closed network connection"]
***** 2. row ****
TIME: 2020/03/08 12:17:41.338
TYPE: pd
INSTANCE: 127.0.0.1:2382
LEVEL: WARN
MESSAGE: [grpclog.go:60] ["transport: http2Server.HandleStreams failed to read frame: read tcp 127.0.0.1:2382->127.0.0.1:57029: use of closed network connection"]
***** 3. row ****
TIME: 2020/03/08 12:17:41.361
TYPE: pd
INSTANCE: 127.0.0.1:2382
LEVEL: WARN
MESSAGE: [grpclog.go:60] ["transport: http2Server.HandleStreams failed to read frame: read tcp 127.0.0.1:2382->127.0.0.1:57031: use of closed network connection"]
3 rows in set (0.01 sec)

```

字段解释：

- TIME：日志打印时间
- TYPE：对应于节点信息表 `information_schema.cluster_info` 中的 TYPE 字段，可取值为 `tidb/pd/tikv`，且均为小写
- INSTANCE：对应于节点信息表 `information_schema.cluster_info` 中的 INSTANCE 字段
- LEVEL：日志级别
- MESSAGE：日志内容

注意事项：日志表的所有字段都会下推到对应节点执行，所以为了降低使用集群日志表的开销，需尽可能地指定更多的条件，比如 `select from cluter_log where instance='tikv-1'` 只会在 `tikv-1` 执行日志搜索。`message` 字段支持 `like` 和 `regexp` 正则表达式，对应的 `pattern` 会编译为 `regexp`，同时指定多个 `message` 条件，相当于 `grep` 命令的 `pipeline` 形式，例如：  
`select from cluster_log where message like 'coprocessor%' and message regexp '.slow.'` 相当于在集群所有节点执行 `grep 'coprocessor' xxx.log | grep -E '.slow.'`。

在TiDB 4.0 之前，要获取集群的日志，需要逐个登录各个节点汇总日志。TiDB 4.0 有了集群日志表后，可以更高效地提供一个全局时间有序的日志搜索结果。这为全链路事件跟踪提供了便利的手段。比如按照某一个 region id 搜索日志，可以查询该 region 生命周期的所有日志。类似的，通过慢日志的 txn id 搜索全链路日志，可以查询该事务在各个节点扫描的 key 数量以及流量等信息。

## 3.2 监控表

前一章节介绍了 TiDB 4.0 新增诊断功能中的集群信息表。本章节主要介绍诊断功能中的监控表。

TiDB 4.0 诊断系统添加了集群监控系统表，所有表都在 `metrics_schema` 中，可以通过 SQL 的方式查询。通过 SQL 查询监控的好处在于可以对整个集群的所有监控进行关联查询，并对比不同时间段的结果，迅速找出性能瓶颈。

### 3.2.1 监控表示例

`tidb_query_duration` 表用来查询 TiDB query 执行的百分位时间，如 P999，P99，P90 的查询耗时，单位是秒。其表结构如下：

| 字段名      | 类型              | 字段解释                                  |
|----------|-----------------|---------------------------------------|
| TIME     | unsigned        | query 执行的具体时间                         |
| INSTANCE | varchar(512)    | 运行 query 的 TiDB 实例地址，以 IP:PORT 格式组织   |
| SQL_TYPE | varchar(512)    | query 的具体类型，比如 Select , internal      |
| QUANTILE | double unsigned | query 执行的时间百分位                        |
| VALUE    | double unsigned | 与 QUANTILE 字段对应执行时间百分位的查询耗时，如上所述，单位为秒 |

下面 SQL 查询当前时间的 P90 的 TiDB Query 耗时。可以看出，`Select` 类似的 Query 的 P90 耗时是 0.0384 秒，`internal` 类型的 P90 耗时是 0.00327。`instance` 字段是 TiDB 示例的地址。

```
metrics_schema> select * from tidb_query_duration where value is not null and time=now() and quantile=0.90;
+-----+-----+-----+-----+
| time      | instance      | sql_type | quantile | value      |
+-----+-----+-----+-----+
| 2020-03-08 13:34:40 | 172.16.5.40:10089 | Select    | 0.9       | 0.0384     |
| 2020-03-08 13:34:40 | 172.16.5.40:10089 | internal  | 0.9       | 0.00327692307692 |
+-----+-----+-----+-----+
```

### 3.2.2 监控表列表

由于目前监控表非常多，本小节不会完全列举所有监控表。系统表 `information_schema.metrics_tables` 存储所有监控系统表的元数据信息，所有的监控表可以通过 SQL `select * from information_schema.metrics_tables` 查询。

系统表表结构如下所示：

| 字段名        | 类型              | 字段解释   |
|------------|-----------------|--|
| TABLE_NAME | varchar(64)     | 对于 metrics_schema 中的表名   |
| PROMQL     | varchar(64)     | 监控表的主要原理是将 SQL 映射成 PromQL，并将 prometheus 结果转换成 SQL 查询结果，这个字段是 PromQL 的表达式模板，获取监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式 |
| LABELS     | varchar(64)     | 监控定义的 label，每一个 label 会对应监控表中的一列，SQL 中如果包含对应列的过滤，对应的 PromQL 也会改变   |
| QUANTILE   | double unsigned | 百分位，对于直方图的监控数据，指定一个默认百分位，如果值为 0，表示该监控表对应的监控不是直方图   |
| COMMENT    | varchar(256)    | 对该监控表的解释   |

### 3.2.3 监控表实现方式

上一小节描述的表结构中有一列 `PROMQL` , TiDB 会根据 SQL 生成一条 `PromQL` 的查询 , 然后把查询请求发给 prometheus 查询相应的监控信息。

通过以下 SQL 的执行计划 , 可以发现在 `MemTableScan` 中 , 有一个 `PromQL` , 其中 `start_time` 和 `end_time` 分别表示查询监控的时间范围的起止 , `step` 表示查询的分辨率步长 , 默认值是 1 分钟。这几个参数和 [prometheus 的 range query HTTP API](#) 的参数是一样的。

```
metrics_schema> desc select * from tidb_query_duration where value is not null and time=now() and quantile=0.90;
+-----+-----+-----+
| id | estRows | task | operator info
|-----+-----+-----+
| Selection_5 | 8000.00 | root | not(isnull(Column#5))
|-----+
|   | MemTableScan_6 | 10000.00 | root | PromQL:histogram_quantile(0.9, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[60s])) by (le,sql_type,instance)), start_time:2020-03-08 13:13:15, end_time:2020-03-08 13:13:15, step:1m 0s |
|-----+
|-----+
```

如果 SQL 的 `where` 中没有 `time` 条件 , 默认会查询最近 10 分钟的监控数据。

### 3.2.4 监控表 session 变量

和监控表查询相关的 2 个 `session` 变量 , 可以通过修改 `session` 的变量来调整监控查询的默认行为。相关参数如下 :

- `tidb_metric_query_step` : 查询的分辨率步长。从 prometheus 的 `query_range` 数据时需要指定 `start` , `end` , `step` , 其中 `step` 会使用该变量的值
- `tidb_metric_query_range_duration` : 生成 PromQL 语句时 , 会将 PromQL 中的 `$RANGE_DURATION` 替换成该变量的值 , 默认值是 60 秒

补充知识点 :

`range query` 是 prometheus 非常常见的一种 `query` , 以下是它的参数 :

- `query=: PromQL 表达式`
- `start=: 时间范围的开始`
- `end=: 时间范围的结束`
- `step=: 查询解析度 (query resolution)`
- `timeout=: 执行超时 , 这个参数是可选的`

prometheus 在对 PromQL 表达式求值的逻辑是这样的 :

- 对于 `[start, end]` 时间区间 , 从 `start` 开始 , 以 `step` 为长度 , 把时间区间分成若干段
- 对每个段进行求值 举例 : `start=10 , end=20 , step=2` , 那么就会有 `ts=10 , ts=12 , ts=14 , ts=16 , ts=18 , ts=206` 共 6 段 , 然后为这 6 个段进行求值。

例如 , 将步长调整为 60

```
set @@session.tidb_metric_query_step=60
```

目前 TiDB 会从 PD 中查询 `prometheus` 的地址 , 然后将查询请求发给 `prometheus` , 后续 PD 会考虑内置监控组件 , 就不用再部署 `prometheus` 组件了。下面例子按照 `instance` 和 `sql_type` 聚合后 , 查询 `['2020-03-08 13:23:00' , '2020-03-08 13:33:00']` 范围内的 P99 耗时的 avg, max, min 值。

```
metrics_schema> select instance,sql_type, avg(value),max(value),min(value) from tidb_query_duration where time >= '2020-03-08 13:23:00' and time < '2020-03-08 13:33:00' and value is not null and quantile=0.99 group by instance,sql_type;
```

| instance          | sql_type | avg(value)       | max(value)       | min(value)         |
|-------------------|----------|------------------|------------------|--------------------|
| 172.16.5.40:10089 | Select   | 0.00800917072138 | 0.00824108821892 | 0.00790462559176   |
| 172.16.5.40:10089 | internal | 0.012384         | 0.01554          | 0.0062             |
| 172.16.5.40:10089 | Insert   | 0.00687276884265 | 0.0069763539823  | 0.00670463917526   |
| 172.16.5.40:10089 | general  | 0.00092395833333 | 0.00133333333333 | 0.0006666666666667 |

## 3.3 诊断结果表

### 1. 背景

在 3.1 和 3.2 中介绍了 TiDB 4.0 引入的集群信息表和集群监控表，也通过 SQL 演示了如何通过查询这些表来发现集群问题，比如通过 `information_schema.cluster_config` 发现集群不同节点配置不一致，通过 `information_schema.cluster_info` 发现是否存在组件版本不一样。

手动执行固定模式的 SQL 排查集群问题是低效的，所以 TiDB 4.0 利用已有的基础信息表提供了诊断相关的系统表来自动执行诊断：

- `information_schema.inspection_result`
- `information_schema.inspection_summary`

诊断功能可以帮助用户快速发现问题，减少用户的重复性手动工作。诊断功能使用 SQL `select * from information_schema.inspection_result` 触发内部的诊断。

诊断模块内部包含一系列的规则，这些规则会通过查询已有的监控表和集群信息表，对结果和预先设定的阈值进行对比，如果结果超过阈值或低于阈值将生成 `warning / critical` 的结果，并在 `details` 列中提供进一步信息。

### 2. 诊断结果表

诊断结果表 `information_schema.inspection_result` 的表结构如下：

```
mysql> desc inspection_result;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
RULE	varchar(64)	YES		NULL	
ITEM	varchar(64)	YES		NULL	
TYPE	varchar(64)	YES		NULL	
INSTANCE	varchar(64)	YES		NULL	
VALUE	varchar(64)	YES		NULL	
REFERENCE	varchar(64)	YES		NULL	
SEVERITY	varchar(64)	YES		NULL	
DETAILS	varchar(256)	YES		NULL	
+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

字段解释：

- RULE：诊断规则，由于规则在持续添加，以下列表可能已经过时，最新的规则列表可以通过 `select * from inspection_rules where type='inspection'` 查询
  - config：配置一致性检测，如果同一个配置在不同节点配置值不同，会生成 warning 级别的诊断结果
  - version：版本一致性检测，如果同一类型的节点版本 githash 不同，会生成 critical 级别的诊断结果
  - current-load：如果当前系统负载太高，会生成对应的 warning 诊断结果
  - critical-error：系统各个模块定义了严重的错误，如果某一个严重错误在对应时间段内超过阈值，会生成 warning 诊断结果
  - threshold-check：诊断系统会对大量指标进行阈值判断，如果超过阈值会生成对应的诊断信息
- ITEM：每一个规则会对不同的项进行诊断，这个用来表示对应规则下面的具体诊断项。
- TYPE：诊断的实例类型，可能是 tidb/tikv/pd
- INSTANCE：诊断的具体实例
- VALUE：针对这个诊断项得到的值
- REFERENCE：针对这个诊断项的参考值（阈值），如果 VALUE 和阈值差距比较大，就会产生对应的结果
- SEVERITY：严重程度，warning/critical

- DETAILS：诊断的详细信息，可能包含进一步调查的 SQL 或文档链接

查询已有的诊断规则：

```
mysql> select * from inspection_rules where type='inspection';
+-----+-----+-----+
| NAME | TYPE | COMMENT |
+-----+-----+-----+
config	inspection	
version	inspection	
current-load	inspection	
critical-error	inspection	
threshold-check	inspection	
+-----+-----+-----+
5 rows in set (0.00 sec)
```

### 3. 诊断汇总表

诊断结果需要基于确定性的阈值进行判断，比如当前 Coprocessor 配置的线程池为 8，如果 Coprocessor 的 CPU 使用率达到了 75%，可以确定这里有风险，或者可能提前成为瓶颈。但是部分监控会因为用户的 workload 不同而差异较大，所以难以定义确定的阈值。这部分场景问题排查也非常重要，所以新增了 `information_schema.inspection_summary` 对特定链路或模块的监控汇总，是用户可以根据整个模块或链路的上下文来排查定位问题。

诊断结果表 `information_schema.inspection_summary` 的表结构如下：

```
mysql> desc inspection_summary;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
RULE	varchar(64)	YES		NULL	
INSTANCE	varchar(64)	YES		NULL	
METRICS_NAME	varchar(64)	YES		NULL	
LABEL	varchar(64)	YES		NULL	
QUANTILE	double unsigned	YES		NULL	
AVG_VALUE	double(22,6) unsigned	YES		NULL	
MIN_VALUE	double(22,6) unsigned	YES		NULL	
MAX_VALUE	double(22,6) unsigned	YES		NULL	
+-----+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

字段解释：

- RULE：汇总规则，由于规则在持续添加，以下列表可能已经过时，最新的规则列表可以通过 `select * from inspection_rules where type='summary'` 查询
- INSTANCE：监控的具体实例
- METRIC\_NAME：监控表
- QUANTILE：对于包含 QUANTILE 的监控表有效，可以通过谓词下推指定多个百分位，比如 `select * from inspection_summary where rule='ddl' and quantile in (0.80, 0.90, 0.99, 0.999)` 来汇总 ddl 相关监控，查询百分位为 80/90/99/999 的结果。AVG\_VALUE/MIN\_VALUE/MAX\_VALUE 分别表示聚合的平均、最小、最大值。

注意事项：

由于汇总所有结果有一定开销，所以 `information_summary` 中的规则是惰性触发的，即通过 SQL 的谓词中显示指定的 rule 才会运行。比如 `select * from inspection_summary` 语句会得到一个空的结果集。`select * from inspection_summary where rule in ('read-link', 'ddl')` 会汇总读链路和 DDL 相关的监控。

### 4. 诊断时间范围

诊断结果表和诊断监控汇总表都可以通过 hint 的方式指定诊断的时间范围，比如 `select /*+ time_range('2020-03-07 12:00:00', '2020-03-07 13:00:00') */ * from inspection_summary` 对 `2020-03-07 12:00:00 - 2020-03-07 13:00:00` 时间段的监控汇总。和监控汇总表一样，通过两个不同时间段的数据进行对比，快速发现差异较大的监控项。以下为一个例子：

```
mysql> SELECT
    t1.avg_value / t2.avg_value AS ratio,
    t1.*,
    t2.*
FROM
(
    SELECT
        /*+ time_range("2020-01-16 16:00:54.933", "2020-01-16 16:10:54.933") */ *
        FROM inspection_summary WHERE rule='read-link'
) t1
JOIN
(
    SELECT
        /*+ time_range("2020-01-16 16:10:54.933", "2020-01-16 16:20:54.933") */ *
        FROM inspection_summary WHERE rule='read-link'
) t2
ON t1.metrics_name = t2.metrics_name
and t1.instance = t2.instance
and t1.label = t2.label
ORDER BY
ratio DESC;
```

## 3.4 监控汇总表

由于 TiDB 集群的监控指标数量较大，需要提供便捷的方式从众多监控中找出异常的监控项，TiDB 4.0 提供了监控汇总表，监控汇总表 `information_schema.metrics_summary` 和 `information_schema.metrics_summary_by_label` 用于汇总所有监控数据，来提升用户对各个监控指标进行排查的效率。两者不同在于 `information_schema.metrics_summary_by_label` 会对不同的 `label` 使用区分统计。

### 1. 查询示例

以查询 `['2020-03-08 13:23:00', '2020-03-08 13:33:00']` 时间范围内 TiDB 集群中平均耗时最高的 3 组监控项为例。通过直接查询 `information_schema.metrics_summary` 表，并通过 `/*+ time_range() */` 这个 hint 来指定时间范围来构造以下 SQL：

```
mysql> select /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
       from information_schema.`METRICS_SUMMARY`
      where metrics_name like 'tidb%duration'
        and avg_value > 0
        and quantile = 0.99
     order by avg_value desc
    limit 3\G
*****
[ 1. row ]*****
METRICS_NAME | tidb_get_token_duration
QUANTILE     | 0.99
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
COMMENT      | The quantile of Duration (us) for getting token, it should be small until concurrency limit is reached(second)
*****
[ 2. row ]*****
METRICS_NAME | tidb_query_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.269079
AVG_VALUE    | 0.007272
MIN_VALUE    | 0.000667
MAX_VALUE    | 0.01554
COMMENT      | The quantile of TiDB query durations(second)
*****
[ 3. row ]*****
METRICS_NAME | tidb_kv_request_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.170232
AVG_VALUE    | 0.004601
MIN_VALUE    | 0.000975
MAX_VALUE    | 0.013
COMMENT      | The quantile of kv requests durations by store
```

注意：其中 `tidb_get_token_duration` 在 `COMMENT` 列中注释了值的单位是微秒 (us)

类似的，查询 `metrics_summary_by_label` 监控汇总表结果如下：

```

mysql> select /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
       from information_schema.`METRICS_SUMMARY_BY_LABEL`*
      where metrics_name like 'tidb%duration'
        and avg_value > 0
        and quantile = 0.99
      order by avg_value desc
     limit 10\G
*****
[ 1. row ]*****
INSTANCE | 172.16.5.40:10089
METRICS_NAME | tidb_get_token_duration
LABEL |
QUANTILE | 0.99
SUM_VALUE | 8.972509
AVG_VALUE | 0.996945
MIN_VALUE | 0.996515
MAX_VALUE | 0.997458
COMMENT | The quantile of Duration (us) for getting token, it should be small until concurrency limit is reached(second)
*****
[ 2. row ]*****
INSTANCE | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL | Select
QUANTILE | 0.99
SUM_VALUE | 0.072083
AVG_VALUE | 0.008009
MIN_VALUE | 0.007905
MAX_VALUE | 0.008241
COMMENT | The quantile of TiDB query durations(second)
*****
[ 3. row ]*****
INSTANCE | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL | Rollback
QUANTILE | 0.99
SUM_VALUE | 0.072083
AVG_VALUE | 0.008009
MIN_VALUE | 0.007905
MAX_VALUE | 0.008241
COMMENT | The quantile of TiDB query durations(second)

```

前文提到 `metrics_summary_by_label` 表结构相对于 `metrics_summary` 多了一列 `LABEL`，以上面查询结果的第 2, 3 行为例：分别表示 `tidb_query_duration` 的 `Select / Rollback` 类型的语句平均耗时非常高。

## 2. 推荐用法

除以上示例之外，监控汇总表可以通过两个时间段的全链路监控对比，迅速找出监控数据变化最大的模块，快速定位瓶颈，以下对比两个时间段的所有监控（其中 t1 为 baseline），并按照差别最大的监控排序：

- 时间段 t1 : ("2020-03-03 17:08:00", "2020-03-03 17:11:00")
- 时间段 t2 : ("2020-03-03 17:18:00", "2020-03-03 17:21:00")

对两个时间段的监控按照 `METRICS_NAME` 进行 join，并按照差值排序。其中 `/*+ time_range() */` 是用于指定查询时间的 hint。

1. 查询 `t1.avg_value / t2.avg_value` 差异最大的 10 个监控项

```

mysql> SELECT
    t1.avg_value / t2.avg_value AS ratio,
    t1.metrics_name,
    t1.avg_value,
    t2.avg_value,
    t2.comment
  FROM
  (
    SELECT /*+ time_range("2020-03-03 17:08:00", "2020-03-03 17:11:00")*/
    *
   FROM information_schema.metrics_summary
  ) t1
 JOIN
  (
    SELECT /*+ time_range("2020-03-03 17:18:00", "2020-03-03 17:21:00")*/
    *
   FROM information_schema.metrics_summary
  ) t2
 ON t1.metrics_name = t2.metrics_name
 ORDER BY
    ratio DESC limit 10;
+-----+-----+-----+-----+
| ratio      | metrics_name          | avg_value     | avg_value     | comment
+-----+-----+-----+-----+
| 17.6439787379 | tikv_region_average_written_bytes | 30816827.0953 | 1746591.71568 | The average rate of writing bytes to Regions per TiKV instance
| 8.88407551364 | tikv_region_average_written_keys | 108557.034612 | 12219.283193 | The average rate of written keys to Regions per TiKV instance
| 6.4105335594  | tidb_kv_write_num        | 4493.293654   | 700.923505   | The quantile of kv write times per transaction execution
| 2.99993333333 | tidb_gc_total_count      | 1.0           | 0.333341    | The total count of kv storage garbage collection time durations
| 2.66412165823 | tikv_engine_avg_seek_duration | 6569.879007 | 2466.05818  | The time consumed when executing seek operation, the unit is microsecond
| 2.66412165823 | tikv_engine_max_seek_duration | 6569.879007 | 2466.05818  | The time consumed when executing seek operation, the unit is microsecond
| 2.49994444321 | tikv_region_change         | -0.277778    | -0.111114   | The count of region change per TiKV instance
| 2.16063829787 | etcd_wal_fsync_duration   | 0.002539     | 0.001175    | The quantile time consumed of writing WAL into the persistent storage
| 2.06089264604 | node_memory_free           | 4541448192.0 | 2203631616.0 |
| 1.96749064186 | tidb_kv_write_size         | 514489.28    | 261495.159902 | The quantile of kv write size per transaction execution
+-----+-----+-----+-----+

```

查询结果表示：

- t1 时间段内的 tikv\_region\_average\_written\_bytes (region 的平均写入字节数) 比 t2 时间段高了 17.6 倍
- t1 时间段内的 tikv\_region\_average\_written\_keys (region 的平均写入 keys 数) 比 t2 时间段高了 8.8 倍
- t1 时间段内的 tidb\_kv\_write\_size (tidb 每个事务写入的 kv 大小) 比 t2 时间段高了 1.96 倍

通过以上结果可以轻易看出 t1 时间段的写入要比 t2 时间段高。

1. 反过来，查询  $t2.\text{avg\_value} / t1.\text{avg\_value}$  差异最大的 10 个监控项

```

mysql> SELECT
    t2.avg_value / t1.avg_value AS ratio,
    t1.metrics_name,
    t1.avg_value,
    t2.avg_value,
    t2.comment
  FROM
  (
    SELECT /*+ time_range("2020-03-03 17:08:00", "2020-03-03 17:11:00")*/
    *
   FROM information_schema.metrics_summary
  ) t1
 JOIN
  (
    SELECT /*+ time_range("2020-03-03 17:18:00", "2020-03-03 17:21:00")*/
    *
   FROM information_schema.metrics_summary
  ) t2
 ON t1.metrics_name = t2.metrics_name
 ORDER BY
    ratio DESC limit 10;
+-----+-----+-----+-----+
| ratio      | metrics_name          | avg_value     | avg_value     | comment
+-----+-----+-----+-----+
| 5865.59537065 | tidb_slow_query_cop_process_total_time | 0.016333 | 95.804724 | The total time of Ti
DB slow query statistics with slow query total cop process time(second) |
| 3648.74109023 | tidb_distsql_partial_scan_key_total_num | 10865.666667 | 39646004.4394 | The total num of dis
tsql partial scan key numbers
| 267.002351165 | tidb_slow_query_cop_wait_total_time | 0.003333 | 0.890008 | The total time of Ti
DB slow query statistics with slow query total cop wait time(second) |
| 192.43267836 | tikv_cop_total_response_total_size | 2515333.66667 | 484032394.445 | |
| 192.43267836 | tikv_cop_total_response_size        | 41922.227778 | 8067206.57408 | |
| 152.780296296 | tidb_distsql_scan_key_total_num    | 5304.333333 | 810397.618317 | The total num of dis
tsql scan numbers
| 126.042290167 | tidb_distsql_execution_total_time | 0.421622 | 53.142143 | The total time of di
stsql execution(second)
105.164020657	tikv_cop_scan_details	134.450733	14139.379665	
105.164020657	tikv_cop_scan_details_total	8067.043981	848362.77991	
101.635495394	tikv_cop_total_kv_cursor_operations	1070.875	108838.91113	
+-----+-----+-----+-----+

```

上面查询结果表示：

- t2 时间段内的 tidb\_slow\_query\_cop\_process\_total\_time (tidb 慢查询中的 cop process 耗时) 比 t1 时间段高了 5865 倍
- t2 时间段内的 tidb\_distsql\_partial\_scan\_key\_total\_num (tidb 的 distsql 请求扫描key 的数量) 比 t1 时间段高了 3648 倍
- t2 时间段内的 tikv\_cop\_total\_response\_size (tikv 的 cop 请求结果的大小) 比 t1 时间段高了 192 倍
- t2 时间段内的 tikv\_cop\_scan\_details (tikv 的 cop 请求的 scan) 比 t1 时间段高了 105 倍

通过上面两个时间段对比查询可以大致了解集群在这 2 个时间段的负载情况。t2 时间段的 Cop 请求要比 t2 时间段高很多，导致 TiKV 的 Coprocessor 过载，出现了 cop task 等待，可以猜测可能是 t2 时间段出现了一些大查询，或者是查询较多的负载。

实际上，在 t1 ~ t2 整个时间段内都在跑 go-ycsb 的压测，然后在 t2 时间段跑了 20 个 tpch 的查询，所以是因为 tpch 大查询导致了很多的 cop 请求。



## 3.5 SQL 慢查询内存表

TiDB 默认会启用慢查询日志，并将执行时间超过规定阈值的 SQL 保存到日志文件。慢查询日志常用于定位慢查询语句，分析和解决 SQL 的性能问题。通过系统表 `information_schema.slow_query` 也可以查看当前 TiDB 节点的慢查询日志，其字段与慢查询日志文件内容一致。TiDB 4.0 又新增了系统表 `information_schema.cluster_slow_query`，可以用于查看全部 TiDB 节点的慢查询。

本节将首先简要介绍慢查询日志的格式和字段含义，然后针对上述两种慢查询系统表给出一些常见的查询示例。

### 慢查询日志示例及字段说明

下面是一段典型的慢查询日志：

```
# Time: 2019-08-14T09:26:59.487776265+08:00
# Txn_start_ts: 410450924122144769
# User: root@127.0.0.1
# Conn_ID: 3086
# Query_time: 1.527627037
# Parse_time: 0.000054933
# Compile_time: 0.000129729
# Process_time: 0.07 Wait_time: 0.002 Backoff_time: 0.002 Request_count: 1 Total_keys: 131073 Process_keys: 131072 Pr
ewrite_time: 0.335415029 Commit_time: 0.032175429 Get_commit_ts_time: 0.000177098 Local_latch_wait_time: 0.106869448
Write_keys: 131072 Write_size: 3538944 Prewrite_region: 1
# DB: test
# Is_internal: false
# Digest: 50a2e32d2abbd6c1764b1b7f2058d428ef2712b029282b776beb9506a365c0f1
# Stats: t:414652072816803841
# Num_cop_tasks: 1
# Cop_proc_avg: 0.07 Cop_proc_p90: 0.07 Cop_proc_max: 0.07 Cop_proc_addr: 172.16.5.87:20171
# Cop_wait_avg: 0 Cop_wait_p90: 0 Cop_wait_max: 0 Cop_wait_addr: 172.16.5.87:20171
# Mem_max: 525211
# Succ: true
# Plan_digest: e5f9d9746c756438a13c75ba3eedf601eecf555cdb7ad327d7092bdd041a83e7
# Plan: tidb_decode_plan('ZJAwCTMyXzcJMAkyMalkYXRh01RhYmxlU2Nhb182CjEJMTBfNgkxAR0AdAEY1Dp0LCByYW5nZTpblWluZiwraW5mXS
ga2VlcCBvcmRlcjpmYWxzZSwgc3RhdHM6cHNldwRvCg==')
insert into t select * from t;
```

以下逐一介绍慢查询日志中各个字段的含义。

注意：慢查询日志中所有时间相关字段的单位都是秒。

#### (1) 慢查询基础信息：

- `Time`：表示日志打印时间。
- `Query_time`：表示执行该语句花费的时间。
- `Parse_time`：表示该语句在语法解析阶段花费的时间。
- `Compile_time`：表示该语句在查询优化阶段花费的时间。
- `Digest`：表示该语句的 SQL 指纹。
- `Stats`：表示 table 使用的统计信息版本时间戳。如果时间戳显示为 `pseudo`，表示用默认假设的统计信息。
- `Txn_start_ts`：表示事务的开始时间戳，也就是事务的唯一 ID，可以用该值在 TiDB 日志中查找事务相关的其他日志。
- `Is_internal`：表示是否为 TiDB 内部的 SQL 语句。`true` 表示是 TiDB 系统内部执行的 SQL 语句，`false` 表示是由用户执行的 SQL 语句。
- `Index_ids`：表示该语句使用的索引 ID。
- `Succ`：表示该语句是否执行成功。
- `Backoff_time`：表示遇到需要重试的错误时该语句在重试前等待的时间。常见的需要重试的错误有以下几种：遇到了 lock、Region 分裂、tikv server is busy。
- `Plan_digest`：表示 plan 的指纹。
- `Plan`：表示该语句的执行计划，运行 `select tidb_decode_plan('...')` 可以解析出具体的执行计划。

- `Query` : 表示该 SQL 语句。慢日志里不会打印字段名 `Query` , 但映射到内存表后对应的字段叫 `query` 。

(2) 和事务执行相关的字段 :

- `Prewrite_time` : 表示事务两阶段提交中第一阶段 ( `prewrite` 阶段) 的耗时。
- `Commit_time` : 表示事务两阶段提交中第二阶段 ( `commit` 阶段) 的耗时。
- `Get_commit_ts_time` : 表示事务两阶段提交中第二阶段 ( `commit` 阶段) 获取 `commit` 时间戳的耗时。
- `Local_latch_wait_time` : 表示事务两阶段提交中第二阶段 ( `commit` 阶段) 启动前在 TiDB 侧等锁的耗时。
- `Write_keys` : 表示该事务向 TiKV 的 Write CF 写入 Key 的数量。
- `Write_size` : 表示事务提交时写 key 和 value 的总大小。
- `Prewrite_region` : 表示事务两阶段提交中第一阶段 ( `prewrite` 阶段) 涉及的 TiKV Region 数量。每个 Region 会触发一次远程过程调用。

(3) 和内存使用相关的字段 :

- `Memory_max` : 表示执行期间 TiDB 使用的最大内存空间 , 单位为 `byte` 。

(4) 和用户相关的字段 :

- `User` : 表示执行语句的用户名。
- `Conn_ID` : 表示用户的连接 ID , 可以用类似 `con:3` 的关键字在 TiDB 日志中查找该链接相关的其他日志。
- `DB` : 表示执行语句时使用的 database。

(5) 和 TiKV Coprocessor Task 相关的字段 :

- `Process_time` : 该 SQL 在 TiKV 上的处理时间之和。因为数据会并行发到 TiKV 执行 , 该值可能会超过 `Query_time` 。
- `Wait_time` : 表示该语句在 TiKV 上的等待时间之和。因为 TiKV 的 Coprocessor 线程数是有限的 , 当所有的 Coprocessor 线程都在工作的时候 , 请求会排队 ; 若队列中部分请求耗时很长 , 后面的请求的等待时间会增加。
- `Request_count` : 表示该语句发送的 Coprocessor 请求的数量。
- `Total_keys` : 表示 Coprocessor 扫过的 key 的数量。
- `Process_keys` : 表示 Coprocessor 处理的 key 的数量。相较于 `total_keys` , `processed_keys` 不包含 MVCC 的旧版本。如果 `processed_keys` 和 `total_keys` 相差很大 , 说明旧版本比较多。
- `Cop_proc_avg` : cop-task 的平均执行时间。
- `Cop_proc_p90` : cop-task 的 P90 分位执行时间。
- `Cop_proc_max` : cop-task 的最大执行时间。
- `Cop_proc_addr` : 执行时间最长的 cop-task 所在地址。
- `Cop_wait_avg` : cop-task 的平均等待时间。
- `Cop_wait_p90` : cop-task 的 P90 分位等待时间。
- `Cop_wait_max` : cop-task 的最大等待时间。
- `Cop_wait_addr` : 等待时间最长的 cop-task 所在地址。

## 慢查询内存表查询示例

下面通过一些示例展示如何通过 SQL 查看 TiDB 的慢查询。

### 检索当前节点 Top N 慢查询

以下 SQL 用于检索当前 TiDB 节点的 Top 2 慢查询 :

```
> select query_time, query
  from information_schema.slow_query    -- 检索当前 TiDB 节点的慢查询
  where is_internal = false           -- 排除 TiDB 内部的慢查询
  order by query_time desc
  limit 2;
+-----+-----+
| query_time | query
+-----+-----+
| 12.77583857 | select * from t_slim, t_wide where t_slim.c0=t_wide.c0; |
| 0.734982725 | select t0.c0, t1.c1 from t_slim t0, t_wide t1 where t0.c0=t1.c0; |
+-----+-----+
```

## 检索全部节点上指定用户的 Top N 慢查询

以下 SQL 会检索全部 TiDB 节点上指定用户 `test` 的 Top 2 慢查询：

```
> select query_time, query, user
  from information_schema.cluster_slow_query   -- 检索全部 TiDB 节点的慢查询
  where is_internal = false
  and user = "test"
  order by query_time desc
  limit 2;
+-----+-----+-----+
| Query_time | query                                | user |
+-----+-----+-----+
| 0.676408014 | select t0.c0, t1.c1 from t_slim t0, t_wide t1 where t0.c0=t1.c1; | test |
+-----+-----+-----+
```

## 检索同类慢查询

在得到 Top N 慢查询后，可通过 SQL 指纹继续检索同类慢查询。

```
-- 先获取 Top N 的慢查询和对应的 SQL 指纹
> select query_time, query, digest
  from information_schema.cluster_slow_query
  where is_internal = false
  order by query_time desc
  limit 1;
+-----+-----+-----+
| query_time | query          | digest
+-----+-----+-----+
| 0.302558006 | select * from t1 where a=1; | 4751cb6008fda383e22dacb601fde85425dc8f8cf669338d55d944bafb46a6fa |
+-----+-----+-----+

-- 再根据 SQL 指纹检索同类慢查询
> select query, query_time
  from information_schema.cluster_slow_query
  where digest = "4751cb6008fda383e22dacb601fde85425dc8f8cf669338d55d944bafb46a6fa";
+-----+-----+
| query          | query_time |
+-----+-----+
| select * from t1 where a=1; | 0.302558006 |
| select * from t1 where a=2; | 0.401313532 |
+-----+-----+
```

## 检索统计信息为 `pseudo` 的慢查询

如果慢查询日志中的统计信息被标记为 `pseudo`，往往说明 TiDB 表的统计信息更新不及时，需要运行 `analyze table` 手动收集统计信息。以下 SQL 可以找到这一类慢查询：

```
> select query, query_time, stats
  from information_schema.cluster_slow_query
  where is_internal = false
    and stats like '%pseudo%';
+-----+-----+-----+
| query | query_time | stats |
+-----+-----+-----+
| select * from t1 where a=1; | 0.302558006 | t1:pseudo |
| select * from t1 where a=2; | 0.401313532 | t1:pseudo |
| select * from t1 where a>2; | 0.602011247 | t1:pseudo |
| select * from t1 where a>3; | 0.50077719 | t1:pseudo |
| select * from t1 join t2; | 0.931260518 | t1:407872303825682445, t2:pseudo |
+-----+-----+-----+
```

## 查询执行计划发生变化的慢查询

由于统计信息不准，可能导致同类型 SQL 的执行计划发生意料之外的改变。用以下 SQL 可以检索到哪些慢查询具有多种不同的执行计划：

```
> select count(distinct plan_digest) as count, digest,min(query)
  from information_schema.cluster_slow_query
  group by digest
  having count>1
  limit 3\G
*****
[ 1. row ]*****
count | 2
digest | 17b4518fde82e32021877878bec2bb309619d384fc944106fcfa9c93b536e94
min(query) | SELECT DISTINCT c FROM sbtest25 WHERE id BETWEEN ? AND ? ORDER BY c [arguments: (291638, 291737)];
*****
[ 2. row ]*****
count | 2
digest | 9337865f3e2ee71c1c2e740e773b6dd85f23ad00f8fa1f11a795e62e15fc9b23
min(query) | SELECT DISTINCT c FROM sbtest22 WHERE id BETWEEN ? AND ? ORDER BY c [arguments: (215420, 215519)];
*****
[ 3. row ]*****
count | 2
digest | db705c89ca2dfc1d39d10e0f30f285cbbadec7e24da4f15af461b148d8ffb020
min(query) | SELECT DISTINCT c FROM sbtest11 WHERE id BETWEEN ? AND ? ORDER BY c [arguments: (303359, 303458)];

-- 借助 SQL 指纹进一步查询执行计划的详细信息
> select min(plan),plan_digest
  from information_schema.cluster_slow_query
  where digest='17b4518fde82e32021877878bec2bb309619d384fc944106fcfa9c93b536e94'
  group by plan_digest\G
*****
  1. row *****
min(plan): Sort_6          root 100.00131380758702      sbtest.sbtest25.c:asc
  └HashAgg_10           root 100.00131380758702      group by:sbtest.sbtest25.c, funcs:firstrow(sbtest.sbtest25.c)->sbtest.sbtest25.c
    └TableReader_15     root 100.00131380758702      data:TableRangeScan_14
      └TableScan_14     cop   100.00131380758702      table:sbtest25, range:[502791,502890], keep order:false
plan_digest: 6afbbd21f6ca6c6fdf3d3cd94f7c7a49dd93c00fcf8774646da492e50e204ee
*****
  2. row *****
min(plan): Sort_6          root 1                  sbtest.sbtest25.c:asc
  └HashAgg_12           root 1                  group by:sbtest.sbtest25.c, funcs:firstrow(sbtest.sbtest25.c)->sbtest.sbtest25.c
    └TableReader_13     root 1                  data:HashAgg_8
      └HashAgg_8         cop   1                  group by:sbtest.sbtest25.c,
        └TableScan_11     cop   1.2440069558121831  table:sbtest25, range:[472745,472844], keep order:false
se
```

## 统计各个节点的慢查询数量

以下 SQL 统计指定时段内各个 TiDB 节点上出现过的慢查询数量：

```
> select instance, count(*)
  from information_schema.cluster_slow_query
  where time >= "2020-03-06 00:00:00"
    and time < now()
  group by instance;
+-----+-----+
| instance | count(*) |
+-----+-----+
| 0.0.0.0:10081 | 124      |
| 0.0.0.0:10080 | 119771   |
+-----+-----+
```

## 检索异常时段的慢查询

假定 2020-03-10 13:24:00 至 2020-03-10 13:27:00 期间发现 QPS 降低和查询响应时间升高等问题，可以用以下 SQL 过滤出仅仅出现在异常时段的慢查询：

```
> select * from
  (select /*+ AGG_TO_COP(), HASH_AGG() */ count(*),
    min(time),
    sum(query_time) AS sum_query_time,
    sum(Process_time) AS sum_process_time,
    sum(Wait_time) AS sum_wait_time,
    sum(Commit_time),
    sum(Request_count),
    sum(process_keys),
    sum(write_keys),
    max(Cop_proc_max),
    min(query),min(prev_stmt),
    digest
  from information_schema.cluster_slow_query
  where time >= '2020-03-10 13:24:00'
    and time < '2020-03-10 13:27:00'
    andn Is_internal = false
  group by digest) AS t1
  where t1.digest not in
  (select /*+ AGG_TO_COP(), HASH_AGG() */ digest
  from information_schema.cluster_slow_query
  where time >= '2020-03-10 13:20:00' -- 排除正常时段 `2020-03-10 13:20:00` ~ `2020-03-10 13:23:00` 期间的慢查询
    and time < '2020-03-10 13:23:00'
  group by digest)
  order by t1.sum_query_time desc
  limit 10\G
*****[ 1. row ]*****
count(*) | 200
min(time) | 2020-03-10 13:24:27.216186
sum_query_time | 50.114126194
sum_process_time | 268.351
sum_wait_time | 8.476
sum(Commit_time) | 1.044304306
sum(Request_count) | 6077
sum(process_keys) | 202871950
sum(write_keys) | 319500
max(Cop_proc_max) | 0.263
min(query) | delete from test.tcs2 limit 5000;
min(prev_stmt) |
digest | 24bd6d8a9b238086c9b8c3d240ad4ef32f79ce94cf5a468c0b8fe1eb5f8d03df
```

## 3.6 processlist

processlist 用来查看 TiDB 服务器当前的会话信息。

### 3.6.1 查看 processlist 的方式

可以用以下 3 种方式查看 processlist 信息：

#### 1. 执行 SQL: SHOW [FULL] PROCESSLIST

此语句可以查看当前 TiDB 服务器节点的会话信息。下面给出一个实际的例子：

```
SHOW FULL PROCESSLIST;
```

| Id | User | Host      | db     | Command | Time | State | Info                  |
|----|------|-----------|--------|---------|------|-------|-----------------------|
| 1  | root | 127.0.0.1 | <null> | Sleep   | 127  | 2     | <null>                |
| 3  | root | 127.0.0.1 | <null> | Sleep   | 75   | 2     | <null>                |
| 5  | root | 127.0.0.1 | <null> | Query   | 0    | 2     | show full processlist |

输出字段含义说明如下：

- Id：连接 TiDB 服务器会话的唯一标识，可以通过 `KILL TiDB {ID}` 来终止同一服务器上会话 Id 的连接。
- User：当前会话的用户。
- Host：会话客户端的主机名。
- db：会话连接的数据库，如果没有则为 null。
- Command：会话正在执行的命令类型，一般是休眠（Sleep），查询（Query）。
- Time：会话处于当前状态的时间（以秒为单位）。
- State：显示当前sql语句的状态，比如：Sending data，Sorting for group，Creating tmp table，Locked 等等。
- Info：会话正在执行的语句，为 NULL 时表示没有在执行任何语句。

这里需要注意的是，如果不指定可选关键字 `FULL`，输出文本会被截断。

#### 2. 查询系统表 INFORMATION\_SCHEMA.PROCESSLIST

这种方式和第一种方式输出结果类似，不同点在于查询结果会比 `show processlist` 多 `MEM` 和 `TxnStart` 列。这两列具体含义如下：

```
* `MEM` : 指正在处理的请求已使用的内存，单位是 byte。（从 v3.0.5 开始引入）
* `TxnStart` : 指当前处理请求的事务开始的时间戳。（从 v4.0.0 开始引入）
```

下面是一个实际的例子：

```
select *
from information_schema.processlist
where command != 'Sleep'
order by time desc\G
```

```
***** 1. row *****
ID: 1
USER: root
HOST: 172.16.5.169
DB: NULL
COMMAND: Query
TIME: 0
STATE: 2
INFO: select * from information_schema.processlist where command != 'Sleep' order by time desc
MEM: 4588
TxnStart:
1 row in set (0.00 sec)
```

### 3. 查询系统表 INFORMATION\_SCHEMA.CLUSTER\_PROCESSLIST

查询 TiDB 的 INFORMATION\_SCHEMA.PROCESSLIST 系统表时，用户一定会遇到的问题是：此表只包含了当前 TiDB 节点的数据，而不是所有节点的数据。为了解决这个问题，TiDB 4.0 中新增了 INFORMATION\_SCHEMA.CLUSTER\_PROCESSLIST 系统表，用来查询所有 TiDB 节点的 PROCESSLIST 数据。其使用方式和 PROCESSLIST 一致。CLUSTER\_PROCESSLIST 表比 PROCESSLIST 多一个 INSTANCE 列，用来表示该条数据属于哪一个 TiDB 节点。具体示例如下：

```
SELECT * FROM INFORMATION_SCHEMA.CLUSTER_PROCESSLIST\G
```

```
***** 1. row *****
INSTANCE: 172.16.4.235:10070
ID: 1
USER: root
HOST: 172.16.5.169
DB: NULL
COMMAND: Query
TIME: 0
STATE: 2
INFO: SELECT * FROM INFORMATION_SCHEMA.CLUSTER_PROCESSLIST
MEM: 0
TxnStart: 04-12 16:51:39.735(415939035066531841)
***** 2. row *****
INSTANCE: 172.16.5.189:10070
ID: 1
USER: root
HOST: 172.16.5.169
DB: NULL
COMMAND: Sleep
TIME: 6
STATE: 2
INFO: NULL
MEM: 0
TxnStart:
2 rows in set (0.00 sec)
```

注意：只有 root 或者具有 ProcessPriv 权限的 User 能看到所有的会话信息，其他 User 只能看到和自己同一 User 的会话信息。

## 3.6.2 KILL [TIDB]

KILL TIDB 语句用于终止当前 TiDB 服务器中某个会话的连接。

### KILL 语句的兼容性

TiDB 中 KILL xxx 的行为和 MySQL 不相同。在 TiDB 中，需要加上 TIDB 关键词，即 KILL TIDB xxx。但是在 TiDB 的配置文件中设置 compatible-kill-query = true 后，则不需要加上 TIDB 关键词。

## KILL 语句兼容性的设计考量

当用户按下 Ctrl+C 时，MySQL 命令行客户端的默认行为是：创建与后台的新连接，并在该新连接中执行 KILL 语句。然而 TiDB 是一个分布式数据库，一个集群可能部署了多个 TiDB 实例。如果负载均衡器或代理已将该新连接发送到与原始会话不同的 TiDB 服务器实例，则该错误会话可能被终止，从而导致使用 TiDB 集群的业务中断。只有当您确定在 KILL 语句中引用的连接正好位于 KILL 语句发送到的服务器上时，才可以启用 compatible-kill-query。因此如果正尝试终止的会话位于同一个 TiDB 服务器上，可在配置文件里设置 `compatible-kill-query = true`。

## KILL 示例

```
SHOW PROCESSLIST;
```

| ID | User | Host      | db     | Command | Time | State | Info                  |
|----|------|-----------|--------|---------|------|-------|-----------------------|
| 5  | root | 127.0.0.1 | <null> | Query   | 0    | 2     | show full processlist |

```
KILL TiDB 5;
```

```
Query OK, 0 rows affected (0.00 sec)
```

再次查看，可以发现 ID 为 5 的查询已经被 kill 掉了。

### 3.6.3 示例：利用 INFORMATION\_SCHEMA.PROCESSLIST 定位问题查询

通过执行 `show full processlist` 可以看到所有连接的情况，但是大多连接的 `state` 其实是 `Sleep` 的。这种连接其实处于空闲状态，没有太多查看价值。我们要观察的是有问题的连接，可以使用 `select` 查询对 `PROCESSLIST` 系统表进行过滤：

```
-- 查询非 Sleep 状态的链接，按消耗时间倒序展示，加条件过滤
select *
from information_schema.processlist
where command != 'Sleep'
order by time desc \G
```

这样就过滤出来哪些是正在运行的连接。之后再按照消耗时间倒序展示，排在最前面的，极大可能就是有问题的连接了。最后，通过查看 `info` 一列，我们就能看到具体执行的什么 SQL 语句了。

```
***** 1. row *****
ID: 1
USER: root
HOST: 172.16.5.169
DB: NULL
COMMAND: Query
TIME: 0
STATE: 2
INFO: select *
from information_schema.processlist
where command != 'Sleep'
order by time desc
MEM: 4588
TxnStart:
1 row in set (0.00 sec)
```

找出运行较长的 SQL 后，我们可以通过运行 `EXPLAIN ANALYZE` 语句获得一个 SQL 语句执行过程中的一些具体信息。关于 `EXPLAIN ANALYZE` 的具体使用，可查看[使用 EXPLAIN 来优化 SQL 语句](#)

当 TiDB 节点内存不足时，可以用同样方式过滤出消耗内存最多的 10 条运行中的命令在具体执行的什么 SQL 语句。

```
-- 查询非 Sleep 状态的链接，按消耗内存展示其 top10
select *
from information_schema.processlist
where command != 'Sleep'
order by mem desc
limit 10 \G
```

```
***** 1. row *****
ID: 1
USER: root
HOST: 172.16.5.169
DB: NULL
COMMAND: Query
TIME: 0
STATE: 2
INFO: select *
from information_schema.processlist
where command != 'Sleep'
order by mem desc
limit 10
MEM: 4588
TxnStart:
1 row in set (0.00 sec)
```

找出内存消耗较多的 SQL 后，同样可以通过运行 `EXPLAIN ANALYZE` 语句来获具体 SQL 执行信息。

我们可以根据 `command` 和 `time` 条件找出有问题的执行语句，并根据其 ID 将其 kill 掉。如果觉得一个个 ID 来 kill 太慢，还可以通过 `concat()` 内置函数来实现快速 kill

```
-- 查询执行时间超过2分钟，且非 sleep 的会话，然后拼接成 kill 语句
select concat('kill ', 'TiDB', id, ';')
from information_schema.processlist
where command != 'Sleep'
and time > 2*60
order by time desc
```

```
-- 然后再来查看执行时间超过2分钟，且非 sleep 的会话，发现已经全部被 kill 了
select *
from information_schema.processlist
where command != 'Sleep'
and time > 2*60
order by time desc
```

输出样例：

| ID | USER | HOST | DB | COMMAND | TIME | STATE | INFO | MEM | TxnStart |
|----|------|------|----|---------|------|-------|------|-----|----------|
|    |      |      |    |         |      |       |      |     |          |
|    |      |      |    |         |      |       |      |     |          |

### 3.6.4 示例：利用 INFORMATION\_SCHEMA.CLUSTER\_PROCESSLIST 定位问题查询

下面一个例子演示了如何利用 `INFORMATION_SCHEMA.CLUSTER_PROCESSLIST` 查询集群各个 TiDB 节点的运行时间超过2分钟的会话数量：

```
select instance, count(*)
from INFORMATION_SCHEMA.CLUSTER_PROCESSLIST
where command != 'Sleep'
and time > 2*60
group by instance;
```

输出样例：

| instance      | count(*) |
|---------------|----------|
| 0.0.0.0:10081 | 1        |
| 0.0.0.0:10080 | 3        |

如果某个节点的长时间运行会话较多，可以进一步查看该节点的具体会话情况，并结合 `EXPLAIN ANALYZE` 分析具体 SQL。

### 3.6.5 与 MySQL 兼容性

- KILL TIDB 语句是 TiDB 的扩展语法。如果正尝试终止的会话位于同一个 TiDB 服务器上，可在配置文件里设置 `compatible-kill-query = true`。
- TiDB 中的 `state` 列是非描述性的。在 TiDB 中，将状态表示为单个值更复杂，因为查询是并行执行的，而且每个 GO 线程在任一时刻都有不同的状态。
- TiDB 的 `show processlist` 与 MySQL 的 `show processlist` 显示内容基本一样，不会显示系统进程号，而 ID 表示当前的 session ID。其中 TiDB 的 `show processlist` 和 MySQL 的 `show processlist` 区别如下：
  - 由于 TiDB 是分布式数据库，tidb-server 实例是无状态的 SQL 解析和执行引擎（详情可参考 TiDB 整体架构），用户使用 MySQL 客户端登录的是哪个 tidb-server，`show processlist` 就会显示当前连接的这个 tidb-server 中执行的 session 列表，不是整个集群中运行的全部 session 列表；而 MySQL 是单机数据库，`show processlist` 列出的是当前整个 MySQL 数据库的全部执行 SQL 列表。

## 3.7 Statement Summary

在 2.2 节中，介绍过 TiDB Dashboard 中的 Statements 页面，本章节主要介绍 Statements 页面使用到的 Statement Summary 系统表。

### 3.7.1 Statement Summary 是什么

首先来对 Statement Summary 做个简单介绍。如果你用过其他产品，你可能听说过 "Dynamic Performance Views" "Profile Tables" "SQL Audit" 等功能。它们的本质都一样，通过系统表的形式，把 SQL 的性能指标暴露给用户，可以定位性能问题、排查原因。

为了补足现有工具在排查 SQL 性能方面的空缺，TiDB 4.0 提供了 [Statement Summary Tables](#)，这个名字从 MySQL 继承而来。

简单来说，Statement Summary 把相似的 SQL 和执行计划汇总到一组，然后统计每一组的各项性能指标。我们只需要查询系统表，TiDB 就可以把这些指标输出。

#### 3.7.1.1 什么是“相似的 SQL 和执行计划”

我们要排查的经常不是一条 SQL，而是一类 SQL。比如有这样两条 SQL：

```
select * from `order` where item_id=1000;

SELECT *
FROM `order`
WHERE item_id=1001;
```

可以看到两条 SQL 虽然大小写、常量、空白符都不一样，但语法完全一样，那么它们就是同类的 SQL。通过把同类的 SQL 归到一组，我们就可以排查这一类的性能问题了。

但是同类的 SQL 也会生成不同的执行计划，执行计划不同也可能导致性能问题。为了排查这种问题，还要把类型相似、执行计划不同的 SQL 归到不同的组。

还是上面的例子，可能生成 IndexLookup，也可能生成 TableScan。通过把它们归到不同的组，就可以比较各种执行计划下的运行耗时了。

#### 3.7.1.2 有哪些监控指标

为了尽可能地排查出原因，TiDB 在这些系统表中定义了很多字段。截止 4.0 版本，共有 60 多项指标。总共分为几类：

- 基本信息：查询语句、原 SQL 语句、执行计划等等。
- TiDB 上的执行数据：总次数、平均延时、平均内存等等。
- TiKV 上的执行数据：CopTask 数量、平均耗时、扫描行数等等。
- 事务相关：写入的数据量、重试次数等等。

可以看到，这些指标与 slow log 非常相似，但几乎每种指标都有最大值、平均值两项，方便排除执行时间不稳定等因素。

试着运行一下：

```

mysql> select * from events_statements_summary_by_digest limit 1\G
***** 1. row ****
SUMMARY_BEGIN_TIME: 2020-03-04 13:00:00
SUMMARY_END_TIME: 2020-03-04 13:30:00
STMT_TYPE: select
SCHEMA_NAME: test
DIGEST: d8cc0047ec3514d418e6f425d6203966de0094f025dab14babab8f4db0947736
DIGEST_TEXT: select buyer_id , item_id from order where order_id = ?
TABLE_NAMES: test.order
INDEX_NAMES: NULL
SAMPLE_USER: root
EXEC_COUNT: 8
SUM_LATENCY: 2591978
MAX_LATENCY: 1345860
MIN_LATENCY: 135860
AVG_LATENCY: 323997
AVG_PARSE_LATENCY: 69993
MAX_PARSE_LATENCY: 78761
AVG_COMPILE_LATENCY: 137604
MAX_COMPILE_LATENCY: 881880
-----
AVG_MEM: 0
MAX_MEM: 0
AVG_AFFECTED_ROWS: 0
FIRST_SEEN: 2020-03-04 13:20:55
LAST_SEEN: 2020-03-04 13:21:24
QUERY_SAMPLE_TEXT: select buyer_id, item_id from `order` where order_id=1001221
PREV_SAMPLE_TEXT:
PLAN_DIGEST: 28ccfb38e96b6e4aab31e82d12f349eb4edcb97bed3be49cb3f73051f2cd9d2
PLAN:      Point_Get_1      root      1      table:order, handle:1001221
1 row in set (0.00 sec)

```

### 3.7.1.3 数据如何刷新

通常我们在排查问题时，问题是最近才出现的。为了查看最近的监控指标，`events_statements_summary_by_digest` 会定时清空，默认半小时清空一次。这样，我们查看这张表时，总是查看到最新的数据。

但是即使我们看到了最新的指标，也并不知道算不算正常。为了与历史的时间段进行比较，TiDB 还新增了一张历史表 `events_statements_summary_by_digest_history`，它存放着从 `events_statements_summary_by_digest` 清掉的历史数据。通过指定 `SUMMARY_BEGIN_TIME` 和 `SUMMARY_END_TIME` 两个字段的值，可以查看特定时间段的指标。

但是历史表也是内存表，所以有数量限制，默认只保存 24 段历史。

### 3.7.2 使用示例

相比于用图形化界面来呈现结果，查询系统表也有它独特优势：通过借助 SQL 强大的语言表达能力，我们可以挖掘更有价值的内容。下面通过几个案例，演示如何使用 Statement Summary 来排查性能问题。

#### 3.7.2.1 案例一

业务更新后，发现某条 SQL 延时上升到了 10ms，但是 Grafana 上没有异常，如何判定是客户端问题还是服务端问题呢？

可以用 `QUERY_SAMPLE_TEXT` 进行模糊查询：

```

mysql> select avg_latency, query_sample_text from events_statements_summary_by_digest where QUERY_SAMPLE_TEXT LIKE '%select buyer_id, item_id from `order`%\G
***** 1. row ****
avg_latency: 202225
query_sample_text: select buyer_id, item_id from `order` where order_id=1001221
1 row in set (0.00 sec)

```

上面看到 `avg_latency` 是 0.2ms，远低于 10ms，说明服务端没有问题，继而排查网络或客户端问题。

### 3.7.2.2 案例二

业务监控显示凌晨三点整个业务的延时出现了波动，怎么看当时耗时最高的几条 SQL 呢？

```
mysql> select sum_latency, query_sample_text, digest
->     from events_statements_summary_by_digest
->     where summary_begin_time='2020-3-1 3:00:00'
->     order by sum_latency desc limit 3\G
***** 1. row *****
sum_latency: 120663574508
query_sample_text: select buyer_id, item_id from `order` where order_id=1001221
digest: d8cc0047ec3514d418e6f425d6203966de0094f025dab14babab8f4db0947736
***** 2. row *****
sum_latency: 97252462621
query_sample_text: select count(1) from item
digest: 1626b9d694faefc4e88ae9fd5e8917e85ed26fe62dbe781eb65edad3aa939ae8
***** 3. row *****
sum_latency: 65914323442
query_sample_text: select count(1) from buyer
digest: b07c73323c511d3c18407771adb8aec865c0409f4e9a4f53baedc409fd5a1cd0
3 rows in set (0.00 sec)
```

可以看到这几条 SQL 的总耗时最高，可以继续排查它们的其他指标。

`digest` 是这类 SQL 的唯一 ID，所以之后的语句，可以带上 `digest` 来过滤，不需要再用模糊查询了。

### 3.7.2.3 案例三

有条 SQL 上午 10 点还是好的，下午 2 点明显变慢了。我怎么知道它是哪里变慢了呢？

可以比较这条 SQL 在两个时间段各项指标的差异：

```
mysql> select abnormal.avg_latency/normal.avg_latency,
->     abnormal.avg_process_time/normal.avg_process_time,
->     abnormal.avg_total_keys/normal.avg_total_keys,
->     abnormal.avg_wait_time/normal.avg_wait_time
->     from events_statements_summary_by_digest_history abnormal,
->     events_statements_summary_by_digest_history normal
->     where normal.summary_begin_time='2020-3-1 10:00:00'
->     and abnormal.summary_begin_time='2020-3-1 14:00:00'
->     and abnormal.digest=normal.digest
->     and normal.digest = 'd8cc0047ec3514d418e6f425d6203966de0094f025dab14babab8f4db0947736'\G
***** 1. row *****
abnormal.avg_latency/normal.avg_latency: 6.3433
abnormal.avg_process_time/normal.avg_process_time: 8.9993
abnormal.avg_total_keys/normal.avg_total_keys: 12.6666
abnormal.avg_wait_time/normal.avg_wait_time: 0.9883
1 row in set, 1 warning (0.00 sec)
```

上面看到，平均扫描行数（`avg_total_keys`）变大了，导致 TiKV 上的处理时间（`avg_process_time`）变长了。所以需要接着排查为什么扫描行数变大。可能是执行计划、表的数据量、过滤条件、TiKV 的 GC 周期等因素。

### 3.7.3 配置项

上面介绍了一些使用案例，但是实际场景中，往往需要修改各类配置。下面是与 Statement Summary 相关的配置：

- `tidb_enable_stmt_summary`：打开或关闭该功能
- `tidb_stmt_summary_refresh_interval`：监控指标的刷新周期
- `tidb_stmt_summary_history_size`：历史表保存的历史数量
- `max-stmt-count`：保存的 SQL 的种类数量
- `max-sql-length`：显示的 SQL 的最大长度

更具体的使用方法及细节，参照[文档](#)。

## 3.7.4 FAQ

介绍了如何使用 Statement Summary，接下来对常见问题做一下说明。

### 3.7.4.1 配置越大越好吗？

Q：我想查看尽可能多的 SQL、保存的历史尽可能多，可以把 `tidb_stmt_summary_history_size` 和 `max-stmt-count` 改成非常大吗？

A：因为 Statement Summary Tables 是内存表，把配置项改得过大，会占用更多的内存。所以不是越大越好，需要根据每台 TiDB server 的物理内存、实际需求而定。

### 3.7.4.2 显示 commit 语句慢了，怎么查呢？

Q：因为 TiDB 是乐观事务，只有在 commit 时才写数据，导致经常看到 commit 语句慢了，我要怎么确认是哪个事务呢？

A：这种情况确实不好处理。目前的做法是按 commit 的前一个语句进行分类，也就是按 `prev_sample_text` 的 digest 来把 commit 分到不同的组中。这基于一个假设：commit 的前一条语句相同，就算同一类事务。

### 3.7.4.3 `schema_name` 为什么总是空的？

Q：SQL 里明明有表名，但 `schema_name` 这个字段却是空的，怎么回事？

A：这里的 `schema_name` 并不是该语句涉及的所有 schema，而是执行当前语句时所在的 schema（例如执行 `use db` 之后的 db 的名字）。因为 `table_names` 里表名的格式是 "`{schema}.{table}`"，要根据 schema 过滤，就要在 `table_names` 里用正则表达式匹配。

### 3.7.4.4 这个功能有性能影响吗？

Q：Statement Summary 看起来要统计所有 SQL，会有性能影响吗？

A：凡事都有弊端，该功能也一样。Sysbench 的结果表明几乎没有性能下降，但是 TPCC 有 2% 的性能下降。但是 TiDB 认为这个功能带来的价值要大于 2% 的性能影响，所以在 4.0 中默认打开。

## 第4章 TiDB 集群监控与报警

### TiDB 监控框架概述

TiDB 使用开源时序数据库 [Prometheus](#) 作为监控和性能指标信息存储方案，使用 [Grafana](#) 作为可视化组件进行展示。

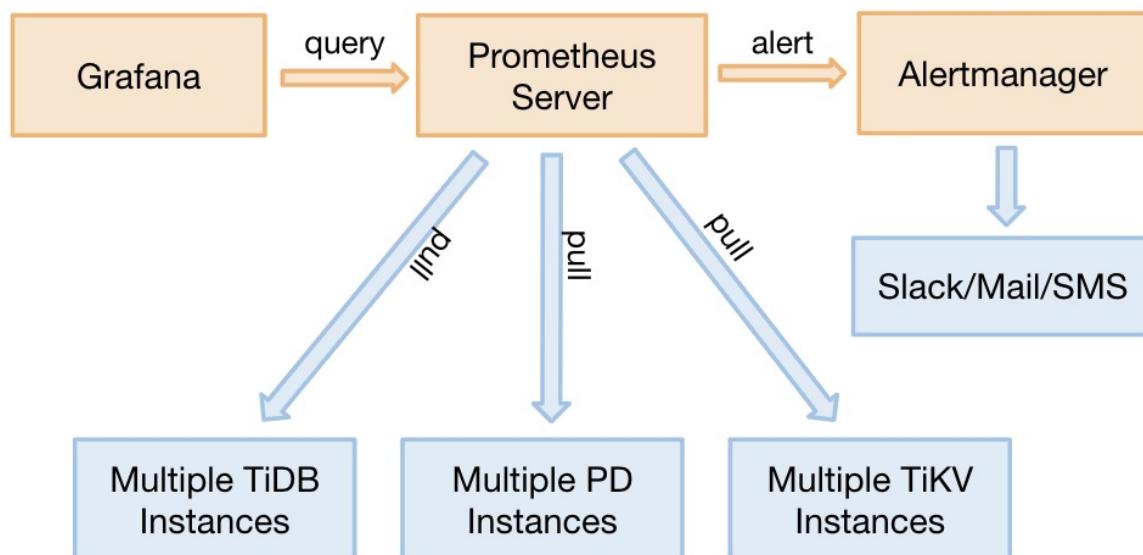
### Prometheus 在 TiDB 中的应用

Prometheus 是一个拥有多维度数据模型的、灵活的查询语句的时序数据库。Prometheus 作为热门的开源项目，拥有活跃的社区及众多的成功案例。

Prometheus 提供了多个组件供用户使用。目前，TiDB 使用了以下组件：

- Prometheus Server：用于收集和存储时间序列数据
- Client 代码库：用于定制程序中需要的 Metric
- Alertmanager：用于实现报警机制

其结构如下图所示：



### Grafana 在 TiDB 中的应用

Grafana 是一个开源的 metric 分析及可视化系统。TiDB 使用 Grafana 来展示 TiDB 的各项性能指标。如下图所示：

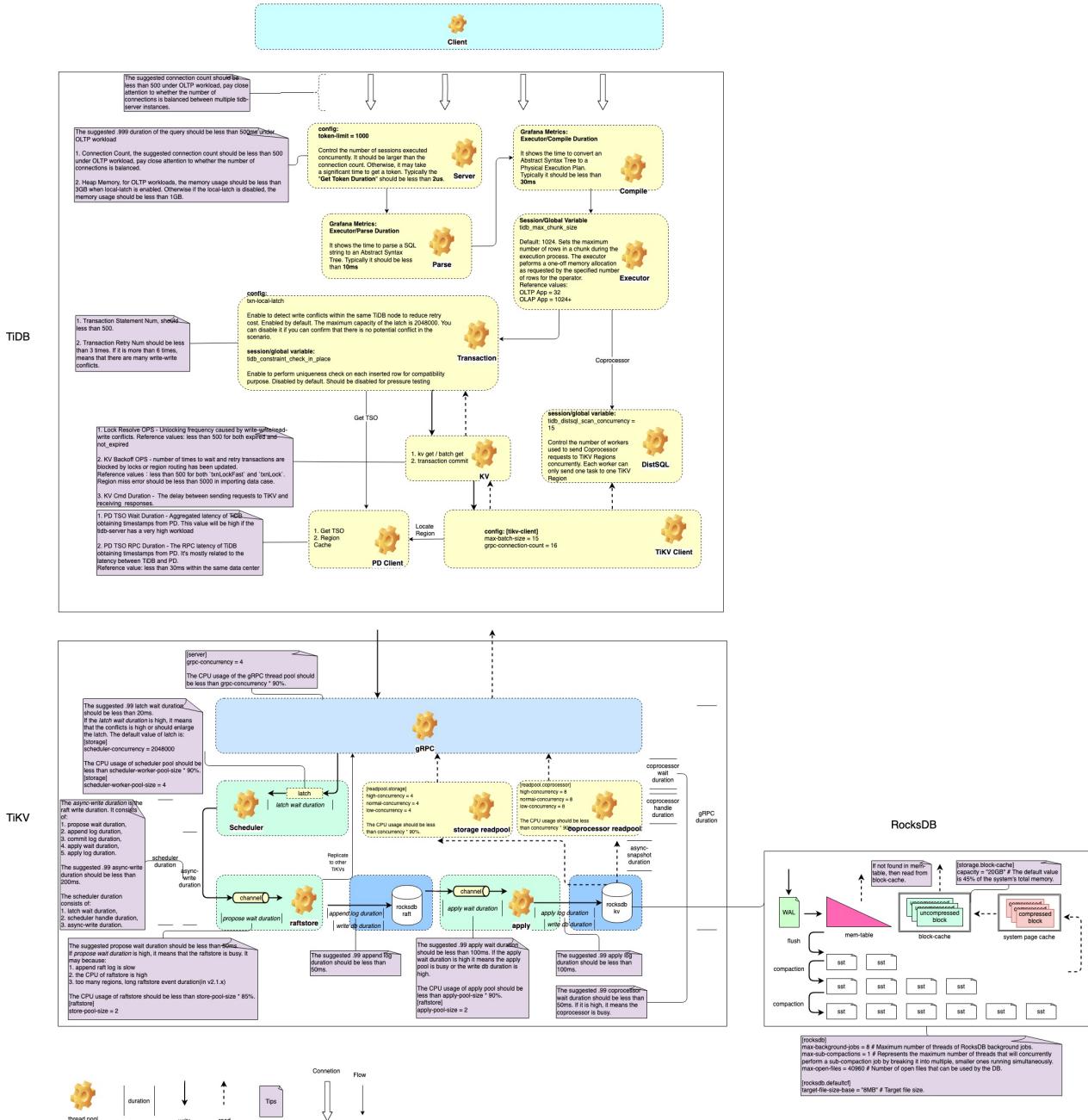


本章将详细介绍 TiDB 各项指标监控指标含义，以及如何借助监控判断 TiDB 集群表现。希望通过本章的介绍，能够让读者对于如何通过监控分析 TiDB 性能，以及设置合理的告警阀值有所了解。

## 4.1 性能调优地图

作为一款分布式数据库，性能调优是 TiDB 中较为复杂且重要的部分。

下图展示 TiDB 各个模块地图，通过这张地图可以清晰展现 TiDB 如何处理一条 SQL。



## 1. TiDB 地图说明

- TiDB 的【Server】模块会为 MySQL 客户端的每一个连接创建一个 session 并且分配一个 token。可以通过 Connection Count 这项监控来查看 TiDB 的连接数，通常建议单个 TiDB 的连接数不超过 500 个
- 经由这个 session 发送到 TiDB 的 SQL 会在【Parse】中被转化为能够被 TiDB 所理解的语法树。转化时间可通过监控【Executor/Parse Duration】查看，通常应该在 10ms 以内
- TiDB 根据语义树生成物理执行计划，决定选择哪些索引或者算子进行计算，这个过程被称作【Compile】。执行时间可通过监控【Executor/Compile Duration】查看

- 生成的物理执行计划会在【Executor】中进行执行
  - 如果是 DML 语句，TiDB 会将用户更新的内容先缓存在【Transaction】模块中，等到用户执行事务的 Commit 时再进行两阶段提交，将结果写入到 TiKV
  - 对于复杂查询请求，TiDB 会通过【DistSQL】模块并行地向 TiKV 的多个 region 发送查询请求，然后再按照执行计划中的流程计算出查询结果来
  - TiDB 发送给 TiKV 的各种请求耗时可以通过监控【KV Request/KV Request Duration 99 by type】查看
- TiDB 采用两阶段提交的事务模型。为此需要向 PD 请求一个全局逻辑时间戳 TSO，用来表明事务的开始时间与提交时间。为了不给 PD 造成过多的请求压力，TiDB 通过单个线程一次为多个事务分配时间
  - 事务在 channel 中等待的时间为监控【PD Client/PD TSO Wait Duration】
  - TiDB 向 PD 请求时间戳的网络请求耗时为监控【PD Client/PD TSO RPC duration】

## 2. TiKV 地图说明

- 参考 [读写流程分析](#) 一章
- TiKV 的主要由 6 个模块构成。分别是 gRPC、Scheduler、raftstore、apply、storage-readpool 以及 coprocessor-readpool
  - gRPC 是 TiKV 所有请求的入口，他会将外界的请求转发给各个模块
    - 对于写事务的请求，会转发给 scheduler 线程
    - 对于简单的读取请求 kv get 或者 kv batch get 会转发给 storage-readpool
    - 对于 TiDB 的 DistSQL 发送过来的 Coprocessor 请求会转发给 coprocessor-readpool（在 4.0 中 storage-readpool 与 coprocessor-readpool 已经被合并为了同一个线程池）
    - 如果是 PD 操作 region 的命令，则直接发送给 raftstore 线程
  - scheduler 负责检测事务冲突，将复杂的事务操作转换为简单的 key-value 插入、删除，发送给 raftstore 线程
    - scheduler 线程执行各个命令的时间可以通过监控【Scheduler-/Scheduler command duration】查看。例如，Prewrite 请求的执行时间在【Scheduler-prewrite】面板
    - 写请求完成 raft 日志复制，到写入 RocksDB 的全部时间可以通过监控【Storage/async write duration】查看
  - raftstore 负责执行 raft 日志复制，将数据复制给多个副本。当日志在多个副本上达成一致后，会发送给 apply 线程
    - 写请求消耗在 raftstore 线程的时间为【Raft IO/Commit log duration】（raft 日志在多数副本上达成一致所需的时间）与【Raft Propose/Propose wait duration】（在队列中等待被处理的时间）之和
  - apply 线程负责将 scheduler 线程的 key-value 操作写入 RocksDB。然后通知 gRPC 线程返回结果给客户端
    - 写请求消耗在 apply 线程的时间为【Raft IO/Apply log duration】（从队列中取出，插入到 RocksDB 的时间）与【Raft Propose/Apply wait duration】（在队列中等待被处理的时间）之和
  - storage-readpool 处理 kv get 以及 kv batch get 等简单的查询请求
  - coprocessor-readpool 处理复杂的范围查询以及表达式计算
- TiKV 共持有两个 RocksDB 实例，一个用于 raftstore 线程记录 raft 日志与 raft 元信息。另一个用于记录用户写入的数据，以及 TiKV 事务中的锁信息
- 线程池调优请见 [8.2.1 TiKV 线程池优化](#) 章节

## 3. RocksDB 地图说明

RocksDB 是一款优秀的开源单机存储引擎，负责将 TiDB 的数据存储在磁盘上。

- RocksDB 的三种基本文件格式
  - Memtable 是一种内存文件数据系统，新数据会被写进 Memtable

- 读取数据如果 Memtable 没有，会访问 Block-Cache
  - Block-Cache 默认设置为系统总内存的 45%，单机多实例的情况下，按照实例个数分配
- WAL Write Ahead Log 写操作先写入 logfile，再写入 Memtable
- SST 在 Memtable 写满以后，将数据写入磁盘中的 SST 文件，对应 logfile 里的 log 会被安全删除。
  - SST 文件中的内容是有序的
  - 根据一定的 Compaction 规则压缩数据

## 4.2 TiDB 读写流程相关监控原理解析

通常来说，监控存在的意义是帮助定位系统的瓶颈和异常。不管是常规巡检还是异常问题排查，系统的瓶颈主要出现在 CPU、内存、网络、磁盘，但是每个瓶颈有不同的横向维度。比如磁盘的 IOPS、吞吐量、延迟、抖动，所以在 TiDB 分布式数据库中具有大量的监控。

本节将会从读写流程的角度，介绍各个部分的关键监控。

为了保持行文连贯，不会插入大量的监控面板图，而是对监控图意义的更加详细解释，例如：

1. 为什么需要这个监控？
2. 这个监控在什么情况下会记录？
3. 如果这个监控的值出现了异常，反应的是系统在哪一方面的瓶颈？

希望读者对监控理解之后，不仅可以看懂监控，还可以举一反三，能对监控的表达式按照业务需要进行调整。

### 4.2.1 读流程

如前文所述，对于一个读取的 SQL，需要将 TiKV 获取到的 Key-Value 转换为用户期望的 Schema 关系型数据。可以对这一句话使用问题的方式进行展开：

1. TiKV 如何存取 Key-Value ？
2. TiKV 如何知道需要获取哪些 Key-Value ？
3. TiDB 如何构造这些请求？
4. TiDB 拿到 Key-Value 之后如何根据 Schema 返回数据？
5. TiDB 如何把最终数据返回给客户端？

希望读者可以根据以上问题阅读后续内容，同时在阅读完成以后，回顾问题加深理解。接下来的内容会按照自底向上的方式来回答以上问题，并分析监控存在的必要性以及监控和实际 SQL 执行过程的关联关系。

#### (1) TiKV 如何获取 Key-Value

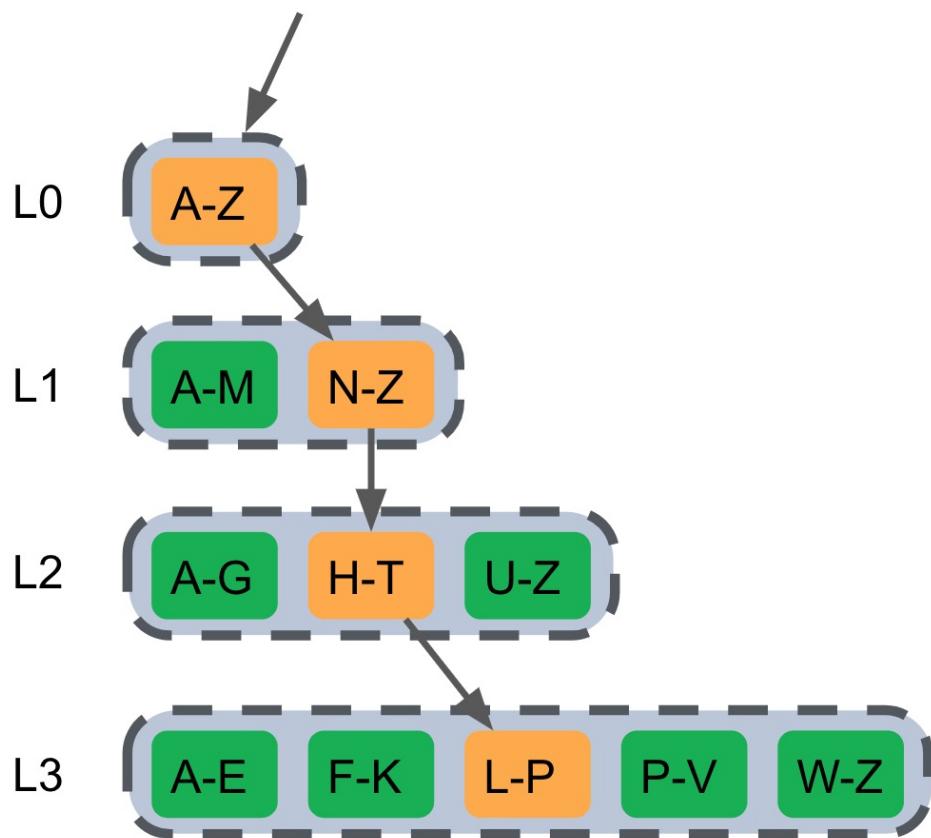
TiKV 底层使用 RocksDB 存储引擎，TiKV 中处理 Query 相关的所有 KV 读取都是基于 RocksDB 的 Snapshot。Snapshot 是 RocksDB DB 实例某一个时间点的视图，可以通过 Snapshot 上的 Get 接口获取单个 KV，也可以通过获取 Iterator 来扫描一个范围。

##### Point-lookup: Get(key)

比如下图查找 “P”，需要从 L0 向下遍历每一层文件直到找到对应的 value：

1. 从 Memtable 中查找，如果没有找到，则从 SST 文件中查找
2. 除了 L0 外，对于每一层二分查找包含这个 key 的文件（每一个文件包含一个 key 的范围，如果目标 key 没有在这个范围之中，则跳过这个文件，另 L0 的文件是可能重叠的，需要查 L0 的所有文件）
3. SST 文件中的查询
  - 通过 SST 的 Index Block 二分定位到 Key 所在的 Block
  - 在这个 Block 中进行 Bloom filter 检查，确定这个文件是否可能包含这个 Key：
    - 如果 Bloom filter 判断为 false，则该 key 一定不在本 block 中。
    - 如果 Bloom filter 判断为 true，则该 key 可能在本 block 中。
4. 通过 Index Block 二分查定位 key 所在的 Block（一个 SST 文件中包含多个 Block）

5. 在 Data Block 中二分查找，找到对应的 value

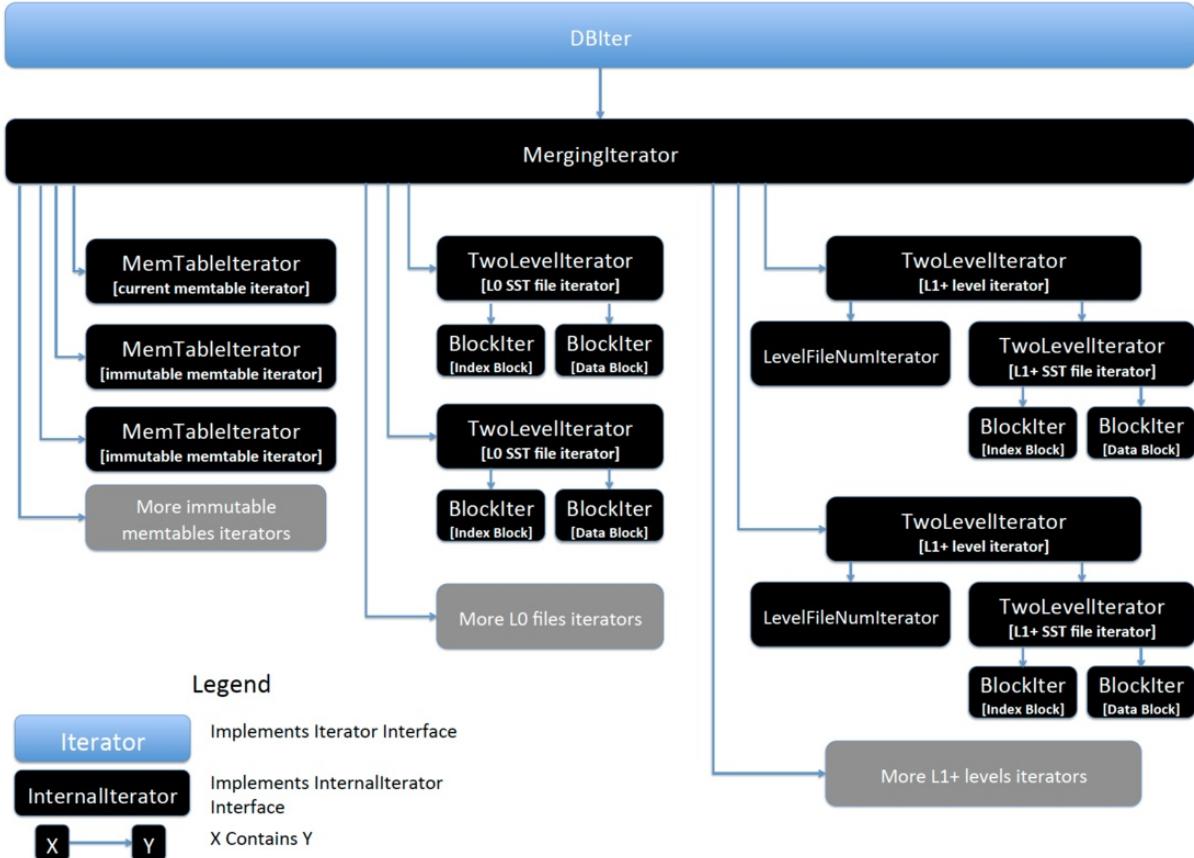


### Range-lookup: Iterator

Iterator 是一个归并迭代器，是一个包含多个子 Iterator 的树结构（如下图所示），不同的子 Iterator 适用 LSM-tree 中不同的数据结构，可以简单的按照以下分类理解：

1. MemTable (Mutable / Immutable)

2. Block (Index Block / Data Block) , RocksDB 使用 TwoLevelIterator 读取 SST File , 其中使用 first\_level\_iter 读取 Index Block , 使用 second\_level\_iter 读取 Data Block



通过上面内容可以发现，不管是 Point-lookup 还是 Range-lookup 都需要读取 SST File 中的 Data Block（对于 point get, 如果在 Memtable 中找到了对应值，就可以直接返回。不需要再往下扫 SST 了），如果每一个 KV 请求都需要通过 IO 从 SST File 的 Data Block 中读取数据，整个系统肯定会因为 IO 而出现瓶颈，所以 RocksDB 中有 Block Cache 在内存中缓存数据。有了以上预备知识之后，就可以来分析在读取 Key-Value 的过程中使用的系统资源，以及查看对应资源的监控信息。以上过程中主要包含：

1. Block Cache , 这一部分依赖 Memory , 对应的监控信息有：
  - a. Block cache size : block cache 的大小。如果将 shared-block-cache 禁用，即为每个 CF 的 block cache 的大小
  - b. Block cache hit : block cache 的命中率
  - c. Block cache flow : 不同 block cache 操作的流量
  - d. Block cache operations 不同 block cache 操作的个数
2. 读取 SST Files , 这部分主要依赖 IO , 对应的监控信息为：
  - a. SST read duration : 读取 SST 所需的时间
3. 二分查找过程中会比较 Key 的大小 , 这部分依赖 CPU , 对应的监控有 :
  - a. RocksDB CPU : RocksDB 线程的 CPU 使用率

以上任何一个环节出现瓶颈，都可能导致读取 Key-Value 的延迟比较高，比如 Block cache size 非常小，命中率很低，那么每次都需要通过 SST File 读取数据必然延迟会加高，如果更糟糕的情况发生，比如 IO 打满或 CPU 打满，延迟会进一步加大。

如果 Block cache 命中率小，可以通过 Block cache flow 和 Block cache operations 进一步定位 Block cache 上面发生的事件。

## (2) TiKV 如何知道需要获取哪些 Key-Value

TiDB 通过 gRPC 发送读取请求，读相关的请求主要包含 kv\_get / kv\_batch\_get / coprocessor 三个接口，可以通过以下监控查看请求的数量、失败数量、延迟等情况：

- gRPC message count : 每种 gRPC 消息的个数

- gRPC message failed : 失败的 gRPC 消息的个数
- gRPC message duration : gRPC 消息的执行时间
- gRPC poll CPU : gRPC 线程的 CPU 使用率, gRPC 线程是一个纯 CPU 的线程, 不涉及 IO 操作, 如果使用率过高, 可以考虑调整 gRPC 线程数量, 不然客户端请求会因为处理不及时导致延迟加大, 此时 gRPC 会成为瓶颈

以上三个监控能对整个过程进行一个概览性的分析, 如果其中某一个监控指标出现了异常, 就需要进一步分析。在 TiDB 4.0 引入统一线程池之后, kv\_get / kv\_batch\_get / coprocessor 以上这三个接口的流量分别在两个线程池执行, 分别为 Storage read threads pool (kv\_get / kv\_batch\_get 接口) 和 Coprocessor threads pool (coprocessor 接口), 这两个线程池中都由低中高三优先级的线程。在 TiDB 4.0 中引入了 Unified Read Pool 之后, 这些请求都会在同一个线程池中执行。

接下来分别解释如何将请求发送到对应的线程池, 以及各个线程池内部的执行流程。

## Storage read threads pool

kv\_get / kv\_batch\_get 比较简单, 在收到 gRPC 请求之后, 根据优先级选择对应的线程池即 FuturePool, 然后分发到对应的 FuturePool 之中。当这个请求被 Pool 调度起来之后, 执行流程为:

1. 从 RocksDB 获取一个 Snapshot
2. 然后再在 Snapshot 上面使用 Point-lookup 获取相应的 Key-Value

获取 Snapshot 和通过 Snapshot 获取 KV 可以参考之前的内容。Storage read threads pool 可以通过以下监控来排查:

- Storage async snapshot duration : 执行 get 命令所需花费的时间 (按照之前的说明, 这部分时间就是调度之后, RocksDB->GetSnapshot() + Snapshot->Get(key) 的时间总和, 所以不包含调度本身花费的时间)
- Storage command total : 可以查看 get 命令的数量
- Storage ReadPool CPU : 线程的 CPU 使用率

比较套路的排查思路是先查看 CPU 是否被打满。如果被打满, 则查看是不是命令的数量比较多; 如果 CPU 没有打满, 但是 Storage async snapshot duration 延迟比较高, 就按照获取 Snapshot 和从 Snapshot 中获取 KV 比较慢进行排查 (参考前文)。

## Coprocessor threads pool

Coprocessor 内部细节较多, 下面进行一个简化的描述:

1. TiKV 收到一个 coprocessor 的请求之后, 先解析这个请求, 然后根据优先级放入 Coprocessor 相应的线程池
2. 请求被 Coprocessor 线程池调度起来之后, 第一步先获取 Snapshot
3. 然后根据 Snapshot 构建一个 Handler 用来处理这个请求

在大致了解以上流程之后, Coprocessor 相关的监控就比较容易理解了:

1. Coprocessor CPU : 线程的 CPU 使用率
2. Wait duration : 请求被调度 + 获取 Snapshot + 构建 Handler 的时间总和 (监控中包含 max / .99 / .95 对应的延迟)
  - a. 请求被调度这个延迟不涉及 IO 和网络, 仅仅和 CPU 的负载有关, 如果 CPU 的负载不高, 调度的时间应该可以忽略不计
  - b. 构建 Handler 的时间在 CPU 负载不高的情况也可以忽略不计
  - c. Snapshot 获取时间如果过长, 可以按照之前的内容进行排查
 综上: 首先应该查看 CPU 的负载情况, 如果 CPU 的负载高则说明瓶颈在 CPU, 否则进一步排查 Snapshot 获取慢的原因
3. Handle duration : 使用 Handler 处理这个请求的时间, Handle 过程包含:
  - a. 算子的执行依赖 CPU, 可以通过 CPU 的负载来定位瓶颈是否在 CPU
  - b. 最底层的算子 TableScan / IndexScan 需要使用 Snapshot 的 Range-lookup, 如果 CPU 负载不高, 但是 Handle duration 延迟高, 则按照 Range-lookup 慢的问题进一步排查

除了以上比较重要的监控指标, 也可以通过以下监控辅助排查:

- Total Request errors : 下推的请求发生错误的个数, 正常情况下, 短时间内不应该有大量的错误

- Total DAG executors : DAG executor 的个数，重点查看 TableScan 算子的数量，排查是否有大查询扫描将 CPU 资源占用
- Total KV Cursor Operations : 每个请求 scan key 的个数，如果 Scan keys 的数量特别大，会占用大量的系统资源，导致请求的延迟变大
- Total Ops Details (Table Scan) : table scan 对从 Snapshot 中获取的 Iterator 执行的操作统计
- Total Ops Details (Index Scan) : index scan 对从 Snapshot 中获取的 Iterator 执行的操作统计
- Total RocksDB Perf Statistics : 执行 scan 的时候，根据 perf 统计的 RocksDB 内部 operation 的个数

## Unified Read Pool

在 TiDB 4.0 中参照了多级反馈队列调度算法构建了一个新的线程池 (yatp, yet another thread pool)，并且将前面介绍的两种请求都统一到了这个线程池中进行处理，称为 Unified Read Pool。

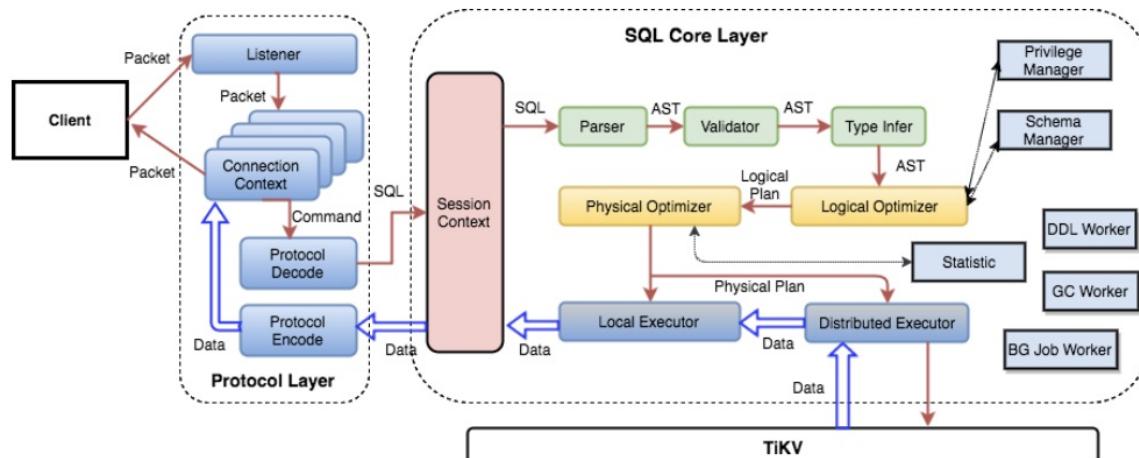
在 Unified Read Pool 中，任务会根据优先级以及执行时长的不同被放到不同的队列中，主要目的是为了减少执行时间长、优先级低的读请求对其他请求的影响，减少延时和吞吐量的抖动。Unified Read Pool 可通过以下可以通过以下监控来排查：

- Unified ReadPool CPU : 线程的 CPU 使用率
- Time used by level : 各个 level 任务的 CPU 使用时间
- Running tasks : 当前读请求的数量

## (3) TiDB 如何构造这些请求

在描述了 TiKV 如何处理 KV 相关请求，以及各个处理环节如何通过监控排查问题后，接下来将描述一条 SQL 经过哪些步骤最终构建一个合适的请求发送到 TiKV，以及如何将获取到的 KV 最终映射为客户端期望的结果。为了屏蔽细节对于读者的干扰，在不影响正确性的情况下对整个主流程进行简化为如下步骤：

1. 从客户端的 Socket 读取一条 SQL
2. 获得一个 Token
3. 从 PD 获取 TSO (异步获取，此处拿到一个 tsFuture，后续的流程中可以通过 tsFuture 结构拿到真正的 TSO)
4. 使用 Parser 将 SQL parse 为 AST
5. 将 AST compile 为执行计划 (此过程包含很多细节，比如 Validator / LogicalPlanOptimizer / PhysicalPlanOptimizer / Executor builder 等，由于都反映在 Compile duration 的监控之中，此处合为一个步骤)
6. 执行上一步得到的执行计划
  - a. Executor.Open() : 最底层的 Executor 会根据这条 SQL 处理的 Key 范围构建出多个要下发到 TiKV 的请求，并通过 distsql 的 API 将这些请求分发到 TiKV
  - b. Executor.Next() : 最底层的 Executor 会将 distsql 返回的数据返回给上层 Executor



由于这一部分涉及的监控非常多且复杂，本小节先从概览到细节对监控进行梳理：

1. QPS：每秒的查询数量
2. Duration：SQL 执行的耗时统计
3. Get Token Duration：建立连接后获取 Token 耗时
4. Parse Duration：SQL 语句解析耗时统计
5. Compile Duration：将 SQL AST 编译成执行计划耗时统计
6. Execution Duration：SQL 语句执行耗时统计

以上几个监控可以反映一个查询的时间消耗主要是在哪个大模块，比如：

1. Parse Duration / Compile Duration 是纯 CPU 操作，如果 CPU 负载不高，但是耗时比较高，大部分情况是 insert ... values 太多。Compile 耗时比较高，可能是带了非关联子查询
2. Get Token Duration 耗时比较高说明已经在执行的 SQL 达到了 TokenLimiter 的上限，具体情况可能很复杂，比如可能是简单的数量达到了上限，或则内部出现了卡死导致 Token 没有释放
3. Execution Duration 包含了 Executor 执行过程中的总耗时。内部涉及的组件比较多，下文将专门对这一部分进行解释
4. TSO 获取比较慢，下文会详细介绍

## TSO 获取相关细节

因为 TSO 的监控对于很多使用者来说比较难懂，所以专门用一个小节对其内部细节进行分析。

所有 TiDB 与 PD 交互的逻辑都是通过一个 PD Client 的对象进行的，这个对象会在服务器启动时创建 Store 时创建出来，创建之后会开启一个新线程，专门负责批量从 PD 获取 TSO，这个线程的工作流程大致流程如下：

1. TSO 线程监听一个 channel，如果这个 channel 里有 TSO 请求，那么就会开始向 PD 请求 TSO（如果这个 channel 有多个请求，本次请求会进行 batch）
2. 将批量请求的 TSO 分配给这些请求

对于这些 TSO 请求，分为三个阶段：

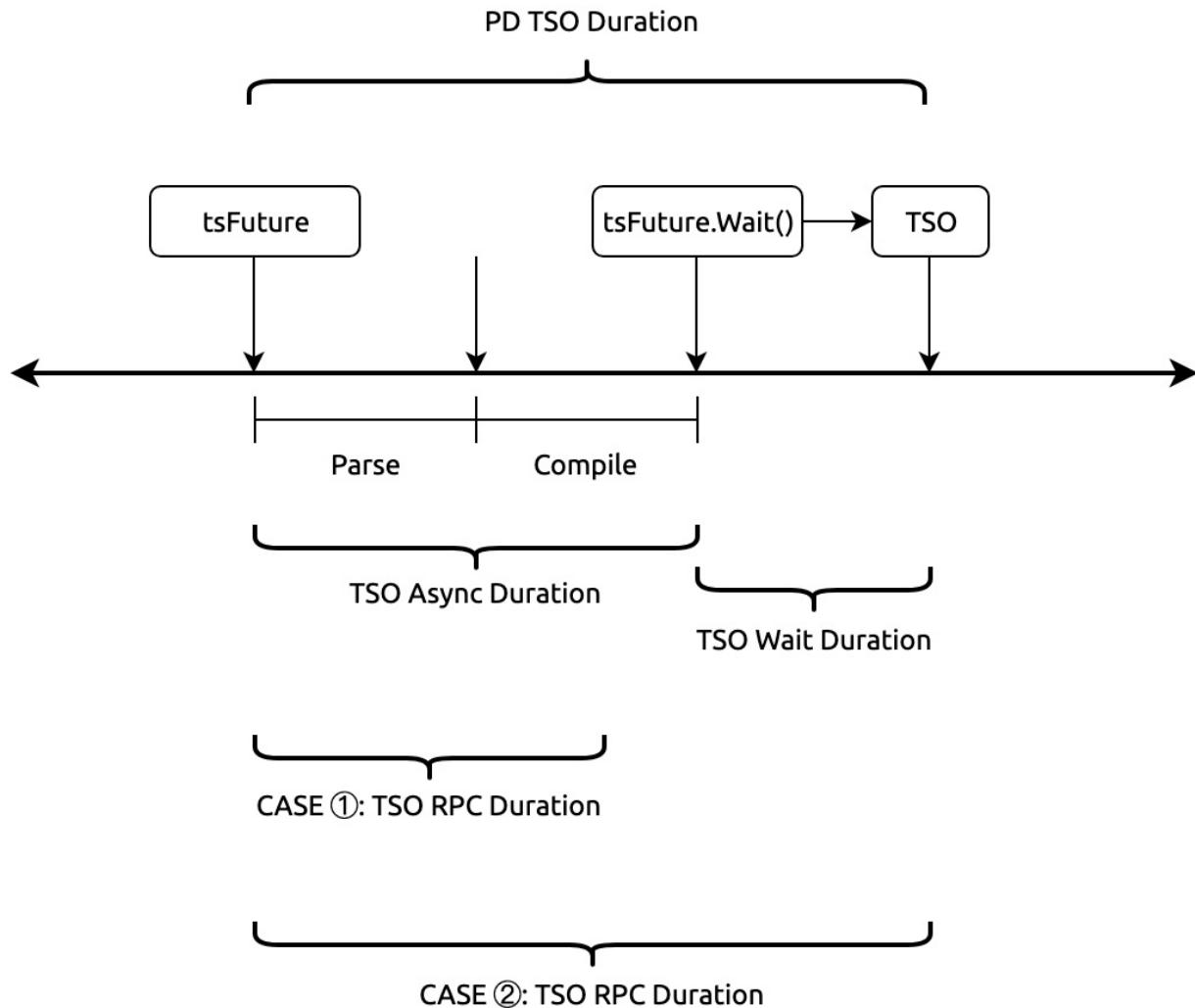
1. 将一个 TSO 请求放入这个 channel，对应的函数为 GetTSAsync，调用这个函数会得到一个 tsFuture 对象
2. TSO 线程从 channel 拿到请求向 PD 发起 RPC 请求，获取 TSO，获取到 TSO 之后分配给相应的请求
3. TSO 分配给相应的请求之后，就可以通过持有的 tsFuture 获取 TSO（调用 tsFuture.Wait()）

目前系统没有对第一个阶段设置相应的监控信息，这个过程通常很快，除非 channel 满了，否则 GetTSAsync 函数很快返回，而 channel 满表示 RPC 请求延迟可能过高，可以通过 RPC 请求的 Duration 进一步分析。

- PD TSO RPC Duration：反映向 PD 发起 RPC 请求的耗时，这个过程慢有两种可能：
  - TiDB 和 PD 之间的网络延迟高
  - PD 负载太高，能不能及时处理 TSO 的 RPC 请求
- PD TSO Wait Duration：我们可能拿到一个 tsFuture 之后，很快做完 Parse 和 Compile 的过程，这个时候去调用 tsFuture.Wait()，但是这个时候 PD 的 TSO RPC 还没有返回，我们就需要等。该项监控反映的是这一过程的等待时间。

所以这部分的正确分析方式是先查看整个 TSO 是否延迟过大，再查看具体是哪个阶段延迟过大。

有了上面的讲解，下面用一个图来辅助理解：



注意图中标记了两种 TSO RPC Duration 的情况，因为 TSO 是异步获取的，后台线程在异步获取 TSO 过程中，处理 SQL 的线程在进行 Parse + Compile (Parse + Compile 完成后调用异步对象 `tsFuture` 获取真正的 TSO，所以 TSO Async Duration = Parse + Compile Duration)，两种情况分别为：

- CASE 1 : TSO RPC Duration  $\geq$  Parse + Compile Duration，这种情况调用 Wait 时，TSO 还未就绪，所以需要等待 RPC 返回 TSO，所以 TSO Wait Duration  $> 0$
- CASE 2 : TSO RPC Duration  $<$  Parse + Compile Duration，这种情况调用在异步对象调用 `tsFuture.Wait()` 时，由于 RPC 已经返回真实的 TSO，所以 Wait Duration = 0

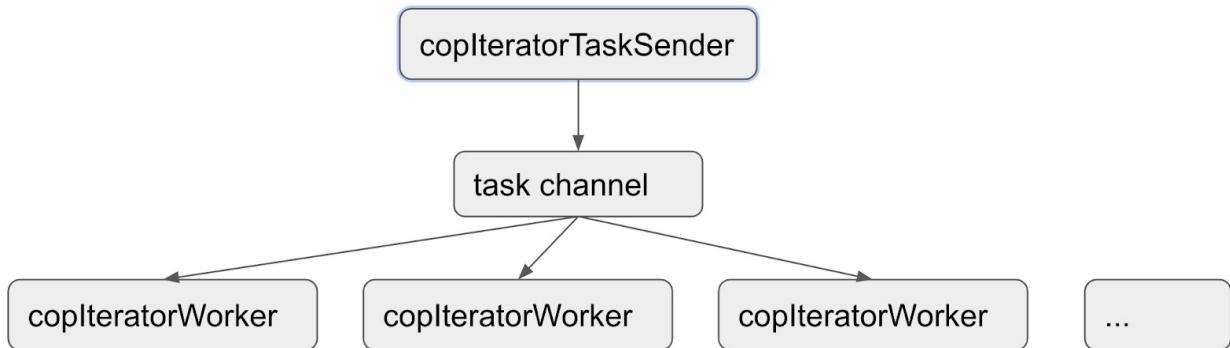
## DistSQL 内部执行流程

`distsql` API 可以简单的认为包含一下几个重要步骤：

1. Build Request
2. Send Request
3. Recv Response
4. Decode Response

其中 Build Request / Decode Response 比较简单，Send Request / Recv Response 相对比较复杂，接下来先看看 Send Request 和 Recv Response 的过程，然后再分析其中各个环节可能出现的问题，以及对应的监控信息：

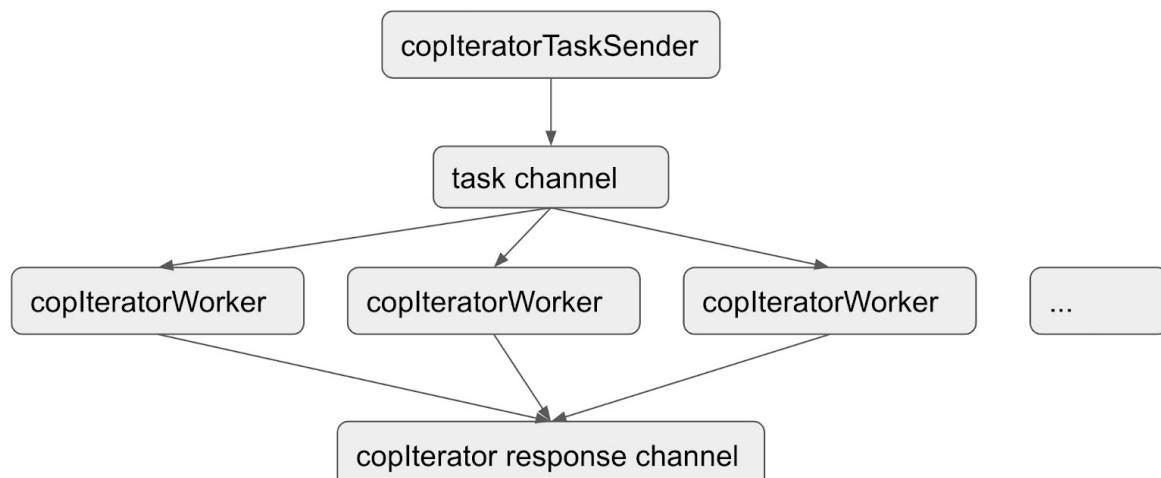
1. Send Request 发送的过程，如下图所示



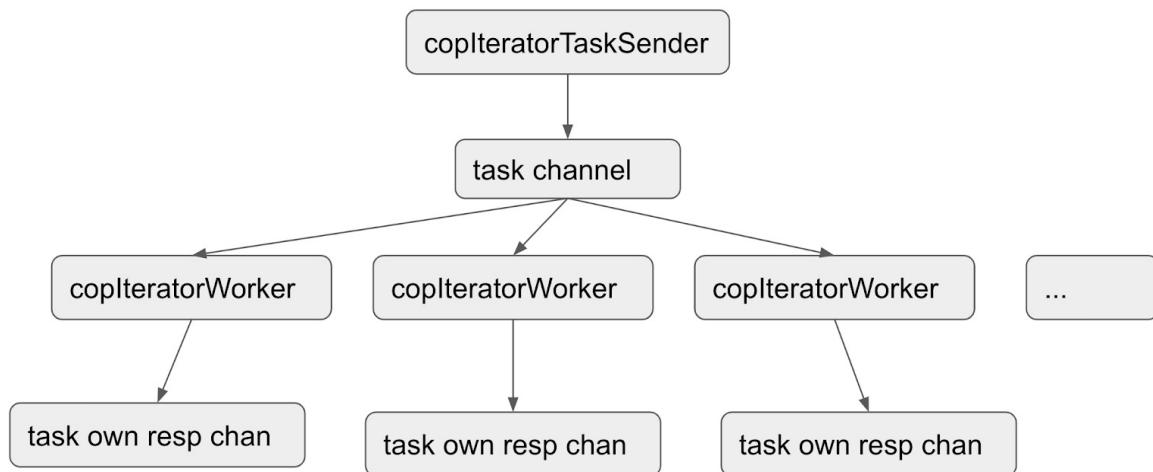
- a. 首先根据请求中 Key 的范围在 RegionCache 找到对应的 Region，然后构造一个 copTask  
 b. 根据并发配置，开启多个 copIteratorWorker goroutine  
 c. 开启一个 copIteratorTaskSender goroutine 分发 copTask  
 d. 各个 copIteratorWorker 并行处理 copTask (构造发往 TiKV 的请求，通过 gRPC 调用对应的接口)

1. Recv Response 的接口过程分为两种情况

- a. 不需要对发送请求的返回结果保持发送之前的顺序



- b. 需要对发送请求的返回结果保持发送之前的顺序



以上是对 Send Request 和 Recv Response 的流程介绍。接下来介绍正常情况下需要关注哪些监控，查看这个过程中可能存在异常结果，以及对应的监控。

- DistSQL

- Distsql Duration : Distsql 处理的时长
- Distsql QPS : Distsql 的数量统计
- Distsql Partial QPS : 每秒 Partial Results 的数量

- Scan Keys Num : 每个 Query 扫描的 Key 的数量
- Scan Keys Partial Num : 每一个 Partial Result 扫描的 Key 的数量
- Partial Num : 每个 SQL 语句 Partial Results 的数量
- KV Request
  - KV Request Duration 999 by store : KV Request 执行时间，根据 TiKV 显示
  - KV Request Duration 999 by type : KV Request 执行时间，根据请求类型显示
  - KV Request OPS : KV 命令执行数量统计
- Transaction
  - Transaction OPS : 启动事务的数量统计
  - Transaction Regions Num 90 : 事务使用的 Region 数量统计
  - Transaction Max Write Size Bytes : 事务写入的最大字节数统计
  - Transaction Max Write KV Num : 事务写入的最大 Key-Value 数量统计
  - Load SafePoint OPS : 更新 SafePoint 的数量统计

在请求处理的过程中，TiKV 可能会返回一些错误，对于这可能出现的错误以及它们的处理方式如下：

1. 如果是 Region 相关的错误（Region 发生分裂、合并、调度等），会先进行 Backoff (sleep 一小段时间)，然后进行重试，比如：a. Region 信息过期会使用新的 Region 信息重试任务 b. Region 的 leader 切换，会把请求发送给新的 Leader
2. 如果部分 Key 上有锁，会进行 Resolve lock
3. 如果有其他错误，则立即向上层返回错误，中断请求

错误相关的监控在 KV Errors 分类下：

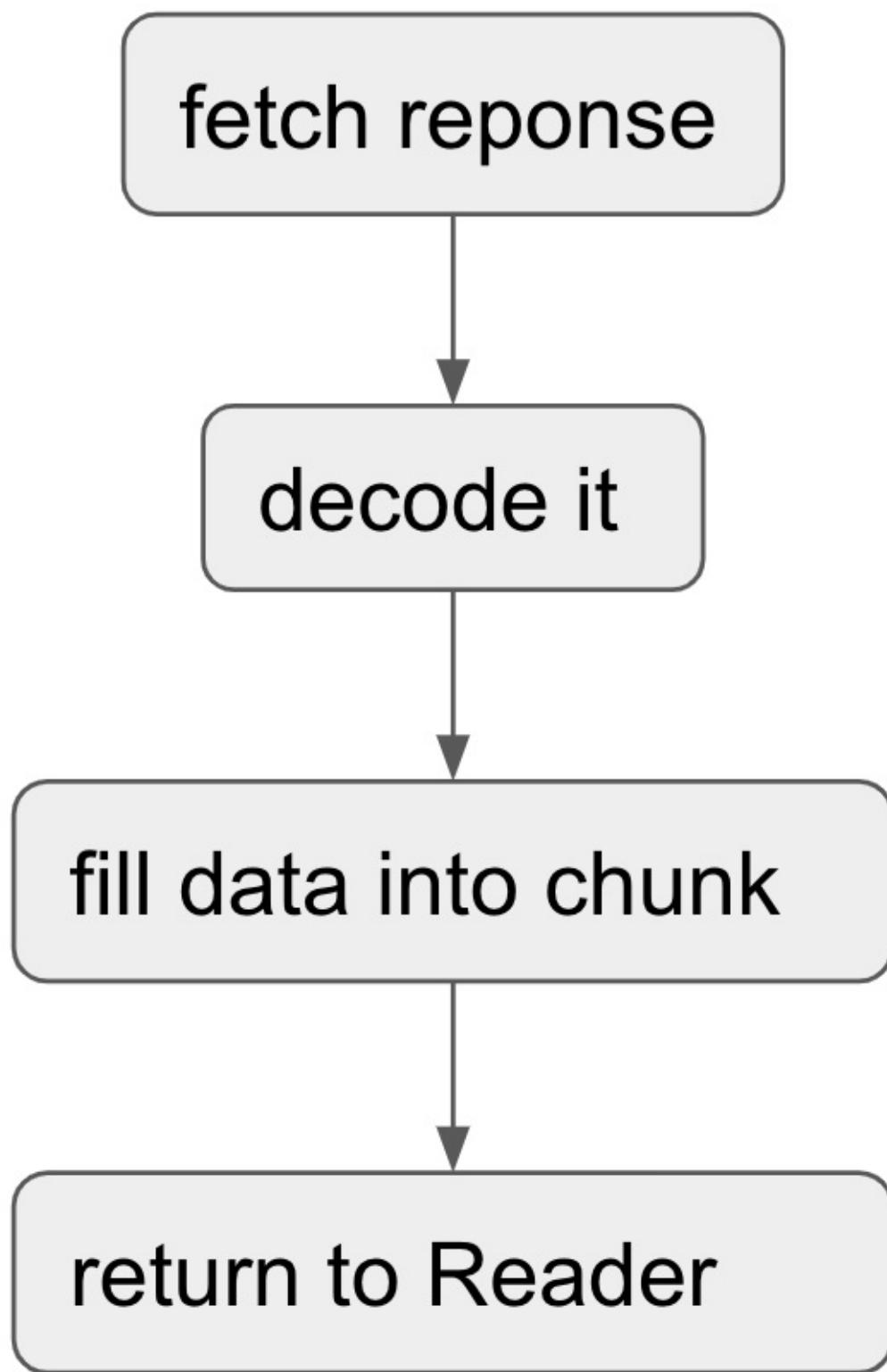
- KV Backoff Duration : 重试请求的时间
- TiClient Region Error OPS : TiKV 返回 Region 相关错误信息的数量
- KV Backoff OPS : TiKV 返回错误信息的数量（事务冲突等）
- Lock Resolve OPS : 事务冲突相关的数量
- Other Errors OPS : 其他类型的错误数量，包括清锁和更新 SafePoint

#### (4) TiDB 拿到 Key-Value 之后如何根据 Schema 返回数据

TiDB 拿到 Key-Value 数据可能有两种格式：

1. Coprocessor 返回的 chunk 格式
2. Get / Batch Get 返回的 Key-Value 格式

对于这两种格式，Coprocessor 返回的 chunk 格式已经从 value 中把的 column 信息提取出来了，而 Get / Batch Get 返回的 Key-Value 还是按照 encode 之后的字节数组，所以需要进一步 decode，大致流程如下：



### (5) TiDB 如何把最终数据返回给客户端的

从最底层的算子一层一层向上返回，最上层的算子拿到数据之后，通过 Socket 将结果返回给客户端。

## 4.2.2 写流程

如前文所述，对于一个写入的 SQL，需要根据已经定义的 Schema 转换为 Key-Value，并将这些 Key-Value 写入 TiKV。读取与写入在 TiDB 端的不同主要是最底层的 Executor，读取的底层 Executor 是 TableReaderExecutor / IndexReaderExecutor / IndexLookupExecutor，这些 Executor 主要依赖底层的 distsql API，而写入的底层 Executor 主要是 Insert / Update / Delete，主要依赖底层的 2PC 模块。所以与读流程相似的原理和监控不在本小节重复说明。本小节主要自顶向下梳理：

1. TiDB 2PC (两阶段提交)
2. TiKV Scheduler
3. TiKV Raftstore

## (2) 两阶段提交

当我们在谈论两阶段提交，我们在谈论什么？

### 提交的是什么

写入的算子 Insert / Update / Delete 在执行过程中对于写入的数据编码为 Key-Value 并先写入事务的 In-Memory Buffer，写入的数据包含两类：

1. 普通数据插入
2. 索引数据插入

普通数据和索引数据使用不同的编码方式转换 Key 和 Value，如何将数据编码成对应的 Key-Value 的细节与监控以及原理无关，这一部分可以暂时忽略，统一理解为 Key-Value 即可。

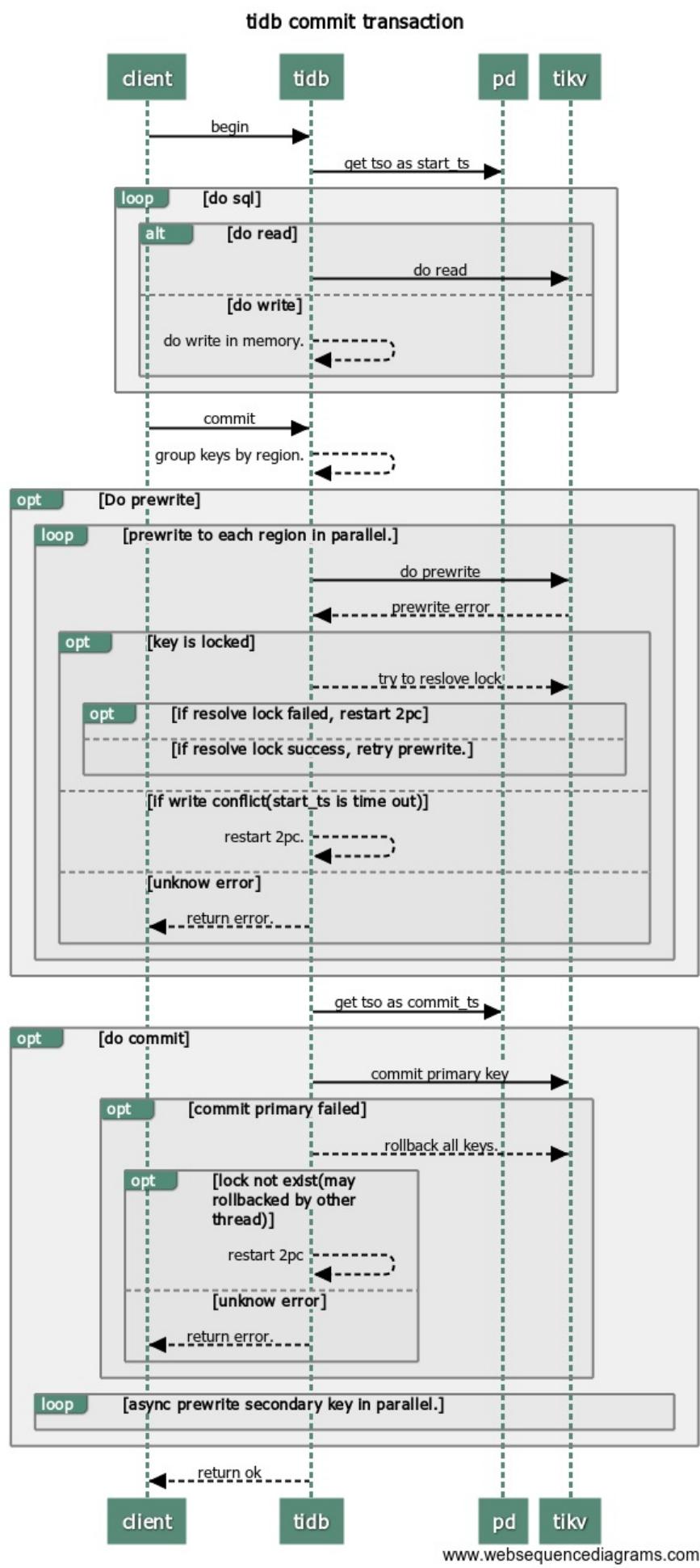
两阶段提交的本质就是将 In-Memory Buffer 中的 Key-Value 通过 tikv client 写入到 TiKV 中。

### 如何提交

顾名思义，两阶段提交就是将事务的提交分成两个阶段：

1. Prewrite
2. Commit

在每个 Transaction 开启时会获取一个 TSO 作为 start\_ts，在 Prewrite 成功后 Commit 前获取 TSO 作为 commit\_ts，如下图：



了解以上基础认识之后，可以进一步查看这个过程中的监控信息，以及监控信息背后所代表的详细信息。

### (3) 本地事务冲突检测

如果配置文件中启用了本地事务冲突检测

```
[txn-local-latches]
enabled = true
capacity = 2048000
```

在 Prewrite 之前，会先在本地等待内存锁，如果冲突比较严重，等待内存锁的时间也可能比较长，具体可以参考监控：

- Local Latch Wait Duration：本地事务等待时间

### (4) 事务相关监控

在介绍 Distsql 部分时已经罗列过：

- Transaction 面板
  - Transaction OPS：事务执行数量统计
  - Duration：事务执行的时间（如果启用 Latches，则包含 Latches 的时间）
  - Transaction Retry Num：事务重试次数
  - Transaction Statement Num：一个事务中的 SQL 语句数量
  - Session Retry Error OPS：事务重试时遇到的错误数量
  - KV Transaction OPS：启动事务的数量统计
  - Transaction Regions Num 90：事务使用的 Region 数量统计
  - Transaction Max Write Size Bytes：事务写入的最大字节数统计
  - Transaction Max Write KV Num：事务写入的最大 Key-Value 数量统计
- KV Errors
  - KV Backoff Duration：KV 重试请求的时间
  - TiClient Region Error OPS：TiKV 返回 Region 相关错误信息的数量
  - KV Backoff OPS：TiKV 返回错误信息的数量（事务冲突等）
  - Lock Resolve OPS：事务冲突相关的数量

以上主要是 TiDB 端 2PC 的基本知识，以及需要关注的监控，接下来关注 TiKV 端。

### (5) TiKV Scheduler

#### Scheduler 是什么

Scheduler 可以简单的认为是包含以下两部分的一个对象：

1. 内存锁 Latches
2. 两个线程池（普通优先级 / 高优先级）

其中 Latches 的实现与 TiDB 的实现类似，都用于等锁，一个 Prewrite 和 Commit 涉及多个 Key-Value 时，只有所有的 Key 都拿到内存锁，才会真正进入 Scheduler 的线程池执行具体的写入逻辑。

而线程池内执行的具体逻辑就比较复杂，对于不同的写入请求逻辑也不一致，可以简化为：

1. 获取 Snapshot
2. 执行具体的命令（依赖于前一步获取到的 Snapshot），普通优先级的线程池负责处理 Prewrite、Commit 等写入请求，而高优先级的线程池负责处理 Resolve Lock 等高优先级的任务

对于具体的命令执行，这里对于 Prewrite 和 Commit 也给出一个最简化的流程：

1. Prewrite :

- a. 对于每一个 Key : load lock , 如果这个 key 已经存在锁，则对于这个 key 返回 KeyIsLocked 错误
- b. 为每一个没有被锁住的 key 新建一个锁信息（此时锁信息还在内存中）
- c. 如果之前已经发现有 Key 被锁住，则返回这些冲突的 Key ; 如果所有的 Key 都没有被锁住，则将内存中的锁信息异步写入到 raftstore

2. Commit

- a. 对于每一个 Key : load lock , 这个 lock key 是在 Prewrite 的阶段写入的，如果已经不存在了，则返回 TxnLockNotFound 错误，如果存在则从锁信息中拿到 Value
- b. 将 Key-Value 异步写入到 raftstore

从上面的信息我们可以发现有哪些环节是需要监控的：

1. Scheduler 中包含两个线程池，所以需要关注这部分线程的 CPU 负载
2. 当前线程池正在处理的 Command 数量
3. 内存锁的等待时间
4. 拿到 Snapshot 的时间
5. Load lock 需要的时间
6. 异步写入的时间
7. Scheduler 执行一个命令的时间

## Scheduler 相关监控

- gRPC

- gRPC duration : 请求在 TiKV 端的总耗时（主要关心 Prewrite / Commit 两种类型），通过对比 TiKV 的 gRPC duration 以及 TiDB 中的 KV duration 可以发现潜在的网络问题。比如 gRPC duration 很短但是 TiDB 的 KV duration 显示很长，说明 TiDB 和 TiKV 之间网络延迟可能很高，或者 TiDB 和 TiKV 之间的网卡带宽打满了。如果 TiDB 和 TiKV 的响应时间相符，但是时间比较长，可以查看与 Scheduler Duration 是否相符进一步排查。gRPC duration 的时间等于 gRPC 线程池将这个请求分发给 Scheduler 模块以及这个命令在 Scheduler 内部执行时间的总和。

- Scheduler

- Scheduler stage total : 每种命令不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler priority commands : 不同优先级命令的个数
- Scheduler pending commands : 每个 TiKV 实例上 pending 命令的个数
- Scheduler duration : 等于 latch-wait duration + async-snapshot duration + load lock (all keys total) + async-write duration 的时间总和。
- Scheduler worker CPU : scheduler worker 线程的 CPU 使用率
- Scheduler latch wait duration : latch wait 延迟高的原因：
  - 多个事务同时对相同的 key 进行写操作导致冲突，每一个 key 都需要等前一个事务在该 region 上的涉及到这个 key 的写请求（可能是 prewrite、commit 或者 acquire-pessimistic-lock）完成 raft 日志复制后，才能获得 latch 锁
- Scheduler keys read : commit 命令读取 key 的个数
- Scheduler keys written : commit 命令写入 key 的个数
- Scheduler scan details : 执行 commit 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock] : 执行 commit 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write] : 执行 commit 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default] : 执行 commit 命令时，扫描每个 default CF 中 key 的详细情况

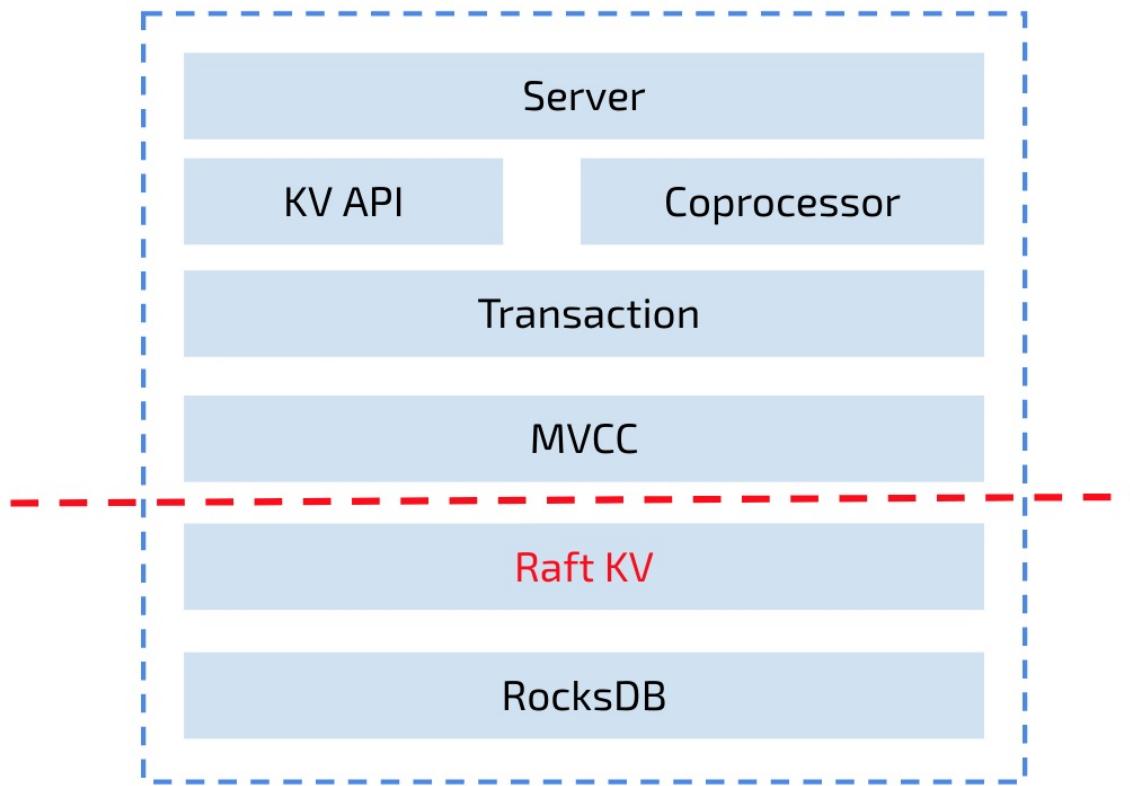
- Storage

- Storage async-snapshot duration : 请求 raft snapshot 的延迟（读之前需要取一个 snapshot）
- Storage async-write duration : raft 写入延迟，如果发现部分延迟高，则需要通过后续 Raftstore 的部分进一步定位

总的来说，Scheduler 这部分的主要工作就是拿到一个 Snapshot 然后根据这个请求的具体类型执行一些具体逻辑，比如读锁信息，然后将最终要写入的数据写入 Raftstore。所以 Scheduler 本身是否达到瓶颈，一部分可以根据之前的读取流程排查获取 Snapshot 和读取锁信息这部分是否达到了瓶颈；另一部分是查看 Scheduler worker CPU 执行具体逻辑时 CPU 的负载是否较高。如果都没有问题，则进入下部分 Raftstore 的写入过程进一步排查。

## (6) TiKV Raftstore

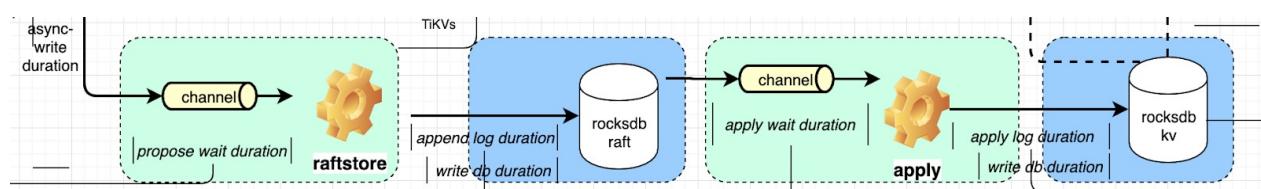
下图展示 Raftstore 在 TiKV 体系中所处的层级，同一个 Region 在一个集群中有多个副本。Raftstore 可以理解为使用 Raft 协议使一个 Region 的 Key-Value 数据在多个 RocksDB 中的保持一致。



Raftstore 整体上来看可以分为 Write Raft Log 和 Apply Raft Log 两个部分，其中包含大量的细节，简单地从以下维度理解：

1. IO : 两个 RocksDB 实例
  - Raft RocksDB 用于保存 Raft 日志
  - KV RocksDB 用于保存 Key-Value 数据
2. CPU : 两个线程池 (每个线程池默认为两个线程)
  - raft 线程池
  - apply 线程池
3. Network : leader 向 follower 同步日志

上层的写入请求会构造一个 PeerMsg::RaftCommand，里面包含写入的 Key-Value，并将 RaftCommand 通过 Router 发送到 raft 线程池，raft 线程将 Raft 日志写入到 Raft RocksDB 并同步给其他 Peer，当日志提交之后，apply 线程将数据写入 KV RocksDB，整体流程如下图所示：



相关的监控在 TiKV dashboard 中按照 Raft IO / Raft process / Raft messages / Raft propose 分类组织：

- Thread CPU
  - Raft store CPU : raft 线程的 CPU 使用率，通常应低于 80%

- Async apply CPU : apply 线程的 CPU 使用率，通常应低于 90%
- Raft propose
  - Raft proposals per ready : 在一个 mio tick 内，所有 Region proposal 的个数
  - Raft read/write proposals : 不同类型的 proposal 的个数
  - Propose wait duration : 发送请求给 raftstore，到 raftstore 真正处理请求之间的延迟。如果该延迟比较长，说明 raftstore 比较繁忙或者 append log 比较耗时导致 raftstore 不能及时处理请求
  - Apply wait duration : committed raft log 发送给 apply 线程到 apply 线程真正 apply 这些 log 之间的延迟。如果该延迟比较高说明 apply 线程比较繁忙。需要查看是否由于 apply CPU 过高导致，或者磁盘负载比较高导致
  - Raft log speed : peer propose 日志的速度
- Raft IO
  - Apply log duration : Raft apply 日志所花费的时间，执行 raft log 的耗时，apply log 是指把用户数据写入到 kvdb
  - Append log duration : Raft append 日志所花费的时间，写 raft log 的耗时。把 raft log 写入到 raftdb 中，建议小于 50ms，如果延时比较高，进一步排查 Raft RocksDB 慢的问题
  - Commit log duration : Raft 将日志发送到其他节点上，并收到过半的确认的时间
- Raft message
  - Sent messages per server : 每个 TiKV 实例发送 Raft 消息的个数
  - Flush messages per server : 每个 TiKV 实例持久化 Raft 消息的个数
  - Receive messages per server : 每个 TiKV 实例接受 Raft 消息的个数
  - Messages : 发送不同类型的 Raft 消息的个数
  - Vote : Raft 投票消息发送的个数
  - Raft dropped messages : 丢弃不同类型的 Raft 消息的个数

## (7) RocksDB 和 Titan

TiKV 中所有的持久化数据都会写入到 RocksDB 中，每个 TiKV 都会创建两个 RocksDB 实例：

- Raft RocksDB : 负责存储 Raft Log
- KV RocksDB : 负责存储 KV 数据以及 Region 的元信息

RocksDB 是一个 LSM-tree 结构，写入的文件被分为多个不同的 level，其写入过程以及相关的重点监控有：

1. 数据写入 Memtable 中，同时也会写入 WAL
  - Memtable size : 每个 CF 的 memtable 的大小
  - WAL sync operations : sync WAL 操作的个数
  - WAL sync duration : sync WAL 操作的耗时
  - Write operations : write 操作的个数
  - Write duration : write 操作的耗时
  - Write flow : 不同写操作的流量
2. 当 Memtable 到达容量设置上限时，会转换为 Immutable Memtable，异步刷入持久化设备中，最初写入的文件被放到第一个 level 中 (level0)
3. 当持久化设备中的文件到达设定的条件时，会进行 Compaction，重新组织数据
  - Compaction operations : Compaction 和 flush 操作的个数
  - Compaction duration : Compaction 和 flush 操作的耗时
  - Compaction flow : Compaction 相关的流量
  - Compaction pending bytes : 等待 Compaction 的大小

当 level0 文件数量 / Memtable 数量 / 等待 compaction 的数据大小到达上限时，会触发 RocksDB 的流控机制，称为 Write Stall，降低写入速度甚至停止写入，其相关的监控有：

- Write stall duration : 由于 write stall 造成的时间开销，正常情况下应为 0
- Stall conditions changed of each CF : 每个 CF stall 的原因

关于 Titan 的详细介绍见第 1 章 8 节。简单来说，Titan 为了解决 LSM-tree 在 Key-Value 较大时存在的写放大问题，将较大的 Value 存储到 Blob 文件中。对于 Titan，LSM-tree 部分监控与 RocksDB 部分的监控，我们需要关注的是 Blob 文件相关的监控：

- Blob file size : Blob 文件的大小
- Blob seek duration : Blob 文件中进行 seek 操作的耗时
- Blob get duration : Blob 文件进行 get 操作的耗时
- Blob bytes/keys flow : Blob 文件的写入流量

同时，Titan 也会对 Blob 文件进行 GC，相关的监控有：

- Blob GC file : Blob 文件 GC 的个数
- Blob GC duration : Blob 文件 GC 操作的耗时
- Blob GC bytes / keys flow : Blob 文件 GC 的流量

## 4.4 Prometheus 使用指南

本节将介绍监控工具 Prometheus 在 TiDB 中的应用，包括 Prometheus 本身的介绍以及如何通过 Prometheus 查看 TiDB 的监控和利用 Prometheus 的 alertmanager 进行告警。

### 4.4.1 Prometheus 简介

TiDB 使用开源时序数据库 Prometheus 作为监控和性能指标信息存储方案，使用 Grafana 作为可视化组件进行信息的展示。

Prometheus 狹义上是软件本身，即 prometheus server，广义上是基于 prometheus server 为核心的各类软件工具的生态。除 prometheus server 和 grafana 外，Prometheus 生态常用的组件还有 alertmanager、pushgateway 和非常丰富的各类 exporters。

prometheus server 自身是一个时序数据库，相比使用 MySQL 做为底层存储的 zabbix 监控，拥有非常高效的插入和查询性能，同时数据存储占用的空间也非常小。另外不同于 zabbix，prometheus server 中的数据是从各种数据源主动拉过来的，而不是客户端主动推送的。如果要使用 prometheus server 接收推送的信息，数据源和 prometheus server 中间需要使用 pushgateway。

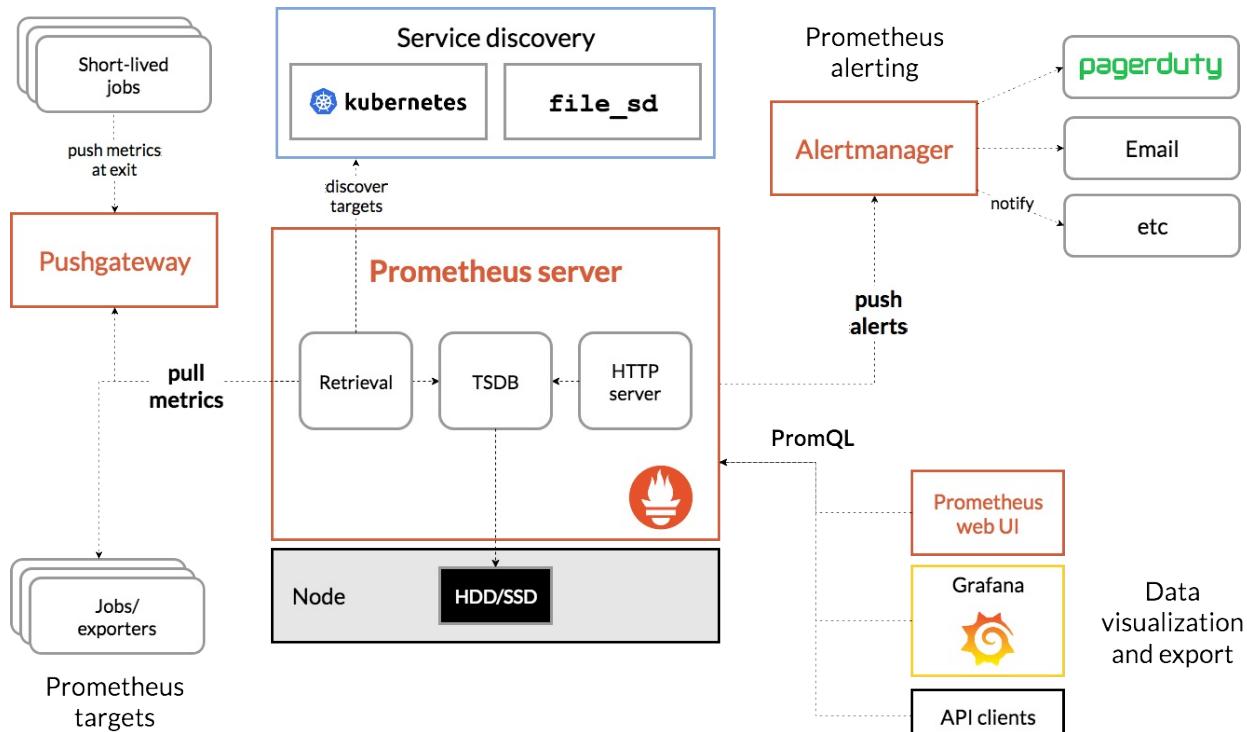
Prometheus 监控生态非常完善，能监控的对象非常丰富。详细的 exporter 支持对象可参考官方介绍 [exporters列表](#)。

Prometheus 可以监控的对象远不止官方 exporters 列表中的产品，有些产品原生支持不在上面列表，如 TiDB；有些可以通过标准的 exporter 来监控一类产品，如 snmp\_exporter；还有些可以通过自己写个简单的脚本往 pushgateway 推送；如果有一定开发能力，还可以通过自己写 exporter 来解决。同时有些产品随着版本的更新，不需要上面列表中的 exporter 就可以支持，比如 ceph。

随着容器和 kubernetes 的不断落地，以及更多的软件原生支持 Prometheus，相信很快 Prometheus 会成为监控领域的领军产品。

### 4.4.2 架构介绍

Prometheus 的架构图如下：



Prometheus 生态中 prometheus server 软件用于监控信息的存储、检索，以及告警消息的推送，是 Prometheus 生态最核心的部分。

Alertmanger 负责接收 prometheus server 推送的告警，并将告警经过分组、去重等处理后，按告警标签内容路由，通过邮件、短信、企业微信、钉钉、webhook 等发送给接收者。

大部分软件在用 Prometheus 作为监控时还需要部署一个 exporter 做为 agent 来采集数据，但是有部分软件原生支持 Prometheus，比如 TiDB 的组件，在不用部署 exporter 的情况下就可以直接采集监控数据。

PromQL 是 Prometheus 数据查询语言，用户可以通过 prometheus server 的 web UI，在浏览器上直接编写 PromQL 来检索监控信息。也可以将 PromQL 固化到 grafana 的报表中做动态的展示，另外用户还可以通过 API 接口做更丰富的自定义功能。

Prometheus 除了可以采集静态的 exporters 之外，还可要通过 service discovery 的方式监控各种动态的目标，如 kubernetes 的 node,pod,service 等。

除 exporter 和 service discovery 之外，用户还可以写脚本做一些自定义的信息采集，然后通过 push 的方式推送到 pushgateway，pushgateway 对于 prometheus server 来说就是一个特殊的 exporter，prometheus server 可以像抓取其他 exporters 一样抓取 pushgateway 的信息。

## 4.4.3 安装运行

Prometheus 可以运行在 kubernetes 中，也可以运行在虚拟机中。Prometheus 的大部分组件都已经有编译好的二进制文件和 docker 镜像。对于二进制文件，从官方网站下载解压后就可以启动运行，命令如下：

```
prometheus --config.file=conf/prometheus.yml
```

建议将二进制文件做成 systemd 的一个服务，这部分可以参考 [TiDB 上运行 prometheus 的方式](#)。

### 1. Prometheus server 配置文件

prometheus server 的配置文件是 yaml 格式，由参数 --config.file 指定需要使用的配置文件。配置文件一般命名为 prometheus.yml。

配置文件示例

```
global:
  scrape_interval: 15s
  scrape_timeout: 10s
  external_labels:
    monitor: 'codelab-monitor'

rule_files:
  - rules/centos7.rules.yml
  - rules/mariadb.rules.yml

alerting:
  alertmanagers:
    - static_configs:
      - targets:
          - 21.129.127.3:9093

scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'node'
    file_sd_configs:
      - files:
          - conf.d/centos.yml
```

配置文件说明：

- global: 指的是全局配置
- scrape\_interval: 抓取目标监控信息的间隔，默认 15 秒
- scrape\_timeout: 抓取时的超时时间，默认 10 秒
- external\_labels: 额外添加的标签，这个标签可以在多个外部系统流转，如：federation, remote storage, Alertmanager
- rule\_files: 这个写的是生成告警规则的配置文件，具体写法见 alertmanger 章节的介绍
- alerting: 这个是用来配置 alertmanager 地址，可以写多个 alertmanager 的地址
- scrape\_configs：从这开始，后面是采集对象的配置
- job\_name：可以定义多个 job，每个 job 里有一类的采集对象
- static\_configs: 后面可以写一些静态的监控对象
- targets：要抓取的具体对象 (instance)
- file\_sd\_configs: 如果监控对象过多，可使用这种方式写到独立的文件中

## 2. 告警规则配置示例

```

groups:
- name: alert.rules
  rules:
    - alert: InstanceDown
      expr: up == 0
      for: 1s
      labels:
        level: emergency
      annotations:
        summary: "该实例抓取数据超时"
        description: "项目： , service: " 当前值
  
```

告警配置说明：

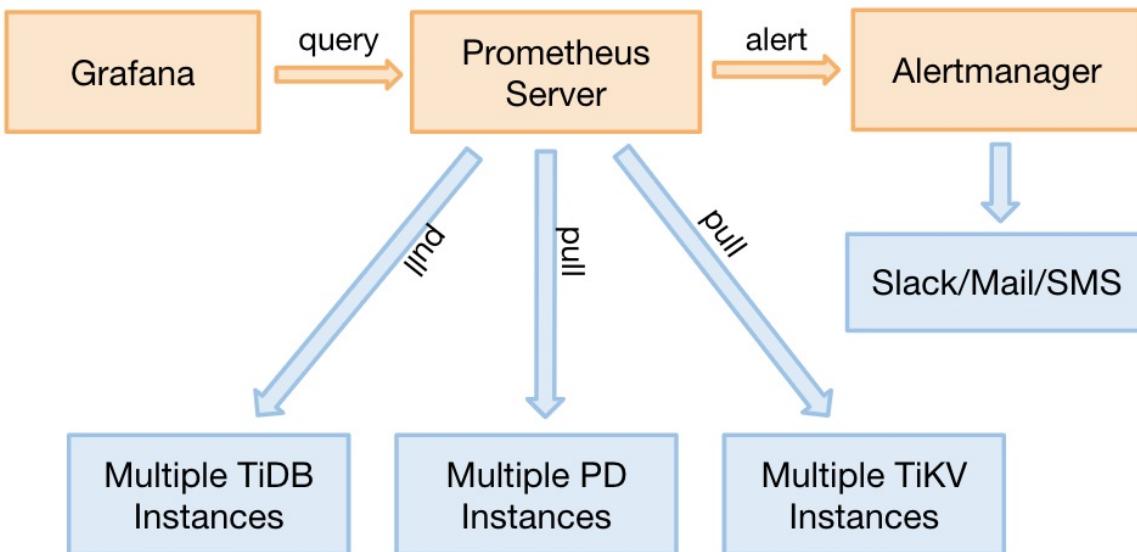
- groups：标记当前所有的告警规划为同一组
- name: 这告警组的自定义名称
- alert：告警规则的名称
- expr：告警的表达式
- for: 问题发生后保持多长时间再推送给 client，调低该值可以提高告警的敏感度，调高会减少告警毛刺
- labels：可以加一些自定义的键值对标签
- annotations: 可以加一些描述信息

### 4.4.4 Prometheus 在 TiDB 集群中的应用

本节介绍 Prometheus 在 TiDB 集群中的应用，主要包括通过 Prometheus PromQL 语言查看 TiDB 的监控，以及告警配置的讲解。

#### 1. TiDB 集群中 Prometheus 的部署架构

TiDB 已经原生支持 Prometheus，在 2.1 之前的版本，TiDB 的监控信息是由各 TiDB 的各个组件主动上报给 pushgateway，再由 prometheus server 去 pushgateway 上主动抓取监控信息。从 2.1 版本开始，TiDB 暴露 Metrics 接口，由 prometheus server 主动抓取信息，这样的架构更符合 Prometheus 的设计思想，整个数据采集路径少了一层 pushgateway。数据采集完成后由 grafana 做报表展示，同时告警信息主动推送给 alertmanager，再由 alertmanager 将告警推送到不同的消息渠道。



## 2. 通过 Prometheus PromQL 语言查看 TiDB 的监控

PromQL(Prometheus Query Language) 是 Prometheus 提供的函数查询语言，可以进行实时查询，也可以通过函数做聚合运算。本节介绍下如何通过 PromQL 对 TiDB 的监控信息进行查询。

### (1) 数据类型

Prometheus 中的数据类型分 4 类：

- Instant vector - 一个时间点的时序数据;
- Range vector - 一个时间段的时序数据;
- Scalar - 数字，浮点值;
- String - 字符串，当前还没有用。

### (2) 通过 web UI 执行查询

下图是在 web UI (<http://prometheus-server:9090/graph>) 上执行 `up{instance="21.129.14.103:2998"}` 表达式查询到的某个实例的存活状态。

The screenshot shows the Prometheus web interface with the following details:

- Header:** Prometheus, Alerts, Graph, Status, Help, Classic UI.
- Query Bar:** Enable query history (checked), Use local time (checked). Search input: `up{instance="21.129.14.103:2998"}`. Execute button.
- Result Panel:**
  - Table tab selected, Graph tab available.
  - Evaluation time: < > (with arrows).
  - Result: `up{alert_lev="0", instance="21.129.14.103:2998", job="hadoop", project="dtl", service="hadoop"}` with value 1.
  - Metrics listed: Load time: 51ms, Resolution: 14s, Result series: 1.
  - Buttons: Remove Panel, Add Panel.

### (3) 结果中各个字段的意义：

- `up`: 是一条具体的时序记录名字，同时 `up` 又是一条特殊的时序名称，他是 Prometheus 对每个监控对象自动生成的，指示该对象的起停状态，1 表示可连通，0 表示不可能连通（注意，0 不一定是服务挂了，也有可能是获取记录的时候超时了）。
- 表达式中的 `instance`, `job`, `project`, `service`, `alert_lev` 都是该条的记录的标签，相对于关系型数据库中的字段。其中 `instance` 和 `job` 是基于 `prometheus.yaml` 中的内容自动生成的，`project`, `service`, `alert_lev` 是用户自定义的标签。`instance` 一般是

prometheus 里的 target，但是也可以在标签里重写。

- 最后的 1 是这条记录在查询时的结果。

### 3. Instant vector 查询

下面列举下几种 Instant vector 查询的常见用法：

- 直接写时序名称，例如: server\_query\_total
- 在 {} 中加一些标签作为过滤条件，例如: server\_query\_total{job="tikv"}
- 一个标签匹配多个值，例如: server\_query\_total{job=~"tikv|tidb"}
- 指定需要过滤掉值，例如: server\_query\_total{job!~"tikv|pd"}
- 匹配正则表达式，例如: up{tidb=~".+"}，可以匹配所以包含 tidb 的 up 时序数据
- 使用算术运算和比较运算过滤结果: tikv\_engine\_bytes\_written{instance="21.129.14.104:21910"}/1024/1024 > 500

### 4. Range vector 查询

Range vector 查询类似于 instance vector 查询，不同之处在于通过 [] 加上时间范围限制，时间单位可以设置为：

- s - seconds
- m - minutes
- h - hours
- d - days
- w - weeks
- y - years

下面看看监控 TiDB QPS 的例子，展示的是 172.16.4.51:10080 这台 TiDB 实例的 QPS 情况：



### 5. offset 查询

通过 offset 能够查询过去某个时间点的监控结果，如下查询的是一天前 TiDB 的请求数总量：

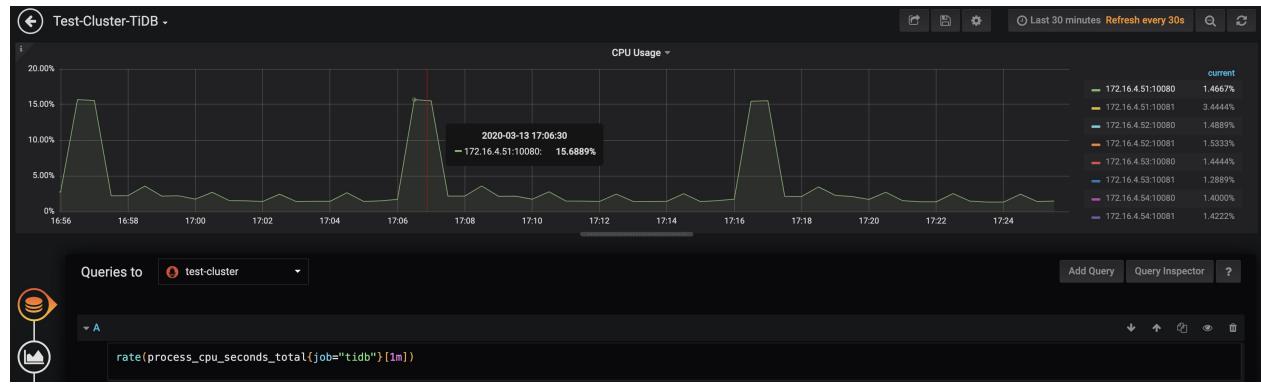
```
sum((tidb_server_query_total{result="OK"} offset 1d))
```

#### 4.4.5 TiDB 监控中常用函数

本节结合实际例子，介绍下 TiDB 监控中经常用到的一些函数。

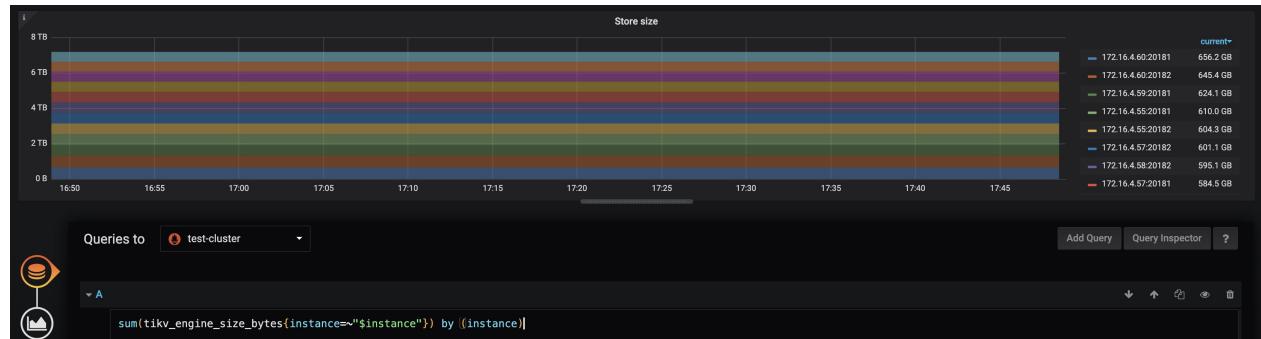
##### rate 和 irate

这两个函数一般作用于计数器 counter 类型的数据，这类数据会一直增加，使用这两个函数后，展示的是一定时间范围内的变化情况。但它俩的计算方式是有差异，irate() 是基于时间范围内连续的两个时间点，而 rate() 是基于时间范围内的所有时间点，所以 irate() 展示的数据更为精确些，做图毛刺也会更明显。下图展示的是 TiDB 集群中节点的 CPU 使用率的监控，对应的表达式是 rate(process\_cpu\_seconds\_total{job="tidb"}[1m])。



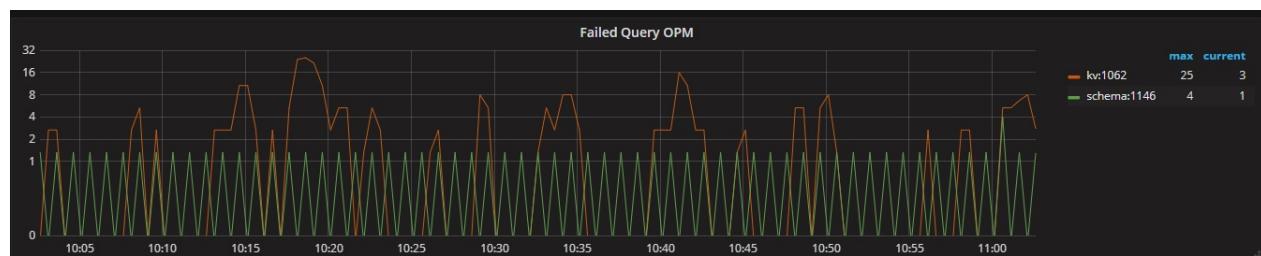
### sum 和 avg

sum 是求和函数，avg 是求均值函数。表达式 `sum(tikv_store_size_bytes{instance=~"$instance"}) by (instance)` 查询的是各个 TiKV 实例的容量总和。



### increase

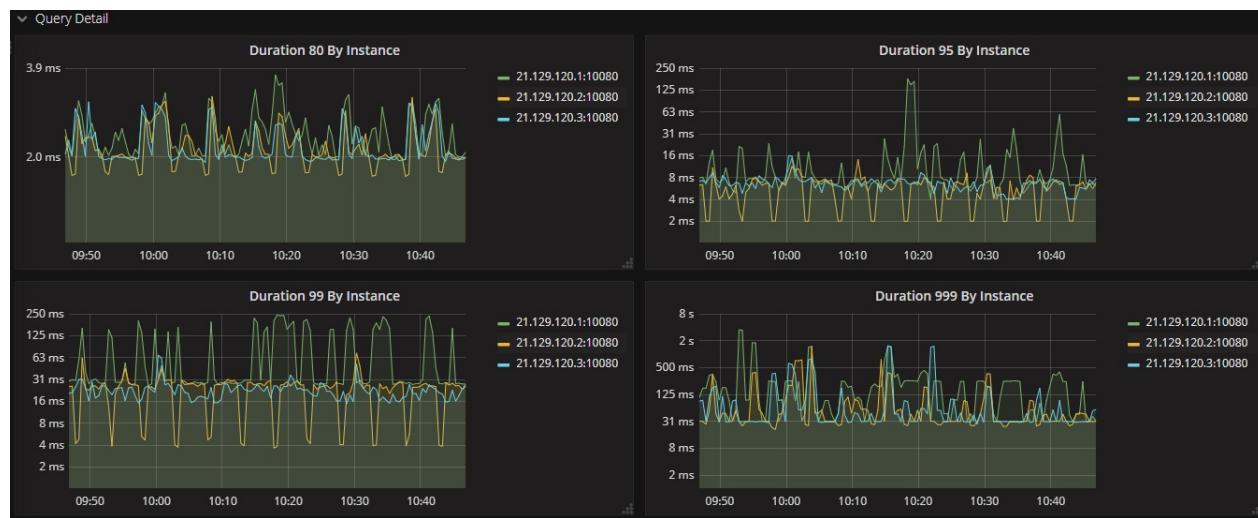
increase 函数计算的是指定时间范围内的变化量，例如表达式 `sum(increase(tidb_server_execute_error_total[1m])) by (type)` 是以 type 为聚合条件，显示 1 分钟内 Failed Query OPM 总数



### histogram\_quantile

histogram\_quantile 是累积直方图百分位函数，用法 `histogram_quantile(phi float, b instant-vector)`，其中百分位  $\phi$  是介于 0 和 1 之间的值。这个函数计算的结果是直方图中指定百分比的最大值，例如 0.95 的百分位的结果是 200，说明所有数据中，小于 200 的占总数据的比例为 95%。下面表达式是展示各个 tidb-server 请求的 99% 延迟情况。

```
histogram_quantile(0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket[1m])) by (le, instance))
```



## 4.4.6 通过配置 alertmanager 对 TiDB 故障进行报警

本节介绍下 TiDB 中是如何配置 Prometheus 的报警的。如果是通过 tidb-ansible 方式部署的集群，Prometheus 的报警配置文件对应的路径是 `tidb-ansible/roles/prometheus/files/tidb.rules.yml`。

### 1. TiDB 告警级别

TiDB 组件的报警项，根据严重级别可分为三类，按照严重程度由高到低依次为：紧急级别、重要级别、警告级别。

#### 紧急级别报警项

紧急级别的报警通常由于服务停止或节点故障导致，此时需要马上进行人工干预操作。告警规则里的标签 `level: emergency`。下面展示的是 `TiDB_schema_error` 的告警示例：TiDB 在一个 Lease 时间内没有重载到最新的 Schema 信息，导致 TiDB 无法继续对外提供服务，需要报警。该问题通常由于 TiKV Region 不可用或超时导致，需要看 TiKV 的监控指标定位问题，比如确认 TiKV 实例是否还存活着。

```
- alert: TiDB_schema_error
  expr: increase(tidb_session_schema_lease_error_total{type="outdated"}[15m]) > 0
  for: 1m
  labels:
    env: ENV_LABELS_ENV
    level: emergency
  expr:  increase(tidb_session_schema_lease_error_total{type="outdated"}[15m]) > 0
  annotations:
    description: 'cluster: ENV_LABELS_ENV, instance: , values:'
    value: ''
  summary: TiDB schema error
```

#### 重要级别报警项

对于重要级别的报警，需要密切关注异常的指标。告警规则里的标签 `level: critical`。下面示例展示的是 `tidb-server` 进程发生崩溃的时候进行报警。收到该报警的一般处理方式是收集 TiDB 的 panic 日志，定位 panic 的原因，比如是否是 `tidb-server` 实例 OOM 导致的问题。

```
- alert: TiDB_server_panic_total
  expr: increase(tidb_server_panic_total[10m]) > 0
  for: 1m
  labels:
    env: ENV_LABELS_ENV
    level: critical
  expr:  increase(tidb_server_panic_total[10m]) > 0
  annotations:
    description: 'cluster: ENV_LABELS_ENV, instance: , values:'
    value: ''
  summary: TiDB server panic total
```

### 警告级别报警项

警告级别的报警是对某一问题或错误的提醒。告警规则里的标签 level: warning。下面展示的是对于 tidb-server 实例内存异常的报警，当 tidb-server 实例的内存占用大于 10GB 的时候进行报警。收到该报警的时候，需要注意是否有大查询在执行，比如大表的 Join 查询。

```
- alert: TiDB_memory_abnormal
  expr: go_memstats_heap_inuse_bytes{job="tidb"} > 1e+10
  for: 1m
  labels:
    env: ENV_LABELS_ENV
    level: warning
  annotations:
    description: 'cluster: ENV_LABELS_ENV, instance: , values:'
    value: ''
  summary: TiDB heap memory usage is over 10 GB
```

更多关于 TiDB 报警规划，以及 TiDB 详细告警的处理方法，请参考 [官网介绍](#)。

## 2. 为 TiDB 集群配置 alertmanager 告警路由

由于往外发送告警需要邮箱、短信、企业微信等外部消息通道打通，一般企业内部都有各自不同的安全要求和操作规范。另外像短信接口并不是统一标准的，大部分也不是原生支持 Prometheus 的，所以需要用户自己编写适配脚本，以 webhook 的方式与 alertmanger 进行适配。

建议使用 TiDB 时，用户自己创建一个独立的 alertmanager，用于接收来自不同 prometheus server 的告警，统一集中路由发送，既可以有效安全管理，也可以减少用户自己的部署操作。如果用户采用的是 tidb-ansible 方式部署的 TiDB 集群，alertmanager 的配置文件位于 tidb-ansible/conf/alertmanager.yml。

### 告警路由配置

```
routes:
- match:
  env: test-cluster
  level: emergency
  receiver: tidb-emergency
  group_by: [alertname, cluster, service]
```

下面简单解释下各个字段的含义：

- match 是一条路由规划。
- env 和 level 是从 Prometheus 发送过来的记录所携带的标签，如果能够匹配该标签，则符合当前这条路由规则。
- receiver 表示接收人，和后面接收部分的 name 一致。
- group\_by 里面的内容是分组聚合的标签。

### 告警接收配置

```
receivers:
- name: 'tidb-emergency'
  webhook_configs:
  - url: 'xxxx'
  wechat_configs:
  - corp_id: 'xxxxx'
    to_party: 'xxx'
    agent_id: 'xxxx'
    api_url: 'https://qyapi.weixin.qq.com/cgi-bin/'
    api_secret: 'xxxxxx'
```

- name：和上面路由规则的 receiver 对应。
- webhook\_configs：以 webhook 的方式发送。
- wechat\_configs：以企业微信的方式发送，具体要求参考 [企业微信](#) 文档；

- 由于默认的告警发送的内容过多，包含注释等信息，影响可读性。建议用户自己写 webhook 的方式发送告警。

## 第5章 灾难快速恢复

数据库是每个公司的重中之重，他往往存储了公司的核心数据，一旦出现永久性损坏，对公司的打击会是致命的。

数据损坏的原因可能是多种多样的，本章针对不同的场景以及相应的处理办法，做出了详细解答，例如：

- 用户在处理数据的过程中，执行了错误的更新、删除操作。这时在 TiDB 中如何快速恢复误操作前的数据，请参考 [【5.1 利用 GC 快照读恢复数据】](#)
- 如果直接执行了 drop table，或者 truncate table，将整张表瞬间删除，这时如何恢复，请参考 [【5.2 利用 Recover/Flashback 命令秒恢复误删表】](#)
- TiKV 虽然已有高可用方案，采用 3 副本的方式冗余数据，但如果特殊意外丢失了 2 副本该如何恢复，请参考 [【5.3 多数副本丢失数据恢复指南】](#)

## 5.1 利用 GC 快照读恢复数据

当遇到数据被误更新或误删除的情况，很多人想到的是 Oracle 的闪回或者 MySQL 的基于 binlog 实现的闪回工具。作为 NewSQL 的佼佼者，TiDB 可以直接通过标准 SQL 读取历史数据，无需特殊的 client 或者 driver。即使在更新/删除数据后，表结构发生了变化，TiDB 依旧能够按照旧的表结构定义将数据读取出来。

### 5.1.1 历史数据保留策略

TiDB 事务的实现采用了 MVCC（多版本并发控制）机制，当更新/删除数据时，不会做真正的数据删除，只会添加一个新版本数据，并以时间戳来区分版本。当然历史数据不会永久保留，超过一定时间的历史数据将会被彻底删除，以减小空间占用，同时避免因历史版本过多引起的性能开销。

TiDB 使用周期性运行的 GC（Garbage Collection，垃圾回收）来进行清理，默认情况下每 10 分钟一次。每次 GC 时，TiDB 会计算一个称为 safe point 的时间戳（默认为上次运行 GC 的时间减去 10 分钟），接下来 TiDB 会在保证在 safe point 之后的快照都能够读取到正确数据的前提下，删除更早的过期数据。

TiDB 的 GC 相关的配置存储于 mysql.tidb 系统表中，可以通过 SQL 语句对这些参数进行查询和更改：

```
select VARIABLE_NAME, VARIABLE_VALUE from mysql.tidb where VARIABLE_NAME like 'tikv_gc%';

+-----+-----+
| VARIABLE_NAME | VARIABLE_VALUE
+-----+-----+
| tikv_gc_leader_uuid | 5afd54a0ea40005
| tikv_gc_leader_desc | host:tidb-cluster-tidb-0, pid:215, start at 2019-07-15 11:09:14.029668932 +0000 UTC m=+0
.463731223 |
| tikv_gc_leader_lease | 20190715-12:12:14 +0000
| tikv_gc_enable | true
| tikv_gc_run_interval | 10m0s
| tikv_gc_life_time | 10m0s
| tikv_gc_last_run_time | 20190715-12:09:14 +0000
| tikv_gc_safe_point | 20190715-11:59:14 +0000
| tikv_gc_auto_concurrency | true
| tikv_gc_mode | distributed
+-----+
-----+
13 rows in set (0.00 sec)
```

例如，如果需要将 GC 调整为保留最近一天以内的数据，只需执行下列语句即可：

```
update mysql.tidb set VARIABLE_VALUE="24h" where VARIABLE_NAME="tikv_gc_life_time";
```

注意：mysql.tidb系统表中除了下文将要列出的GC的配置外，还包含一些TiDB用于储存部分集群状态（包括GC状态）的记录。请勿手动更改这些记录。其中，与GC有关的记录如下：

- . tikv\_gc\_leader\_uuid , tikv\_gc\_leader\_desc 和 tikv\_gc\_leader\_lease 用于记录 GC leader 的状态
- . tikv\_gc\_last\_run\_time : 上次 GC 运行时间
- . tikv\_gc\_safe\_point : 当前 GC 的 safe point
- . tikv\_gc\_life\_time: 用于配置历史版本保留时间，可以手动修改
- . tikv\_gc\_safe\_point: 记录了当前的 safe point，用户可以安全地使用大于 safe point 的时间戳创建 Snapshot 读取历史版本。safe point 在每次 GC 开始运行时自动更新。

## 5.1.2 查询历史数据

读取历史版本数据前，需设定一个系统变量: tidb\_snapshot，这个变量是 Session 范围有效级别，可以通过标准的 Set 语句修改其值。其值可以是 TSO 或日期时间。TSO 是全局授时的时间戳，是从 PD 端获取的；日期时间的格式可以为：“2016-10-08 16:45:26.999”，一般来说可以只写到秒，比如“2016-10-08 16:45:26”。当这个变量被设置后，TiDB 会用这个时间戳建立 Snapshot（没有开销，只是创建数据结构），之后所有的查询操作都会在这个 Snapshot 上读取数据。

注意：TiDB 的事务是通过 PD 进行全局授时，所以存储的数据版本也是以 PD 所授时间戳作为版本号。在生成 Snapshot 时，是以 tidb\_snapshot 变量的值作为版本号，如果 TiDB Server 所在机器和 PD Server 所在机器的本地时间相差较大，需要以 PD 的时间为基准。

当读取历史版本数据操作结束后，可以结束当前 Session 或者是通过 Set 语句将 tidb\_snapshot 变量的值设为 “”，即可读取最新版本的数据。

## 5.1.3 查询历史数据示例：

1. 初始化阶段，创建一个表，并插入几行数据：

```
create table t (c int);
Query OK, 0 rows affected (0.01 sec)

insert into t values (1), (2), (3);
Query OK, 3 rows affected (0.00 sec)
```

2. 查看表中的数据：

```
select * from t;
+---+
| c |
+---+
| 1 |
| 2 |
| 3 |
+---+
3 rows in set (0.00 sec)
```

3. 查看当前时间：

```
select now();
+-----+
| now()           |
+-----+
| 2016-10-08 16:45:26 |
+-----+
1 row in set (0.00 sec)
```

## 4. 更新某一行数据：

```
update t set c=22 where c=2;
Query OK, 1 row affected (0.00 sec)
```

## 5. 确认数据已经被更新：

```
select * from t;
+----+
| c |
+----+
| 1 |
| 22 |
| 3 |
+----+
3 rows in set (0.00 sec)
```

## 6. 设置一个特殊的环境变量，这个是一个 session scope 的变量，其意义为读取这个时间之前的最新的一个版本。

```
set @@tidb_snapshot="2016-10-08 16:45:26";
Query OK, 0 rows affected (0.00 sec)
```

注意：这里的时间设置的是 update 语句之前的那个时间。在 tidb\_snapshot 前须使用 @@ 而非 @，因为 @@ 表示系统变量，@ 表示用户变量。

这里读取到的内容即为 update 之前的内容，也就是历史版本：

```
select * from t;
+----+
| c |
+----+
| 1 |
| 2 |
| 3 |
+----+
3 rows in set (0.00 sec)
```

## 7. 清空这个变量后，即可读取最新版本数据：

```
set @@tidb_snapshot="";
Query OK, 0 rows affected (0.00 sec)

select * from t;
+----+
| c |
+----+
| 1 |
| 22 |
| 3 |
+----+
3 rows in set (0.00 sec)
```

注意：在 tidb\_snapshot 前须使用 @@ 而非 @，因为 @@ 表示系统变量，@ 表示用户变量。

## 5.1.4 恢复被更新/删除的数据

通过读取历史数据可以快速恢复被更新/删除的数据，大致步骤如下：

注意：此方法仅适用变化的数据量较少的情况，进行恢复时需要调整gc的生命周期。

- 1、调整 GC 保留时间，如将 GC 调整为保留最近一天以内的数据。

```
update mysql.tidb set VARIABLE_VALUE="24h" where VARIABLE_NAME="tikv_gc_life_time";
```

说明：具体保留多长时间，需要结合业务进行评估

2、创建一个与待恢复的数据表同结构的临时表，如：

```
set @@tidb_snapshot="2016-10-08 16:45:26";
create table t_20161008 like t;
```

3、按照业务逻辑将需要的数据插入到临时表：

```
insert into t_20161008 select * from t where c=2;
```

4、按照业务逻辑将数据从临时表反更新或插入到原表

5、按照业务逻辑校验数据

6、将 GC 保留时长调整为恢复之前的设置

```
update mysql.tidb set VARIABLE_VALUE="10m0s" where VARIABLE_NAME="tikv_gc_life_time"
```

7、根据需要删除临时表

## 5.2 利用 Recover/Flashback 命令秒恢复误删表

对于 DBA 来说，删库跑路永远是被调侃的眼，近在眼前的某微商平台的大面积宕机事件给企业管理人员及数据库运维团队带来的启示仍在被不断讨论。当然为了应对大面积的恶意删库跑路行为，可能真的是防不胜防，不过这种情况也非常罕见。但是大家尤其是线上运维 DBA 在维护数据库的过程中，出现删错表/库的情况却是很容易出现的，下面来看一下 TiDB 提供的快速恢复误删表的功能。

### 5.2.1 Recover 实现原理

TiDB 在删除表时，实际上只删除了表的元信息，并将需要删除的表数据范围（行数据和索引数据）写一条数据到 mysql.gc\_delete\_range 表。TiDB 后台的 GC Worker 会定期从 mysql.gc\_delete\_range 表中取出超过 GC lifetime 相关范围的 key 进行删除。

所以，RECOVER TABLE 只需要在 GC Worker 还没删除表数据前，恢复表的元信息并删除 mysql.gc\_delete\_range 表中相应的行记录就可以了。恢复表的元信息可以用 TiDB 的快照读实现，TiDB 中表的恢复是通过快照读获取表的元信息后，再走一次类似于 CREATE TABLE 的建表流程，所以 RECOVER TABLE 实际上也是一种 DDL。

### 5.2.2 简单实践

```
MySQL [test]> show tables;

+-----+
| Tables_in_test |
+-----+
| t1           |
| t3           |
+-----+

**2 rows in set (0.00 sec)**

MySQL [test]> create table t2 like t1;

**Query OK, 0 rows affected (0.10 sec)**

MySQL [test]> insert into t2 select * from t1;

**Query OK, 524288 rows affected (17.10 sec)**

**Records: 524288  Duplicates: 0  Warnings: 0**

MySQL [test]> show tables;

+-----+
| Tables_in_test |
+-----+
| t1           |
| t2           |
| t3           |
+-----+

**3 rows in set (0.00 sec)**
```

执行 DDL 删除 t2 表。

```

MySQL [test]> drop table t2;

**Query OK, 0 rows affected (0.24 sec)**

MySQL [test]> show tables;

+-----+
| Tables_in_test |
+-----+
| t1           |
| t3           |
+-----+

**2 rows in set (0.00 sec)**

MySQL [test]> select * from t2 limit 1;

ERROR 1146 (42S02): Table 'test.t2' doesn't exist

```

这个时候执行 recover 操作

```

MySQL [test]> recover table t2;

**Query OK, 0 rows affected (1.17 sec)**

MySQL [test]> show tables;

+-----+
| Tables_in_test |
+-----+
| t1           |
| t2           |
| t3           |
+-----+

**3 rows in set (0.00 sec)**

MySQL [test]> select count(1) from t2;

+-----+
| count(1) |
+-----+
| 524288   |
+-----+

**1 row in set (0.55 sec)**

```

可以看到table t2 已经回来了。当然根据前面的原理介绍，如果超过 gc 时间，这里就会返回下面错误

```

MySQL [test]> recover table t2;

ERROR 8055 (HY000): snapshot is older than GC safe point 2020-03-08 15:06:19 +0800 CST

```

另外还有一种用法，可以通过查询 ddl jobs 队列，指定 job id 的方式进行恢复。

```
MySQL [test]> ADMIN SHOW DDL JOBS;

+-----+-----+-----+-----+-----+-----+
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE | SCHEMA_ID | TABLE_ID | ROW_COUNT | START_TIME
|        |          |           |           |           |           |           |           |           |
+-----+-----+-----+-----+-----+-----+
|    73 | test     | t2         | drop table   | none       | 1          | 71        | 0          | 2020-03-08 15:38:
47.076 +0800 CST | 2020-03-08 15:38:47.276 +0800 CST | synced |
|    72 | test     | t2         | create table | public     | 1          | 71        | 0          | 2020-03-08 15:38:
37.626 +0800 CST | 2020-03-08 15:38:37.726 +0800 CST | synced |
|    70 | test     | t2         | drop table   | none       | 1          | 66        | 0          | 2020-03-08 15:38:
20.576 +0800 CST | 2020-03-08 15:38:20.776 +0800 CST | synced |
+-----+-----+-----+-----+-----+-----+
```

从 DDL 记录可以看到，我们一共执行了 2 次 drop table t2 的操作，如果只执行 recover table t2 恢复的是最近一次删除的表。如果我希望恢复前面一次删除的动作，就需要用到下面的命令

```
MySQL [test]> recover table by job 70;

**Query OK, 0 rows affected (1.15 sec)**
```

### 5.2.3 使用限制

recover table 的一些使用限制：

- 只能用来恢复误删除表的 DDL 操作，例如 truncate table，delete 操作没有办法恢复。
- 只能在 GC 回收数据之前完成，超过 GC 时间后会报错无法成功恢复。
- 如果在使用 binlog 的情况下，上游执行 recover table 可能会出现一些非预期的结果。例如下游是 MySQL 数据库，对于这个语法不兼容。上下游的 GC 策略配置不同，再加上复制延迟可能会引起下游的数据在 apply recover table 的时候已经被 GC 了从而导致语句执行失败。

### 5.2.4 Flashback 介绍

Flashback 的原理和 Recover table 比较类似，不过是 recover 的升级版，在覆盖 recover 的所有功能之外，还可以支持 truncate table 的操作，未来也会逐渐取代 recover 命令。下面是简单的使用示例，其他不再累述。

```
MySQL [test]> show tables;

+-----+
| Tables_in_test |
+-----+
| t1           |
| t2           |
| t3           |
+-----+

**3 rows in set (0.00 sec)**

MySQL [test]> truncate table t2;

**Query OK, 0 rows affected (0.11 sec)**

MySQL [test]> flashback table t2 to t4;

**Query OK, 0 rows affected (1.16 sec)**

MySQL [test]> show tables;

+-----+
| Tables_in_test |
+-----+
| t1           |
| t2           |
| t3           |
| t4           |
+-----+

**4 rows in set (0.00 sec)**

MySQL [test]> select count(1) from t2;

+-----+
| count(1) |
+-----+
|      0   |
+-----+

**1 row in set (0.01 sec)**

MySQL [test]> select count(1) from t4;

+-----+
| count(1) |
+-----+
|  524288  |
+-----+

**1 row in set (0.39 sec)**
```

需要注意的是，目前 flashback 命令还不支持指定 ddl job id 来恢复表。

## 5.3 多数副本丢失数据恢复指南

### 5.3.1 问题背景

TiDB 默认配置为 3 副本，每一个 Region 都会在集群中保存 3 份，它们之间通过 Raft 协议来选举 Leader 并同步数据。Raft 协议可以保证在数量小于副本数（注意，不是节点数）一半的节点挂掉或者隔离的情况下，仍然能够提供服务，并且不丢失任何数据。对于 3 副本集群，挂掉一个节点除了可能会导致性能有抖动之外，可用性和正确性理论上不会受影响；但是挂掉 2 个副本，一些 region 就会不可用，而且如果这 2 个副本无法完整地找回了，还存在永久丢失部分数据的可能。这里主要讨论 2 个或 3 个副本丢失的问题。

在实际生产环境中，TiDB 集群是可能会出现丢失数据情况，如：

- 一个 TiDB 集群可能会出现多台 TiKV 机器短时间内接连故障且无法短期内恢复
- 一个双机房部署的 TiDB 集群的其中一个机房整体故障等

在上述这些情形下，会出现部分 Region 的多个副本（包含全部副本的情况）同时故障，进而导致 Region 的数据部分或全部丢失的问题。这个时候，最重要的是快速地最大程度地恢复数据并恢复 TiDB 集群正常服务。

### 5.3.2 副本数据恢复思路简析

副本数据恢复包含两个部分：故障 Region 处理和丢失数据处理

- 故障 Region 处理，针对 Region 数据丢失的严重情况，可分为两种：
  - Region 至少还有 1 个副本，恢复思路是在 Region 的剩余副本上移除掉所有位于故障节点上的 Peer，这样可以用这些剩余副本重新选举和补充副本恢复，但这些剩余副本中可能不包含最新的 Raft Log 更新，这个时候就会丢失部分数据
  - Region 的所有副本都丢失了，这个 Region 的数据就丢失了，无法恢复。可以通过创建 1 个空 Region 来解决 Region 不可用的问题
  - 在恢复 Region 故障的过程中，要详细记录下所处理 Region 的信息，如 Region ID、Region 丢失副本的数量等
- 丢失数据处理
  - 根据故障 Region ID 找到对应的表，找到相关用户并询问用户在故障前的某一段时间内（比如 5 min），大概写入了哪些数据表，是否有 DDL 操作，是否可以重新消费更上游的数据来再次写入，等等
  - 如果可以重导，则是最简单的处理方式。否则的话，则只能对重要的数据表，检查数据索引的一致性，保证还在的数据是正确无误的

### 5.3.3 故障 Region 的处理操作步骤

故障 Region 处理步骤包括：处理前禁用 PD 调度、处理还有剩余副本的 Region、处理所有副本都丢失的 Region、处理后恢复 Region 调度。

- 处理前后的 PD 调度处理
  - 为将恢复过程中可能的异常情况降到最少，需在故障处理期间禁用相关的调度：
    - 通过 `pd-ctl config get` 获取 `region-schedule-limit`、`replica-schedule-limit`、`leader-schedule-limit`、`merge-schedule-limit` 这 4 个参数的值，并记录下来用于后面恢复设置
    - 通过 `pd-ctl config set` 将这 4 个参数设为 0
    - 处理完之后需要将这 4 个参数进行恢复
- 处理还有剩余副本的 Region
  - 使用 `pd-ctl` 检查大于等于一半副本数在故障节点上的 Region，并记录它们的 ID（假设故障节点为 1, 4, 5）：

```
pd-ctl -u <endpoint> -d region --jq='$.regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length as $total | map(if .==(1,4,5) then . else empty end) | length>=$total-length) }'
```

- 根据上面的 Region 的个数，可以采取 2 种不同的解决方式（运行以下命令时需关闭相应 Store 上面的 TiKV）：

- Region 比较少，则可以在给定 Region 的剩余副本上，移除掉所有位于故障节点上的 Peer，在这些 Region 的未发生掉电故障的机器上运行：

```
tikv-ctl --db /path/to/tikv-data/db unsafe-recover remove-fail-stores -s <s1,s2> -r <r1,r2,r3>
```

- 反之，则可以在所有未发生掉电故障的实例上，对所有 Region 移除掉所有位于故障节点上的 Peer，在所有未发生掉电故障的机器上运行：

```
tikv-ctl --db /path/to/tikv-data/db unsafe-recover remove-fail-stores -s <s1,s2> --all-regions
```

- 执行后所有仍然有副本健在的 Region 都可以选出 Leader

- 处理所有副本都丢失的 Region

- 重启 PD，重启 TiKV 集群，使用 `pd-ctl` 检查没有 Leader 的 Region：

```
pd-ctl -u <endpoint> -d region --jq '.regions[]|select(has("leader")|not)|{id: .id, peer_stores: [.peers[]|.store_id]}'
```

- 创建空 Region 解决 Unavailable 报错。任选一个 Store，关闭上面的 TiKV，然后执行：

```
tikv-ctl --db /path/to/tikv-data/db recreate-region --pd <endpoint> -r <region_id>
```

# 第6章 TiDB Operator 故障诊断

本文将介绍 Kubernetes 上 TiDB 集群的一些常见故障以及诊断解决方案。

## 6.1 创建集群

### 6.1.1 镜像拉取失败

#### 问题分析

由于网络限制访问相关镜像仓库，部署 Kubernetes 时常拉取失败，给使用带来了较大不便。

#### 解决方案

主要通过镜像和代理的方式拉取，这里推荐一个好用的开源工具 docker-wrapper 方便手动拉取。

```
git clone https://github.com/silenceshell/docker-wrapper.git
cd docker-wrapper/
# 测试拉取 k8s.gcr.io 的一个例子
./docker-wrapper.py pull k8s.gcr.io/kube-apiserver:v1.16.0
-- pull k8s.gcr.io/kube-apiserver:v1.16.0 from gcr.azk8s.cn/google-containers/kube-apiserver:v1.16.0 instead --
...
-- pull k8s.gcr.io/kube-apiserver:v1.16.0 done --
```

如果是生产环境，推荐采用 Kubernetes 的一些管理平台来部署和管理 Kubernetes 集群，同时在管理平台上设置镜像代理。如果采用镜像拉取到内网 image registry 的话，TiDB 的 helm chart 要更改对应的 image 设置。

### 6.1.2 PD 处于 Running 运行中状态，但 TiKV 未开始创建

#### 问题分析

Pod 的状态处于 Running，表示容器已经绑定到一个节点，同时 Pod 中所有容器都已被创建。此时容器可能处于正在运行中、或者正处于启动或重启中。

PD 启动逻辑：Pod 启动时依赖启动脚本 pd\_start\_script.sh，通过 Service DNS 循环调用 discovery 进行注册及获得启动参数，discovery 也通过 Service DNS 访问 PD 获取最新的集群 members 地址。

由于无明显表象，只能通过日志等方式深入容器内部排查。

#### 解决方案

##### 查看 Pod 内容器的日志

```
# 查看所有 pods
kubectl get pods -n <namespace>
# 查看 events 是否有异常
kubectl describe -n <namespace> pods <pd pod-name>
# 查看相关日志，检查是否有 nslookup 解析失败的异常问题
kubectl logs -n <namespace> pods <pd pod-name>
# 管理工具
tkctl debug <pd-name>
```

根据经验主要是网络相关问题引起，建议从如下两方面进行排查：

- DNS 解析是否正常

```
# 检查 DNS 是否成功解析
kubectl exec -it <discovery pod name> -n=<namespace> -- nslookup <service dns>
# 成功解析 headless service dns 的示例
$ kubectl exec -it tidb-cluster-discovery-6b7f8d9954-48ngh -n=tidb -- nslookup tidb-cluster-pd-peer.tidb.svc
Name:      tidb-cluster-pd-peer.tidb.svc
Address 1: 10.244.2.19 tidb-cluster-pd-0.tidb-cluster-pd-peer.tidb.svc.cluster.local
Address 2: 10.244.1.13 10-244-1-13.tidb-cluster-pd.tidb.svc.cluster.local
Address 3: 10.244.3.11 10-244-3-11.tidb-cluster-pd.tidb.svc.cluster.local
# 成功解析 cluster ip service dns 的示例
$ kubectl exec -it tidb-cluster-discovery-6b7f8d9954-48ngh -n=tidb -- nslookup tidb-cluster-prometheus.tidb.svc
Name:      tidb-cluster-prometheus.tidb.svc
Address 1: 10.107.103.198 tidb-cluster-prometheus.tidb.svc.cluster.local
```

若 DNS 解析失败，请检查是否已成功启动 CoreDNS 组件或配置是否正确。

```
# 失败解析的示例
nslookup: can't resolve 'tidb-cluster-prometheus.tidb.svc1': Name does not resolve
command terminated with exit code
# 检查 CoreDNS 组件是否正常运行
kubectl get pods -n kube-system | grep core
coredns-9d85f5447-642rs     1/1     Running   1            13h
coredns-9d85f5447-8swr2     1/1     Running   1            13h
# 若 CoreDNS 正常运行仍不能成功解析 DNS
# 登陆 pd pod, 检查 /etc/resolv 中的 nameserver 是否配置正确
kubectl exec -it tidb-cluster-pd-0 -n=tidb -- cat /etc/resolv.conf
```

- IP 网络是否连通

详见 1.5 Pod 之间网络不通

### 6.1.3 容器处于 CrashLoopBackOff 状态

#### 问题分析

Pod 处于 CrashLoopBackOff 状态意味着 Pod 内的容器重复地异常退出。可能导致 CrashLoopBackOff 的原因有很多，最有效的定位办法是查看 Pod 容器的日志

#### 解决方案

- 查看 Pod 内容器的日志

```
# 寻找容器进程退出或者健康检查失败退出相关日志
kubectl -n <namespace> logs -f <pod-name>
# 当前 Pod 可能发生了多次 CrashLoopBackoff, 根因可能在上一次的启动日志的情况
kubectl -n <namespace> logs -p <pod-name>
```

确认日志中的错误信息后，可以根据 [tidb-server 启动报错](#)，[tikv-server 启动报错](#)，[pd-server 启动报错](#) 中的指引信息进行进一步排查解决

- 案例一：TiKV Pod 日志中出现“cluster id mismatch”

TiKV Pod 使用的数据可能是其他或之前的 TiKV Pod 的旧数据。在集群配置本地存储时未清除机器上本地磁盘上的数据，或者强制删除了 PV 导致数据并没有被 local volume provisioner 程序回收，可能导致 PV 遗留旧数据，导致错误。

在确认该 TiKV 应作为新节点加入集群、且 PV 上的数据应该删除后，可以删除该 TiKV Pod 和关联 PVC。TiKV Pod 将自动重建并绑定新的 PV 来使用。集群本地存储配置中，应对机器上的本地存储删除，避免 Kubernetes 使用机器上遗留的数据。集群运维中，不可强制删除 PV，应由 local volume provisioner 程序管理。用户通过创建、删除 PVC 以及设置 PV 的 reclaimPolicy 来管理 PV 的生命周期。

- 案例二：Node 节点 ulimit 设置不足

```

root      soft   nofile     1000000
root      hard   nofile     1000000
root      soft   core       1048576
root      hard   core       1048576

```

TiKV 在 ulimit 不足时也会发生启动失败的状况，对于这种情况，可以修改 Kubernetes 节点的 /etc/security/limits.conf 调大 ulimit

- 案例三：部分 pd 的 pod 反复 CrashLoopBackOff

在节点维护或者自动故障转移的场景下，部分 pd 的 pod 反复 CrashLoopBackOff。查看异常 pod 的日志可见：

```
waiting for discovery service to return start args...
```

可在 Kubernetes 内部尝试访问 discovery 的 service：

```
kubectl get service -n {your namespace} | grep discovery //get ip port
telnet {ip} {port}
```

如果访问失败，可尝试重启 discovery 服务

```
kubectl delete po {clusternamespace}-discovery-xxxxxxxx-xxxxx -n {your namespace}
```

如果重启后问题仍然存在，可尝试查看 discovery 的 pod 日志：

```
kubectl logs -f {clusternamespace}-discovery-xxxxxxxx-xxxxx -n {your namespace}
```

如有类似日志：

```
E1218 15:49:00.395064      1 mux.go:58] failed to discover: {cluster-name}-pd-3.{cluster-name}-pd-peer.{cluster-name}.svc:2380, Get http://{cluster-name}-pd.{cluster-name}:2379/pd/api/v1/members: dial tcp x.x.x.x:2379: connect: connection refused
```

可以观察 pd 的 service 状态：

```
wget -c http://{cluster-name}-pd.{cluster-name}:2379/pd/api/v1/members
```

同时观察 pd 的 pod 状态：

```
wget -c http://{ip of pd's pod}:2379/pd/api/v1/members
```

如果第一个 wget 命令失败，但是第二个 wget 命令成功，则可判断 Kubernetes 集群的 kube-proxy 是否有问题。关于 kube-proxy 的状态分析，可参考 Kubernetes 文档分析 kube-proxy 日志。也可通过实验证 kube-proxy 的有效性：

首先部署一个 nginx 服务：

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/website/master/content/en/examples/service/networking/run-my-nginx.yaml
```

然后将它暴露成一个 service：

```
kubectl expose deployment/my-nginx
```

之后访问这个 service 验证是否成功，如果失败则 kube-proxy 是有问题的。可以重启 kube-proxy 来尝试解决问题。如果问题仍然存在，则需要检查 Kubernetes 的网络配置情况，具体可参考 Kubernetes 文档。

- 案例四部分 pd 的 pod 反复 CrashLoopBackOff 的第二种情况

chart 中预设的 pd replicas 为 3，但是出现了 pd 的 pod 为 5 个的情况，3 个非 running 的状态，这 3 个互相 join，且反复 CrashLoopBackoff。现象如下：

| NAME   | READY | STATUS           | RESTARTS | AGE |
|--|-------|------------------|----------|-----|
| tidbcluster-...-i-discovery-68d7c75556-956kw | 1/1   | Running          | 1        | 15d |
| tidbcluster-...-i-monitor-84cc9cc5-m6rlm     | 3/3   | Running          | 3        | 15d |
| tidbcluster-...-i-pd-0                       | 1/1   | Running          | 1        | 15d |
| tidbcluster-...-i-pd-1                       | 1/1   | Running          | 15       | 12d |
| tidbcluster-...-i-pd-2                       | 0/1   | CrashLoopBackoff | 283      | 24h |
| tidbcluster-...-i-pd-3                       | 0/1   | CrashLoopBackoff | 308      | 26h |
| tidbcluster-...-i-pd-4                       | 0/1   | CrashLoopBackoff | 3032     | 11d |
| tidbcluster-...-i-tidb-0                     | 2/2   | Running          | 0        | 15d |
| tidbcluster-...-i-tidb-1                     | 2/2   | Running          | 0        | 15d |
| tidbcluster-...-i-tidb-initializer-p8r2l     | 0/1   | Completed        | 129      | 29d |
| tidbcluster-...-i-tikv-0                     | 1/1   | Running          | 0        | 15d |
| tidbcluster-...-i-tikv-1                     | 1/1   | Running          | 0        | 15d |
| tidbcluster-...-i-tikv-2                     | 1/1   | Running          | 0        | 15d |

查看 PD 的 pod 日志：

```
[~]# kubectl logs --tail=200 tidbcluster-...-i-pd-2
Name: tidbcluster-...-i-pd-2
Address 1: 10.42.40.80
nslookup domain
starting pd-server...
/pd-server --data-dir=/var/lib/pd --name=tidbcluster-...-i-pd-2 --pd-peer=tidbcluster-...-i-pd-2 --pd-peer=tidbcluster-...-i-pd-2 --pd-peer=tidbcluster-...-i-pd-2 --peer-urls=http://0.0.0.0:2380 --advertise-peer-urls=http://tidbcluster-...-i-pd-2.tidbcluster-...-i-pd-2.svc:2380 --client-urls=http://0.0.0.0:2379 --advertise-client-urls=http://tidbcluster-...-i-pd-2.tidbcluster-...-i-pd-2.svc:2379 --config=/etc/pd/pd.toml --join=http://tidbcluster-...-i-pd-3.tidbcluster-...-i-pd-3.svc:2380,http://tidbcluster-...-i-pd-3.tidbcluster-...-i-pd-3.svc:2380 --pd=tidbcluster-...-i-pd-2 --pd-peer=tidbcluster-...-i-pd-2 --pd-peer=tidbcluster-...-i-pd-2 --pd-peer=tidbcluster-...-i-pd-2
[2020/02/06 16:50:46.071 +08:00] [INFO] [util.go:59] ["Welcome to Placement Driver (PD)"]
[2020/02/06 16:50:46.071 +08:00] [INFO] [util.go:60] [PD] [release-version=v3.0.5]
[2020/02/06 16:50:46.071 +08:00] [INFO] [util.go:61] [PD] [git-hash=4934a651d2d428411a9610a5b6b9d7156a439355]
[2020/02/06 16:50:46.071 +08:00] [INFO] [util.go:62] [PD] [git-branch=HEAD]
[2020/02/06 16:50:46.071 +08:00] [INFO] [util.go:63] [PD] [utc-build-time="2019-10-25 03:24:57"]
[2020/02/06 16:50:46.071 +08:00] [INFO] [metricutil.go:80] ["disable Prometheus push client"]
[2020/02/06 16:50:46.071 +08:00] [ERROR] [join.go:212] ["failed to open directory"] [error="open /var/lib/pd/member: no such file or directory"]
2020/02/06 16:50:46.071 grpclog.go:45: [info] parsed scheme: "endpoint"
2020/02/06 16:50:46.071 grpclog.go:45: [info] ccResolverWrapper: sending new addresses to cc: {[http://tidbcluster-...-i-pd-3.tidbcluster-...-i-pd-3.svc:2380 0 <nil>] {http://tidbcluster-...-i-pd-1.tidbcluster-...-i-pd-1.svc:2380 0 <nil>} {http://tidbcluster-...-i-pd-0.tidbcluster-...-i-pd-0.svc:2380 0 <nil>}}
2020/02/06 16:50:46.079 grpclog.go:60: [warning] grpc: addrConn.createTransport failed to connect to {http://tidbcluster-...-i-pd-3.tidbcluster-...-i-pd-3.svc:2380 0 <nil>}. Err :connection error: desc = "transport: Error while dialing dial tcp: lookup tidbcluster-wuhanwuli-pd-3.tidbcluster-...-i-pd-3.svc on 10.43.0.10:53: no such host". Reconnecting...
[2020/02/06 16:50:46.081 +08:00] [FATAL] [main.go:91] ["join meet error"] [error="there is a member that has not joined successfully"]
[stack=github.com/pingcap/log@v0.0.2-0190715063458-479153f07ebd/global.go:59/main.main@nкт/home/jenkins/agent/workspace/release_tidb_3.0/go/pkg/mod/github.com/pingcap/log@v0.0.2-0190715063458-479153f07ebd/global.go:59/main.main@nкт/home/jenkins/agent/workspace/release_tidb_3.0/go/src/github.com/pingcap/pd/cm/d/pd-server/main.go:91/nruntime.main@nкт/usr/local/go/src/runtime/proc.go:209"]
2020/02/06 16:50:46.081 grpclog.go:60: [warning] Failed to dial http://tidbcluster-...-i-pd-3.tidbcluster-...-i-pd-3.svc:2380: context canceled; please retry.
]
```

日志中可见打开一些文件失败，可以查看 pod 的本地目录文件状态。如果发现目录是不完整的，需要考虑 PV 设置是否正常，比如是否为 Retain 状态等。

查看 provisioner 的日志，如下：

```
I0206 09:47:40.383224 1 cache.go:64] Updated pv "local-pv-863ba702" to cache
I0206 09:47:40.383244 1 cache.go:64] Updated pv "local-pv-ca3501f4" to cache
I0206 09:51:42.745063 1 cache.go:64] Updated pv "local-pv-863ba702" to cache
I0206 09:51:52.419372 1 deleter.go:195] Start cleanup for pv local-pv-863ba702
I0206 09:51:52.419505 1 deleter.go:266] Deleting PV file volume "local-pv-863ba702" contents at hostpath "/mnt/disks/tikv1", mountpath "/mnt/disks/tikv1"
I0206 09:52:02.420739 1 deleter.go:165] Deleting pv local-pv-863ba702 after successful cleanup
I0206 09:52:02.430763 1 cache.go:64] Updated pv "local-pv-863ba702" to cache
I0206 09:52:02.439564 1 cache.go:73] Deleted pv "local-pv-863ba702" from cache
I0206 09:52:12.433960 1 discovery.go:299] Found new volume at host path "/mnt/disks/tikv1" with capacity 3935254495232, creating Local PV "local-pv-863ba702", required volumeMode "Filesystem"
I0206 09:52:12.443227 1 discovery.go:332] Created PV "local-pv-863ba702" for volume at "/mnt/disks/tikv1"
I0206 09:52:12.443520 1 cache.go:55] Added pv "local-pv-863ba702" to cache
I0206 09:52:12.443520 1 cache.go:64] Updated pv "local-pv-863ba702" to cache
```

可见 PV 曾经被删除。这是 pd 数据目录异常的原因。可继续查看 kube-apiserver 的日志，寻找删除 PV 操作的触发原因，这需要 kube-apiserver 通过启动参数配置日志级别：

```
--v=4
```

此时的 pd 集群需要通过 pd-recover 恢复。

### 6.1.4 容器处于 Pending 状态

#### 问题分析

```
kubectl get pod
NAME    READY  STATUS    RESTARTS   AGE
pod-pvc-pv  0/1    Pending    0          4s
```

Pod 处于 Pending 状态，通常都是资源不满足导致的，比如：

- 使用持久化存储的 PD/TiKV/Monitor Pod 使用的 PVC 的 StorageClass 不存在或 PV 不足
- Kubernetes 集群中没有节点能满足 Pod 申请的 CPU 或内存
- PD 或者 TiKV Replicas 数量和集群内节点数量不满足 tidb-scheduler 高可用调度策略

## 解决方案

- kubectl describe 查看 Pending 的具体原因

```
kubectl describe pod -n <namespace> <pod-name>
kubectl describe pod pod-pvc-pv
Name:          pod-pvc-pv
...
Status:        Pending
...
Warning FailedScheduling 21s (x4 over 4m28s) default-scheduler 0/1 nodes are available: 1 node(s) didn't find available persistent volumes to bind.
```

- 资源不足解决办法

降低对应组件的 CPU 或内存资源申请使其能够得到调度，或是增加新的 Kubernetes 节点。

- StorageClass 不存在

需要在 values.yaml 里面将 storageClassName 修改为集群中可用的 StorageClass 名字，执行 helm upgrade，然后将 Statefulset 删除，并且将对应的 PVC 也都删除，可以通过以下命令获取集群中可用的 StorageClass：

```
kubectl get storageclass
```

- Local PV 不足

需要添加对应的 PV 资源。对于 Local PV，可以参考 [本地 PV 配置](#) 进行扩充。

### 6.1.5 Pod 之间网络不通

#### 问题分析

针对 TiDB 集群而言，绝大部分 Pod 间的访问均通过 Pod 的域名（使用 Headless Service 分配）进行，例外的情况是 TiDB Operator 在收集集群信息或下发控制指令时，会通过 PD Service 的 service-name 访问 PD 集群。

当通过日志或监控确认 Pod 间存在网络连通性问题，或根据故障情况推断出 Pod 间网络连接可能不正常时，可以按照下面的流程进行诊断，逐步缩小问题范围。

## 解决方案

1. 确认 Service 和 Headless Service 的 Endpoints 是否正常：

```
kubectl -n <namespace> get endpoints <release-name>-pd
kubectl -n <namespace> get endpoints <release-name>-tidb
kubectl -n <namespace> get endpoints <release-name>-pd-peer
kubectl -n <namespace> get endpoints <release-name>-tikv-peer
kubectl -n <namespace> get endpoints <release-name>-tidb-peer
```

以上命令展示的 ENDPOINTS 字段中，应当是由逗号分隔的 cluster\_ip:port 列表。假如字段为空或不正确，请检查 Pod 的健康状态以及 kube-controller-manager 是否正常工作。

#### 1. 进入 Pod 的 Network Namespace 诊断网络问题：

```
tkctl debug -n <namespace> <pod-name>
```

远端 shell 启动后，使用 dig 命令诊断 DNS 解析，假如 DNS 解析异常，请参照 [诊断 Kubernetes DNS 解析](#) 进行故障排除：

```
dig <HOSTNAME>
```

使用 ping 命令诊断到目的 IP 的三层网络是否连通（目的 IP 为使用 dig 解析出的 ClusterIP）：

```
ping <TARGET_IP>
```

假如 ping 检查失败，请参照 [诊断 Kubernetes 网络](#) 进行故障排除。假如 ping 检查正常，继续使用 telnet 检查目标端口是否打开：

```
telnet <TARGET_IP> <TARGET_PORT>
```

假如 telnet 检查失败，则需要验证 Pod 的对应端口是否正确暴露以及应用的端口是否配置正确：

```
# 检查端口是否一致
kubectl -n <namespace> get po <pod-name> -ojson | jq '.spec.containers[].ports[].containerPort'

# 检查应用是否被正确配置服务于指定端口上
# PD, 未配置时默认为 2379 端口
kubectl -n <namespace> -it exec <pod-name> -- cat /etc/pd/pd.toml | grep client-urls
# TiKV, 未配置时默认为 20160 端口
kubectl -n <namespace> -it exec <pod-name> -- cat /etc/tikv/tikv.toml | grep addr
# TiDB, 未配置时默认为 4000 端口
kubectl -n <namespace> -it exec <pod-name> -- cat /etc/tidb/tidb.toml | grep port
```

## 6.1.6 无法访问 TiDB 服务

### 问题分析

由于远程访问，所以优先排查网络链路。

### 解决方案

TiDB 服务访问不了时，首先确认 TiDB 服务是否部署成功，确认方法如下：

查看该集群的所有组件是否全部都启动了，状态是否为 Running。

```
kubectl get po -n <namespace>
```

检查 TiDB 组件的日志，看日志是否有报错。

```
kubectl logs -f <tidb-pod-name> -n <namespace> -c tidb
```

如果确定集群部署成功，则进行网络检查：

- 如果你是通过 NodePort 方式访问不了 TiDB 服务，请在 node 上尝试使用 service domain 或 clusterIP 访问 TiDB 服务，假如 serviceName 或 clusterIP 的方式能访问，基本判断 Kubernetes 集群内的网络是正常的，问题可能出在下面两个方面：
  - 客户端到 node 节点的网络不通。

- 查看 TiDB service 的 externalTrafficPolicy 属性是否为 Local。如果是 Local 则客户端必须通过 TiDB Pod 所在 node 的 IP 来访问。
2. 如果 service domain 或 clusterIP 方式也访问不了 TiDB 服务，尝试用 TiDB 服务后端的 :4000 连接看是否可以访问，如果通过 PodIP 可以访问 TiDB 服务，可以确认问题出在 service domain 或 clusterIP 到 PodIP 之间的连接上，排查项如下：

- 检查 DNS 服务是否正常：

```
kubectl get po -n kube-system -l k8s-app=kube-dns
dig <tidb-service-domain>
```

- 检查各个 node 上的 kube-proxy 是否正常运行：

```
kubectl get po -n kube-system -l k8s-app=kube-proxy
```

- 检查 node 上的 iptables 规则中 TiDB 服务的规则是否正确

```
iptables-save -t nat |grep <clusterIP>
```

- 检查对应的 endpoint 是否正确

3. 如果通过 PodIP 访问不了 TiDB 服务，问题出在 Pod 层面的网络上，排查项如下：

- 检查 node 上的相关 route 规则是否正确
- 检查网络插件服务是否正常
- 参考上面的 [Pod 之间网络不通](#) 章节

## 6.2 集群运行过程中

### 6.2.1 删除 TiKV 或 PD，再次创建启动不成功

#### 问题分析

#### 解决方案

### 6.2.2 并发扩缩容，TiKV Store 异常进入 Tombstone 状态

#### 问题分析

正常情况下，当 TiKV Pod 处于健康状态时（Pod 状态为 Running），对应的 TiKV Store 状态也是健康的（Store 状态为 UP）。但并发进行 TiKV 组件的扩容和缩容可能会导致部分 TiKV Store 异常并进入 Tombstone 状态。

对比 Store 状态与 Pod 运行状态。假如某个 TiKV Pod 所对应的 Store 处于 Offline 状态，则表明该 Pod 的 Store 正在异常下线中。此时，可以通过下面的命令取消下线进程，进行恢复：

#### 解决方案

此时，可以按照以下步骤进行修复：

1. 查看 TiKV Store 状态：

```
kubectl get -n <namespace> tidbcluster <release-name> -ojson | jq '.status.tikv.stores'
```

1. 查看 TiKV Pod 运行状态：

```
kubectl get -n <namespace> po -l app.kubernetes.io/component=tikv
```

1. 对比 Store 状态与 Pod 运行状态。假如某个 TiKV Pod 所对应的 Store 处于 Offline 状态，则表明该 Pod 的 Store 正在异常下线中。此时，可以通过下面的命令取消下线进程，进行恢复：

- 打开到 PD 服务的连接：

```
kubectl port-forward -n <namespace> svc/<cluster-name>-pd <local-port>:2379 &>/tmp/portforward-pd.log &
```

- 上线对应 Store：

```
curl -X POST http://127.0.0.1:2379/pd/api/v1/store/<store-id>/state?state=Up
```

2. 假如某个 TiKV Pod 所对应的 lastHeartbeatTime 最新的 Store 处于 Tombstone 状态，则表明异常下线已经完成。此时，需要重建 Pod 并绑定新的 PV 进行恢复：

- 将该 Store 对应 PV 的 reclaimPolicy 调整为 Delete：

```
kubectl patch $(kubectl get pv -l app.kubernetes.io/instance=<release-name>,tidb.pingcap.com/store-id=<store-id> -o name) -p '{"spec":{"persistentVolumeReclaimPolicy":"Delete"}}'
```

- 删除 Pod 使用的 PVC：

```
kubectl delete -n <namespace> pvc tikv-<pod-name> --wait=false
```

- 删除 Pod，等待 Pod 重建：

```
kubectl delete -n <namespace> pod <pod-name>
```

Pod 重建后，会以在集群中注册一个新的 Store，恢复完成。

### 6.2.3 TiDB 长连接被异常中断

#### 问题分析

许多负载均衡器 (Load Balancer) 会设置连接空闲超时时间。当连接上没有数据传输的时间超过设定值，负载均衡器会主动将连接中断。若发现 TiDB 使用过程中，长查询会被异常中断，可检查客户端与 TiDB 服务端之间的中间件程序

#### 解决方案

若其连接空闲超时时间较短，可尝试增大该超时时间。若不可修改，可打开 TiDB tcp-keep-alive 选项，启用 TCP keepalive 特性。

- 如果 Kubernetes 集群内的 [kubelet](#) 允许配置 `--allowed-unsafe-sysctls=net.*`，请为 kubelet 配置该参数，并按如下方式配置 TiDB：

```
tidb:
...
podSecurityContext:
sysctls:
- name: net.ipv4.tcp_keepalive_time
value: "300"
```

- 如果 Kubernetes 集群内的 [kubelet](#) 不允许配置 `--allowed-unsafe-sysctls=net.*`，请按如下方式配置 TiDB：

```

tidb:
  annotations:
    tidb.pingcap.com/sysctl-init: "true"
  podSecurityContext:
    sysctls:
      - name: net.ipv4.tcp_keepalive_time
        value: "300"
      ...

```

注意：进行以上配置要求 TiDB Operator 1.1 及以上版本。

## 6.3 诊断模式

当 Pod 处于 CrashLoopBackoff 状态时，Pod 内容器会不断退出，导致无法正常使用 kubectl exec 或 tkctl debug，给诊断带来不便。为了解决这个问题，TiDB in Kubernetes 提供了 PD/TiKV/TiDB Pod 诊断模式。在诊断模式下，Pod 内的容器启动后会直接挂起，不会再进入重复 Crash 的状态，此时，便可以通过 kubectl exec 或 tkctl debug 连接 Pod 内的容器进行诊断。

操作之前，Pod 处于 CrashLoopBackoff 状态：

```

$ kubectl get pods -n tidbcluster1
NAME           READY   STATUS      RESTARTS   AGE
demo-discovery-5c78d6bcd8-5ttcl  1/1     Running    0          20h
demo-monitor-6ddc6d6674-kcmhh   3/3     Running    0          3d22h
demo-pd-0       1/1     Running    0          3d22h
demo-pd-1       0/1     CrashLoopBackOff  911       3d22h
demo-pd-2       1/1     Running    10         3d22h
demo-pd-3       0/1     CrashLoopBackOff  932       3d22h
demo-pd-4       1/1     Running    0          3d22h
demo-pd-5       1/1     Running    0          3d22h
demo-tidb-0     2/2     Running    1          3d22h
demo-tidb-1     2/2     Running    0          3d22h
demo-tikv-0     1/1     Running    0          3d22h
demo-tikv-1     1/1     Running    0          3d22h
demo-tikv-2     1/1     Running    0          3d22h
demo-tikv-3     1/1     Running    0          3d22h
demo-tikv-4     1/1     Running    0          3d22h
demo-tikv-5     1/1     Running    0          3d22h

```

首先，为待诊断的 Pod 添加 Annotation：

```

kubectl annotate pod <pod-name> -n <namespace> runmode=debug
# kubectl annotate pod demo-pd-1 -n tidbcluster1 runmode=debug

```

在 Pod 内的容器下次重启时，会检测到该 Annotation，进入诊断模式。等待 Pod 进入 Running 状态即可开始诊断：

```

watch kubectl get pod <pod-name> -n <namespace>
# watch kubectl get pod demo-pd-1 -n tidbcluster1

```

Every 2.0s: kubectl get pod demo-pd-1 -n tidbcluster1

| NAME      | READY | STATUS  | RESTARTS | AGE   |
|-----------|-------|---------|----------|-------|
| demo-pd-1 | 1/1   | Running | 912      | 3d22h |

下面是使用 kubectl exec 进入容器进行诊断工作的例子：

```
# 进入诊断模式后 kubectl logs 会提示当前 pod 已经进入诊断模式
$ kubectl logs demo-pd-1 -n tidbcluster1
entering debug mode.
# 进入 pod 进行诊断
kubectl exec -it <pod-name> -n <namespace> -- /bin/sh
# kubectl exec -it demo-pd-1 -n tidbcluster1 -- /bin/sh
```

诊断完毕，修复问题后，删除 Pod：

```
kubectl delete pod <pod-name> -n <namespace>
# kubectl delete pod demo-pd-1 -n tidbcluster1
```

输出为：

```
pod "demo-pd-1" deleted
```

监控：Pod 重建后会自动回到正常运行模式。

# 第1章 适用场景介绍

谈到场景我们不禁会想到各种各样的案例，单纯只是罗列案例无疑会浪费大量篇幅，读者也未必会找到一条清晰的脉络来回答一个问题“TiDB是否适合我现在的工程环境？”。我们不妨换个角度，所谓适用场景，首先要明确解决什么问题，然后考察产品的哪些特性有利于解决问题，对比类似方案的优缺点，最后通过阅读实际案例增强感性认知。通过前述过程，在面对具体环境时，就可以快速判断产品是否适用，保证选型的正确性。在本节中我们会给出常见的场景描述和经典案例，让读者对TiDB的场景有更具体的认识。

## 1. TiDB 要解决什么问题

在 20 年前，大家对数据库的认知可能比较单一，大致的概念是“支持 SQL 的数据存储软件产品”，基于等同于关系型数据库 (RDBMS)。单机关系型数据库如日中天，掩盖了其他类型数据库的光芒。随着互联网的发展，开始认识到传统单机关系型数据库的局限。数据的概念被泛化，如今支持数据存储和查询的软件系统都被叫做数据库，于是出现了文档数据库、时序数据库、分布式数据库、KV 数据库等等。简而言之，都是为了满足传统单机数据库扩展性的不足。很多新的数据库支持横向扩展的同时放弃了 SQL 语言和事务支持，这类数据库被称为 NoSQL。大部分 NoSQL 产品牺牲强一致性换取扩展性，开发者要自己实现业务出错时的补偿机制。没有标准的数据操作规范，程序员需要面对各种数据操作接口，增加了出错的概率。最近几年，人们开始意识到了标准的数据操作对于开发和维护的意义。简单来讲，数据库的发展趋势就是：SQL -> NoSQL -> No only SQL。

回归并非回到原点，而是螺旋上升。大家期待一种数据库，既兼容SQL，又可以横向扩展，并支持事务，这也是 NewSQL 的定义。TiDB 正是一款 NewSQL 数据库产品。如果你现在正在使用 MySQL 数据库，面对无法承载的海量业务又不想让分库分表折磨，TiDB 无疑是目前最好的选择，这也是 TiDB 最为经典的场景。泛化一点，需要在数据层面横向扩展又需要兼容 SQL 和事务 ACID 特点的技术团队都可以尝试 TiDB。

在进行 TiDB 产品适用场景的正式介绍之前，可以站在使用者的角度，来观察 TiDB 的最为突出的特质：

- 架构经过重新设计，可以横向扩展的 MySQL。
- Raft 多数派一致性协议实现数据的高可用和存储的灵活性。
- 不同计算场景的需求可以灵活地访问共存的行式和列式数据。
- 计算存储分离的分布式架构易于与云的弹性特质结合。

## 2. 适用场景

### 高并发 OLTP

MySQL 是第一个广泛使用的开源关系型数据库，也是国内互联网业务数据库的事实标准。面对终端用户的迅猛增长，MySQL 数据库的架构方案很快就会面临承载能力上限。在 NewSQL 还没有出现前，MySQL 遭遇业务迅速增长造成的瓶颈时，数据库架构的演化方向通常会选用分库分表方案。

分库分表将高吞吐的大表按主键值的 Hash 值进行切分（称为 Sharding），表上数据的分发、路由引入中间件进行处理。自下而上分为三层，分别 DB 层、中间层、应用层。分库分表方案部分解决了业务扩展的问题，但对开发和运维造成巨大的压力。业务需要提供切分维度，不支持在线 DDL 操作，不能跨维度 Join / 聚合 / 子查询，不支持分布式事务，无法实现多维度的业务需求。业务程序从单机数据库迁移到分库分表方案时，通常要完成大量的开发适配改造。不能在线进行扩缩容，不能实现一致性的备份还原，难以实现异地容灾等。

TiDB 最初的目标使用场景就是通过计算存储分层的分布式设计实现单机 MySQL 性能的突破。TiDB 之于分库分表方案是一种颠覆，分布式架构优雅地实现了水平扩展，解决了诸多的开发限制。计算层关系型二维表与存储层 KV 键值对两种模式的转换关系，支持多维度的业务需求和复杂 SQL 查询，支持在线 DDL。Percolator 事务模型能保证 ACID。基于 Region 的 Multi-Raft 设计，支持以较小的成本进行在线扩缩容，支持无人工介入的高可用等。

| 类型                        | 分库分表 | TiDB |
|---------------------------|------|------|
| 强一致的分布式事务                 | 不支持  | 支持   |
| 水平扩展                      | 不支持  | 支持   |
| 复杂查询 (JOIN/ GROUP BY/...) | 不支持  | 支持   |
| 无人工介入的高可用                 | 不支持  | 支持   |
| 业务兼容性                     | 低    | 高    |
| 多维度支持                     | 不友好  | 友好   |
| 全局 ID 支持                  | 不友好  | 友好   |
| 机器容量                      | 很浪费  | 随需扩容 |

TiDB 兼容 MySQL 的开发生态，基于 MySQL 的业务只需要修改数据源连接 TiDB 就能运行，节点数可横向扩展适配业务变化。在 TiDB 4.0 中，弹性调度特性能根据现有的热点统计，利用 K8s 容器编排平台上灵活的调度能力，自动化地扩展 TiKV Pod 并迁移热点 Region，实现对前台业务负载变化的快速响应。

#### 案例参考

- TiDB 在知乎万亿量级业务数据下的实践和挑战

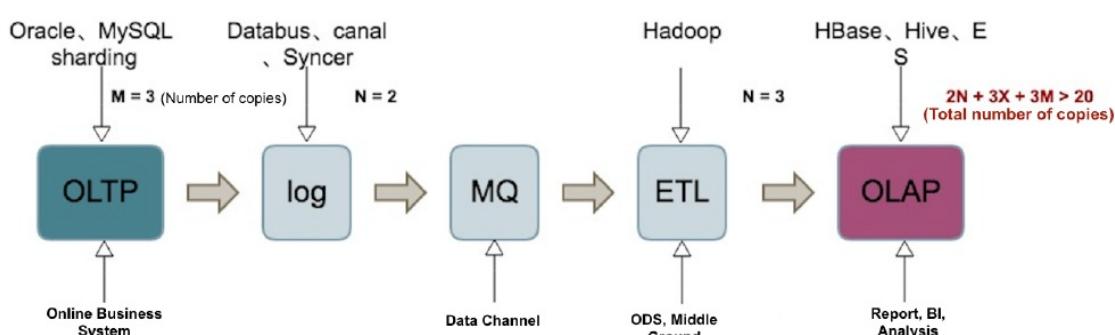
<https://pingcap.com/cases-cn/user-case-zhihu/>

- TiDB at 丰巢：尝鲜分布式数据库

<https://pingcap.com/cases-cn/user-case-fengchao/>

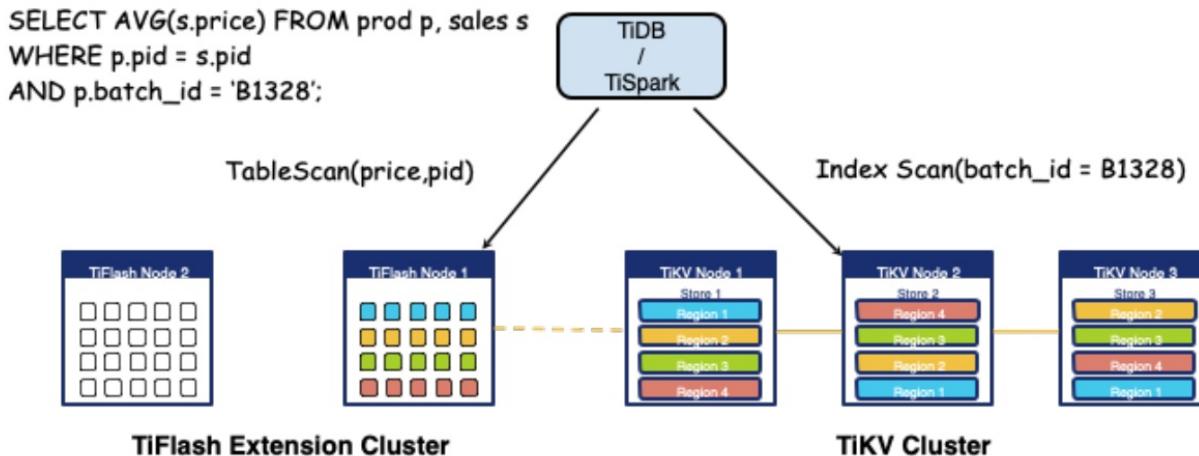
## 实时分析

传统的数据架构设计中，每个数据库都有一个明显角色身份，比如业务系统库，经营分析库，报表库，数据仓库等。使用 ETL 工具按照时间和数据获取策略将交易数据抽取到数据分析平台，满足业务的分析需求。受制于数据抽取方式的时间策略和分析平台的性能，业务部门最常抱怨的莫过于分析时效性，实时分析的概念应运而生。在实时分析领域，离线和在线的边界越来越模糊，一切数据皆服务化，一切分析皆在线化。数据的实时分析结果直接服务于业务，这对系统处理延时提出了新的挑战。



在 TiDB 4.0 版本之前，很难在一个集群内既支持实时分析处理，又要支持高并发的在线交易。3.0 版本中的 TiSpark 组件，将 Spark 的计算能力嫁接到 TiDB 的存储引擎 TiKV 之上。由于 Spark 的计算模型重且资源消耗高，在没有资源隔离支持的情况下，通常会将存储引擎的处理能力消耗殆尽，实时分析和交易处理成为“鱼和熊掌不可兼得”。

在 TiDB 4.0 版本中，引入列存引擎 TiFlash，既加速了分析运算，又解决了资源隔离的问题。通过 Raft learner 机制，将行存数据复制出列存数据，使用 Engine tag 实现访问资源的物理隔离。两种不同资源需求的场景实现共存，实时分析使用列存引擎 TiFlash，交易处理使用行存引擎 TiKV。列存引擎 TiFlash 组件的加入，补全 TiDB 的 HATP 版图。



对于业务更具价值的是，Raft learner 机制能将行存引擎上最为新鲜的业务数据复制到 TiFlash 中，实现业务数据的实时分析，分析的结果可以回写到行存引擎，为实时数据服务提供更多的想象空间。熟悉 Oracle 的读者，对于报表系统使用 DataGuard 备库，计算结果写回主库的架构设计应该不陌生。TiDB HTAP 架构的优势在于，在一套集群内通过扩展存储引擎组件的方式实现计算访问的灵活性，列式的存储不仅能够极大地加速分析场景的计算，同时交易场景可以利用列存实现类似索引的效果。在表级别实现灵活的配置方式，也免去整个过程中的人工介入减少大量的维护成本。

#### 案例参考

- TiDB / TiSpark 在易果集团实时数仓中的创新实践

<https://pingcap.com/cases-cn/user-case-yiguo/>

## 多活场景

多个数据中心内部署业务系统组件，如数据库服务器、应用服务器，并组成一个有机的整体，用户能够接入任意一个数据中心的业务系统实现多活访问，能有效提高系统健壮性和业务流量承载能力。

多活场景最主要的难点在于业务系统需要在同一份数据上提供数据服务。假设多活数据中心共有 A、B、C 三个站点，结合业务对于数据一致性的要求，需要保证在 A 站点发生故障后，业务系统此前发生的操作，在另外两个站点上也能访问到。常见的多活基础架构方式有：

- 数据库集群结合裸光纤互连的存储容灾复制方案，比如 Oracle Extended RAC。
- 按站点进行应用数模设计结合数据复制的，比如 A 站点的记录号为奇数，B 站点的记录号为偶数，利用序列的步长避免记录操作冲突，同时使用 Oracle GoldenGate 进行双向复制。
- 在应用层设计共享中心和业务中心，终端用户绑定业务中心属主，当用户访问非属主业务中心时，共享中心自动实现用户的漫游和数据跨中心的数据访问。

以上的多活设计方式中，如果优先保证一致性就会影响性能，如果优先满足性能就需要在一致性上做妥协。

在 TiDB 的多活场景设计中，根据各个分布式组件的高可用机制实现多活部署。TiDB Server 属于无状态应用，类似 Web 服务器，在多个站点部署结合负载均衡设备实现高可用和多活访问。TiKV Server 和 PD Server 基于 Raft 多数派一致性协议实现高可用。TiKV Server 以 Region 为单位，按指定的数据副本数进行存储，属于 Multi-Raft 设计。在高可用设计上还引入 DC / Zone / Rack / Host 的四层标签体系和 Raft Leader 的 Reject 排斥策略，能灵活地指定在多个站点的数据副本分布和 IO 的流量导向。数据写入时，只需要在延时较低的站点内写入足够的数据副本数量就可以返回写入成功，同时满足性能和数据一致性要求。PD Server 属于单 Raft 组设计，节点数等于数据副本数，在多个站点均衡配置 3 个或者更多节点。

TiDB 的多活架构设计，不需要在应用层数模做特殊设计，实现原生业务的多活。Raft 的多数派一致性设计，既降低了多活的网络要求，又满足了数据的高可用要求。同时整个多活体制的高可用机制，均由底层体制自动实现，不需要人工介入和额外的操作流程。TiDB 4.0 的 Follower Read 特性，能实现同站点读取操作的亲和性，有效提高存储层的数据吞吐能力并降低跨站点的网络流量，进一步降低了网络成本。

- TiDB 在银行核心金融领域的研究与两地三中心实践

<https://pingcap.com/cases-cn/user-case-beijing-bank/>

- 微众银行数据库架构演进及 TiDB 实践经验

<https://pingcap.com/cases-cn/user-case-webank/>

## 3.经典案例

经过几百个用户实际使用，TiDB 产品积累了大量的案例。在这些案例中，也许读者会找到自己场景的影子，或者案例本身就是你面对场景的 TiDB 解决方案。欢迎访问以下链接阅读用户案例。

用户案例汇总链接 (<https://pingcap.com/cases-cn/>)

## 第2章 硬件选型规划

TiDB 集群的硬件要求在[官方文档](#)中进行了总结。本节将介绍更实用的硬件选择和优化。请注意，如果工作负载不同，下文的细节可能有所不同，需要根据实际情况调整。

### 2.1 TiDB server 组件

如果是典型的 TP 类场景，业务并发较高，但几乎都是点查点写，那么整个系统的瓶颈大概率先出现在 tidb-server，所以 tidb-server 的 cpu 高一点，数量多一点是一个更好的选择。如果你有更多 OLAP 类的查询，内存通常会成为瓶颈。在以下情况下，可以考虑分配超过官方文档建议的 16GB 的内存：

- 工作负载中 OLAP 查询的比例很高；
- 工作负载中包含很复杂的 OLAP 查询；
- 计划使用 mydumper 进行全量备份

### 2.2 PD 组件

主要负责全局唯一的事务号的分配，以及整个集群的元数据信息。所以 PD 不需要太多的资源，在集群规模较大，元数据较多的情况下，推荐使用 SSD 磁盘。但是 PD 的稳定性对于整个集群来说至关重要，生产环境推荐单独的服务器部署 PD，而且要有至少三个节点。对于非关键性工作负载或者非生产环境，你可以考虑将 PD 和 TiDB 组件部署在同一台服务器上，或减少 PD 实例数以降低成本。

### 2.3 TiKV 组件

TiKV 节点除了负责数据存储之外，也会负责部分的计算功能，通常情况下，TiKV 组件是最消耗资源的部分。为避免出现任何问题，建议你为 TiKV 多分配或预留些资源，特别是如果你预计不久后流量会大幅增加，则更应注意这一点。同时 TiKV 实例的个数必须不小于副本数(max-replicas)，考虑到 location-label，TiKV 实例个数推荐是副本数的倍数，并且应当与你的数据量/业务相关。

### 2.4 监控组件

在监控组件的硬件资源上，官方的建议比较大方，但实际情况你可以考虑缩小规模以节省一些硬件成本，4-8GB 内存、2-4 核 CPU 在大多数情况下已经够用了。建议不要把监控组件部署在一个 16GB 内存、8 核 CPU 的实例上，而是可以考虑部署在两个分别有 8GB 内存、4 核 CPU 的实例上，这样能保证 Prometheus/Alertmanager/Grafana 监控系统的高可用性。TiDB 后续也会推出支持监控高可用的部署方式。

### 2.5 Pump 组件

建议使用：

- CPU：4 核
- 内存：8GB
- 磁盘：SSD
- 网络：万兆网卡（建议两块）

Pump 非常占用磁盘和网络资源。你需要确保具有足够的磁盘空间和网络带宽，特别是如果业务不能承受较大的同步延迟，则更应注意这一点。

## 2.6 Drainer 组件

建议使用：

- CPU：4 核
- 内存：8GB
- 磁盘：如果 Drainer 生成了文件，建议使用 SSD；其他情况无特殊要求。
- 网络：万兆网卡（建议两块）

Drainer 主要占用很多网络资源。如果你的 Drainer 输出类型是文件，请确保具有足够的磁盘带宽，以进行准实时增量备份。

## 2.7 传统备份恢复

如果你计划使用 mydumper/myloader 进行备份与恢复，请确保使用具有足够磁盘和网络带宽的专用节点。建议使用：

- CPU：16 核
- 内存：32GB
- 磁盘：SSD
- 网络：万兆网卡

如果你需要压缩转储文件，则应将内存增加一倍（即 64GB）。

## 第3章 常见性能压测

由于不同的数据库产品存在架构的不同、硬件环境的不同，很难直观的将不同的数据库直接进行横向对比，为此我们需要进行较为通用的数据库基准测试。基准测试是针对数据库的一种压力测试，但基准测试不关心业务逻辑，更加简单、直接、易于测试，不要求数据的真实性和逻辑关系。目前业界常用的数据库基准测试有 Sysbench、TPC-C、TPC-H 等，每种测试工具针对一种或者多种场景进行测试，本章将会介绍 Sysbench 测试和 TPC-C 测试的方法和步骤，上述两种测试均针对数据库 OLTP 场景。因硬件环境的不同，所以测试结果可能并不一致，测试结果仅供参考。

希望通过本章的描述，能够让大家对于如何进行数据库基准测试有所认识。

## 3.1 Sysbench 基准性能测试

TiDB 兼容 MySQL，支持无限的水平扩展，具备强一致性和金融级高可用性，为 OLTP 和 OLAP 场景提供一站式的解决方案。但想要使用 TiDB 时，都会被要求做基准测试并与 MySQL 对比，本文基于 sysbench 工具进行基准测试做简要说明。

### 3.1.1 工具集方案

- sysbench 安装

```
mkdir -p /tmp/sysbench
cd /tmp/sysbench
wget https://github.com/akopytov/sysbench/archive/1.0.14.tar.gz
yum -y install make automake libtool pkgconfig libaio-devel
yum -y install mariadb-devel
./autogen.sh
./configure
make -j
make install
sysbench --version
```

- 硬件配置

| 项目        | 配置   | 台数 | 说明                                    |
|-----------|--|----|---------------------------------------|
| TiDB & PD | CPU：2*E5-2650 v4@2.20GHz<br>内存：128G<br>硬盘：2*800G 固态、3*1.6T SSD<br>网卡：2 * 万兆 做 bond-1       | 3  | TiDB 和 PD 应用部署，文件系统 ext4              |
| TiKV      | CPU：2*E5-2650 v4@2.20GHz<br>内存：256G<br>硬盘：2*480G 固态、4*1.92T NVMe SSD<br>网卡：2 * 万兆 做 bond-1 | 3  | TiKV 应用部署，文件系统 ext4，PCIe 盘直接挂载到操作系统目录 |
| monitor   | 8 核，32G，800G 硬盘虚拟机   | 3  | 部署：Grafana + Prometheus               |

- 环境说明

|           |   |
|-----------|---|
| 项目        |   |
| 操作系统      | Redhat 7.4  |
| TiDB 版本   | TiDB-v3.0.5   |
| TiDB & PD | 每台 "TiDB 服务器" 部署 2 个 tidb-server + 1 个 pd-server  |
| TiKV      | 每台 "TiKV 服务器" 部署 4 个 tikv-server  |
| TiDB 关键参数 | performance:<br>max-procs: 24   |
| TiKV 关键参数 | readpool:<br>coprocessor:<br>high-concurrency: 8<br>normal-concurrency: 8<br>low-concurrency: 8<br>storage:<br>block-cache:<br>capacity: "32GB" |

## 3.1.2 测试实操

### 1. 测试准备工作

- sysbench 配置

```
mysql-host=192.168.xxx.xxx
mysql-port=4000
mysql-user=sysbench
mysql-password=*****
mysql-db=test
time=60
threads=16
report-interval=10
db-driver=mysql
```

- sysbench 关键参数说明

```
--threads=8 表示发起 8 个并发连接
--report-interval=10 表示每 10 秒输出一次测试进度报告
--rand-type=uniform 表示随机类型为固定模式，其他几个可选随机模式：uniform（固定），gaussian（高斯），special（特定的），pareto（帕累托）
--time=120 表示最大执行时长为 120 秒
--events=0 表示总请求数为 0，因为上面已经定义了总执行时长，所以总请求数可以设定为 0；也可以只设定总请求数，不设定最大执行时长
--percentile=99 表示设定采样比例，默认是 95%，即丢弃 1% 的长请求，在剩余的 99% 里取最大值
```

- sysbench 结果解读

```

sysbench 1.0.14 (using bundled LuaJIT 2.1.0-beta2)
Running the test with following options:
Number of threads: 16
Report intermediate results every 10 second(s)
Initializing random number generator from current time
Initializing worker threads...
Threads started!
# 每 10 秒钟报告一次测试结果，tps、每秒读、每秒写、95% 以上的响应时长统计
[ 10s ] thds: 16 tps: 21532.38 qps: 21532.38 (r/w/o: 21532.38/0.00/0.00) lat (ms,95%): 1.04 err/s: 0.00 reconn/s: 0.0
0
[ 20s ] thds: 16 tps: 21617.20 qps: 21617.20 (r/w/o: 21617.20/0.00/0.00) lat (ms,95%): 1.01 err/s: 0.00 reconn/s: 0.0
0
[ 30s ] thds: 16 tps: 21550.98 qps: 21550.98 (r/w/o: 21550.98/0.00/0.00) lat (ms,95%): 1.03 err/s: 0.00 reconn/s: 0.0
0
[ 40s ] thds: 16 tps: 21544.16 qps: 21544.16 (r/w/o: 21544.16/0.00/0.00) lat (ms,95%): 1.01 err/s: 0.00 reconn/s: 0.0
0
[ 50s ] thds: 16 tps: 21639.76 qps: 21639.76 (r/w/o: 21639.76/0.00/0.00) lat (ms,95%): 0.99 err/s: 0.00 reconn/s: 0.0
0
[ 60s ] thds: 16 tps: 21597.56 qps: 21597.56 (r/w/o: 21597.56/0.00/0.00) lat (ms,95%): 1.01 err/s: 0.00 reconn/s: 0.0
0
SQL statistics:
  queries performed:
    read:          1294886          # 读总数
    write:         0                # 写总数
    other:        0                # 其他操作总数 (COMMIT 等)
    total:        1294886          # 全部总数
  transactions: 1294886 (21579.74 per sec.)          # 总事务数 (每秒事务数)
  queries:      1294886 (21579.74 per sec.)          # 读总数 (每秒读次数)
  ignored errors: 0 (0.00 per sec.)          # 忽略错误数 (每秒忽略错误数)
  reconnects:   0 (0.00 per sec.)          # 重连次数 (每秒重连次数)
General statistics:
  total time:    60.0029s          # 总共耗时
  total number of events: 1294886          # 总共发生多少事务数

Latency (ms):
  min:           0.36            # 最小延时
  avg:           0.74            # 平均延时
  max:           8.90            # 最大延时
  95th percentile: 1.01          # 超过 95% 平均耗时
  sum:          959137.19
Threads fairness:
  events (avg/stddev): 80930.3750/440.48          # 平均每线程完成 80930.3750 次 event, 标准差为 44
0.48
  execution time (avg/stddev): 59.9461/0.00          # 每个线程平均耗时 59.9 秒, 标准差为 0

```

- 准备数据

```
sysbench --config-file=sysbench-thread-16.cfg oltp_point_select --tables=32 --table-size=10000000 prepare
```

- 数据预热与统计信息收集

数据预热可将磁盘中的数据载入内存的 block cache 中，预热后的数据对系统整体的性能有较大的改善，建议在每次重启集群后进行一次数据预热。以表 sbtest1 为例，执行如下 SQL 语句 进行数据预热，命令如下：

```
SELECT COUNT(pad) FROM sbtest1 USE INDEX(k_1);
```

统计信息收集有助于优化器选择更为准确的执行计划，可以通过 analyze 命令来收集系列表 sbtest\* 的统计信息，每个表都需要收集统计信息，以表 sbtest1 为例，命令如下：

```
ANALYZE TABLE sbtest1;
```

## 2. 测试命令举例

- Point select 测试命令

```
sysbench --config-file=sysbench-thread-16.cfg oltp_point_select --tables=32 --table-size=10000000 run
```

- Update index 测试命令

```
sysbench --config-file=sysbench-thread-16.cfg oltp_update_index --tables=32 --table-size=10000000 run
```

- Read-only 测试命令

```
sysbench --config-file=sysbench-thread-16.cfg oltp_read_only --tables=32 --table-size=10000000 run
```

- Write-only 测试命令

```
sysbench --config-file=sysbench-thread-16.cfg oltp_write_only --tables=32 --table-size=10000000 run
```

- Read-Write 测试命令

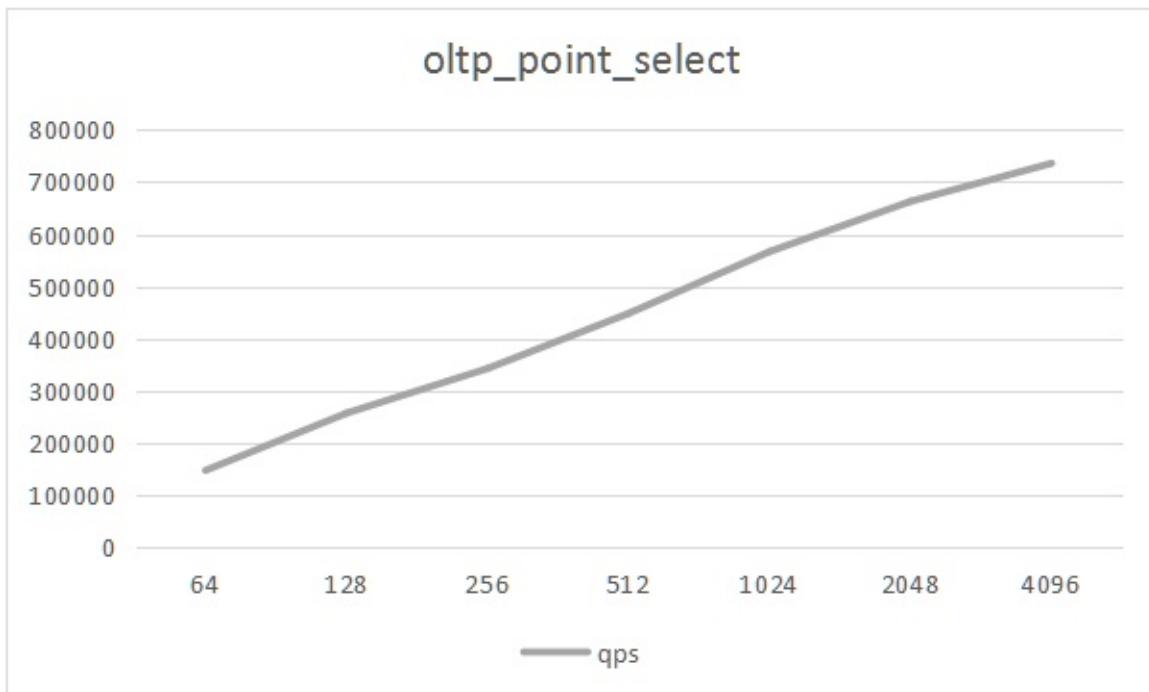
```
sysbench --config-file=sysbench-thread-16.cfg oltp_read_write --tables=32 --table-size=10000000 run
```

### 3. 测试结果举例

笔者测试数据 32 张表，每张表有 10MB 数据。对集群所有 tidb-server 都同时进行 sysbench 测试，将结果相加，得出最终结果：

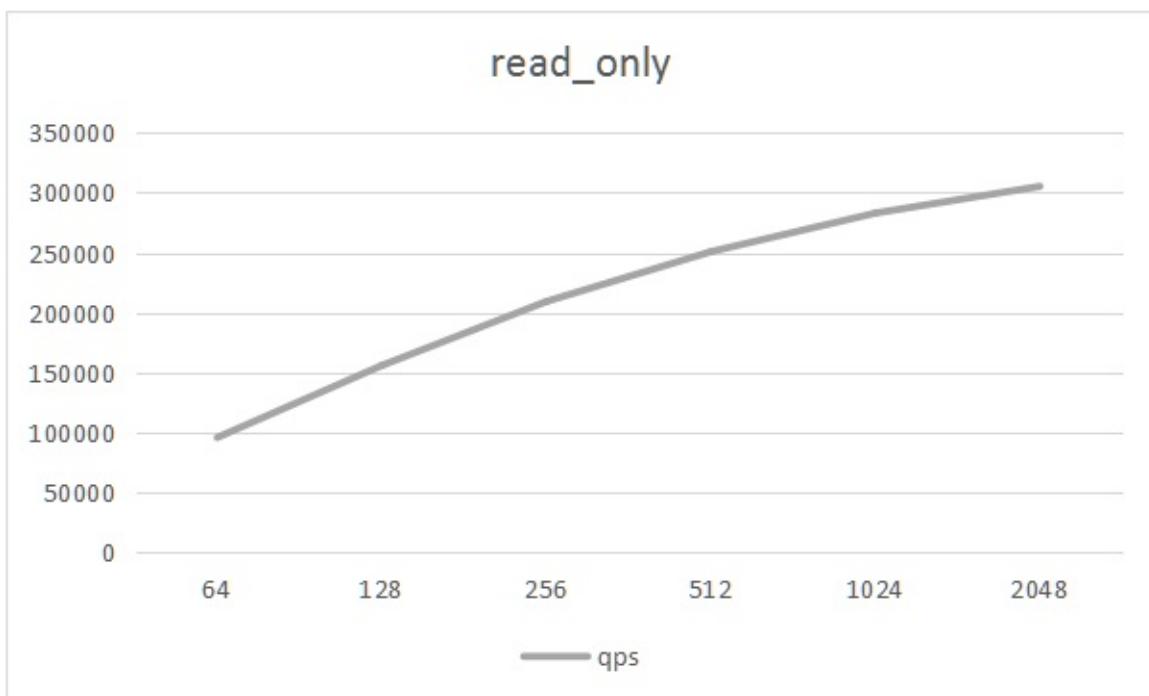
- oltp\_point\_select

| <b>type</b>  | <b>thread</b> | <b>tps</b> | <b>qps</b> | <b>min latency</b> | <b>avg latency</b> | <b>95th latency</b> | <b>max latency</b> |
|--------------|---------------|------------|------------|--------------------|--------------------|---------------------|--------------------|
| point_select | 64            | 148098.00  | 148098.00  | 0.26               | 0.43               | 0.52                | 276.54             |
| point_select | 128           | 257760.00  | 257760.00  | 0.27               | 0.50               | 0.65                | 261.51             |
| point_select | 256           | 343215.00  | 343215.00  | 0.28               | 0.75               | 1.89                | 253.23             |
| point_select | 512           | 448683.00  | 448683.00  | 0.29               | 1.14               | 3.55                | 290.85             |
| point_select | 1024          | 567063.00  | 567063.00  | 0.30               | 1.80               | 5.57                | 70.21              |
| point_select | 2048          | 663217.00  | 663217.00  | 0.29               | 3.08               | 8.90                | 330.19             |
| point_select | 4096          | 736094.00  | 736094.00  | 0.33               | 5.55               | 15.00               | 431.72             |



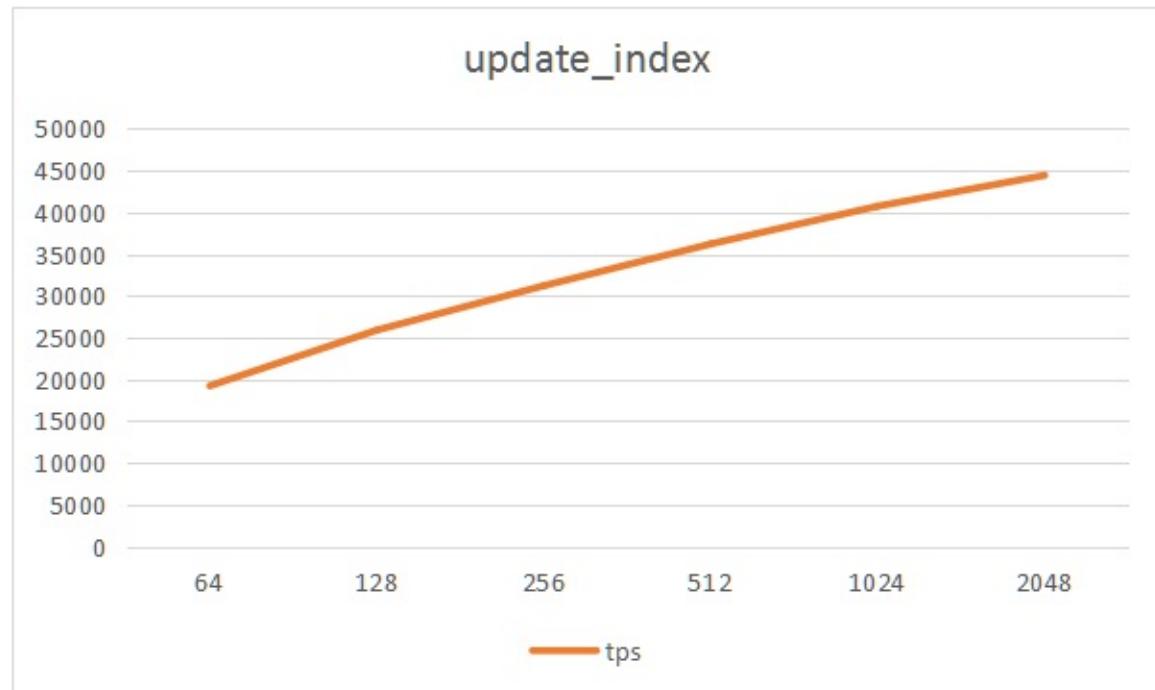
- `read_only`

| type      | thread | tps      | qps       | min latency | avg latency | 95th latency | max latency |
|-----------|--------|----------|-----------|-------------|-------------|--------------|-------------|
| read_only | 64     | 5984.48  | 95751.60  | 7.87        | 10.69       | 14.21        | 85.24       |
| read_only | 128    | 9741.39  | 155862.00 | 7.64        | 13.14       | 18.28        | 236.37      |
| read_only | 256    | 13080.20 | 209284.00 | 9.22        | 19.56       | 28.16        | 99.79       |
| read_only | 512    | 15678.40 | 250854.00 | 10.40       | 32.62       | 49.34        | 115.78      |
| read_only | 1024   | 17691.40 | 283063.00 | 10.87       | 57.73       | 87.56        | 378.12      |
| read_only | 2048   | 19086.60 | 305386.00 | 7.68        | 107.12      | 164.45       | 710.91      |



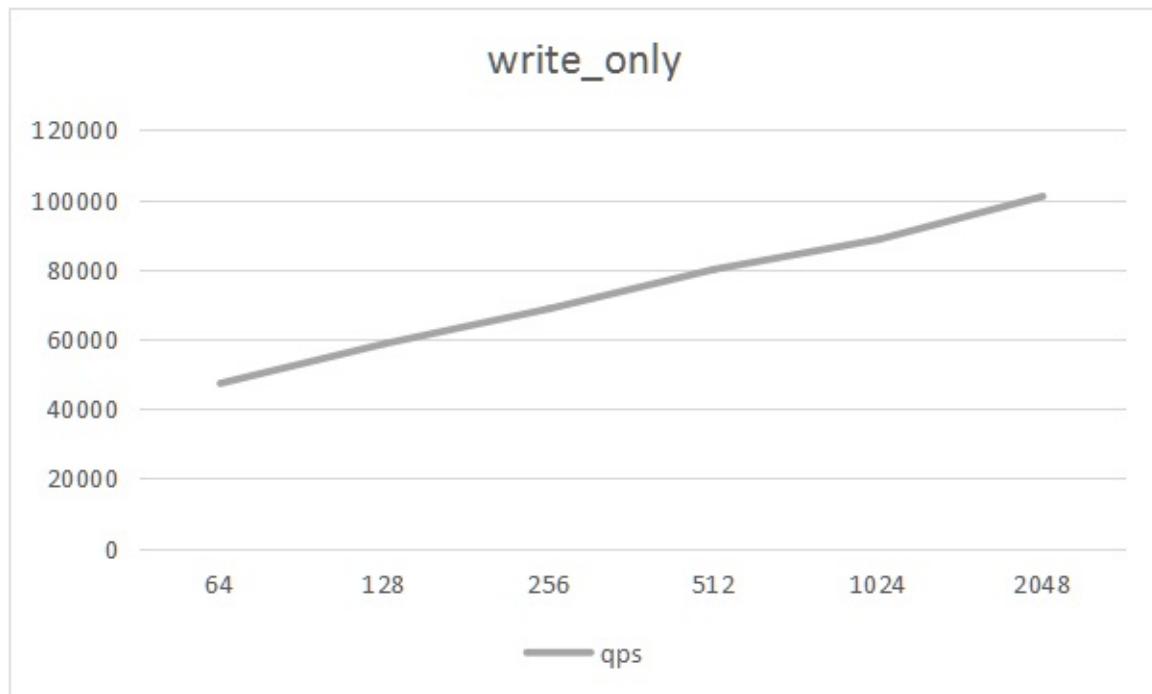
- `oltp_update_index`

| type         | thread | tps      | qps      | min latency | avg latency | 95th latency | max latency |
|--------------|--------|----------|----------|-------------|-------------|--------------|-------------|
| update_index | 64     | 19232.10 | 19232.10 | 1.75        | 3.33        | 4.74         | 274.86      |
| update_index | 128    | 25898.20 | 25898.20 | 1.67        | 4.94        | 7.98         | 330.88      |
| update_index | 256    | 31214.00 | 31214.00 | 1.67        | 8.20        | 14.73        | 5189.46     |
| update_index | 512    | 36213.50 | 36213.50 | 1.74        | 14.13       | 27.66        | 5487.91     |
| update_index | 1024   | 40731.20 | 40731.20 | 1.74        | 25.12       | 52.89        | 7395.50     |
| update_index | 2048   | 44423.50 | 44423.50 | 1.77        | 46.04       | 99.33        | 5563.36     |



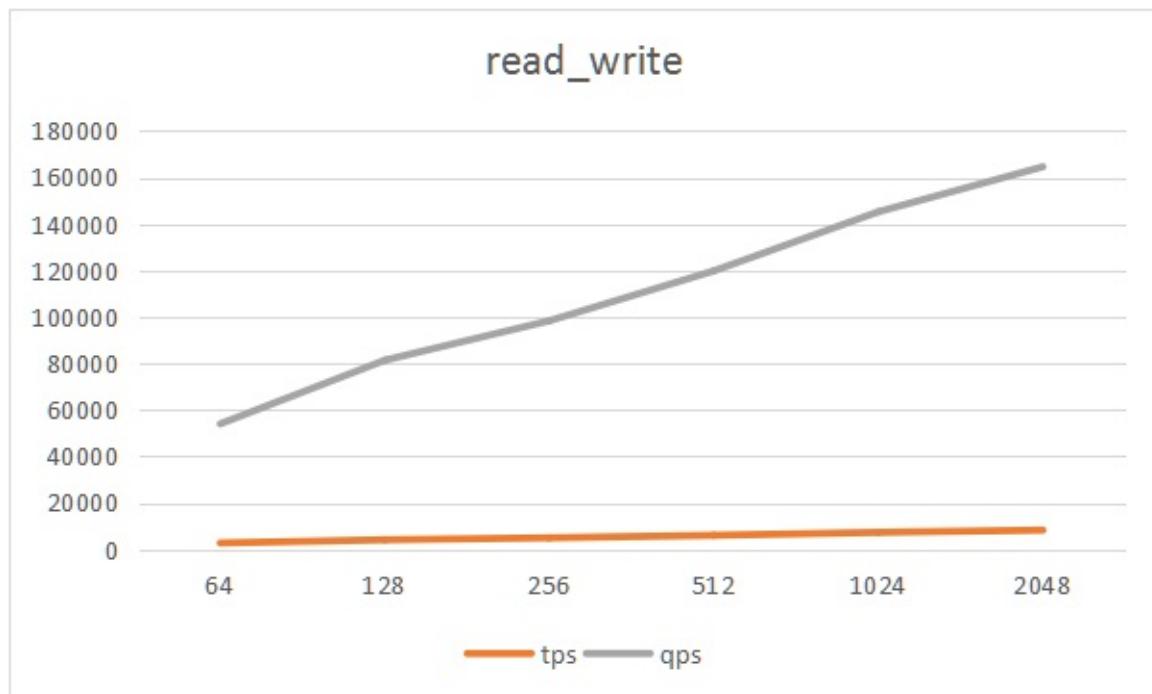
- write\_only

| type       | thread | tps      | qps       | min latency | avg latency | 95th latency | max latency |
|------------|--------|----------|-----------|-------------|-------------|--------------|-------------|
| write_only | 64     | 7882.92  | 47297.50  | 3.05        | 8.12        | 12.52        | 341.78      |
| write_only | 128    | 9780.01  | 58680.10  | 3.07        | 13.08       | 21.50        | 504.41      |
| write_only | 256    | 11450.20 | 68701.20  | 3.12        | 22.34       | 36.89        | 6874.97     |
| write_only | 512    | 13330.00 | 79979.20  | 3.04        | 38.39       | 65.65        | 6316.33     |
| write_only | 1024   | 14761.20 | 88567.40  | 3.30        | 68.39       | 118.92       | 5426.65     |
| write_only | 2048   | 16825.20 | 100951.00 | 3.25        | 121.50      | 223.34       | 5551.31     |



- **read\_write**

| type       | thread | tps     | qps       | min latency | avg latency | 95th latency | max latency |
|------------|--------|---------|-----------|-------------|-------------|--------------|-------------|
| read_write | 64     | 2698.01 | 53960.20  | 13.91       | 23.72       | 29.72        | 321.56      |
| read_write | 128    | 4066.40 | 81328.10  | 12.19       | 31.47       | 42.85        | 411.31      |
| read_write | 256    | 4915.23 | 98304.50  | 12.94       | 52.06       | 70.55        | 626.57      |
| read_write | 512    | 5988.96 | 119779.00 | 12.99       | 85.42       | 121.08       | 5023.20     |
| read_write | 1024   | 7260.19 | 145204.00 | 13.25       | 140.67      | 196.89       | 5767.52     |
| read_write | 2048   | 8228.84 | 164577.00 | 13.96       | 248.19      | 376.49       | 5475.98     |



### 3.1.3 总结

由于 TiDB 与 MySQL 在体系架构上的差别非常大，很多方面都很难找到一个共同的基准点，所以大家不要消耗过多精力在这类基准测试上，应该更多关注 TiDB 和 MySQL 在应用程序使用场景上的区别。MySQL 读扩容可以通过添加从库进行扩展，但单节点写入不具备扩展能力只能通过分库分表，而分库分表会增加开发维护方面成本。TiDB 不管是读流量还是写流量都可以通过添加节点的方式进行快速方便的扩展。

TiDB 设计的目标就是针对 MySQL 单台容量限制而被迫做出分库分表的场景，或者需要强一致性和完整分布式事务的场景。它的优势是通过尽量将并行计算下推到各个存储节点。对于小表（比如千万级以下）不适合 TiDB。因为数据量少导致表的 region 数量有限，发挥不了并行计算的优势。最极端的就是计数器表，几行记录高频更新，会变成存储引擎上的几个 KV 高频更新，然后数据都存储在一个 region 里，而这个 region 的流量都在一个计算节点上，再加上后台强一致性复制的开销，以及 TiDB 到 TiKV 的开销，最后表现出来的就是没有单个 MySQL 好。

## 3.2 TPC-C 基准性能测试

本文介绍如何对 TiDB 进行 [TPC-C](#) 测试。

### 1. TPC-C 简介

TPC 是一系列事务处理和数据库基准测试的规范。其中 TPC-C (Transaction Processing Performance Council) 是针对 OLTP 的基准测试模型。TPC-C 测试模型给基准测试提供了一种统一的测试标准，可以大体观察出数据库服务稳定性、性能以及系统性能等一系列问题。对数据库展开 TPC-C 基准性能测试，一方面可以衡量数据库的性能，另一方面可以衡量采用不同硬件软件系统的性价比，也是被业内广泛应用并关注的一种测试模型。

我们这里以经典的开源数据库测试工具 BenchmarkSQL 为例，其内嵌了 TPCC 测试脚本，可以对 PostgreSQL、MySQL、Oracle、TiDB 等行业内主流的数据库产品直接进行测试。

### 2. BenchmarkSQL

TPC-C 是一个对 OLTP (联机交易处理) 系统进行测试的规范，使用一个商品销售模型对 OLTP 系统进行测试，其中包含五类事务：

- NewOrder – 新订单的生成
- Payment – 订单付款
- OrderStatus – 最近订单查询
- Delivery – 配送
- StockLevel – 库存缺货状态分析

在测试开始前，TPC-C Benchmark 规定了数据库的初始状态，也就是数据库中数据生成的规则，其中 ITEM 表中固定包含 10 万种商品，仓库的数量可进行调整，假设 WAREHOUSE 表中有  $W$  条记录，那么：

- STOCK 表中应有  $W * 10$  万条记录（每个仓库对应 10 万种商品的库存数据）
- DISTRICT 表中应有  $W * 10$  条记录（每个仓库为 10 个地区提供服务）
- CUSTOMER 表中应有  $W * 10 * 3000$  条记录（每个地区有 3000 个客户）
- HISTORY 表中应有  $W * 10 * 3000$  条记录（每个客户一条交易历史）
- ORDER 表中应有  $W * 10 * 3000$  条记录（每个地区 3000 个订单），并且最后生成的 900 个订单被添加到 NEW-ORDER 表中，每个订单随机生成 5 ~ 15 条 ORDER-LINE 记录。

TPC-C 使用 tpmC 值 (Transactions per Minute) 来衡量系统最大有效吞吐量 (MQTh, Max Qualified Throughput)，其中 Transactions 以 NewOrder Transaction 为准，即最终衡量单位为每分钟处理的新订单数。

### 3. TiDB 测试环境部署

对于 1000 warehouse 我们将在 3 台服务器上部署集群。

在 3 台服务器的条件下，建议每台机器部署 1 个 TiDB，1 个 PD 和 1 个 TiKV 实例。

比如这里采用的机器硬件配置是：

| 类别   | 名称  |
|------|---|
| OS   | Linux (CentOS 7.3.1611)                             |
| CPU  | 40 vCPUs, Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz |
| RAM  | 128GB   |
| DISK | Optane 500GB SSD                                    |

因为该型号 CPU 是 NUMA 架构，建议先用 `taskset` 进行绑核，首先用 `lscpu` 查看 NUMA node，比如：

```
NUMA node0 CPU(s):    0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38
NUMA node1 CPU(s):    1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39
```

之后可以通过下面的命令来启动 TiDB：

```
nohup taskset -c 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38 bin/tidb-server && \
nohup taskset -c 1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39 bin/tidb-server
```

最后，可以选择部署一个 HAProxy 来进行多个 TiDB node 的负载均衡，推荐配置 nbproc 为 CPU 核数。

|       | <b>TIDB</b> | <b>TIKV</b> | <b>PD</b> |
|-------|-------------|-------------|-----------|
| node1 | 1           | 1           | 1         |
| node2 | 1           | 1           | 1         |
| node3 | 1           | 1           | 1         |

## 4. TiDB 调优配置

1、升高日志级别，可以减少打印日志数量，对性能有积极影响。

```
[log]
level = "error"
```

2、性能相关配置，可以根据机器的 CPU 核数设置，设置 TiDB 的 CPU 使用数量。

```
performance:
# Max CPUs to use, 0 use number of CPUs in the machine.
max-procs: 20
```

3、缓存语句数量设置，开启 TiDB 配置中的 prepared plan cache，可减少优化执行计划的开销。

```
prepared_plan_cache:
enabled: true
```

4、与 TiKV 客户端相关的设置，默认值为 16；当节点负载比较低时，可适当调大该值。

```
tikv_client:
# Max gRPC connections that will be established with each tikv-server.
grpc-connection-count: 4
```

5、本地事务冲突检测设置，并发压测时建议开启，可减少事务的冲突。

```
txn_local_latches:
# Enable local latches for transactions. Enable it when
# there are lots of conflicts between transactions.
enabled: true
```

## 5. TiKV 调优配置

1、调整日志级别，升高 TiKV 的日志级别同样有利于性能表现。

```
global:
  log-level = "error"
```

2、关闭 sync-log，由于TiKV 是以集群形式部署，在 Raft 算法的作用下，能保证大多数节点已经写入数据，除了对数据安全极端敏感的场景之外，raftstore 中的 sync-log 选项可以关闭。

```
[raftstore]
sync-log = false
```

3、块缓存配置，在 TiKV 中需要根据机器内存大小配置 RocksDB 的 block cache，以充分利用内存。以 20 GB 内存的虚拟机部署一个TiKV 为例，其 block cache 建议配置如下。

```
[storage.block-cache]
capacity = "10GB"
```

3、开始可以使用基本的配置，压测运行后可以通过观察 Grafana 并参考 [TiKV 调优说明]进行调整。如出现单线程模块瓶颈，可以通过扩展 TiKV 节点来进行负载均摊；如出现多线程模块瓶颈，可以通过增加该模块并发度进行调整。

## 6. BenchmarkSQL 配置

修改 benchmarksql/run/props.mysql 文件

```
conn=jdbc:mysql://{{HAPROXY-HOST}}:{{HAPROXY-PORT}}/tpcc?useSSL=false&useServerPrepStmts=true&useConfigs=maxPerformance
warehouses=1000 # 使用 1000 个 warehouse
terminals=500 # 使用 500 个终端
loadWorkers=32 # 导入数据的并发数
```

## 7. 导入数据

(导入数据通常是整个 TPC-C 测试中最耗时，也是最容易出问题的阶段)

1、首先连接到 TiDB-Server 并执行：

```
create database tpcc;
```

2、之后在 shell 中运行 BenchmarkSQL 建表脚本：

```
cd run && \
./runSQL.sh props.mysql sql.mysql/tableCreates.sql && \
./runSQL.sh props.mysql sql.mysql/indexCreates.sql
```

3、数据导入有两种方式可以选取，主要如下：

(1) 直接使用 BenchmarkSQL 导入（根据机器配置这个过程可能会持续几个小时）；

```
./runLoader.sh props.mysql
```

(2) 通过 TiDB Lightning 导入（由于导入数据量随着 warehouse 的增加而增加，当需要导入 1000 warehouse 以上数据时，可以先用 BenchmarkSQL 生成 csv 文件，再将文件通过 TiDB Lightning（以下简称 Lightning）导入的方式来快速导入。生成的 csv 文件也可以多次复用，节省每次生成所需要的时间）；

a、修改 BenchmarkSQL 的配置文件 warehouse 的 csv 文件需要 77 MB 磁盘空间，在生成之前要根据需要分配足够的磁盘空间来保存 csv 文件。可以在 `benchmarksql/run/props.mysql` 文件中增加一行：

```
fileLocation=/home/user/csv/ # 存储 csv 文件的目录绝对路径, 需保证有足够的空间
```

因为最终要使用 Lightning 导入数据，所以 csv 文件名需要符合 Lightning 要求，即 `{database}.{table}.csv` 的命名法。可以将以上配置改为：

```
fileLocation=/home/user/csv/tpcc. # 存储 csv 文件的目录绝对路径 + 文件名前缀 (database)
```

这样生成的 csv 文件名将会是类似 `tpcc.bmsql_warehouse.csv` 的样式，符合 Lightning 的要求。

b、生成 csv 文件

```
./runLoader.sh props.mysql
```

c、修改 inventory.ini

建议手动指定清楚部署的 IP、端口、目录，避免各种冲突问题带来的异常。

```
[importer_server]
IS1 ansible_host=172.16.5.34 deploy_dir=/data2/is1 tikv_importer_port=13323 import_dir=/data2/import

[lightning_server]
LS1 ansible_host=172.16.5.34 deploy_dir=/data2/ls1 tidb_lightning_pprof_port=23323 data_source_dir=/home/user/csv
```

d、修改 conf/tidb-lightning.yml

```
mydumper:
  no-schema: true
csv:
  separator: ','
  delimiter: ''
  header: false
  not-null: false
  'null': 'NULL'
  backslash-escape: true
  trim-last-separator: false
```

e、部署 Lightning 和 Importer

```
ansible-playbook deploy.yml --tags=lightning
```

f、启动

- 登录到部署 Lightning 和 Importer 的服务器；
- 进入部署目录；
- 在 Importer 目录下执行 `scripts/start_importer.sh`，启动 Importer；
- 在 Lightning 目录下执行 `scripts/start_lightning.sh`，开始导入数据。

由于是用 ansible 进行部署的，可以在监控页面看到 Lightning 的导入进度，或者通过日志查看导入是否结束。数据导入完成之后，可以运行 `sql.common/test.sql` 进行数据正确性验证，如果所有 SQL 语句都返回结果为空，即为数据导入正确。

## 8. 运行测试

执行 BenchmarkSQL 测试脚本：

```
nohup ./runBenchmark.sh props.mysql &> test.log &
```

运行结束后通过 `test.log` 查看结果：

```
07:09:53,455 [Thread-351] INFO jTPCC : Term-00, Measured tpmC (NewOrders) = 77373.25
07:09:53,455 [Thread-351] INFO jTPCC : Term-00, Measured tpmTOTAL = 171959.88
07:09:53,455 [Thread-351] INFO jTPCC : Term-00, Session Start      = 2019-03-21 07:07:52
07:09:53,456 [Thread-351] INFO jTPCC : Term-00, Session End        = 2019-03-21 07:09:53
07:09:53,456 [Thread-351] INFO jTPCC : Term-00, Transaction Count = 345240
```

tpmC 部分即为测试结果。

## 第4章 跨数据中心方案

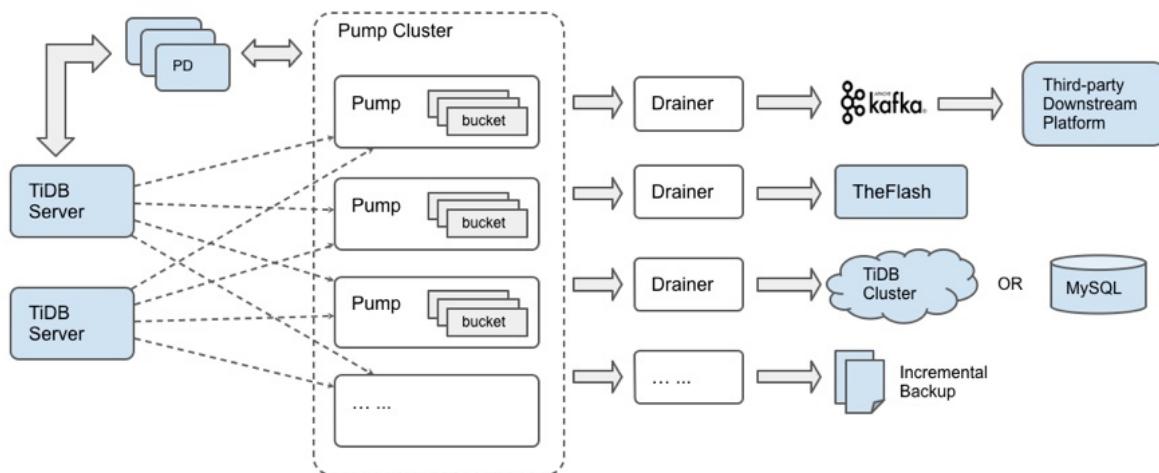
在部署 TiDB 集群的不同需求下，拓扑设计会涉及到跨数据中心的需求，对于不同的跨数据中心方案，本章做了介绍。

## 4.1 两中心异步复制方案 (binlog 复制)

### TiDB-Binlog 简介

TiDB Binlog 组件用于收集 TiDB 的 binlog，并提供实时备份和同步功能。该组件在功能上类似于 MySQL 的主从复制，MySQL 的主从复制依赖于记录的 binlog 文件，TiDB Binlog 组件也是如此，主要的不同点是 TiDB 是分布式的，因此需要收集各个 TiDB 实例产生的 binlog，并按照事务提交的时间排序后才能同步到下游。如果你需要部署 TiDB 集群的从库，或者想订阅 TiDB 数据的变更输出到其他的系统中，TiDB Binlog 则是必不可少的工具。

### TiDB Binlog 组件架构



该架构的主要特点是：

1. 多个 Pump 形成一个集群，可以水平扩容，各个 Pump 可以均匀地承担业务的压力。
2. TiDB 通过内置的 Pump Client 将 binlog 分发到各个 Pump，即使有部分 Pump 出现故障也不影响 TiDB 的业务。
3. Pump 内部实现了简单的 kv 来存储 binlog，方便对 binlog 数据的管理。
4. binlog 排序逻辑由 Pump 来做，而 Pump 是可扩展的，这样就能提高整体的同步性能。
5. Drainer 只需要依次读取各个 Pump 的 binlog 进行归并排序，这样可以大大节省内存的使用，同时也更容易做内存控制。

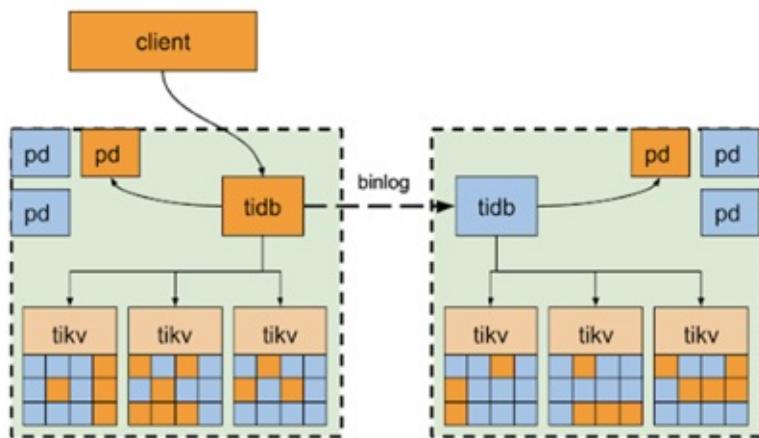
### 两中心异步复制方案

利用 TiDB-Binlog 异步复制数据的特性，我们可以利用其搭建一套主从集群，准实时的将主机群的数据同步到异地的从集群。这个场景在金融行业是比较常见的，除了承接主要业务访问的主机群需要两地三中心的高可用部署之外，还要有一个数据异地容灾备份系统。这样的考虑出发点在于当出现极端场景，导致主机群完全无法提供服务后，异地容灾机房可以顶上来；同时异地容灾机房还可以承载业务 T+1 的报表查询业务，减少了对主机群的影响。

两中心双集群部署方案，顾名思义是在两个中心分别部署独立 TiDB 集群，通过 binlog 进行数据同步，类似于传统的 MySQL 中 master/slave 主从复制方案。一个集群作为主集群，负责应用接入提供读写服务，另一个集群作为从集群，负责同步主集群数据以及故障接管。

#### 优势

- 避免因网络传输延迟带来的性能影响。
- 两中心集群各自独立，减少集群间互相影响，减少两集群同时故障的概率。



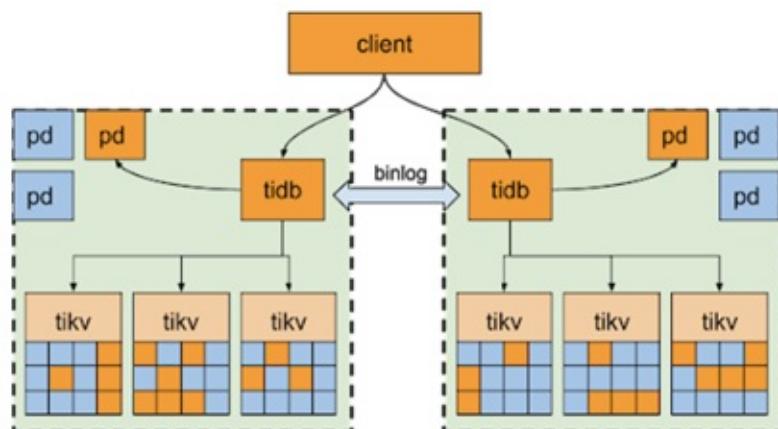
当主集群所在的中心发生故障时，业务可以切换至从集群。因两集群采用 binlog 异步复制，无法保证数据一致性，所以极端情况下会有部分数据缺失。

### 劣势

- 无法保证两集群数据一致性。 \*正常情况下从集群只负责同步数据，存在较大的资源浪费。

### 双集群互为备份方案

另一种两中心架构部署方案是在之前方案的基础上进行一定的延伸，同样两个中心部署两套独立 TiDB 集群，将业务拆分为两个库，每个库分别放在一个集群（数据中心）中，接入每个数据中心的业务请求只访问本地库，两个集群之间通过 binlog 将本数据中心业务所涉及的库中的数据变更同步到对端，互为备份。



这样就能实现两数据中心分库双活，解决从集群资源浪费的问题，但两集群的架构方案仍然无法保证数据一致性，而且此分库方案对应用可能有一定侵入性。

### 环境要求

- 服务器要求

Pump 和 Drainer 均可部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台上。在开发、测试和生产环境下，对服务器硬件配置的要求和建议如下：

| 服务      | 部署数量 | CPU | 磁盘   | 内存  |
|---------|------|-----|--|-----|
| Pump    | 3    | 8核+ | SSD, 200 GB+                                   | 16G |
| Drainer | 1    | 8核+ | SAS, 100 GB+ (如果输出 binlog 为本地文件，磁盘大小视保留数据天数而定) | 16G |

- 网络要求

单数据中心内部网络要求：

业务网络满足千兆及以上带宽，私网满足万兆及以上带宽。

跨数据中心之间网络要求：

跨数据中心带宽至少 300M 及以上，同时确保数据中心内部及跨数据中心的所有节点之间私网互联互通，业务网络之间不需要互联互通。

## 搭建步骤

### 1. 部署 Pump

#### i. 中控机上修改 `tidb-ansible/inventory.ini` ( 这里默认用户在上下游已经成功部署好了 TiDB 集群 )

- i. 设置 `enable_binlog = True`，表示 TiDB 集群开启 binlog。

```
enable_binlog = True
```

#### 2. 为 `pump_servers` 主机组添加部署机器 IP。

```
```
## Binlog Part
[pump_servers]
172.16.10.72
172.16.10.73
172.16.10.74
```
```

#### 3. 默认 Pump 保留 7 天数据，如需修改可修改 `tidb-ansible/conf/pump.yml` (TiDB 3.0.2 及之前版本中为 `tidb-ansible/conf/pump-cluster.yml`) 文件中 `gc` 变量值，并取消注释。

```
```
global:
  # an integer value to control the expiry date of the binlog data, which indicates for how long (in days) the bin
  log data would be stored
  # must be bigger than 0
  # gc: 7
```
```

#### 4. 请确保部署目录有足够的空间存储 binlog，详见[调整部署目录](<https://pingcap.com/docs-cn/stable/how-to/deploy/orchestrated/ansible/#E8%80%83%E6%95%B4%E9%83%A8%E7%BD%B2%E7%9B%AE%E5%BD%95>)，也可为 Pump 设置单独的部署目录。

```
```
## Binlog Part
[pump_servers]
pump1 ansible_host=172.16.10.72 deploy_dir=/data1/pump
pump2 ansible_host=172.16.10.73 deploy_dir=/data2/pump
pump3 ansible_host=172.16.10.74 deploy_dir=/data3/pump
```
```

### 1. 部署并启动含 Pump 组件的 TiDB 集群。

#### i. 部署 `pump_servers` 和 `node_exporters`

```
$ ansible-playbook deploy.yml --tags=pump -l ${pump1_ip},${pump2_ip},[$alias1_name,$alias2_name]
### 上述命令中，逗号后不要加空格，否则会报错
```

#### i. 启动 `pump_servers`

```
$ ansible-playbook start.yml --tags=pump
```

i. 更新并重启 tidb\_servers

```
$ ansible-playbook rolling_update.yml --tags=tidb
```

i. 更新监控信息

```
$ ansible-playbook rolling_update_monitor.yml --tags=prometheus
```

2. 查看 Pump 服务状态

使用 binlogctl 查看 Pump 服务状态，pd-urls 参数请替换为集群 PD 地址，结果 State 为 online 表示 Pump 启动成功

```
$ cd /home/tidb/tidb-ansible &&
resources/bin/binlogctl -pd-urls=http://172.16.10.72:2379 -cmd pumps
INFO[0000] pump: {NodeID: ip-172-16-10-72:8250, Addr: 172.16.10.72:8250, State: online, MaxCommitTS: 4030515256908
84099, UpdateTime: 2018-12-25 14:23:37 +0800 CST}
INFO[0000] pump: {NodeID: ip-172-16-10-73:8250, Addr: 172.16.10.73:8250, State: online, MaxCommitTS: 4030515257039
91299, UpdateTime: 2018-12-25 14:23:36 +0800 CST}
INFO[0000] pump: {NodeID: ip-172-16-10-74:8250, Addr: 172.16.10.74:8250, State: online, MaxCommitTS: 4030515257173
60643, UpdateTime: 2018-12-25 14:23:35 +0800 CST}
```

1. 记录 TSO 断点信息

Drainer 在初次启动时需要获取 initial\_commit\_ts 这个时间戳信息，为了保证数据的完整性，需要进行全量数据的备份与恢复。此时 initial\_commit\_ts 的值必须是全量备份的时间戳。为了保证这个时间戳包含备份数据之后的所有数据变更，我们把这一步操作放到全备之前来做。

```
$ cd /home/tidb/tidb-ansible &&
resources/bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd generate_meta
INFO[0000] [pd] create pd client with endpoints [http://192.168.199.118:32379]
INFO[0000] [pd] leader switches to: http://192.168.199.118:32379, previous:
INFO[0000] [pd] init cluster id 6569368151110378289
2018/06/21 11:24:47 meta.go:117: [info] meta: &{CommitTS:400962745252184065}
```

该命令会输出 meta: &{CommitTS:400962745252184065}，其中 CommitTS 的值即所需的时间戳。

1. 逻辑备份全量数据，使用 mydumper 备份主库的数据。
2. 全量数据传输并恢复到备端，使用 loader 恢复数据。
3. 部署 Drainer 服务并开启 Drainer 同步实时增量数据

i. 修改 tidb-ansible/inventory.ini 文件。为 drainer\_servers 主机组添加部署机器 IP，initial\_commit\_ts 请设置为获取的 initial\_commit\_ts，仅用于 Drainer 第一次启动。` `` [drainer\_servers]

```
drainer_mysql ansible_host=172.16.10.71 initial_commit_ts="402899541671542785"
```

2. 修改配置文件

```
cd /home/tidb/tidb-ansible/conf && cp drainer.toml drainer_mysql_drainer.toml && vi drainer_mysql_drainer.toml
```

```
```
[syncer]
# downstream storage, equal to --dest-db-type
# Valid values are "mysql", "file", "tidb", "kafka".
db-type = "mysql"
# the downstream MySQL protocol database
[syncer.to]
host = "172.16.10.72"
user = "root"
password = "123456"
port = 3306
```

i. 部署 Drainer

```
ansible-playbook deploy_drainer.yml
```

ii. 启动 Drainer

```
ansible-playbook start_drainer.yml
```

这时 Drainer 已经开始同步数据了，在监控中能够观察到数据同步情况。

## 数据校验

为了验证主从集群的数据是一致的，我们提供了数据校验工具：sync-diff-inspector，该工具同时提供了数据修复功能（适用于修复少量不一致的数据），具体使用方法可以在登录 PingCAP 官网搜索 sync-diff-inspector，这里就不再赘述。

## 总结

TiDB-Binlog 提供了两套 TiDB 集群之间，或者上游是 TiDB 下游是 MySQL 之间的数据同步。同时 Drainer 提供了输出到 PB 文件，通过这个功能可以完成增量数据的备份目的。需要注意的地方是 TiDB-Binlog 的同步是异步的，主从数据同步延迟受上下游集群硬件性能影响而有所不同，正常情况下延迟能稳定在秒级。其次使用过程中需要注意同步表必须要有主键。定期对主从集群的校验能够及时发现数据同步发生的异常问题。使用中如有其它问题可以联系 PingCAP 官方人员跟进解决。

## 4.2 两中心同步复制方案（三副本）

### 4.2.1 Raft 算法

1.Raft 是一种分布式一致性算法，在 TiDB 集群的多种组件中，PD 和 TiKV 都通过 Raft 实现了数据的容灾。Raft 的灾难恢复能力通过如下机制实现：

- Raft 成员的本质是日志复制和状态机。Raft 成员之间通过复制日志来实现数据同步；Raft 成员在不同条件下切换自己的成员状态，其目标是选出 leader 以提供对外服务。
- Raft 是一个表决系统，它遵循多数派协议，在一个 Raft Group 中，某成员获得大多数投票，它的成员状态就会转变为 leader。也就是说，当一个 Raft Group 还保有大多数节点（majority）时，它就能够选出 leader 以提供对外服务。

2.Raft 算法本身以及 TiDB 中的 Raft 实现都没有限制一个 Raft Group 的副本数，这个副本数可以为任意正整数，当副本数为 n 的时候，一个 Raft Group 的可靠性如下：

- 若 n 为奇数，该 Raft Group 可以容忍  $(n-1)/2$  个副本同时发生故障
- 若 n 为偶数，该 Raft Group 可以容忍  $n/2 - 1$  个副本同时发生故障

3.在一般情况下，我们建议将 Raft Group 的副本数设置为奇数，其原因如下：

- 避免造成存储空间的浪费：三副本可以容忍 1 副本故障，增加 1 个副本变为 4 副本后，容灾能力维持不变。
- 当副本数为偶数时，如果发生了一个网络隔离，刚好将隔离开的两侧的副本数划分为两个  $n/2$  副本的话，由于两边都得不到大多数成员，因此都无法选出 leader 提供服务，这个网络隔离将直接导致整体的服务不可用。
- 当副本数为奇数时，在只发生一个网络隔离的情况下，网络隔离的两侧中总有一侧能分到大多数的成员，可以选出 leader 以提供服务。

4.遵循 Raft 可靠性的特点，放到现实场景中：

- 想克服任意 1 台服务器的故障，应至少提供 3 台服务器。
- 想克服任意 1 个机柜的故障，应至少提供 3 个机柜。
- 想克服任意 1 个数据中心（机房）的故障，应至少提供 3 个数据中心。
- 想应对任意 1 个城市的灾难场景，应至少规划 3 个城市用于部署。

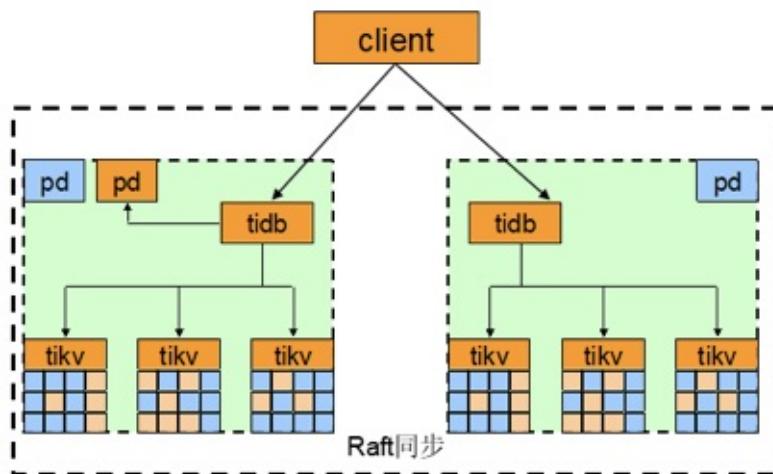
可见，原生的 Raft 协议对于偶数副本的支持并不是很友好，同城 3 中心或许是最适合部署 Raft 的高可用及容灾方案。而在现实情况中，大多数采用 on-premise 部署的用户中，很少具备同城 3 中心的条件。比如数字化程度非常高的银行业，基于传统单机系统点对点复制的特性，绝大多数银行都只建设了同城 2 中心，两地 3 中心的基础设施。

在同城两中心部署原生的 Raft，主备机房按 2:1 的比例来分配 3 个副本，由于网速的差异，备机房会存在异步复制的副本。当主机房由于故障无法恢复时，备机房仅剩的 1 个副本是无法保障 CAP 中的一致性的。

TiKV 对 Raft 做了一些功能上的扩展，基于这个扩展它可以在一定程度上克服主机房故障后使用备机房难以恢复一致性数据的难题。但在这之前，我们先介绍一个基于原生 Raft 偶数副本的双机房部署方案。

### 4.2.2 两中心单集群偶数副本方案

两中心单集群方案，即两种数据中心只部署一个 TiDB 集群，两中心间的数据同步通过集群自身内部（Raft 协议）完成。两中心可同时对外进行读写服务，任意中心发生故障不影响数据一致性。需要注意的是这样的部署方式不仅在设备资源上需要对称部署，副本也需对称，即两中心副本数相同，整个集群为偶数副本。



如上图所示，假设每个中心有 3 个数据副本，整个集群 6 副本，任意一个中心发生灾难，虽然会短暂影响业务连续性，但因为 Raft 协议下多数副本数据强一致的特性，数据不会丢失，而且可以通过降低副本数实现快速恢复业务，即把集群改成 3 副本拉起业务。

优势：

- 保证两中心数据一致性。
- 两中心同时对外提供服务，资源使用合理。

劣势：

- 两中心间网络传输延迟会对集群性能带来影响，不建议远程异地中心使用该架构方案。
- 缺少降级服务机制，机房间网络故障或任一机房整体故障都会导致服务中断，需要通过人工介入的方式来恢复数据，虽然 RPO (Recovery Point Objective) 业务系统所能容忍的数据丢失量为 0，但 RTO (Recovery Time Objective) 灾难发生到恢复的时长较高。

### 4.2.3 两中心单集群三副本同步复制方案

原生的 Raft 协议无法通过人工来干预一个 Raft Group 的 quorum (大多数)，quorum 只能通过 Raft Group 最初的副本数设置自动生成。为了弥补以上方案缺少降级服务的能力不足，TiKV 中提供了配置 Raft Group 的 quorum 属性的接口，并推出了一个全新的方案，该方案采用两中心三副本按照 2:1 方式进行部署，不限定 Raft leader 位置，初始配置 quorum 为 3，即同步写三副本，当发生单个网络隔离时自动缩减 quorum 到 2，让集群继续提供服务。

#### 1. TiKV 技术细节

TiKV 已具备 custom quorum 接口，可供运行时动态修改 quorum。初始化时，采用  $Q1(n) = \text{floor}((n + 1) / 2) + 1$  配置 quorum，譬如三副本 quorum 就是 3。这样保证所有节点写入时才能响应，即 leader 和所有 follower 都写完。

##### (1) 网络隔离时

TiKV 提供配置项 `max-idle-time` 可以控制一个 region 所在的 raft group 中没有 leader 的最长时间。通过调整配置项 `max-idle-time` 的大小可以切换数据保护模式（类似 Oracle DG 的最大可用/最大性能/最大保护模式）。如果一个 region 所在的 raft group 长时间没有 leader，则将 quorum 修改为  $Q2(n) = \text{floor}(n / 2) + 1$ 。三副本情况下，只需要 2 个副本存活即可工作。等所有副本的日志都复制到当前 term 且副本间的 Raft log gap 小于指定值以后，就可以识别为所有副本可回到同步复制模式，重新把 quorum 修改为  $Q1(n)$ ，这个修改由 PD 自动完成。

##### (2) 性能开销

本方案在三副本的情况下，写入需要全部落盘才能响应请求，所以写入性能会有所下降。（根据 5 副本 & 3 副本性能对比测试，写入性能衰减在 5% 以内）。

#### 2. TiDB, PD 部署方式

TiDB 采用一般的高可用方式部署，在两个机房部署若干 tidb-server，各自连接到各自机房的负载均衡器对外提供服务。PD 采用一般的三副本 2:1 部署，不增加 quorum 配置，主机房故障时通过人工重建恢复 PD 服务 (pd-recover)。

### 3. 应用部署技术细节

主机房承载两个数据副本，灾备机房承载一个数据副本。应用可以部署在两个机房，连接到各自机房的负载均衡器，需要注意的是阻塞窗口 `max-idle-time` 的设置不应太短，为了防止在极端情况下的数据丢失。当应用在主机房启动时，它不需要灾备机房参与也可以正常工作，当两机房网络断开一段时间，数据复制由同步转为异步后，应用还可以正常工作，假设此时主机房整体故障，那么灾备机房的数据是不可用的。设想一次机房故障的开始于两机房的数据库专用网络，继而发展为全机房故障，就会遇到这个情况。

#### 4.2.4 应急处置预案

##### (1) 两机房间网络断开

网络抖动期间，在阻塞窗口 `max-idle-time` 时间内，应用请求被阻塞，此时会有网络延迟相关报警发出，用户收到数据库和应用的报警并根据报警定位问题及产生原因。网络连接在 `max-idle-time` 时间内恢复的话，应用阻塞也会随之解除，所有 region 继续保持同步复制状态。假设网络断开超过 `max-idle-time`，应用阻塞会被解除，同时数据库发出同步复制转为异步复制的报警，待用户定位问题并恢复网络后，集群可以自动转为同步复制状态，并发出异步复制转为同步复制的报警解除。

##### (2) 灾备机房整体宕机

灾备机房整体宕机，主机房的 TiKV 无法继续保持同步复制模式，进入阻塞窗口，在阻塞窗口 `max-idle-time` 时间内，应用请求被阻塞此时会有网络延迟相关报警发出（主机房一侧无法判断是网络故障还是灾备机房故障，只会以网络延迟报警形式发出），用户收到数据库和应用的报警并根据报警定位问题及产生原因。超过 `max-idle-time`，应用阻塞会被解除，同时数据库发出同步复制转为异步复制的报警，机房整体故障需要较长时间恢复，数据库在这期间会持续发出网络连接相关报警，可以进入报警系统暂时人为禁用相关报警，集群此时处于两副本的状态，在灾备机房离线时间超过 `max-store-down-time` 之后，PD 会在可用的位置上补齐离线的第三个副本，副本补齐速度视 PD 调度参数而定。待用户恢复灾备机房后，集群可以自动将副本调度会灾备机房，继而转为同步复制状态，并发出异步复制转为同步复制的报警解除。用户需要进入报警系统恢复之前禁用的网络延迟报警。

##### (3) 主机房整体宕机

主机房整体宕机，TiKV 由于 region 丢失了大多数副本而停止服务，旁路报警系统发出网络延迟过高相关报警，用户收到数据库和应用的报警并根据报警，确认主机房整体故障，并且难以在短时间内恢复。使用 `pd-recover` 重建 PD（参考官方手册，并定期演练以确保流程的可用），使用 `tikv-ctl` 从仅存的一副本恢复数据（参考官方手册，并定期演练以确保流程的可用），数据恢复后通过客户端连接到数据库服务，确保数据库可用后启动应用到数据库的连接，恢复业务。接下来的操作视灾备机房环境而定，如果可以提供冗余的服务器，可以在线操作集群的扩容与扩充副本。待主机房恢复之后，需要确保主机房部署的集群进程保持停止的状态，并清除主机房的全部组件的部署目录，之后通过扩容的方式将主机房纳入到集群中，扩副本的操作可以与扩容操作同步进行。

### 4. 关于 RPO 和 RTO

- RPO = 0
  - 排除数据同步转异步，降级以提供服务之后继而发生主机房整体故障的情况，可以实现数据 0 丢失。
- RTO 在不同的场景下的计算方法：
  - 当两机房网络断开时，RTO 为阻塞窗口 `max-idle-time` 所设置的时间。
  - 当灾备机房整体宕机时，RTO 为阻塞窗口 `max-idle-time` 所设置的时间。
  - 主机房整体宕机恢复时间为，RTO 为报警响应时间、重建 PD 操作时间（熟练 DBA 分钟级）、恢复 TiKV 单副本时间（熟练 DBA 分钟级）、集群验证时间（用于验证数据库可用，应用连接顺畅）的总时常，恢复后的单副本集群可以直接提供服务，后续的扩容以及扩副本操作可以在线执行。



## 4.3 两地三中心

TiDB 分布式数据库两地三中心建设架构基于 Raft 算法，保证了集群数据一致性和高可用。两地是同城、异地，同城双中心指在同城或临近城市建立独立数据中心，双中心通过高速链路实时同步数据，网络延迟相对较小，另外一个数据中心在异地城市。在这种场景下，可以把业务流量同时派发到同城两个数据中心，通过控制 Region leader 和 PD leader 分布在同城两个数据中心。

### 1.1 架构图

以北京和西安为例阐述 TiDB 分布式数据库两地三中心架构部署模型，例中北京有两个机房 IDC1 和 IDC2，异地西安一个机房 IDC3。北京同城两机房之间网络延迟低于 3ms，北京与西安之间网络使用 ISP 专线，延迟约 20ms。

如下图 1 所示为集群部署架构图，具体如下：

- 部署采用主从架构，主集群作为生产集群，承担日常生产服务，从集群同通过 TiDB-Binlog 异步同步主集群数据库，作为备份数据库使用。
- 生产集群采用两地三中心，分别为北京 IDC1，北京 IDC2，西安 IDC3；
- 生产集群采用 5 副本模式，其中 IDC1 和 IDC2 分别放 2 个副本，IDC3 放 1 个副本；TiKV 按机柜打 Label，既每个机柜上有一份副本。
- 从集群与主机群直接通过 TiDB-Binlog 完成数据同步工作。

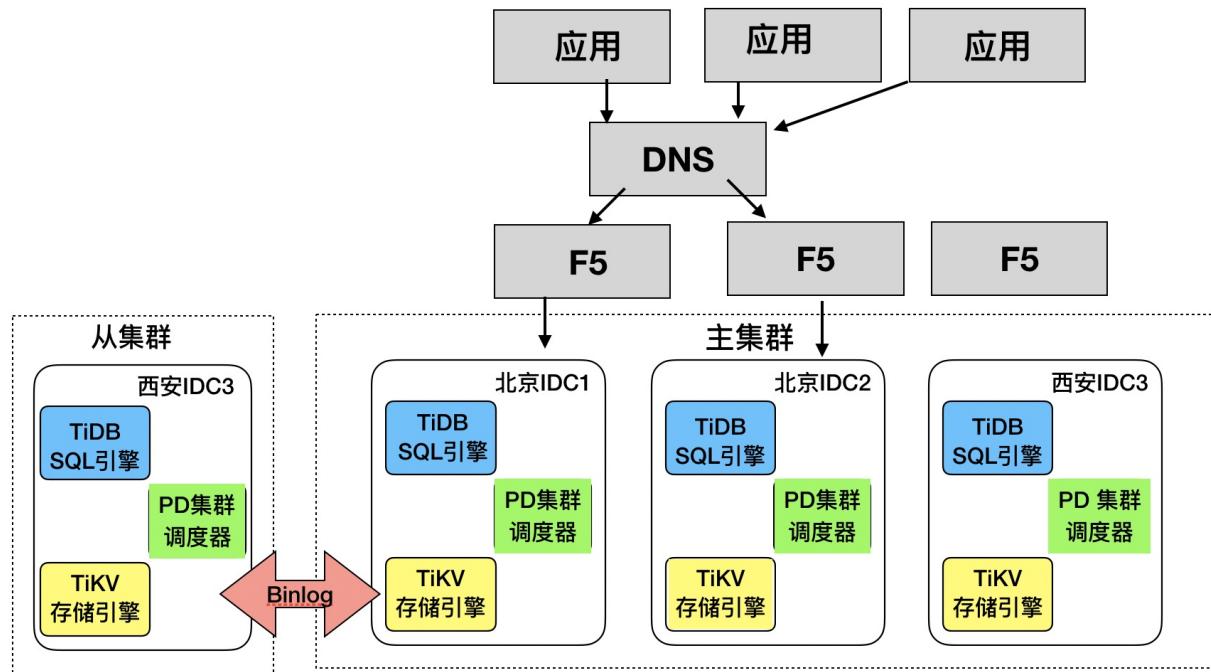


图 1 两地三中心集群架构图

该架构具备高可用和容灾备份能力，同时通过 PD 调度限制了 region leader 尽量只出现在同城的两个数据中心，这相比于三数据中心 region leader 分布不受限制的方案优势如下：

- 写入速度更优。
- 两中心同时提供服务资源利用率更高。
- 可保证任一数据中心失效后，服务可用并且不发生数据丢失。

因为是基于 Raft 算法，同城两个数据中心同时失效时，只有一个节点存在，不满足 Raft 算法大多数节点存在要求，最终将导致集群不可用及部分数据丢失，这种情况出现的概率是比较小的；另外由于使用到了网络专线，导致该架构成本较高。

## 1.2 部署说明

下面具体介绍两地三中心架构部署详情。

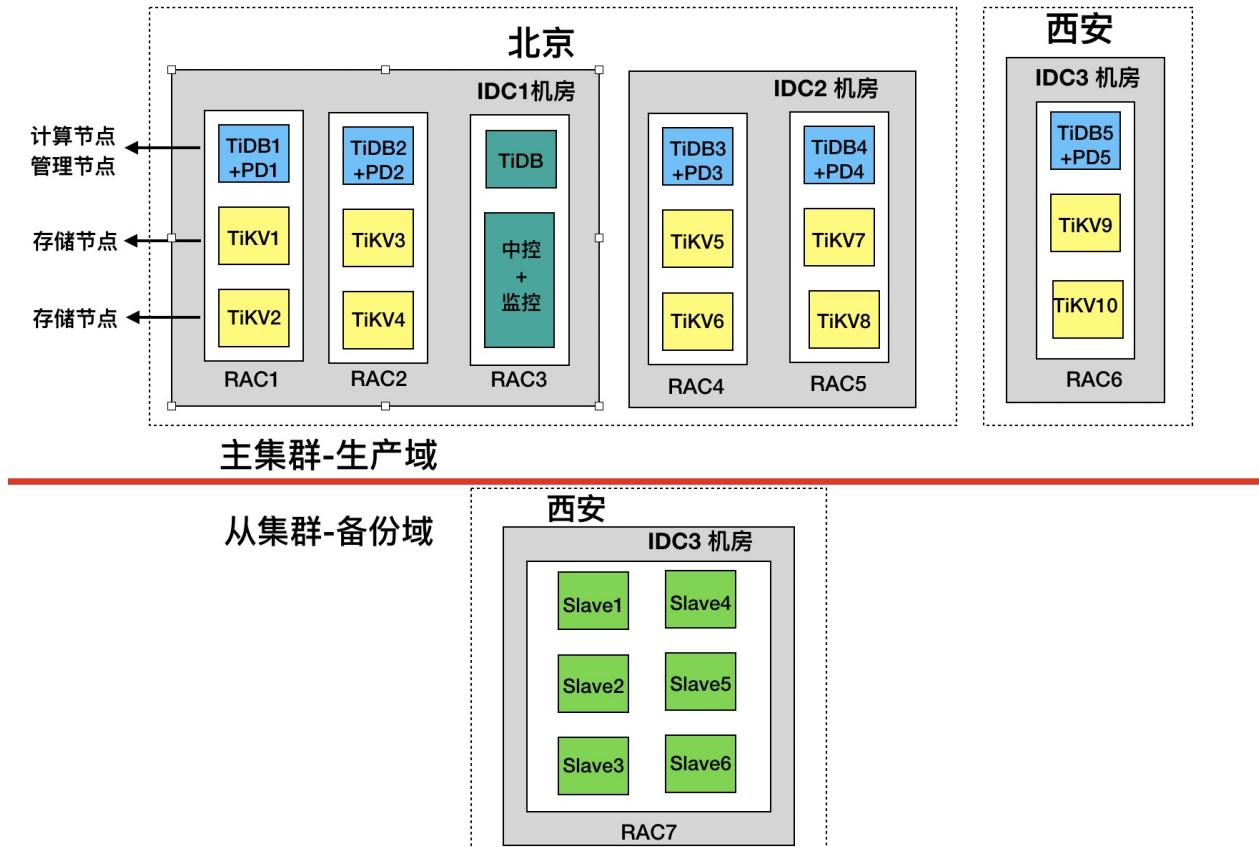


图2 两地三中心配置详图

北京、西安两地三中心配置详解：

- 如图 2 所示，北京有两个机房 IDC1 和 IDC2，机房 IDC1 中有三套机架 RAC1、RAC2、RAC3，机房 IDC2 有机架 RAC4、RAC5；西安机房 IDC3 有机架 RAC6、RAC7，其中机架 RAC7 中的服务器搭建了一套新的 TiDB 分布式数据库，作为灾备准实时同步主机群的数据，同时一些 T+1 的业务报表也放到这个集群中执行，避免影响主集群。
- 如图中 RAC1 机架所示，TiDB、PD 服务部署在同一台服务器上，还有两台 TiKV 服务器；每台 TiKV 服务器部署 2 个 TiKV 实例 (tikv-server)，即 TiKV 服务器上每块 PCIe SSD 上单独部署一个 TiKV 实例；RAC2、RAC4、RAC5、RAC6 类似。
- 机架 RAC3 上安放 TiDB-Server 及中控 + 监控服务器。部署 TiDB-Server，用于日常管理维护、备份使用。中控 + 监控服务器上部署 TiDB-Ansible、Prometheus，Grafana 以及恢复工具；
- 从集群采用混合部署方式，每台服务器上部署 2 个 TiKV 实例，其中的 3 台部署 TiDB 及 PD。
- 备份服务器上部署 Mydumper 及 Drainer，Drainer 以输出 PB 文件的方式将增量数据保存到指定位置，实现增量备份的目的。

## 2. 两地三中心部署配置

### 2.1 inventory 配置模板

## inventory.ini 配置模板信息

```

## TiDB Cluster Part
[tidb_servers]
TiDB-10 ansible_host=10.63.10.10      deploy_dir=/data/tidb_cluster/tidb
TiDB-11 ansible_host=10.63.10.11      deploy_dir=/data/tidb_cluster/tidb
TiDB-12 ansible_host=10.63.10.12      deploy_dir=/data/tidb_cluster/tidb
TiDB-13 ansible_host=10.63.10.13      deploy_dir=/data/tidb_cluster/tidb
TiDB-14 ansible_host=10.63.10.14      deploy_dir=/data/tidb_cluster/tidb

[tikv_servers]
TiKV-30 ansible_host=10.63.10.30      deploy_dir=/data/tidb_cluster/tikv  tikv_port=20171  labels="dc=1, rack=1, zone
=1, host=30"
TiKV-31 ansible_host=10.63.10.31      deploy_dir=/data/tidb_cluster/tikv  tikv_port=20171  labels="dc=1, rack=2, zone
=2, host=31"
TiKV-32 ansible_host=10.63.10.32      deploy_dir=/data/tidb_cluster/tikv  tikv_port=20171  labels="dc=2, rack=3, zone
=3, host=32"
TiKV-33 ansible_host=10.63.10.33      deploy_dir=/data/tidb_cluster/tikv  tikv_port=20171  labels="dc=2, rack=4, zone
=4, host=33"
TiKV-34 ansible_host=10.63.10.34      deploy_dir=/data/tidb_cluster/tikv  tikv_port=20171  labels="dc=3, rack=5, zone
=5, host=34"

[pd_servers]
PD-10  ansible_host=10.63.10.10      deploy_dir=/data/tidb_cluster/pd
PD-11  ansible_host=10.63.10.11      deploy_dir=/data/tidb_cluster/pd
PD-12  ansible_host=10.63.10.12      deploy_dir=/data/tidb_cluster/pd
PD-13  ansible_host=10.63.10.13      deploy_dir=/data/tidb_cluster/pd
PD-14  ansible_host=10.63.10.14      deploy_dir=/data/tidb_cluster/pd

[spark_master]

[spark_slaves]

[lightning_server]

[importer_server]

## Monitoring Part
# prometheus and pushgateway servers
[monitoring_servers]
proto-60 ansible_host=10.63.10.60      prometheus_port=8380  deploy_dir=/data/tidb_cluster/prometheus

[grafana_servers]
graf-60  ansible_host=10.63.10.60      grafana_port=8690  grafana_collector_port=8691  deploy_dir=/data/tidb_cluster/
grafana

# node_exporter and blackbox_exporter servers
[monitored_servers]
10.63.10.10
10.63.10.11
10.63.10.12
10.63.10.13
10.63.10.14
10.63.10.30
10.63.10.31
10.63.10.32
10.63.10.33
10.63.10.34

[alertmanager_servers]
alertm  ansible_host=10.63.10.60      deploy_dir=/data/tidb_cluster/alertmanager

[kafka_exporter_servers]

## Binlog Part
[pump_servers]
pump-10  ansible_host=10.63.10.10      deploy_dir=/data/tidb_cluster/pump
pump-11  ansible_host=10.63.10.11      deploy_dir=/data/tidb_cluster/pump
pump-12  ansible_host=10.63.10.12      deploy_dir=/data/tidb_cluster/pump
pump-13  ansible_host=10.63.10.13      deploy_dir=/data/tidb_cluster/pump

```

```
[drainer_servers]

## Group variables
[pd_servers:vars]
location_labels = ["dc", "rack", "zone", "host"]

## Global variables
[all:vars]
deploy_dir = /data/tidb_cluster/tidb

## Connection
# ssh via normal user
ansible_user = tidb

cluster_name = test

tidb_version = v3.0.5

# process supervision, [systemd, supervise]
process_supervision = systemd

timezone = Asia/Shanghai

enable_firewalld = False
# check NTP service
enable_ntpd = False
set_hostname = False

## binlog trigger
enable_binlog = True

# kafka cluster address for monitoring, example:

# zookeeper address of kafka cluster for monitoring, example:
# zookeeper_addrs = "192.168.0.11:2181,192.168.0.12:2181,192.168.0.13:2181"

# enable TLS authentication in the TiDB cluster
enable_tls = False

# KV mode
deploy_without_tidb = False

# wait for region replication complete before start tidb-server.
wait_replication = True

# Optional: Set if you already have a alertmanager server.
# Format: alertmanager_host:alertmanager_port
alertmanager_target = ""

grafana_admin_user = ""
grafana_admin_password = ""

#### Collect diagnosis
collect_log_recent_hours = 2
enable_bandwidth_limit = True
# default: 10Mb/s, unit: Kbit/s
collect_bandwidth_limit = 10000
```

## 2.2 inventory 配置详解

inventory.ini 作为部署 TiDB 集群的重要配置文件，在配置中建议对所有的组件进行别名设置，以方便使用 ansible-playbook 的 -l 参数操作单一组件的单一实例。

### 1.TiDB Servers

```
[tidb_servers]
TiDB-10 ansible_host=10.63.10.10 deploy_dir=/data/tidb_cluster/tidb
TiDB-11 ansible_host=10.63.10.11 deploy_dir=/data/tidb_cluster/tidb
```

2.TiKV Servers 设置基于 tikv 真实物理部署位置的 label 信息，方便 PD 进行全局管理和调度。

```
[tikv_servers]
TiKV-30 ansible_host=10.63.10.30 deploy_dir=/data/tidb_cluster/tikv1 tikv_port=20171 labels="dc=1,rack=1,zone=1,host=30"
TiKV-31 ansible_host=10.63.10.31 deploy_dir=/data/tidb_cluster/tikv1 tikv_port=20171 labels="dc=1,rack=2,zone=2,host=31"
```

3.PD 设置为 PD 设置 TiKV 部署位置等级信息。

```
[pd_servers:vars]
location_labels = ["dc", "rack", "zone", "host"]
```

## 2.3 Labels 设计

在两地三中心部署方式下，对于 Labels 的设计也需要充分考虑到系统的可用性和容灾能力，建议根据部署的物理结构来定义 DC、AZ、RACK、HOST 四个等级。

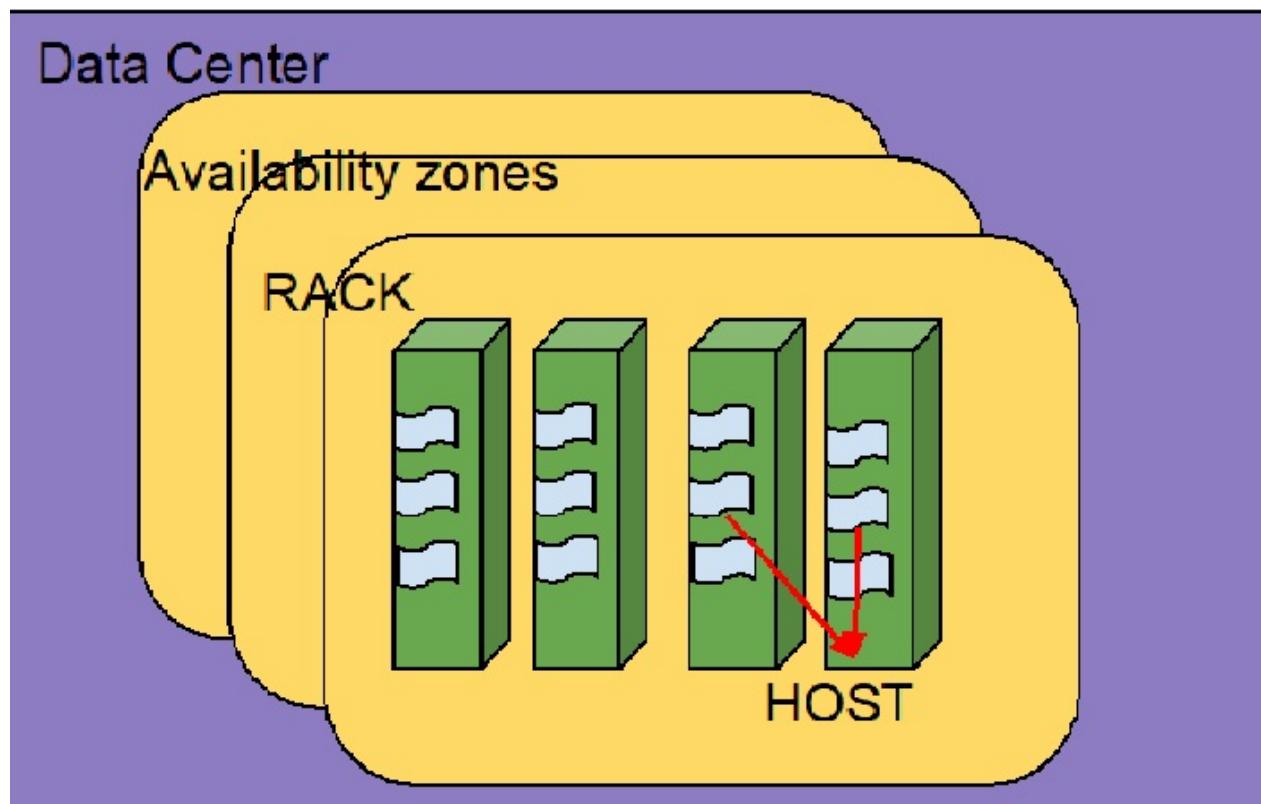


图3 label逻辑定义图

## 2.4 参数配置

在两地三中心的架构部署中，从性能优化的角度，建议对集群中相关组件参数进行调整。

- tikv.yml 中相关参数优化

文件路径：/tidb-ansible/conf/tikv.yml 需要在集群安装前进行设置。

```
+ block-cache-size
在 TiKV 单机多实例环境下，需要按照以下公式调整该值。
```

capacity = MEM\_TOTAL \* 0.5 / TiKV 实例数量 示例如下：

```
...
Storage:
  block-cache:
    capacity: "1G"
...
```

- 启用grpc消息压缩 由于涉及到集群中的数据在网络中传输，需要开启 gRPC 消息压缩，降低网络流量。

```
server:
  grpc-compression-type: gzip
```

- pd.yml 中相关参数优化 文件路径：/tidb-ansible/conf/pd.yml 需要在集群安装前进行设置。

调整 PD balance 缓冲区大小，提高 PD 容忍度，因为 PD 会根据节点情况计算出各个对象的 score 作为调度的一句，当两个 store 的 leader 或 Region 的得分差距小于指定倍数的 Region size 时，PD 会认为此时 balance 达到均衡状态。

```
...
schedule:
  tolerant-size-ratio: 20.0
...
```

- DC3 TiKV 网络优化 文件路径：/tikv/conf/tikv.toml

修改此参数，拉长了异地副本参与选举的时间，尽量避免异地 TiKV 中的副本参与 raft 选举。建议在集群部署完毕后，为 DC3 的 TiKV 增加额外配置后重启 DC3 的TiKV。

```
...
raftstore:
  raft-min-election-timeout-ticks= 1000
  raft-max-election-timeout-ticks= 1020
...
```

- 调度设置 在集群启动后，通过 PD control 工具进行调度策略修改。

- 修改 TiKV raft 副本数 按照安装时规划好的副本数进行设置，在本例中为 5 副本。

```
config set max-replicas 5
```

- 禁止向异地机房调度 raft leader 当 raft leader 在异地数据中心时，会造成不必要的本地数据中心与异地数据中心间的网络消耗，同时由于网络带宽和延迟的影响，也会对 TiDB 的集群性能产生影响。需要禁用 raft leader 的调度。

```
config set label-property reject-leader dc dc3
```

- 设置 PD 的优先级 为了避免出现异地数据中心的 PD 成为 leader，可以将本地数据中心的 PD 优先级调高(数字越大，优先级越高)，将异地的 PD 优先级调低。

```
member leader_priority PD-10 5
member leader_priority PD-11 5
member leader_priority PD-12 5
member leader_priority PD-13 5
member leader_priority PD-14 1
```



## 4.4 AWS 跨 AZ 部署 TiDB

本节介绍了在 AWS 上的单一区域 (region)、跨三个可用区 (Availability Zones，以下简称 AZ) 部署 TiDB 过程中，遇到的问题和相应的变通方案。

### 4.4.1 比本地机房部署更高的故障率

根据我们的经验，每月平均至少有一个部署 TiDB 分布式数据库的 EC2 实例发生故障。在 AWS 上部署 TiDB 的这半年里，我们还经历过一次 AZ 不稳定的问题。下面提供了一些建议，帮助你时刻准备好应对各种类型的故障。

#### 1. 副本数

对于非关键性业务，考虑部署奇数个 PD 实例，并且建议接近选用的 AWS 区域的可用 AZ 数。TiKV region 副本数也是如此。例如，你选用的 AWS 区域提供了三个 AZ，那么建议你：

- 部署三个 PD 实例：每个 AZ 上部署一个 PD 实例
- 设置 max-replicas 为 3

这样即使单个 AZ 出现故障，也能最大程度的确保可用性。建议使用奇数个数的 PD 和 region 副本数以避免脑裂问题。如果你选用的 AWS 区域有三个以上的 AZ（例如 Northern Virginia 有六个 AZ），则可以考虑部署三副本或五副本。对于任何关键性业务，考虑使用五副本或更多副本。在我们的案例中，虽然我们选用的 AWS 区域只提供了三个 AZ，但我们仍然：

- 部署了五个 PD 实例：分布在三个 AZ 上 (2:2:1)
- 设置 max-replicas 为 5

与 max-replicas=3 相比，该设置不会在 AZ 故障期间提高可用性，但会减少由于 EC2 实例的偶然故障而导致集群服务中断的可能性。

#### 2. 位置标签 (location labels)

配置完正确的副本数后，记得给 TiKV 设置适当的位置标签。否则，你的集群将无法承受预期的故障。设置标签的基本原则是：

- 每台主机都有唯一的标签
- 每个机架（如果有）都有唯一的标签
- 放置每个副本的每个逻辑组 (logical group) 都有唯一的标签
- 每个 AZ 都有唯一的标签

以实际情况为例（10 个 TiKV 实例分布在三个 AZ 上，实例数量：1a:1b:1c=2:4:4；max-replicas=5），则恰当的标签为：

- PD : location-labels = ["az", "zone", "host"]
- TiKV :

az	zone	host
az=1a	zone=1	host=1a_1
az=1a	zone=1	host=1a_2
az=1b	zone=2	host=1b_1
az=1b	zone=2	host=1b_2
az=1b	zone=3	host=1b_3
az=1b	zone=3	host=1b_4
az=1c	zone=4	host=1c_1
az=1c	zone=4	host=1c_2
az=1c	zone=5	host=1c_3
az=1c	zone=5	host=1c_4

需要注意的是 zone 是一个逻辑的标签，PD 在调度时首先会尝试将 Region 的 Peer 放置在不同的 AZ，此时无法满足(3 个 AZ ,5 个副本)，下一步保证放置在不同的 az.zone 中(此时 5 个 zone ,5 个副本，满足要求)。这样确保了在正常情况下，使用 az.zone 将每个副本（即 Region peer）放置在每个逻辑组（即 zone）中。如果可用 az.zone 的数量少于 5，比如 zone 或 AZ 级别的故障出现时，将使用 az.zone.host 标签来均匀调度五个 Region peers，以保持最大可用性。

### 3. 置放群组 (placement groups)

如果你想减少相关的硬件故障，可以考虑使用 [Placement Groups](#)。在不同的场景下，Partition Placement Groups 和 Spread Placement Groups 各有用处。记住要额外配置位置标签（例如机架的标签）以充分利用 placement groups。

### 4. 备份

理想情况下，为了防止 AWS 上的单一区域不可用，一般首选某种形式的多区域部署方式进行部署。但如果你的部署被限制在单个区域（比如数据必须与服务位于同一国家），建议将数据备份到另一个位置，以保证业务可用性。

在同一位置进行每日全量和增量备份，能帮助你在该区域恢复可用时恢复业务。请注意，与多区域部署相比，这种部署方式的恢复时间目标 (Recovery Time Objective, RTO) 将更高。

## 4.4.2 更高延迟的 AZ

AWS 区域中的 AZ 可能带有不对称的延迟。例如，与其他两个 AZ（延迟均小于 1 毫秒）相比，同一区域的另一个 AZ 具有更高的延迟（2~3 毫秒）。

以下示例中，假设我们有三个 AZ (1a, 1b, 1c)，并且其中一个 AZ (1a) 的延迟比其他两个 AZ 的延迟高。

### Reject region leaders

在正确设置了位置标签后，你可以配置标签的属性以阻止在特定 AZ 中选举 region leader。

```
$ pd-ctl --pd="http://pd-url:2379"

» config set label-property reject-leader az 1a

» config show label-property

{
  "reject-leader": [
    {
      "key": "az",
      "value": "1a"
    }
  ]
}
```

## 1. 设置 PD leader 选举的优先级

对于延迟较高的 AZ，我们可以降低其中 PD 实例选举 leader 的优先级，这样能让其他延迟低的 AZ 优先选举 PD leader。在以下示例中，对于 1a 中的 PD，我们将 leader\_priority 设置为 3；对于 1b 和 1c 中的 PD，我们将 leader\_priority 设置为 5。leader\_priority 的值越大，优先级越高。

```
» member leader_priority pd_1a_1 3
Success!

» member leader_priority pd_1c_1 5
Success!

» member leader_priority pd_1b_1 5
Success!

» member
{
...
"members": [
  {
    "name": "pd_1a_1",
...
    "leader_priority": 3
  },
  {
    "name": "pd_1b_1",
...
    "leader_priority": 5
  },
...
  {
    "name": "pd_1c_1",
...
    "leader_priority": 5
  }
],
"leader": {
...
},
"etcd_leader": {
...
}
}
```

## 2. 重新放置 TiDB server 服务器

考虑将 TiDB server 服务器从较高延迟的 AZ 移到其他 AZ。例如，如果每个 AZ 中有两台 TiDB 服务器（1a、1b 和 1c 中各放两台），则可以将 1a 中的两台 TiDB 服务器分别移至 1b 和 1c（1b 和 1c 中各放三台）。请注意，该操作牺牲了性能的高可用性——如果放置 TiDB 服务器的其中一个 AZ 发生故障，对工作负载的影响会更大。

## 4.4.3 性能波动

### 1. 报警

如果你确定触发的报警是由于 AWS 硬件引起的错误报警，建议调整报警项。对于我们在 AWS 上的部署，与磁盘延迟有关的报警项必须通过两种方式进行调整：

- 在报警规则里将“for”部分设置的等待时间从 1m 增加到 5m，从而可以忽略短期的，间歇性的性能下降
- 增加阈值，以便解决与本地部署相比较低的硬件性能

### 2. 硬件选择

在相同实例类型的 EC2 实例中，我们观察到性能差异高达 20%。因此，建议考虑以下做法：

- 在最初部署 TiDB 集群时，进行基准测试并选择性能更好的实例。
- 定期替换 EC2 实例，从而将性能不佳的实例淘汰，替换为性能更好的实例。这也是一个很好的演习机会——练习如何轻松地从 TiDB 集群中删除节点或向其中添加节点。

## 第 5 章 数据迁移方案

本章将介绍常用的异构或者同构数据库之间的数据全量和增量同步方案，重点介绍如何将数据从IBM Db2、Mongodb、MySQL、Oracle、阿里云 DRDS、SQL Server同步至TiDB分布式数据库，同时介绍如何使用数据同步方案进行TiDB的高可用数据容灾。

- [IBM Db2 到 TiDB \(CDC\)](#)
- [MongoDB 迁移到 TiDB](#)
- [DM 同步 MySQL 到 TiDB 的实践](#)
- [Oracle 到 TiDB \(OGG\)](#)
- [DM 同步分库分表 MySQL 到 TiDB 的实践](#)
- [DM 同步单机 MySQL 到 TiDB 的实践](#)
- [SQLServer 到 TiDB \(阿里DataX\)](#)
- [SQL Server 迁移到 TiDB \(阿里yutong\)](#)
- [TiDB 到 TiDB \(阿里DataX\)](#)

## 5.1 DM 同步 MySQL 到 TiDB 的实践

DM (Data Migration) 是一体化的数据同步任务管理平台，支持从 MySQL 或 MariaDB 到 TiDB 的全量数据迁移和增量数据同步。

使用 DM 工具有利于简化错误处理流程，降低运维成本。

[官网文档](#) [开源仓库](#)

## 5.1.1 DM 同步单机 MySQL 到 TiDB 的实践

本小节的实践目标是将 单机 MySQL 实例数据同步到 TiDB 集群。

主要包括以下8个小节的内容，所有提到的参数配置及说明基于 DM 1.0.2版本

- [DM 支持的场景](#)
- [DM 使用要求](#)
- [DM 同步原理](#)
- [同步前置条件](#)
- [制定同步规则的 task 文件配置](#)
- [同步状态检查](#)
- [同步过程中可能遇到的问题及如何解决](#)
- [tips](#)

### 1. DM 支持的场景

- 全量&增量同步数据
- 不同维度的过滤规则设定：库表级别，SQL级别
- 上游分库分表合并及变更聚合，具体示例参见下一章节
- 同步延迟监控

### 2. DM 使用要求

- 数据库版本
  - 5.5 < MySQL 版本 < 8.0
  - MariaDB 版本 >= 10.1.2
- 仅支持 TiDB parser 支持的 DDL 语法
- 上下游 sql\_model 检查
- 上游开启 binlog，且 binlog\_format=ROW
- 关于分库分表合并场景的限制，参见下一章节

### 3. DM 同步原理

DM 以一个集群为单位运行，包括以下5个组成部分：

- DM-master，负责管理整个DM集群，以及调度同步任务
- DM-worker，执行具体的同步任务
  - 一个 DM-worker 注册为一个上游 MySQL 或 MariaDB 实例的slave
  - 一个 DM 集群中可以包含多个 DM-worker，也就是说可以同步上游多个 MySQL 或 MariaDB 实例
  - DM-worker 的工作方式是
    - Dumper 从上游 导出全量数据到本地磁盘
    - Loader 读取 dumper 处理单元的数据文件，然后加载到下游 TiDB
    - Syncer 读取 relay log 处理单元的 binlog event，将这些 event 转化为 SQL 语句，再将这些 SQL 语句应用到下游 TiDB
- dmctl，控制DM集群的命令行工具，连接 DM-Master 管理整个集群
- task文件，配置 DM-worker 要执行的同步规则，所有 DM-worker 生效
- Prometheus，监控同步状态

## 4. 同步前置条件

- 确认同步上下游部署结构

组件	主机	端口号
DM-Master	172.16.10.71	8261
DM-worker	172.16.10.72	8262
上游MariaDB	172.16.10.81	3306
下游TiDB的计算节点	172.16.10.83	4000

- 确认同步目标

- 将上游单机 MySQL 实例中的 book 库 session 表全量同步到下游 TiDB 中
- 过滤系统库 : mysql,information\_schema,percona,performance\_schema
- 过滤删除操作 : drop , truncate
- 不同步 book 库的 draft 表

- DM 集群的部署启动文档 , 部署过程与 TiDB 集群的部署启动高度相似 , inventory.ini 需要注意以下几点

- [dm\_worker\_servers] 部分
  - 1.server\_id 在整个同步结构里唯一 , 范围包括上游 MySQL , 下游 TiDB
  - 2.source\_id 在 task 任务配置里标示上游实例
  - 3.mysql\_password 需要通过 dmctl 工具加密 , 这个密码在 task 文件里也需要用到
    - eg: dm-ansible/resources/bin/dmctl -encrypt 密码串
  - 3.enable\_gtid 是否使用 GTID 同步 , 前提是上游 MySQL 实例开启了 GTID , 本案例里没有用到。
  - 4.这个部分完整的例子

```
[dm_worker_servers]
dm-worker1 ansible_host=1.1.1.1 source_id="mariadb-01" server_id=101 mysql_host=172.16.10.81 mysql_user=tidb dm mysql_password="encrytpwd" mysql_port=3306
```

- 同步用户需要上游 MySQL 实例访问授权

- 需要权限有
  - REPLICATION SLAVE
  - REPLICATION CLIENT
  - RELOAD
  - SELECT

- 下游 TiDB 集群部署 及读写访问授权

- 同步需求分类 , 决定 task 文件的配置项复杂程度

- 同步模式 : 全量 , 增量 , 仅备份
  - 这个例子里使用全量
  - 全量备份上游数据库 , 将数据全量导入到下游数据库
  - 全量数据备份时导出的位置信息 (binlog position) 开始通过 binlog 增量同步数据到下游数据库。
- 同步粒度 : 整库 , 指定表 , 指定 Binlog
  - 这个例子里选 book 整库
  - 过滤上游系统库
  - 过滤上游删库删表操作

## 5. 制定同步规则的 task 文件配置

task 文件决定 DM-Worker 按照怎样的规格同步数据 , 主要有以下 9 个区域 :

- 任务全局定义

```
name: "taskname"      # 全局唯一的 task 名称
task-mode: all        # 同步模式, 这里选全量
meta-schema: "dm_meta" # checkpoint 信息存储在下游的数据库名
remove-meta: false    # 是否在任务同步开始前移除该任务名对应的 checkpoint 信息, 删除会重新开始同步, 不删除会从上次停止的位置开始同步
```

- target-database , 下游 TiDB 集群地址用户密码，密码与 DM-Worker 配置里的密码相同
- mysql-instances , 上游 MySQL 实例 source-id 及同步规则模块名称

```
source-id: "mariadb-01"      # dm-worker 定义的 source-id 对应
route-rules: ["book-route-rules-schema", "book-route-rules"] # 需要同步的对应的库表配置名称
filter-rules: ["book-filter-1"] # 需要过滤的 binlog event 配置名称
black-white-list: "bookblack" # 需要过滤的库表配置名称

mydumper-config-name: "global" # mydumper 配置名称
mydumper-thread: 4

loader-config-name: "global"   # loader 配置名称
loader-thread: 8

syncer-config-name: "global"   # syncer 配置名称
syncer-thread: 9
```

- routes , 需要同步的库表信息

```
book-route-rules-schema: # mysql-instances 部分定义的配置名称
  schema-pattern: "book"
  target-schema: "book"
book-route-rules:
  schema-pattern: "book"
  table-pattern: "session"
  target-schema: "book"
  target-table: "session"
```

- filters , 需要过滤的Binlog Event

```
book-filter-1:
  schema-pattern: "book"
  table-pattern: "session"
  events: ["truncate table", "drop table"]
  action: Ignore
```

- black-white-list , 需要过滤的库表

```
bookblack:
  do-dbs: ["~^book.*"]
  ignore-dbs: ["mysql", "performance_schema", "percona", "information_schema"]
  ignore-tables:
    - db-name: "book.*"
      tbl-name: "draft"
```

- mydumpers , 备份控制

```
global:
  mydumper-path: "./bin/mydumper"
  threads: 4
  chunk-filename-size: 64
  skip-tz-utc: true
```

- loaders , 备份导入控制

```
global:
  pool-size: 8
  dir: "./dumped_data"
```

- syncers，同步控制

```
global:
  worker-count: 8
  batch: 100
```

## 6. 同步状态检查

task 配置完成，通过 dmctl 工具检查执行同步

- 通过 dmctl 连接 DM-master 管理 task 任务

```
./dmctl -master-addr 172.16.10.71:8261
```

- 首先检查任务配置是否符合规范

```
» check-task  task-path
{
  "result": true,
  "msg": "check pass!!!"
}
```

- 运行任务

```
» start-task  task-path
{
  "result": true,
  "msg": "",
  "workers": [
    {
      "result": true,
      "worker": "172.16.10.72:8262",
      "msg": ""
    }
  ]
}
```

- 查看详细任务状态，正常状态 result 为 true，worker 内的 binlog 位置一致，同步过程中也会展示同步百分比

```
query-status  taskname
```

- 如果发现启动任务异常，查看详细的错误信息

```
query-error  taskname
```

- 停止任务

```
start-task  taskname
```

- 其他详细的任务管理内容

## 7. 同步过程中可能遇到的问题及如何解决

- 检查 task 失败，根据提示检查对应配置行是否有语法错误
- 启动失败，根据提示信息解决后，resume-task
- 同步过程中，因为 SQL 不兼容，或者异常问题导致复制中断，查看详细错误信息修复
- [Data Migration 常见错误修复](#)

举个例子

- task 状态报错信息

```
"msg": "[code=44003;class=schema-tracker:scope=downstream;level=high] current pos (mysql-bin.000010, 814332497): failed to create table for `db_1`.`tb_1` in schema tracker: [types:1067]Invalid default value for 'expire_time'
```

- 查看上游 db\_1.tb\_1 报错字段定义

```
expire_time  datetime NOT NULL DEFAULT '0000-00-00 00:00:00'
```

- 查看下游列定义一致
- 查看下游 sql\_mode，严格模式下，datetime 类型默认值不能为 '0000-00-00 00:00:00'

```
TiDB> show variables like 'sql_mode';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| sql_mode      | STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION |
+-----+-----+
```

- 修改下游 sql\_mode，重启 task 同步继续

## 8. Tips

- task 配置过滤规则与 MySQL 基本一致
- task 是否需要保留 meta 信息，决定任务重新启动后的 binlog 起点位置
- 配置[监控告警](#)，如果发现同步异常可以及时通知
- task 任务的配置也可以使用 [DM Portal](#) 工具自动生成
- DM-worker 需要配置[日志清理](#)，清理同步完成的 Rely log 以及索引信息

## 5.1.2 DM 同步分库分表 MySQL 到 TiDB 的实践

### 5.1.2.1 DM 分库分表安装部署实战

本实战模拟企业生产环境阿里云 DRDS 中间件对业务表进行分库分表后，这边使用 DM 工具将线上分库分表数据同步至 TiDB 中：

1. 解决跨业务跨库的数据查询分析
2. 结合 DBA 管理平台提供数据排错查询减少因人为慢查询引起的线上故障
3. 线上仅保留半年数据，数据归档至 TiDB 保留。

有部分的分库分表使用了自增长主键 ID，使用 DM 的自增长主键重算机制，解决了上游分库分表合并到下游单表时的主键冲突问题。要说明的是这个功能还是有所限制，上游在设计自增主键的时候最好还是使用全局自增服务组件来做比较好。

此外 dm 在 loader 恢复时支持断点操作，支持幂等 binlog 重做，不用担心恢复中意外而前功尽弃。

#### 1. 环境说明

实验环境宿主机的用户名、密码与数据库的用户名、密码一致。

主机 IP	操作系统	应用部署	说明	帐号密码
192.168.128.131	centos7.3 x86_64	MySQL5.7	3306 端口	root/password
192.168.128.131	centos7.3 x86_64	MySQL5.7	3307 端口	root/password
192.168.128.131	centos7.3 x86_64	MySQL5.7	3308 端口	root/password
192.168.128.132	centos7.3 x86_64	dm-master/dmctl	中控机	root/password
192.168.128.133	centos7.3 x86_64	dm-worker	dm-worker	root/password
192.168.206.28	centos7.3 x86_64	TiDB 库	4000 端口	root/password

#### 2. 准备工作

第一步：使用 **root** 账号登录中控机 **192.168.128.132** 上并安装依赖包

```
[tidb@dm master ~]# yum -y install epel-release git curl sshpass
[tidb@dm master ~]# yum -y install python-pip
```

第二步：在中控机上创建 **tidb** 用户并生成 SSH 密钥

1、创建 tidb 用户

```
[tidb@dm master ~]# useradd -m -d /home/tidb tidb
```

2、为 tidb 用户设置密码

```
[tidb@dm master ~]# echo "password" | passwd --stdin tidb
```

3、为 tidb 用户设置免密使用 sudo

```
[tidb@dm master ~]# echo "tidb ALL=(ALL) NOPASSWD: ALL" >>/etc/sudoers
```

4、切换至 tidb 用户 home 目录并生成 SSH 密钥

```
[tidb@dmmaster ~]# su - tidb
[tidb@dmmaster ~]$ ssh-keygen -t rsa
一路按回车生成密钥
```

### 第三步：使用 **tidb** 用户在中控机下载 DM-Ansible

```
[tidb@dmmaster ~]$ wget https://download.pingcap.org/dm-ansible-v1.0.2.tar.gz
```

### 第四步：安装 DM-Ansible 及其依赖至中控机

```
[tidb@dmmaster ~]$ tar -xf dm-ansible-v1.0.2.tar.gz
[tidb@dmmaster ~]$ mv dm-ansible-v1.0.2.tar.gz dm-ansible
[tidb@dmmaster ~]$ cd /home/tidb/dm-ansible
[tidb@dmmaster dm-ansible]$ sudo pip install --upgrade pip
[tidb@dmmaster dm-ansible]$ sudo pip install -r ./requirements.txt
```

### 第五步：在中控机上配置 ssh 互信和 sudo 规则

```
[tidb@dmmaster dm-ansible]$ cat hosts.ini
[servers]
192.168.128.132
192.168.128.133
[all:vars]
username = tidb
ansible_ssh_port = 22
ntp_server = ntp.aliyun.com
```

dm-worker 主机建立 tidb 用户并完成互信，此处输入远程机器的 root 密码 password

```
[tidb@dmmaster dm-ansible]$ ansible-playbook -i hosts.ini create_users.yml -u root -k
SSH password:
PLAY [all]
*****
TASK [create user]
*****
changed: [192.168.128.133]
TASK [set authorized key]
*****
changed: [192.168.128.133]
TASK [update sudoers file] *****
changed: [192.168.128.133]
PLAY RECAP
*****
192.168.128.133 : ok=3    changed=3    unreachable=0    failed=0
```

### 第六步：下载 DM 及监控组件安装包至中控机

```
[tidb@dmmaster dm-ansible]$ ansible-playbook local_prepare.yml
PLAY [do local preparation] *****
TASK [download : Stop if ansible version is too low, make sure that the Ansible version is Ansible 2.5.0 or later, otherwise a compatibility issue occurs.]
*****
ok: [localhost] => {
    "changed": false,
    "msg": "All assertions passed"
}
此处打印日志省略
localhost : ok=13    changed=5    unreachable=0    failed=0
```

### 第七步：上游 MySQL 数据库建立 TiDB 数据迁移专用帐户

```
root@localhost >grant Reload,Replication slave, Replication client,select on *.* to tidb@'%' IDENTIFIED by 'tidb@2020';
';
```

#### 第八步：使用 dmctl 加密上下游数据库登录密码

```
[tidb@dmworker bin]$ dmctl -encrypt tidb@2020
BXTTVvKeWhXgAefaFRNoN0BS4XjZ85uZByE=
```

### 3. 部署 dm-worker

#### 第一步：编写 inventory.ini 文件

此处我们主要定义 dm-master 和 dm-worker，本章采取单台部署多台 dm-worker。

```
[tidb@dmworker dm-ansible]$ cat inventory.ini
## DM modules
[dm_master_servers]
dm_master ansible_host=192.168.128.132
[dm_worker_servers]
dm_worker3306 ansible_host=192.168.128.133 deploy_dir=/data/mysql3306 dm_worker_port=13306 source_id="mysql3306" server_id=13306 mysql_host=192.168.128.131 mysql_user=tidb mysql_password=BXTTVvKeWhXgAefaFRNoN0BS4XjZ85uZByE mysql_port=3306
dm_worker3307 ansible_host=192.168.128.133 deploy_dir=/data/mysql3307 dm_worker_port=13307 source_id="mysql3307" server_id=13307 mysql_host=192.168.128.131 mysql_user=tidb mysql_password=BXTTVvKeWhXgAefaFRNoN0BS4XjZ85uZByE mysql_port=3307
dm_worker3308 ansible_host=192.168.128.133 deploy_dir=/data/mysql3308 dm_worker_port=13308 source_id="mysql3308" server_id=13308 mysql_host=192.168.128.131 mysql_user=tidb mysql_password=BXTTVvKeWhXgAefaFRNoN0BS4XjZ85uZByE mysql_port=3308
[dm_portal_servers]
dm_portal ansible_host=192.168.128.132
## Monitoring modules
[prometheus_servers]
prometheus ansible_host=192.168.128.132
[grafana_servers]
grafana ansible_host=192.168.128.132
[alertmanager_servers]
alertmanager ansible_host=192.168.128.132
## Global variables
[all:vars]
cluster_name = dm-cluster
ansible_user = tidb
ansible_port = 5622
dm_version = v1.0.2
deploy_dir = /home/tidb/deploy
grafana_admin_user = "admin"
grafana_admin_password = "admin"
```

#### inventory.ini 文件参数说明

[dm_master_servers]	dm-master 选项，用于定义哪台主机是中控 dm-master
[dm_worker_servers]	dm-worker 选项，用于定义 dm-worker 服务
----dm_worker3306	dm服务全局唯一标签，配合ansible-playbook -l 参数使用
----ansible_host	指定 dm-worker 部署在哪台主机
----dm_worker_port	指定 dm-worker 启动服务端口号
----deploy_dir	指定 dm-worker 部署安装目录
----source_id	指定 dm-worker 的 source-id
----mysql_host	上游MySQL主机地址
----mysql_user	上游MySQL登录用户
----mysql_port	上游MySQL服务端口
----mysql_password	上游MySQL登录密码(必须dmctl加密后的值,参考2.2章第八步)

#### 第二步：执行安装并启动 dm-worker

安装 dm-worker:

```
[tidb@dmmaster dm-ansible]$ ansible-playbook deploy.yml
PLAY RECAP ****
alertmanager : ok=13    changed=7      unreachable=0      failed=0
dm_master    : ok=13    changed=8      unreachable=0      failed=0
dm_portal    : ok=12    changed=5      unreachable=0      failed=0
dm_worker3306: ok=14    changed=2      unreachable=0      failed=0
dm_worker3307: ok=14    changed=2      unreachable=0      failed=0
dm_worker3308: ok=14    changed=2      unreachable=0      failed=0
grafana      : ok=17    changed=10     unreachable=0      failed=0
localhost    : ok=4     changed=3      unreachable=0      failed=0
prometheus   : ok=15    changed=13     unreachable=0      failed=0
#出现以上信息表示部署成功
```

启动 dm-worker:

```
[tidb@dmmaster dm-ansible]$ ansible-playbook start.yml
PLAY RECAP ****
alertmanager : ok=10    changed=1      unreachable=0      failed=0
dm_master    : ok=10    changed=1      unreachable=0      failed=0
dm_portal    : ok=9     changed=1      unreachable=0      failed=0
dm_worker3306: ok=11    changed=1      unreachable=0      failed=0
dm_worker3307: ok=11    changed=1      unreachable=0      failed=0
dm_worker3308: ok=11    changed=1      unreachable=0      failed=0
grafana      : ok=13    changed=1      unreachable=0      failed=0
localhost    : ok=4     changed=0      unreachable=0      failed=0
prometheus   : ok=13    changed=4      unreachable=0      failed=0
#出现以上信息表示 dm 启动成功, 此时已经开始同步上游 binlog 至 dm 机器中。
```

## 4. 配置 & 启动 task

上游数据库结构合并至下游 **TiDB** 说明

上游分库	上游分表	下游合并库名	下游合并表名
shard_db01	shard_tb01	merge_db	merge_tb
shard_db02	shard_tb02		
shard_db03	shard_tb03		
shard_db04	shard_tb04		
shard_db05	shard_tb05		
shard_db06	shard_tb06		

上游数据库准备

```

CREATE TABLE shard_tb01~06 (
    id bigint(20) NOT NULL AUTO_INCREMENT COMMENT'主键ID',
    uid bigint(20) NOT NULL COMMENT '用户ID',
    uname varchar(10) NOT NULL DEFAULT '' COMMENT '用户名',
    gender tinyint(1) NOT NULL DEFAULT '0' COMMENT '性别 0-男、1-女',
    shard varchar(50) NOT NULL DEFAULT '' COMMENT '分片信息',
    mobile varchar(15) NOT NULL DEFAULT '' COMMENT '联系电话',
    PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='分库分表';
INSERT INTO shard_db01.shard_tb01
(uid,uname,gender,shard,mobile) VALUES
('10001','tb01001','0','shard_db01_tb01','136*****17'), ('10002','tb01002','1','shard_db01_tb01','136*****17');
INSERT INTO shard_db02.shard_tb02
(uid,uname,gender,shard,mobile) VALUES ('20001','tb02001','1','shard_db02_tb02','136*****33'), ('20002','tb02002','0',
,'shard_db02_tb02','139*****63');
INSERT INTO shard_db03.shard_tb03
(uid,uname,gender,shard,mobile) VALUES ('30001','tb03001','0','shard_db03_tb03','135*****73'),
('30002','tb03002','0','shard_db03_tb03','139*****46');

INSERT INTO shard_db04.shard_tb04
(uid,uname,gender,shard,mobile) VALUES ('40001','tb04001','0','shard_db04_tb04','137*****91'), ('40002','tb04002','1',
,'shard_db04_tb04','138*****91');
INSERT INTO shard_db05.shard_tb05
(uid,uname,gender,shard,mobile) VALUES ('50001','tb05001','1','shard_db05_tb05','158*****96'), ('50002','tb05002','0',
,'shard_db05_tb05','188*****92');
INSERT INTO shard_db06.shard_tb06
(uid,uname,gender,shard,mobile) VALUES ('60001','tb06001','1','shard_db06_tb06','178*****98'), ('60002','tb06002','1',
,'shard_db06_tb06','175*****31');

```

### 合库合表 task 的 yaml 文件

```

[tidb@tidb-dm-4-0-95 task]$ cat shardmysql_to_tidb.yaml
name: "shard_to_tidb" #task 名称, 必须全局唯一
is-sharding: true      #上游是不是进行了分库分表库
task-mode: "all"       #迁移同步方式 full-全量、incremental-增量、all-全量+增加
meta-schema: "tidb_meta" #定义下游保留迁移点位信息库名称
remove-meta: false
target-database:
  host: "192.168.206.28" #下游 TiDB 主机 IP
  port: 4000                #TiDB 访问端口
  user: "root"              #TiDB 登录用户
  password: "vLnqQt44rNFHSxA" #使用 dmctl 加密的登录密码
mysql-instances:
  -
    source-id: "mysql3306"   #必须与 inventory.ini 中对应的 source-id 一致
    route-rules: ["rt000","rt001"]      #库表合并规则
    filter-rules: ["ymdd-filter-rule"]  #过滤规则
    mydumper-config-name: "global"
    loader-config-name: "global"
    syncer-config-name: "global"
    black-white-list: "br01"          #白名单列表
    column-mapping-rules: ["cm001"]    #自增主键重计算规则
  -
    source-id: "mysql3307"
    route-rules: ["rt000","rt001"]
    filter-rules: ["ymdd-filter-rule"]
    mydumper-config-name: "global"
    loader-config-name: "global"
    syncer-config-name: "global"
    black-white-list: "br01"
    column-mapping-rules: ["cm002"]
  -
    source-id: "mysql3308"
    route-rules: ["rt000","rt001"]
    filter-rules: ["ymdd-filter-rule"]
    mydumper-config-name: "global"
    loader-config-name: "global"
    syncer-config-name: "global"
    black-white-list: "br01"
    column-mapping-rules: ["cm003"]

```

```

filters:
  ymdd-filter-rule:
    schema-pattern: "shard_db *"
    #以下 2 行定义忽略的 binlog 事件
    events: ["truncate table","delete","drop table","drop database"]
    action: Ignore

routes:
  rt000:
    #将上游所有 shard_db* 匹配的库合并至 merge_db 库
    schema-pattern: "shard_db*"
    target-schema: "merge_db"
  rt001:
    #将上游所有 shard_tb* 匹配的分表合并至 merge_db 库的 merge_tb 表中
    schema-pattern: "shard_db*"
    table-pattern: "shard_tb??"
    target-schema: "merge_db"
    target-table: "merge_tb"

#由于上游数据库使用了自增主键，此处我们需要定义下游主键重算处理，该功能需要谨慎使用
#特别注意上游的自增主键不能有任何业务关系

column-mappings:
  cm001:
    schema-pattern: "shard_db*"
    table-pattern: "shard_tb??"
    expression: "partition id"
    source-column: "id"
    target-column: "id"
    arguments: ["1","shard_db","shard_tb"]
  cm002:
    schema-pattern: "shard_db*"
    table-pattern: "shard_tb??"
    expression: "partition id"
    source-column: "id"
    target-column: "id"
    arguments: ["2","shard_db","shard_tb"]
  cm003:
    schema-pattern: "shard_db*"
    table-pattern: "shard_tb??"
    expression: "partition id"
    source-column: "id"
    target-column: "id"
    arguments: ["3","shard_db","shard_tb"]

#黑名单定义
black-white-list:
  br01:
    do-dbs: ["~shard_db*"] #需要同步的库
    #需要忽略同步的库
    ignore-dbs: ["mysql","performance_schema","information_schema"]
    #需要忽略同步的哪个库的哪张表
    ignore-tables:
      - db-name: "~shard_db*"
        tbl-name: "~txc_undo_log*"
#以下默认即可

mydumpers:
  global:
    threads:
      chunk-filename-size: 64
      skip-tz-utc: true
      extra-args: " --no-locks "
loaders:
  global:
    pool-size: 64
    dir: "./dumped_data"
syncers:
  global:
    worker-count: 6
    batch: 1000

```

## 启动迁移同步任务

```
[tidb@dmworker dumped_data.shard_to_tidb]# dmctl -master-addr 192.168.128.132:8261
» start-task shard_to_tidb.yaml
» query-status
{
    "result": true,
    "msg": "",
    "tasks": [
        {
            "taskName": "shard_to_tidb",
            "taskStatus": "Running",
            "workers": [
                "192.168.128.133:53306",
                "192.168.128.133:53307",
                "192.168.128.133:53308"
            ]
        }
    ]
}
#任务运行正常
```

上游数据全量导入 TiDB 完成后，我们登录 **dm-worker** 机器，删除全备并保留表结构文件（节约磁盘空间成本），操作如下  
先看一下备份目录的文件

```
[root@dmworker dumped_data.shard_to_tidb]# ll |awk '{print $NF}'
metadata
shard_db01-schema-create.sql
shard_db01.shard_tb01-schema.sql
shard_db01.shard_tb01.sql
shard_db02-schema-create.sql
shard_db02.shard_tb02-schema.sql
shard_db02.shard_tb02.sql
```

删除备份的数据，必须保留表库结构信息否则会出错

```
[root@dmworker dumped_data.shard_to_tidb]# ls | grep -v schema | xargs rm -f
[root@dmworker dumped_data.shard_to_tidb]# ls
shard_db01.shard_tb01-schema.sql      shard_db02.shard_tb02-schema.sql
shard_db02-schema-create.sql
```

查看上游分库分表数据已迁移到下游 **merge\_db** 库 **merge\_tb** 表

id	uid	uname	gender	shard	mobile
580981944116838401	10001	tb01001		0 shard_db01_tb01	136 17
580981944116838402	10002	tb01002		1 shard_db01_tb01	136 17
585503135930253313	20001	tb02001		1 shard_db02_tb02	136 33
585503135930253314	20002	tb02002		0 shard_db02_tb02	139 63
1166485080047091713	30001	tb03001		0 shard_db03_tb03	135 73
1166485080047091714	30002	tb03002		0 shard_db03_tb03	139 46
1171006271860506625	40001	tb04001		0 shard_db04_tb04	137 91
1171006271860506626	40002	tb04002		1 shard_db04_tb04	138 91
1751988215977345025	50001	tb05001		1 shard_db05_tb05	158 96
1751988215977345026	50002	tb05002		0 shard_db05_tb05	188 92
1756509407790759937	60001	tb06001		1 shard_db06_tb06	178 98
1756509407790759938	60002	tb06002		1 shard_db06_tb06	175 31

## 5.1.2.2 DM 常用管理命令

### 1. dm-worker 部署管理

部署命令 `deploy.yml`

```
#部署所有inventory.ini中所有选项中的主机服务
[tidb@dmmaster dm-ansible]$ ansible-playbook deploy.yml
#使用-l 参数部署指定标签，如部署 mysql3306 标签的 dm-worker 主机服务
[tidb@dmmaster dm-ansible]$ ansible-playbook deploy.yml -l mysql3306
#使得 --tags 部署指定部署 deploy.yml 中的某个标签任务，如仅部署所有的 dm-worker
[tidb@dmmaster dm-ansible]$ ansible-playbook deploy.yml --tags=dm-worker
```

**dm-worker** 启动停止更新命令

```
#启动 dm 集群，开始自动拉取上游 MySQL 的 binlog 日志
#相当于开启了 MySQL 的 Slave_IO_Running 线程
[tidb@dmmaster dm-ansible]$ ansible-playbook start.yml
#停止 dm 集群，停止拉取上游 MySQL 的 binlog日志
[tidb@dmmaster dm-ansible]$ ansible-playbook stop.yml
#滚动更新 dm 集群
[tidb@dmmaster dm-ansible]$ ansible-playbook rolling_update.yml
*此三个yml命令也可以配合-l、--tags一起使用。
```

### 2. dm-worker task 管理

管理 task 需要使用 `dmctl` 连接上中控机，输入 `help` 查看所有命令信息

```
#连接中控
[tidb@dmmaster dm-ansible]$ dmctl -master-addr 192.168.128.132:8261
»help
```

**start-task** 命令读取 task 文件启动同步任务，相当于开启 MySQL 的 `Slave_SQL_Running` 线程

```
[tidb@dmmaster dm-ansible]$ dmctl -master-addr 192.168.128.132:8261
#启动 task.yaml 配置文件中的所有 dm-worker 数据写入下游库任务
»start-task shard_to_tidb.yaml
#启动 shard_to_tidb.yaml 配置文件中的某个 dm-worker 数据写入下游库任务
#启动 shard_to_tidb.yaml 对应的子任务 192.168.128.133:53307, 如下:
»start-task -w '192.168.128.133:53307' shard_to_tidb.yaml
```

**stop-task** 命令终止同步任务，相当于停止 MySQL 的 `Slave_SQL_Running` 线程

```
[tidb@dmmaster dm-ansible]$ dmctl -master-addr 192.168.128.132:8261
#停止 shard_to_tidb.yaml 配置文件中的所有 dm-worker 数据写入下游库任务
#也可使用-w 参数停止某个指定的任务[可选 -w IP:PORT]
»start-task shard_to_tidb
```

**query-status** 命令查看任务状态，默认显示所有任务状态，可指定任务名查看

```
[tidb@dmmaster dm-ansible]$ dmctl -master-addr 192.168.128.132:8261
#查看 shard_to_tidb 任务当前状态
»query-status shard_to_tidb
```

**query-error** 命令查看任务错误信息，默认显示所有任务的错误信息，可指定任务名查看

```
[tidb@dmmaster dm-ansible]$ dmctl -master-addr 192.168.128.132:8261
#查看 shard_to_tidb 任务当前状态
»query-error shard_to_tidb
```

**skip\_sql** 跳过正在执行的 SQL 语句

```
#看出错的 binlog 位置(failedBinlogPosition), 确定是否可以路过错误
query-error shard_to_tidb
#跳过当前错误的 binlog
sql-skip --worker=192.168.128.133:53307 --binlog-pos=mysql-bin|000001.000003:737983 shard_to_tidb
#恢复继续任务
resume-task --worker=192.168.128.133:53307 shard_to_tidb
#再次查看错误信息
query-error shard_to_tidb
```

## 背景描述

业务的发展，以及数据的爆发式增长，公司会面临一轮数据库的垂直拆分和水平拆分。拆分后对代码的侵入性较大，后续的不断扩容让 DBA 的管理成本上升。所以急需既支持关系型数据库 RDBMS 和非关系型数据库 NoSQL 分布式的存储计算引擎。

TiDB 分布式数据库结合了传统的 RDBMS 和 NoSQL 的最佳特性。首先，高度兼容 MySQL 协议，大多数情况代码可以直接迁移到 TiDB 分布式数据库，已经分库分表的实例可以在 TiDB 中进行聚合；同时，TiDB 分布式数据库支持水平弹性扩展，通过简单地增加新节点即可实现 TiDB 分布式数据库的水平扩展，按需扩展计算节点或存储节点，轻松应对高并发、海量数据场景。

所以基于 TiDB 分布式数据库的上述特性，本文将介绍通过 OGG 将 Oracle 的数据同步到 TiDB 分布式数据库的相关原理及操作步骤。

### 5.2.1 环境说明

#### 1. 软件版本

Oracle GoldenGate 12.3.0.1.4 for Oracle on Linux x86-64

Oracle GoldenGate 12.3.0.1.5 for MySQL-compatible DBs on Linux x86-64

本文档以 Oracle 11G 数据库为例，对 Oracle 数据库通过 OGG 同步至 TiDB 分布式数据库中进行说明。

Oracle 端字符集为 ZHS16GBK。

#### 2. 同步过程说明

##### 2.1. 表结构转换

由于 Oracle 中表数据类型与 TiDB 分布式数据库中数据类型不一致，需要预先进行表结构的转换（可以使用 Navicat 工具）。

##### 2.2. 初始化同步

初始化数据通过 OGG initial load 进程将 Oracle 数据导入到 TiDB 分布式数据库中，如此时 Oracle 数据库中仍然有写入操作，则 initial load 开始时间之后的数据变化无法同步，此时需要进行增量的数据同步，且在同步多张表时，每个表同步的时间并不一致。如采用停机同步（同步期间 Oracle 中表数据不再变化），仅需要进行初始化同步操作。

##### 2.3. 增量数据同步

在初始化同步的基础上进行后续变化的数据同步，需要在初始化同步之前，先开启日志抽取。由于异构平台，无法基于统一时间点完成数据初始导入操作，所以在完成初始化导入操作之后需要完整应用开始 initial load 同步之后的所有 log，此时会存在重复执行的问题，OGG 中通过 handlecollisions 参数处理冲突的场景，保证最终数据的一致性（根据主键或唯一键进行重复的操作可以保证最终数据一致，在缺少主键的场景可能会导致数据重复）。

#### 3. 源端 Oracle 要求

- 归档模式
- Force logging
- ENABLE\_GOLDENGATE\_REPLICATION 参数为 true (11.2.0.4)
- 最小补全日志（根据同步数据范围选择）
  - 表级别
  - Schema 级别
  - 数据库级
- 用户权限
  - 建议 DBA 角色。

## 4. 目标端 TiDB 分布式数据库要求

- set tidb\_constraint\_check\_in\_place = 1;

该参数将 TiDB 分布式数据库中乐观锁模式下的主键冲突检查由 commit 时检查提升为 insert 时检查，在遇到主键冲突时可配置为忽略冲突，否则在遇到主键冲突时无法忽略，复制进程会 abend。仅需在 OGG 复制进程 session 级别设置，通过配置在复制进程参数中实现，无需全局修改。

- lower-case-table-names = 1

OGG 复制进程需要该参数设置为 1，tidb 中修改此参数并未产生实际效果变化，仅为实现兼容。

## 5. OGG 同步要求

DDL 建表语句需提前转换，并在目标端执行。

如下表的示例：

- 源端 Oracle

```
create table account (
    account_number      number(10,0),
    account_balance     decimal(38,2),
    account_trans_ts    timestamp(6),
    account_trans_type  varchar2(30),
    primary key (account_number)
    using index
);
```

- 目标端 TiDB 分布式数据库

```
create table account (
    account_number int,
    account_balance decimal(38,2),
    account_trans_ts timestamp,
    account_trans_type varchar(30),
    primary key (account_number));
```

## 5.2.2 环境准备

### 1. OGG 安装 - Oracle 端

解压安装包

```
$unzip V975837-01.zip
$ ls -l
total 1201820
drwxr-xr-x 3 oracle oinstall      4096 Apr 16  2018 fbo_ggs_Linux_x64_shiphome
-rw-r--r-- 1 oracle oinstall      1396 May 10  2018 OGG-12.3.0.1.4-README.txt
-rw-r--r-- 1 oracle oinstall    293566 May 10  2018 OGG_WinUnix_Rel_Notes_12.3.0.1.4.pdf
```

采用静默安装方式

(1) 编辑应答文件

```
$vi ./fbo_ggs_Linux_x64_shiphome/Disk1/response/oggcore.rsp
#####
## Copyright(c) Oracle Corporation 2017. All rights reserved.      ##
##   ##
## Specify values for the variables listed below to customize   ##
## your installation.  ##
##   ##
```

```

## Each variable is associated with a comment. The comment      ##
## can help to populate the variables with the appropriate  ##
## values.   ##
##   ##
## IMPORTANT NOTE: This file should be secured to have read  ##
## permission only by the oracle user or an administrator who  ##
## own this installation to protect any sensitive input values.  ##
##   ##
#####
#-----#
# Do not change the following system generated value.
#-----
oracle.install.responseFileVersion=/oracle/install/rspfmt_ogginstall_response_schema_v12_1_2

#####
##   ##
## Oracle GoldenGate installation option and details      ##
##   ##
##   ##
#-----#
# Specify the installation option.
# Specify ORA12c for installing Oracle GoldenGate for Oracle Database 12c and
#       ORA11g for installing Oracle GoldenGate for Oracle Database 11g
#-----
INSTALL_OPTION=ORA11g
##此处为 Oracle 数据库版本, 本文档使用 11.2.0.4, 此处填写 ORA11G, 如果是 12c 版本则填写 ORA12C
#-----
# Specify a location to install Oracle GoldenGate
#-----
SOFTWARE_LOCATION=/home/oracle/ogg12.3
##此处为安装路径
#-----
# Specify true to start the manager after installation.
#-----
START_MANAGER=false
##安装完成不启动 mgr 进程, 设为 false
#-----
# Specify a free port within the valid range for the manager process.
# Required only if START_MANAGER is true.
#-----
MANAGER_PORT=
#-----
# Specify the location of the Oracle Database.
# Required only if START_MANAGER is true.
#-----
DATABASE_LOCATION=

#####
##   ##
## Specify details to Create inventory for Oracle installs  ##
## Required only for the first Oracle product install on a system.  ##
##   ##
##   ##
#-----#
# Specify the location which holds the install inventory files.
# This is an optional parameter if installing on
# Windows based Operating System.
#-----
INVENTORY_LOCATION=
#-----
# Unix group to be set for the inventory directory.
# This parameter is not applicable if installing on
# Windows based Operating System.
#-----
UNIX_GROUP_NAME=

```

## (2) 安装

执行 fbo\_ggs\_Linux\_x64\_shiphome/Disk1/runInstaller

```
./fbo_ggs_Linux_x64_shiphome/Disk1/runInstaller -silent -responseFile /home/oracle/oggsoft/fbo_ggs_Linux_x64_shiphome/Disk1/response/oggcore.rsp
#此处 response 文件不支持相对路径
```

安装完成后日志中会显示如下 successful

```
The installation of Oracle GoldenGate Core was successful
```

在指定安装目录中会生成 OGG 相关文件

```
$ ls -l
total 225044
-rwxr-xr-x 1 oracle oinstall      426 Oct 15 2010 bcpfmt.tpl
-rwxr-xr-x 1 oracle oinstall    1725 Oct 15 2010 bcrypt.txt
-rwxrwxr-x 1 oracle oinstall 1612776 Apr 15 2018 cachefiledump
drwxr-xr-x 4 oracle oinstall    4096 Jul 28 14:45 cfgtoollogs
-rwxrwxr-x 1 oracle oinstall 3563576 Apr 15 2018 checkprm
-rw-rw-r-- 1 oracle oinstall   1021 Apr 15 2018 chkpt_ora_create.sql
-rwxrwxr-x 1 oracle oinstall 3379568 Apr 15 2018 convchk
-rwxrwxr-x 1 oracle oinstall 4716080 Apr 15 2018 convprm
drwxr-xr-x 2 oracle oinstall    4096 Jul 28 14:45 crypto
-rwxr-xr-x 1 oracle oinstall     159 Oct 15 2010 db2cntl.tpl
-rwxrwxr-x 1 oracle oinstall    9696 Apr 15 2018 db_upgrade
-rw-rw-r-- 1 oracle oinstall    455 Apr 15 2018 ddl_cleartrace.sql
-rw-rw-r-- 1 oracle oinstall   8414 Apr 15 2018 ddl_create.sql
-rw-rw-r-- 1 oracle oinstall   3176 Apr 15 2018 ddl_ddl2file.sql
-rw-rw-r-- 1 oracle oinstall     90 Apr 15 2018 ddl_disable.sql
-rw-rw-r-- 1 oracle oinstall     88 Apr 15 2018 ddl_enable.sql
-rw-rw-r-- 1 oracle oinstall    2036 Apr 15 2018 ddl_filter.sql
-rw-rw-r-- 1 oracle oinstall 12220 Apr 15 2018 ddl_ora10.sql
-rw-rw-r-- 1 oracle oinstall   1725 Apr 15 2018 ddl_ora10upCommon.sql
-rw-rw-r-- 1 oracle oinstall 13539 Apr 15 2018 ddl_ora11.sql
-rw-rw-r-- 1 oracle oinstall 12564 Apr 15 2018 ddl_ora9.sql
-rw-rw-r-- 1 oracle oinstall    216 Apr 15 2018 ddl_pin.sql
-rw-rw-r-- 1 oracle oinstall   3184 Apr 15 2018 ddl_remove.sql
-rw-rw-r-- 1 oracle oinstall      1 Apr 15 2018 ddl_session1.sql
-rw-rw-r-- 1 oracle oinstall    629 Apr 15 2018 ddl_session.sql
-rw-rw-r-- 1 oracle oinstall 287877 Apr 15 2018 ddl_setup.sql
-rw-rw-r-- 1 oracle oinstall   8401 Apr 15 2018 ddl_status.sql
-rw-rw-r-- 1 oracle oinstall   2122 Apr 15 2018 ddl_staymetadata_off.sql
-rw-rw-r-- 1 oracle oinstall   2118 Apr 15 2018 ddl_staymetadata_on.sql
-rw-rw-r-- 1 oracle oinstall   2186 Apr 15 2018 ddl_tracelevel.sql
-rw-rw-r-- 1 oracle oinstall   2133 Apr 15 2018 ddl_trace_off.sql
-rw-rw-r-- 1 oracle oinstall   2383 Apr 15 2018 ddl_trace_on.sql
-rwxrwxr-x 1 oracle oinstall 5037440 Apr 15 2018 defgen
drwxr-xr-x 2 oracle oinstall    4096 Jul 28 14:45 deinstall
-rw-rw-r-- 1 oracle oinstall    882 Apr 15 2018 demo_more_ora_create.sql
-rw-rw-r-- 1 oracle oinstall    649 Apr 15 2018 demo_more_ora_insert.sql
-rw-rw-r-- 1 oracle oinstall    583 Apr 15 2018 demo_ora_create.sql
-rw-rw-r-- 1 oracle oinstall    504 Apr 15 2018 demo_ora_insert.sql
-rw-rw-r-- 1 oracle oinstall   3597 Apr 15 2018 demo_ora_lob_create.sql
-rw-rw-r-- 1 oracle oinstall   1943 Apr 15 2018 demo_ora_misc.sql
-rw-rw-r-- 1 oracle oinstall   1056 Apr 15 2018 demo_ora_pk_befores_create.sql
-rw-rw-r-- 1 oracle oinstall   1013 Apr 15 2018 demo_ora_pk_befores_insert.sql
-rw-rw-r-- 1 oracle oinstall   2305 Apr 15 2018 demo_ora_pk_befores_updates.sql
drwxr-xr-x 3 oracle oinstall    4096 Jul 28 14:45 diagnostics
drwxr-xr-x 3 oracle oinstall    4096 Jul 28 14:45 diretc
drwxr-xr-x 2 oracle oinstall    4096 Jul 28 14:45 dirout
drwxr-xr-x 4 oracle oinstall    4096 Jul 28 14:45 dirsca
-rwxrwxr-x 1 oracle oinstall 4272400 Apr 15 2018 emsclnt
-rwxrwx--- 1 oracle oinstall 12545040 Apr 15 2018 extract
-rwxr-xr-x 1 oracle oinstall   1968 Oct 15 2010 freeBSD.txt
-rwxrwxr-x 1 oracle oinstall 4280528 Apr 15 2018 ggcmd
-rwxr-xr-x 1 oracle oinstall 2303056 Apr 15 2018 ggMessage.dat
-rwxr-xr-x 1 oracle oinstall 49675440 Apr 15 2018 ggparam.dat
-rwxrwx--- 1 oracle oinstall 9340192 Apr 15 2018 ggsci
drwxr-xr-x 2 oracle oinstall    4096 Jul 28 14:45 healthcheck
-rwxr-xr-x 1 oracle oinstall 299451 Nov 16 2017 help.txt
```

```

drwxr-xr-x  3 oracle oinstall      4096 Jul 28 14:45 install
drwxr-x--- 12 oracle oinstall     4096 Jul 28 14:45 inventory
drwxr-xr-x  7 oracle oinstall     4096 Jul 28 14:45 jdk
-rwxrwxr-x  1 oracle oinstall    144496 Apr 15 2018 keygen
-rw-rw-r--  1 oracle oinstall      56 Apr 15 2018 label.sql
-rwxrwx---  1 oracle oinstall   102840 Apr 15 2018 libantlr3c.so
-rwxrwxr-x  1 oracle oinstall    12312 Apr 15 2018 libboost_system-mt.so.1.58.0
-rwxrwx---  1 oracle oinstall  2190856 Apr 15 2018 libdb-6.1.so
-rwxrwx---  1 oracle oinstall  2198136 Apr 15 2018 libglog.so
-rwxrwx---  1 oracle oinstall 10524064 Apr 15 2018 libgnnzip.so
-rwxrwx---  1 oracle oinstall 21993240 Apr 15 2018 libggparam.so
-rwxrwx---  1 oracle oinstall  210264 Apr 15 2018 libgperf.so
-rwxrwx---  1 oracle oinstall  779352 Apr 15 2018 libgrep.so
-rwxrwx---  1 oracle oinstall 1108760 Apr 15 2018 libgss.so
-rwxrwx---  1 oracle oinstall 125624 Apr 15 2018 libgutil.so
-rwxrwxr-x  1 oracle oinstall 29764432 Apr 15 2018 libicudata.so.56
-rwxrwxr-x  1 oracle oinstall 2909360 Apr 15 2018 libicui18n.so.56
-rwxrwxr-x  1 oracle oinstall 1995808 Apr 15 2018 libicuuc.so.56
-rwxrwx---  1 oracle oinstall  86960 Apr 15 2018 liblmbdb.so
-rwxrwxr-x  1 oracle oinstall 175136 Apr 15 2018 libPocoCrypto.so.48
-rwxrwxr-x  1 oracle oinstall 3079760 Apr 15 2018 libPocoFoundation.so.48
-rwxrwxr-x  1 oracle oinstall 373232 Apr 15 2018 libPocoJSON.so.48
-rwxrwxr-x  1 oracle oinstall 1326504 Apr 15 2018 libPocoNet.so.48
-rwxrwxr-x  1 oracle oinstall 350064 Apr 15 2018 libPocoNetSSL.so.48
-rwxrwxr-x  1 oracle oinstall 520864 Apr 15 2018 libPocoUtil.so.48
-rwxrwxr-x  1 oracle oinstall 680288 Apr 15 2018 libPocoXML.so.48
-rwxrwx---  1 oracle oinstall 1115360 Apr 15 2018 libudt.so
-rwxrwx---  1 oracle oinstall 4782344 Apr 15 2018 libxerces-c-3.1.so
-rwxrwxr-x  1 oracle oinstall 5025840 Apr 15 2018 logdump
-rw-rw-r--  1 oracle oinstall    1553 Apr 15 2018 marker_remove.sql
-rw-rw-r--  1 oracle oinstall    3309 Apr 15 2018 marker_setup.sql
-rw-rw-r--  1 oracle oinstall    675 Apr 15 2018 marker_status.sql
-rwxrwxr-x  1 oracle oinstall 6570480 Apr 15 2018 mgr
-rwxr-xr-x  1 oracle oinstall  41643 Jun 30 2017 notices.txt
-rwxrwxr-x  1 oracle oinstall 1661024 Apr 15 2018 ogger
drwxr-xr-x 12 oracle oinstall     4096 Jul 28 14:45 OPatch
-rw-r----- 1 oracle oinstall     59 May 30 13:35 oraInst.loc
drwxr-xr-x  8 oracle oinstall     4096 Jul 28 14:45 oui
-rw-rw-r--  1 oracle oinstall    3146 Apr 15 2018 params.sql
-rwxrwxr-x  1 oracle oinstall 11524576 Apr 15 2018 pmsrvr
-rwxr-xr-x  1 oracle oinstall    1272 Dec 28 2010 prvtclkm.plb
-rwxr-xr-x  1 oracle oinstall    9487 May 27 2015 prvtlmpg.plb
-rw-rw-r--  1 oracle oinstall   2724 Apr 15 2018 prvtlmpg_uninstall.sql
-rw-rw-r--  1 oracle oinstall   1532 Apr 15 2018 remove_seq.sql
-rwxrwx---  1 oracle oinstall 10873240 Apr 15 2018 replicat
-rwxrwxr-x  1 oracle oinstall 1656864 Apr 15 2018 retrace
-rw-rw-r--  1 oracle oinstall    3187 Apr 15 2018 role_setup.sql
-rw-rw-r--  1 oracle oinstall   35254 Apr 15 2018 sequence.sql
-rwxrwxr-x  1 oracle oinstall 4659736 Apr 15 2018 server
-rwxr-xr-x  1 oracle oinstall    4917 Jan  5 2017 SQLDataTypes.h
-rwxr-xr-x  1 oracle oinstall    248 Oct 15 2010 sqlldr.tpl
drwxr-xr-x  3 oracle oinstall     4096 Jul 28 14:45 srvm
-rwxrwxr-x  1 oracle oinstall    759 Oct 15 2010 tcperrs
-rwxr-xr-x  1 oracle oinstall   37877 Apr 16 2016 ucharset.h
-rw-rw-r--  1 oracle oinstall   7341 Apr 15 2018 ulg.sql
drwxr-xr-x  7 oracle oinstall     4096 Jul 28 14:45 UserExitExamples
-rwxr-xr-x  1 oracle oinstall   32987 Jun  2 2017 usrdecs.h
-rwxr-xr-x  1 oracle oinstall   1033 Oct 19 2016 zlib.txt

```

## (3) 设置环境变量并将环境变量加入 .bash\_profile 中

```
export LD_LIBRARY_PATH=$ORACLE_HOME/lib
```

## (4) 测试能否正常运行，并创建相关目录

```
$ ./ggsci
Oracle GoldenGate Command Interpreter for Oracle
Version 12.3.0.1.4 OGGCORE_12.3.0.1.0_PLATFORMS_180415.0359_F80
Linux, x64, 64bit (optimized), Oracle 11g on Apr 15 2018 21:16:09
Operating system character set identified as UTF-8.
Copyright (C) 1995, 2018, Oracle and/or its affiliates. All rights reserved.

GGSCI ( hostname ) 1> create subdirs
Creating subdirectories under current directory /home/oracle/ogg12.3
Parameter file          /home/oracle/ogg12.3/dirprm: created.
Report file             /home/oracle/ogg12.3/dir rpt: created.
Checkpoint file         /home/oracle/ogg12.3/dir chk: created.
Process status files   /home/oracle/ogg12.3/dir pcs: created.
SQL script files       /home/oracle/ogg12.3/dir sql: created.
Database definitions files /home/oracle/ogg12.3/dir def: created.
Extract data files     /home/oracle/ogg12.3/dir ddat: created.
Temporary files        /home/oracle/ogg12.3/dir tmp: created.
Credential store files /home/oracle/ogg12.3/dir crd: created.
Masterkey wallet files /home/oracle/ogg12.3/dir wl t: created.
Dump files              /home/oracle/ogg12.3/dir dmp: created.
```

Oracle 端 OGG 完成安装。

## 2. OGG 安装 - TiDB 分布式数据库端

### (1) 解压安装包

```
$ unzip V978711-01.zip
Archive:  V978711-01.zip
  inflating: ggs_Linux_x64_SQL_64bit.tar
  inflating: OGG-12.3.0.1-README.txt
  inflating: OGG_WinUnix_Rel_Notes_12.3.0.1.5.pdf
```

(2) 将 ggs\_Linux\_x64\_SQL\_64bit.tar 解压至安装目录即完成安装。

```
$tar xvf ./ggs_Linux_x64_SQL_64bit.tar -C /home/tidb/ogg12.3
```

(3) 进入解压目录测试 ggsci 是否能正常运行，并创建相应目录

```
$ ./ggsci
Oracle GoldenGate Command Interpreter for MySQL
Version 12.3.0.1.5 OGGCORE_12.3.0.1.0_PLATFORMS_180501.2124
Linux, x64, 64bit (optimized), MySQL Enterprise on May 2 2018 10:58:16
Operating system character set identified as UTF-8.
Copyright (C) 1995, 2018, Oracle and/or its affiliates. All rights reserved.

GGSCI ( hostname ) 1> create subdirs
Creating subdirectories under current directory /home/tidb/ogg12.3
Parameter file          /home/tidb/ogg12.3/dirprm: created.
Report file             /home/tidb/ogg12.3/dir rpt: created.
Checkpoint file         /home/tidb/ogg12.3/dir chk: created.
Process status files   /home/tidb/ogg12.3/dir pcs: created.
SQL script files       /home/tidb/ogg12.3/dir sql: created.
Database definitions files /home/tidb/ogg12.3/dir def: created.
Extract data files     /home/tidb/ogg12.3/dir ddat: created.
Temporary files        /home/tidb/ogg12.3/dir tmp: created.
Credential store files /home/tidb/ogg12.3/dir crd: created.
Masterkey wallet files /home/tidb/ogg12.3/dir wl t: created.
Dump files              /home/tidb/ogg12.3/dir dmp: created.
```

OGG TiDB 分布式数据库端安装完成 (OGG For MySQL 版本)。

### 3. TiDB 分布式数据库端数据库环境准备

(1) 设置 lower-case-table-names 参数为 1

```
$grep lower-case-table-names tidb.toml
lower-case-table-names = 1
```

(2) 检查参数是否正确

```
MySQL [(none)]> show variables like '%lower%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| lower_case_table_names | 1      |
| lower_case_file_system | 1      |
+-----+-----+
MySQL [(none)]> show variables like '%place%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| tidb_constraint_check_in_place | 0      |
+-----+-----+
```

(3) 创建用于同步的用户并赋权

```
MySQL [scott]> create user 'tidb' identified by 'tidb';
Query OK, 1 row affected (0.01 sec)
MySQL [scott]> GRANT ALL PRIVILEGES ON scott.* to oggadmin ;
Query OK, 0 rows affected (0.01 sec)
MySQL [scott]> show create user tidb;
+-----+
| CREATE USER for tidb@%          |
+-----+
| CREATE USER 'tidb'@'%' IDENTIFIED WITH 'mysql_native_password' AS '465D123EE55795DBDBDAE36AFD3DCD9C429B718A' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT UNLOCK |
+-----+
1 row in set (0.00 sec)
```

(4) 创建对应表结构

### 4. Oracle 端数据库环境准备

4.1 查看数据库是否开启归档

```
SQL> archive log list;
```

若归档没有开启，需要开启归档

(1) 关闭数据库

```
SQL> shutdown immediate;
```

(2) 将数据库启动到 mount 状态

```
SQL> startup mount;
```

(3) 开启数据库的归档

```
SQL> alter database archivelog;
```

## (4) 打开数据库

```
SQL> alter database open;
```

## (5) 归档状态

```
SQL> archive log list;
```

## 4.2 开启数据级别的最小增量日志

## (1) 查询是否开启增量日志

```
select log_mode, supplemental_log_data_min, force_logging from v$database;
```

## (2) 开启 force logging

开启强同步，会记录所有的事务日志以及数据导入的日志，即使用户设置了 nolog 也会记录。

```
SQL> ALTER DATABASE FORCE LOGGING;
```

## (3) 附加增量日志

```
SQL> ALTER DATABASE ADD SUPPLEMENTAL LOG DATA;
SQL> ALTER SYSTEM SWITCH LOGFILE;
```

## (4) 数据库中允许启动 OGG

```
SQL> alter system set enable_goldengate_replication=true;
```

## 4.3 创建 ogg 专属账户

创建 goldengate 用户，并赋予相关权限

```
SQL> create tablespace goldengate datafile '/opt/oracle/data/goldengate001.dbf' size 4G;
SQL> create user goldengate identified by goldengate default tablespace goldengate;
SQL> grant dba to goldengate;
```

## 5. OGG 中创建免密登录文件

## (1) 在 ggsic 中创建免密登录秘钥

```
./ggsic
create subdirs
add credentialstore
alter credentialstore add user goldengate,password goldengate
```

## (2) 直接执行下面命令即可在 ogg 中登录到 goldengate 用户

```
dblogin useralias goldengate
```

## 6. 开启 schema 级别的附加日志

强烈建议开启 schema 级别的附加日志，因为这样能够确保 schema 下新表的附加日志也会被自动捕获

(1) 若数据库版本低于 11.2.0.2，则需要打 Oracle Patch 13794550

若以前的 oracle 数据库版本没有打上面的补丁，开启 schema 级别附加日志会报如下错误：

ERROR OGG-06522 Cannot verify existence of table function that is required to ADD schema level supplemental logging.  
failed.

(2) GGSCI 登录有两种方式

- 1) dblogin userid goldengate, password goldengate
- 2) dblogin useridalias goldengate

(3) 为指定 schema 开启日志捕获

```
ADD SCHEMATRANSACTION schema ALLCOLS
```

## 7. 开启表级别的增量日志

当没有启动 schema 级别的附加日志时，可以使用基于表级别的附加日志

(1) GGSCI 登录

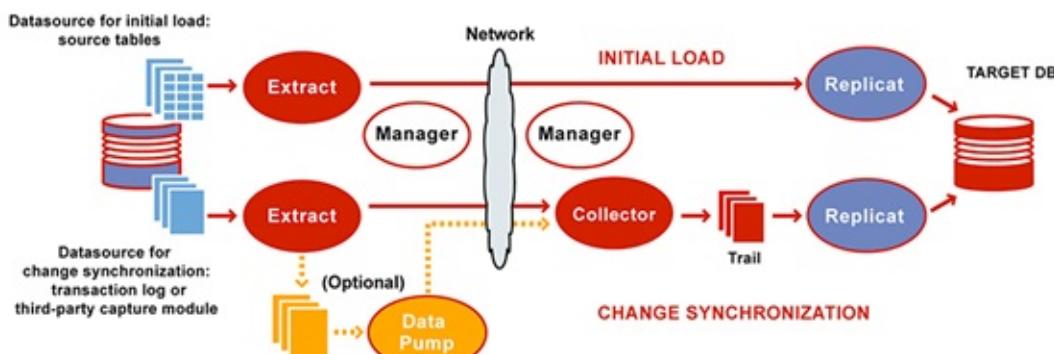
```
dblogin userid goldengate, password goldengate
```

(2) 为指定 schema 开启日志捕获

```
ADD TRANSACTION schema.tablename NOKEY
```

### 5.2.3 OGG 配置

#### 1. OGG 基本配置图



基本配置中，主要是两套进程：

- INITIAL LOAD：即全量同步，主要是一个抽取进程和应用进程，将存量数据从原端同步到目标端。
- CHANGE SYNCHRONIZATION：即增量同步进程，有三个进程，抽取进程（捕获源端数据实时变化日志，并生成 trail 文件。oracle 数据库的话为 oracle 的 redo log 和 archive log），投递进程（将抽取进程产生的 trail 文件投递到目标端，给应用进程回放），应用进程（回放投递进程投递过来的日志文件，应用到目标数据库）

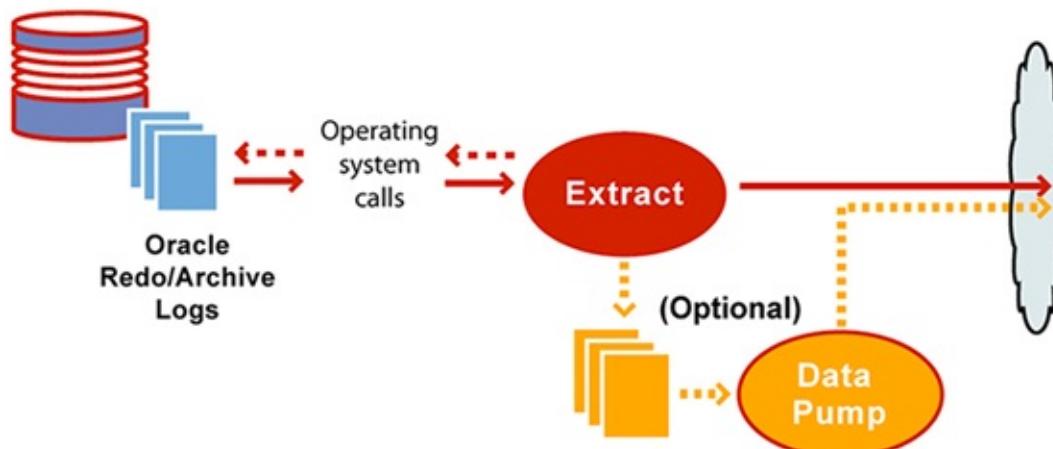
本地配置参数说明：

参数值	参数说明
/opt/oracle/11.2.0	为 ORACLE_HOME 的值
ora	为 Oracle 的 sid 名称
ext_ora	抽取进程名称
tdpm_ora	投递进程名称
rep_ora	应用进程名称
./dirdat/sk	抽取文件存放目录及命名规则
/Data/tidb/ogg/dirdat/sk	Replicat 端接收日志的存放目录

## 2. 配置增量日志抽取端进程

以下示例中抽取是从 ADG 上进行抽取

Extract 和 pump 进程架构图



上图只是通过 Extract 进程增量捕获源端数据库 redo 和 archive log 的变化，然后通过 Data pump 进程将变化的日志投递到远端服务器

### 2.1 MGR 进程配置信息

Manager 进程是 OGG 的控制进程，用于管理 Extract、Pump、Replicat 等进程，在 Extract、Pump、Replicat 进程启动之前，Manager 进程必须先要在源端和目标端启动。

#### (1) 进程配置信息

```

GGSCI (b-db-ps-055) 5> view param mgr

port 9001
DYNAMICPORTLIST 9001-9020
--AUTORESTART ER *,RETRIES 5,WAITMINUTES 7
PURGEOLDEXTRACTS ./dirdat/*,usecheckpoints, minkeepdays 2
LAGREPORTHOURS 1
LAGINFOMINUTES 30
LAGCRITICALMINUTES 45
  
```

参数说明：

参数	说明
port	指定服务的默认端口
DYNAMICPORTLIST	端口列表，用户本地 OGG 进程与远端 OGG 进程通信
AUTORESTART	自动重启参数
PURGEOLDEXTRACTS	定期清理 trail 文件，此处表示超过 2 天的 trail 文件会进行清理
LAGREPORTHOURS	表示检查 lag 的频率（小时），此处表示每隔 1 小时检查一次
LAGINFOMINUTES	延时超过设定值（分钟），会将延时信息记录在错误日志中

## (2) 启动 mgr 进程

```
./ggsci
start mgr
```

## 2.2 Extract 进程配置信息

Extract 进程运行在数据库源端上，它是 OGG 的捕获进程，可以配置 Extract 进程来初始数据装载和同步变化捕获。

### (1) 进程配置信息

```
GGSCI (b-db-ps-055) 2> edit param ext_ora

EXTRACT ext_ora
SETENV (ORACLE_HOME="/opt/oracle/11.2.0")
setenv (NLS_LANG="AMERICAN_AMERICA.AL32UTF8")
setenv (ORACLE_SID="ora")
useralias goldengate
GETTRUNCATES
REPORTCOUNT EVERY 1 MINUTES, RATE
DISCARDFILE ./dir rpt/ext_ora.dsc,APPEND,MEGABYTES 1000
WARNLONGTRANS 2h,CHECKINTERVAL 10m
EXTTRAIL ./dir dat/sk
TRANLOGOPTIONS EXCLUDEUSER goldengate
TRANLOGOPTIONS MINEFROMACTIVEADG
DBOPTIONS ALLOWUNUSEDCOLUMN
DYNAMICRESOLUTION
FETCHOPTIONS FETCHKUPDATECOLS,INCONSISTENTROW ALLOW
ddl include mapped objname hr.*
getupdatebefore
nocompressdeletes
nocompressupdates
table hr.table_name;
```

主要参数说明：|参数|说明|:----|:----| |SETENV|设置环境变量||useralias|根据前面创建的认证登录||GETTRUNCATES|捕获源端数据库的 truncate 操作，默认是不捕获||REPORTCOUNT|报告进程从启动开启处理的记录数，此处表示每分钟收集处理的记录数，并报告处理的统计信息||WARNLONGTRANS|监控 OGG 中的长事务，超过指定时间的长事务，进行定期检查||EXTTRAIL|抽取进程产生 trail 文件的存放目录||TRANLOGOPTIONS EXCLUDEUSER|在抽取时，排除掉指定用户的日志||TRANLOGOPTIONS MINEFROMACTIVEADG|指定抽取进程能够从 ADG 中读取本地日志，若没有这个参数，抽取进程将会异常结束||DBOPTIONS ALLOWUNUSEDCOLUMN|抽取进程在抽取的过程中，遇到未使用的列时，进程会继续处理，并产生一个警告||DYNAMICRESOLUTION|OGG 进程中可能由于表比较多，OGG 为了获取处理记录的元数据，需要查询数据库，然后构建表的相关记录，这样导致 OGG 进程启动非常慢，该参数是一次只生成一个表的记录，其他表的记录是 OGG 进程在抽取的事务日志中，第一次遇到表的 ID 时，再进行创建||FETCHOPTIONS FETCHKUPDATECOLS,INCONSISTENTROW ALLOW|当主键更新时，获取所有列记录；当通过 row id 获取到的列值与主键不匹配时(原记录被删除或者更新了)，设置 ALLOW 表示允许这种情况发生，并继续处理|

### (2) 添加抽取进程

```
./ggsci
add extract ext_ora,TRANLOG, begin now
add ExtTrail ./dirdat/sk, Extract ext_ora, Megabytes 50
```

## 2.3 Pump 进程配置信息

Pump 进程是配置在源端辅助 Extract 进程，Pump 进程将 Extract 进程写好的本地 Trail 文件通过网络发送到目标端的 Trail 文件中。

```
GGSCI (b-db-ps-055) 4> edit param tdpm_ora

EXTRACT tdpm_ora
RMTHOST 31.***.***.***, MGRPORT 9001, compress
PASSTHRU
RMTTRAIL /Data/tidb/ogg/dirdat/sk
DYNAMICRESOLUTION
--table
table hr.table_name;
```

主要参数说明：| 参数 | 说明 |:---|:---| | RMTHOST | 设置远端服务器的 IP 地址，以及远端 MGR 进程的端口 || PASSTHRU | 使用该参数 OGG 不对元数据一致性进行校验，从而提高性能 |

### (1) 添加 pump 进程

```
./ggsci
add extract tdpm_ora, extrailsource ./dirdat/sk
add rmttrail /Data/tidb/ogg/dirdat/sk, extract tdpm_ora, megabytes 100
```

## 3. 配置应用端进程

### 3.1 MGR 进程配置信息

#### (1) 进程配置信息

```
GGSCI (w-db-ps-082) 2> edit param mgr

port 9001
DYNAMICPORTLIST 9001-9120
ACCESSRULE, PROG *, IPADDR 31.*.*.*, ALLOW
--AUTORESTART ER *, RETRIES 5, WAITMINUTES 7
PURGEOLDEXTRACTS ./dirdat/*, usecheckpoints, minkeepdays 10
LAGREPORTHOURS 1
LAGINFOMINUTES 30
LAGCRITICALMINUTES 45
```

#### (2) 启动 mgr 进程

```
./ggsci
start mgr
```

### 3.2 Replicat 进程配置信息

Replicat 进程是运行在目标端系统的一个进程，负责读取 Extract 进程提取到的数据（变更的事务或 DDL 变化）并应用到目标数据库。

#### (1) 创建 checkpoint table (该步骤为一次性工作)

```
dblogin sourcedb tidb@31.***.***.***:4000    userid username password user_password
add checkpointtable tidb.checkpoint_table
```

## (2) 编辑 replicat 进程的参数

```
GGSCI (w-db-ps-082) 4> edit param  rep_ora

replicat rep_ora
targetdb tidb@31.***.***.***:4000 userid  username password user_password
SQLEXEC "set tidb_constraint_check_in_place=1"
handlecollisions
MAXTRANSOPS 10000
discardfile /Data/tidb/ogg/dirrpt/repora.dsc,purge
map hr.table_name,target tidb.table_name,keycols(ID);
```

主要参数说明：| 参数 | 说明 | :---|:---| | SQLEXEC | 可以设置抽取或应用进程与数据库的交互，比如执行相关命令，设置相关参数；tidb\_constraint\_check\_in\_place：默认值为 0，即 INSERT 数据时不做唯一性校验，只在事务提交过程中做校验；OGG 同步时需要将此参数设置为 1 | | handlecollisions | 使用该参数，replicat 处理记录重复冲突和记录丢失：当目标点记录不存在时，replicat 进程转换成 insert；当目标记录存在是，replicat 将进行更新 (该参数一般只在第一次追数据时使用，上下游完全一致的情况下不会出现冲突的情况) |

## (3) 添加 replicat 进程

```
add replicat rep_ora,exttrail /Data/tidb/ogg/dirdat/sk,checkpointtable tidb.checkpoint_table
```

**4. 配置全量抽取****4.1 配置全量抽取进程**

## (1) 全量抽取进程参数

```
GGSCI (w-db-ps-082) 4>edit param  init_ora

extract init_ora
setenv (NLS_LANG=AMERICAN_AMERICA.AL32UTF8)
setenv (ORACLE_SID="ora")
useridalias goldengate
rmthost 31.***.***.***,mgrport 9001
rmttask replicat,group rnit_ora
table hr.table_name;
```

主要参数说明：

参数	说明
rmttask replicat,group	后面为配置初始化 replicat 进程的名称

## (2) 添加进程

```
./ggsci
add extract init_ora,sourceistable
```

**4.2 配置全量应用进程**

## (1) 全量应用进程参数

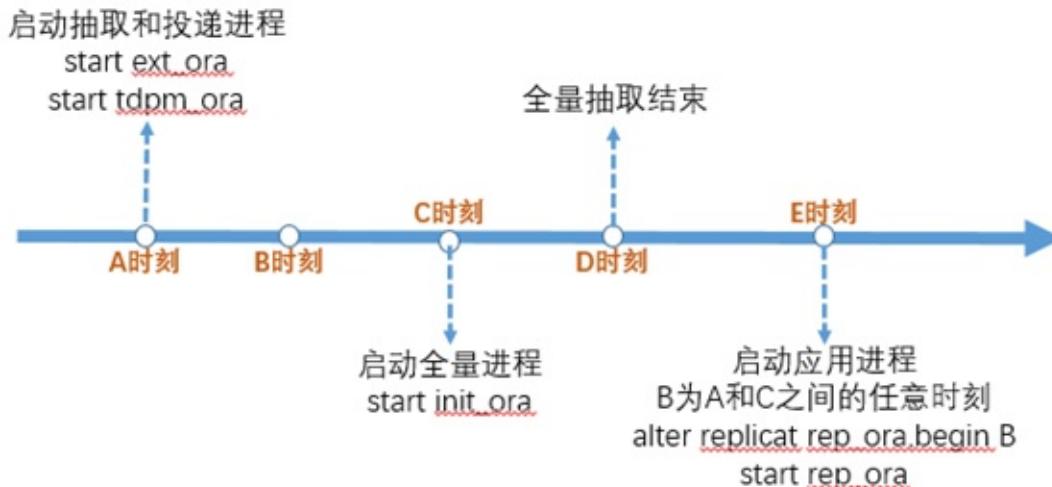
```
GGSCI (w-db-ps-082) 4>edit param  rnit_ora

replicat rnit_ora
SETENV (NLS_LANG = AMERICAN_AMERICA.AL32UTF8)
targetdb tidb@31.***.***.***:4000 userid  username password user_password
discardfile /Data/tidb/ogg/dirrpt/rnit_ora.dsc,purge
map hr.table_name, target  tidb.hr.table_name;
```

## (2) 添加进程

```
./ggsci
add replicat rnit_ora,specialrun
```

## 5. 进程启动顺序



如上图，配置表的全量及增量的完成时间流程图：

1. 首先配置好 MGR 进程，抽取进程，投递进程，应用进程，全量抽取进程，全量应用进程；
2. “A 时刻” 正常启动抽取进程和投递进程，确保投递进程的远端目录有相关的 trail 文件。
3. “C 时刻” 启动全量抽取进程，该时刻必须在“A 时刻”之后，不然会丢失数据；
4. “D 时刻” 全量同步完成；
5. “E 时刻” 准备启动增量 replicat 进程，为了确保数据不丢失，将 replicat 进程要提前到“C 时刻” 和“A 时刻”之间的任意时刻，时间重复部分的日志冲突通过使用参数配置“handlecollisions”自动处理。

### 5.2.4 OGG 日志解析

有时候进程由于某种原因异常，需要对 OGG 的日志进行解析，判断是否有异常数据导致的。

通过下面例子详解分别查看在开启数据库附加日志时，表是否开启附加日志所对应的日志记录。

创建表

```
SQL> CREATE TABLE hr.table_name
  2  (
  3     ID          NUMBER           NOT NULL,
  4     CREATED_AT  DATE            DEFAULT sysdate NOT NULL,
  5     CREATED_BY   VARCHAR2(128 BYTE)  DEFAULT 'SYS'    NOT NULL,
  6     UPDATED_AT  DATE            DEFAULT sysdate NOT NULL,
  7     UPDATED_BY   VARCHAR2(128 BYTE)  DEFAULT 'SYS'    NOT NULL
  8  )
  9 ;
SQL> ALTER TABLE hr.table_name ADD  CONSTRAINT PK_table_name;
```

## 1. 表没有开启附加日志的

### 1.1 插入日志解析

#### (1) 往表中插入一条记录

```
SQL> insert into hr.table_name(id)  values(1);

1 row created.

SQL> commit;

Commit complete.
```

#### (3) 解析 OGG 日志文件

```
Logdump 12 >ghdr on
Logdump 13 >detail on
Logdump 14 >detail data

Logdump 18 >n

Hdr-Ind      : E  (x45)      Partition  : .  (x0c)
UndoFlag     : .  (x00)      BeforeAfter: A  (x41)
RecLength    : 81  (x0051)    IO Time    : 2020/03/06 17:26:30.000.000
IOType       : 5   (x05)      OrigNode   : 255  (xff)
TransInd     : .  (x03)      FormatType: R  (x52)
SyskeyLen    : 0   (x00)      Incomplete : .  (x00)
AuditRBA     : 3539        AuditPos   : 31052304
Continued    : N  (x00)      RecCount   : 1  (x01)

2020/03/06 17:26:30.000.000 Insert          Len     81 RBA 2014
Name: HR.TABLE_NAME (TDR Index: 1)
After Image:
 0000 0005 0000 0001 3100 0100 1500 0032 3032 302d | .....1.....2020-
 3033 2d30 363a 3137 3a32 363a 3238 0002 0007 0000 | 03-06:17:26:28.....
 0003 5359 5300 0300 1500 0032 3032 302d 3033 2d30 | ..SYS.....2020-03-0
 363a 3137 3a32 363a 3238 0004 0007 0000 0003 5359 | 6:17:26:28.....SY
 53           | S
Column 0 (x0000), Len      5  (x0005)
 0000 0001 31           | ....1
Column 1 (x0001), Len      21 (x0015)
 0000 3230 3230 2d30 332d 3036 3a31 373a 3236 3a32 | ..2020-03-06:17:26:2
 38           | 8
Column 2 (x0002), Len      7  (x0007)
 0000 0003 5359 53           | ....SYS
Column 3 (x0003), Len      21 (x0015)
 0000 3230 3230 2d30 332d 3036 3a31 373a 3236 3a32 | ..2020-03-06:17:26:2
 38           | 8
Column 4 (x0004), Len      7  (x0007)
 0000 0003 5359 53           | ....SYS
```

常见参数解析：

**IOType**：表示操作类型，3 表示 delete，5 表示 insert，10 表示 update (full >record)，15 表示 update (compressed record)

**Continued**：该属性有两种值：Y 和 N，用来定义数据片大小，通常 lob，clob 或 varchar 类型，该值是 Y，否则是 N。

**BeforeAfter**：对于 update 操作，表示该数据是 before image(用 B 表示)还是 after image (用 A 表示)。对于 insert 操作，总是 after images，而 delete 操作总是 before images。

**FormatType**：表示数据是从事务日志读取还是直接从数据库中获取的。取值为 F 和 >R，F 表示 fetched from database。R 表示 readable in transaction log。

可以看到，在插入数据的时候，所有列的相关数据都在 trail 文件中。

## 1.2 更新日志解析

### (1) 执行一次 update 操作

```
SQL> update hr.table_name set updated_by='aaaaaaaa';
1 row updated.

SQL> commit;
```

### (2) 解析 OGG 日志文件

```
Logdump 19 >n

Hdr-Ind : E (x45) Partition : . (x0c)
UndoFlag : . (x00) BeforeAfter: B (x42)
Reclength : 19 (x013) IO Time : 2020/03/06 17:26:36.000.000
IOType : 15 (x0f) OrigNode : 255 (xff)
TransInd : . (x00) FormatType : R (x52)
SyskeyLen : 0 (x00) Incomplete : . (x00)
AuditRBA : 3539 AuditPos : 31055376
Continued : N (x00) RecCount : 1 (x01)

2020/03/06 17:26:36.000.000 FieldComp Len 19 RBA 2207
Name: HR.TABLE_NAME (TDR Index: 1)
Before Image: Partition 12 G b
 0000 0004 ffff 0000 0004 0007 0000 0003 5359 53 | ....SYS
Column 0 (x0000), Len 4 (x0004)
ffff 0000 | ....
Column 4 (x0004), Len 7 (x0007)
 0000 0003 5359 53 | ....SYS
Logdump 20 >n

Hdr-Ind : E (x45) Partition : . (x0c)
UndoFlag : . (x00) BeforeAfter: A (x41)
Reclength : 23 (x017) IO Time : 2020/03/06 17:26:36.000.000
IOType : 15 (x0f) OrigNode : 255 (xff)
TransInd : . (x02) FormatType : R (x52)
SyskeyLen : 0 (x00) Incomplete : . (x00)
AuditRBA : 3539 AuditPos : 31055376
Continued : N (x00) RecCount : 1 (x01)

2020/03/06 17:26:36.000.000 FieldComp Len 23 RBA 2334
Name: HR.TABLE_NAME (TDR Index: 1)
After Image: Partition 12 G e
 0000 0004 ffff 0000 0004 000b 0000 0007 6161 6161 | ....aaaa
6161 61 | aaa
Column 0 (x0000), Len 4 (x0004)
ffff 0000 | ....
Column 4 (x0004), Len 11 (x000b)
 0000 0007 6161 6161 6161 61 | ....aaaaaaaa
```

可以看到，对于数据的更新，trail 日志中有两条记录，分别为更新前的记录，和更新后的记录，同时，通过日志可以看出，对于更新的数据，在更新前和更新后的记录中，都只有主键列和更新列的值，没有其他列的值，这个正是由于没有开启表的附加日志导致的。

前面介绍 replicat 中的参数 “handlecollisions” 用于处理数据冲突和记录不存在的情况。若更新时，目标表中不存在该记录，则对应的日志会转化成 insert 语句，这样就会导致其他字段为空插入到目标库，造成数据不一致，若目标库相关字段存在非空约束，将直接导致进程异常。

## 2. 开启表的附加日志

### (1) OGG 账户登录

```
GGSCI (1-db-ps-005) 3> dblogin userid goldengate, password goldengate
Successfully logged into database.
```

### (2) 开启表的附加日志

```
GGSCI (1-db-ps-005 as goldengate@ogg) 4> add trandata hr.table_name,nokey
2020-03-06 21:13:51 WARNING OGG-01387 Table HR.TABLE_NAME has no valid key columns, added unconditional supplemental log group for all table columns.

Logging of supplemental redo data enabled for table HR.TABLE_NAME.
TRANSACTION for scheduling columns has been added on table 'HR.TABLE_NAME'.
TRANSACTION for instantiation CSN has been added on table 'HR.TABLE_NAME'.
```

### (3) 查看表的附加日志是否开启

```
GGSCI (1-db-ps-005 as goldengate@ogg) 7> info trandata hr.table_name
Logging of supplemental redo log data is enabled for table HR.TABLE_NAME.
Columns supplementally logged for table HR.TABLE_NAME: ALL.
Prepared CSN for table HR.TABLE_NAME: 57187843
```

通过上面的命令，已经开启表的附加日志，这样表的所有字段的附加日志都会进行记录，我们可以对更新语句再次做相关的操作，并解析 ogg 日志。

### (4) 再次执行更新日志

```
SQL> update hr.table_name set updated_by='bbb';
1 row updated.

SQL> commit;
Commit complete.
```

### (5) 查看 ogg 日志

```
Logdump 23 >n

Hdr-Ind : E (x45) Partition : . (x0c)
UndoFlag : . (x00) BeforeAfter: B (x42)
RecLength : 85 (x055) IO Time : 2020/03/06 21:22:21.000.000
IOType : 15 (x0f) OrigNode : 255 (xff)
TransInd : . (x00) FormatType : R (x52)
SyskeyLen : 0 (x00) Incomplete : . (x00)
AuditRBA : 3539 AuditPos : 45237264
Continued : N (x00) RecCount : 1 (x01)

2020/03/06 21:22:21.000.000 FieldComp Len 85 RBA 2437
Name: HR.TABLE_NAME (TDR Index: 1)
Before Image: Partition 12 G b
 0000 0005 0000 0001 3100 0100 1500 0032 3032 302d | .....1.....2020-
 3033 2d30 363a 3137 3a32 363a 3238 0002 0007 0000 | 03-06:17:26:28.....
 0003 5359 5300 0300 1500 0032 3032 302d 3033 2d30 | ..SYS.....2020-03-0
 363a 3137 3a32 363a 3238 0004 000b 0000 0007 6161 | 6:17:26:28.....aa
 6161 6161 61 | aaaaaa
Column 0 (x0000), Len 5 (x0005)
 0000 0001 31 | ....1
Column 1 (x0001), Len 21 (x0015)
 0000 3230 3230 2d30 332d 3036 3a31 373a 3236 3a32 | ..2020-03-06:17:26:2
 38 | 8
Column 2 (x0002), Len 7 (x0007)
 0000 0003 5359 53 | ....SYS
Column 3 (x0003), Len 21 (x0015)
 0000 3230 3230 2d30 332d 3036 3a31 373a 3236 3a32 | ..2020-03-06:17:26:2
 38 | 8
Column 4 (x0004), Len 11 (x000b)
 0000 0007 6161 6161 6161 61 | ....aaaaaaaa
Logdump 24 >n

Hdr-Ind : E (x45) Partition : . (x0c)
UndoFlag : . (x00) BeforeAfter: A (x41)
RecLength : 81 (x051) IO Time : 2020/03/06 21:22:21.000.000
IOType : 15 (x0f) OrigNode : 255 (xff)
TransInd : . (x02) FormatType : R (x52)
SyskeyLen : 0 (x00) Incomplete : . (x00)
AuditRBA : 3539 AuditPos : 45237264
Continued : N (x00) RecCount : 1 (x01)

2020/03/06 21:22:21.000.000 FieldComp Len 81 RBA 2630
Name: HR.TABLE_NAME (TDR Index: 1)
After Image: Partition 12 G e
 0000 0005 0000 0001 3100 0100 1500 0032 3032 302d | .....1.....2020-
 3033 2d30 363a 3137 3a32 363a 3238 0002 0007 0000 | 03-06:17:26:28.....
 0003 5359 5300 0300 1500 0032 3032 302d 3033 2d30 | ..SYS.....2020-03-0
 363a 3137 3a32 363a 3238 0004 0007 0000 0003 6262 | 6:17:26:28.....bb
 62 | b
Column 0 (x0000), Len 5 (x0005)
 0000 0001 31 | ....1
Column 1 (x0001), Len 21 (x0015)
 0000 3230 3230 2d30 332d 3036 3a31 373a 3236 3a32 | ..2020-03-06:17:26:2
 38 | 8
Column 2 (x0002), Len 7 (x0007)
 0000 0003 5359 53 | ....SYS
Column 3 (x0003), Len 21 (x0015)
 0000 3230 3230 2d30 332d 3036 3a31 373a 3236 3a32 | ..2020-03-06:17:26:2
 38 | 8
Column 4 (x0004), Len 7 (x0007)
 0000 0003 6262 62 | ....bbb
```

可以看到，开启表的附加日志后，对于更新语句而言，无论是更新前的记录，还是更新后的记录，都包含了全字段，在日志中，任何一个操作都包含被更新表的全字段记录。

## 5.2.5 注意事项

## 1. 大事务

```
MAXTRANSOPS 100
```

TiDB 分布式数据库存在大事务的限制，上述 OGG 复制进程参数，拆分事务为每个事物最大 100 条

```
SQLEXEC "set tidb_constraint_check_in_place = 1"
```

OGG 复制进程参数，启动进程后在 TiDB 分布式数据库中执行 set tidb\_constraint\_check\_in\_place = 1，需要设置该参数将主键冲突检测提前，否则遇到主键冲突复制进程会直接 abend。

## 2. 字符集

在源端为 GBK 字符集时，注意查看复制进程的 report 日志，其中有进行字符集转换相关的信息，以及确保中文字符能正常显示。

```
2019-07-28 21:01:10 INFO     OGG-02243  Opened trail file /home/tidb/ogg12.3/dirdat/p1000000002 at 2019-07-28 21:01:10.232896.

2019-07-28 21:01:10 INFO     OGG-03506  The source database character set, as determined from the trail file, is zhs16gbk.

2019-07-28 21:05:57 INFO     OGG-06505  MAP resolved (entry scott.account): MAP "SCOTT"."ACCOUNT", TARGET "scott"."account".

2019-07-28 21:05:57 INFO     OGG-02756  The definition for table SCOTT.ACOUNT is obtained from the trail file.

2019-07-28 21:05:57 INFO     OGG-06511  Using following columns in default map by name: account_number, account_balance, account_trans_ts, account_trans_type.

2019-07-28 21:05:57 INFO     OGG-06510  Using the following key columns for target table scott.account: account_number.

2019-07-28 21:05:57 INFO     OGG-03010  Performing implicit conversion of column data from character set zhs16gbk to UTF-8.
```

## 3. 版本兼容

请使用与本文相同的 OGG 版本，新版本的 OGG 存在无法通过密码连接 TiDB 分布式数据库，无法进行字符集转换的问题。

## 4. 表名大小写

在 TiDB 分布式数据库中建表使用小写，同时 OGG replicat 进程参数中 map 对应的也为小写。

## 5. 对大表并行 replicat

初始化过程中对大表可以通过 OGG RANGE 方法，通过将多个分区映射给不同的 replicat 进程，达到并行复制提升效率。

例：

```
filter(@RANGE ( 分片 , 总分片数量 , 分片键));
```

以下为两个分片时配置文件，其中分片键在有主键或唯一键时可不指定

```

EXTRACT initext1

userid oggadmin ,password oggadmin

RMTHOST target-ogg, MGRPORT 7909

RMTTASK replicat, GROUP initrep1

TABLE scott.account, filter(@RANGE(1,2,account)) ;

EXTRACT inittext2

userid oggadmin ,password oggadmin

RMTHOST target-ogg, MGRPORT 7909

RMTTASK replicat, GROUP initrep2

TABLE scott.account, filter(@RANGE(2,2,account)) ;

```

## 5.2.6 常见问题处理

### 1. OGG-01201 Error reported by MGR : Access denied

问题原因：

This is due to a new security restriction in GoldenGate 12.2. In order to allow access from a remote system the ACCESSRULE parameter must be put into the manager parameter file on the target in order to allow access from the source.

处理方法：需要在源端 mgr 增加允许访问目标端 ip 的规则

```

(1) 编辑 mgr 参数文件

edit params mgr

(2) 增加以下内容：

ACCESSRULE, PROG *, IPADDR *, ALLOW

```

### 2. replicat 端报 invalid time format

问题原因：

之前把 Oracle 的 date 类型改为了 TiDB timestamp 类型，但是 Oracle 里面部分时间类型数据超过了 TiDB timestamp 的范围。（TiDB 范围为 1970-01-01 00:00:01.000000 到 2038-01-19 03:14:07.999999，Oracle 部分时间数据超过 2038 年，应该是原先遗留的测试数据）。

处理方法：

将 TiDB 的 timestamp 改为支持更大的 datetime 类型；同时在 OGG replicat 端增加异常数据不 abended 而是记录 discard 的方式，具体参数为：REPERROR (default,discard)，防止 OGG 因为异常数据终止。

### 3. extract 报 ORA-01801: date format is too long for internal buffer

问题原因：Oracle 端有时间类型数据通过 OCI 接口入库时，Oracle 不做时间校验，但是查询的时候就会校验时间格式，正常 Oracle select \* 都会报错，属于 Oracle 端数据问题。

处理方法：

用 to\_char (date,'yyyy-mm-dd hh24:mi:ss') 处理错误日期后，Oracle 可以正常查出来，但是显示的时间为 '0000-00-00 00:00:00' 的样式，可以通过类似以下的语句查出错误数据，在源端处理掉错误数据：

(1) 备份错误数据

```
create table t_bak as select * from t where to_char(date1,'yyyy-mm-dd hh24:mi:ss')= '0000-00-00 00:00:00';
```

(2) 删除原表错误数据:

```
delete from t where id in (select id from t_bak);  
commit;
```

## 5.3 SQL Server 迁移到 TiDB

### 5.3.1 迁移背景

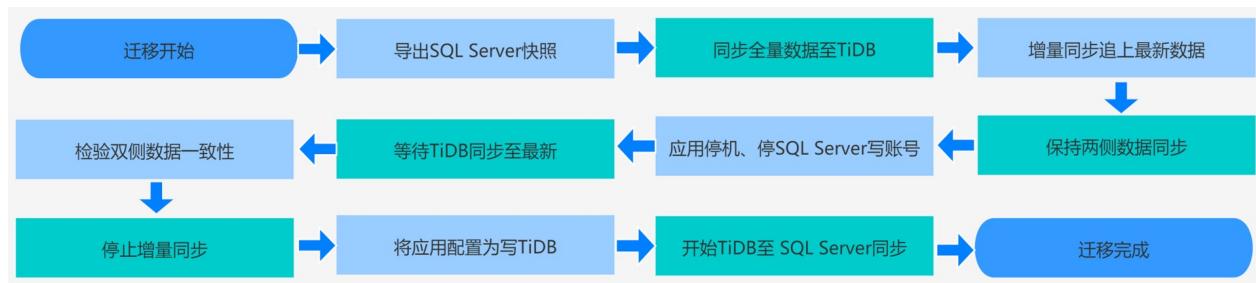
随着业务不断地增长，数据不断地递增，当线上场景的 SQL server 主库有十亿级数据，每天有千万级 DAU、亿级的访问量，接口日均调用量 1000000000+ 次，很可能会触达 SQL server 集群的一些瓶颈。以至于用户不得不寻找一个新的数据库产品替换现有的架构，那么可能会遇到一下的问题：

- 1、数据采用分库分表的设计，来解决当 SQL Server 单表数据量过大，出现的性能下降等问题。假设目前业务场景有 100 多个库包含 1000 多张表，随着应用的不断发展，需求的不断变化，发现在实现某些业务场景中，需要关联很多张表才能满足需求。那么分库分表的结构就难以满足，所以需要数据逻辑最好能在一张表上实现。
- 2、近些年来，随着业务加速成长，数据量突飞猛进。而硬盘容量是有限的，每台服务器上能扩展的硬盘数量也是有限的。一方面每隔一段时间都要增加更大容量的存储服务器来应对数据增长，每次更换服务器的时候需要关注 SQL Server 扩容规划。另外一方面大容量数据库服务器价格昂贵，同时需要做到扩容对应用无感知。

至此本节将介绍如何实现从 SQL Server 到 TiDB 的整体迁移流程，包括全量与增量同步，数据校验及注意事项等内容，希望本节可以帮助想要将 SQL Server 迁移到 TiDB 的用户提供上手操作指导。

### 5.3.2 流程介绍

下图为在线迁移数据库的基本流程，供读者对整体流程有个大致的认识：



执行过程中，可根据实际情况，结合如下表格制定迁移操作手册：

迁移阶段	流程
全量同步	1. 开启 SQL Server 从库的 CDC； 2. 导出 SQL Server 快照到备份服务器； 3. 从备份服务器将存量数据 ETL 至 TiDB； 4. 检查两侧数据条数是否相同
增量同步	1. 读取CDC记录，持续写入 TiDB； 2. 检查增量数据一致性
切换数据库	1. 停止 SQL Server 写入； 2. 等待 CDC 消费完成，检查数据一致性； 3. 业务数据写入 TiDB

### 5.3.3 全量同步

#### 1. 迁移工具

迁移工具采用 yugong。yugong 是阿里开发的一款去 Oracle 数据迁移同步工具，<https://github.com/alswl/yugong> 版本基于阿里的版本增加了对 SQL Server 数据源的支持。

#### 2. 操作步骤

##### (1) 下载 yugong

```
# 方法一：直接下载可执行文件 jar 包  
wget https://github.com/alswl/yugong/releases/download/541e5f8/yugong-shaded.jar  
  
# 方法二：自行编译  
git git@github.com:alswl/yugong.git  
cd yugong  
mvn clean package
```

## (2) 修改配置

运行 yugong 需要用到两个配置文件 `yugong.properties` 和 `yugong.yaml`。

- `yugong.properties` 文件用于配置数据库链接和同步任务

```
# vi 编辑 yugong.properties  
vi yugong.properties
```

```

# 源数据库配置
yugong.database.source.url=jdbc:sqlserver://192.168.1.88:1433;databaseName=example
yugong.database.source.username=sa
yugong.database.source.password=sa
yugong.database.source.type=SQL_SERVER
yugong.database.source.encode=UTF-8
yugong.database.source.poolSize=200

# 目标数据库配置, TiDB 使用 MySQL 协议
yugong.database.target.url=jdbc:mysql://192.168.1.99:3306/example?autoReconnect=true
yugong.database.target.username=root
yugong.database.target.password=root
yugong.database.target.type=MYSQL
yugong.database.target.encode=UTF-8
yugong.database.target.poolSize=200
yugong.table.batchApply=true
yugong.table.onceCrawNum=1000
yugong.table.tpsLimit=0
# 使用数据库链接中的 schema
yugong.table.ignoreSchema=false
# Skip Applier Load Db failed data
yugong.table.skipApplierException=false
# 需要迁移的数据库表
yugong.table.white=user
yugong.table.black=
# 是否开启并发
yugong.table.concurrent.enable=true
# 并发数
yugong.table.concurrent.size=6
# retry times
yugong.table.retry.times=3
# retry interval or sleep time (ms)
yugong.table.retry.interval=1000
# MARK/FULL/INC/ALL(REC+FULL+INC)/CHECK/CLEAR
# 设置为FULL (全量模式)
yugong.table.mode=FULL

# yugong extractor
yugong.extractor.dump=false
yugong.extractor.concurrent.enable=true
yugong.extractor.concurrent.global=false
yugong.extractor.concurrent.size=60
yugong.extractor.noupdate.sleep=1000
yugong.extractor.noupdate.thresold=0
yugong.extractor.once=false

# yugong applier
yugong.applier.concurrent.enable=true
yugong.applier.concurrent.global=false
yugong.applier.concurrent.size=20
yugong.applier.dump=false
# stats
yugong.stat.print.interval=5
yugong.progress.print.interval=1
# alarm email
yugong.alarm.email.host=
yugong.alarm.email.username=
yugong.alarm.email.password=
yugong.alarm.email.smtp.port=
yugong.alarm.email.ssl.support=

```

- `yugong.yaml` 文件用于定制 translator 的定制，直接使用范例中的配置即可

```

# 下载 yugong.yaml.sample 模版文件
wget https://raw.githubusercontent.com/alswl/yugong/feature/sql-server-to-mysql-overview/yugong.yaml.sample

# 重命名为 yugong.yaml
mv yugong.yaml.sample yugong.yaml

```

### (3) 运行 yugong

- 执行 yugong 的运行命令

```
# -c yugong.properties 数据库和任务配置文件
# -y yugong.yaml ETL 中 translator 流程的配置
# 执行命令
java -jar yugong-shaded.jar -c yugong.properties -y yugong.yaml
```

- 程序运行后，控制台会输出运行日志。

```
# 程序启动
2019-12-02 20:49:23.923 [main] INFO com.taobao.yugong.YugongApp - ## start the YuGong.
# 检查源数据库和目标数据库连接情况
2019-12-02 20:49:24.000 [main] INFO com.taobao.yugong.controller.YuGongController - check source database connection ...
2019-12-02 20:49:24.017 [main] INFO com.taobao.yugong.controller.YuGongController - check source database is ok
2019-12-02 20:49:24.017 [main] INFO com.taobao.yugong.controller.YuGongController - check target database connection ...
...
2019-12-02 20:49:24.028 [main] INFO com.taobao.yugong.controller.YuGongController - check target database is ok
2019-12-02 20:49:24.028 [main] INFO com.taobao.yugong.controller.YuGongController - check source tables read privileges ...
# 初始化连接池
2019-12-02 20:49:24.071 [main] INFO com.alibaba.druid.pool.DruidDataSource - {dataSource-1} initied
2019-12-02 20:49:24.277 [main] INFO com.alibaba.druid.pool.DruidDataSource - {dataSource-2} initied
2019-12-02 20:49:25.418 [main] INFO com.taobao.yugong.controller.YuGongController - check source tables is ok.
# 启动同步任务线程
2019-12-02 20:49:26.933 [main] INFO com.taobao.yugong.controller.YuGongController - ## prepare start tables[1] with concurrent[6]
2019-12-02 20:49:26.959 [main] INFO com.taobao.yugong.YugongApp - ## the YuGong is running now .....
# 同步进度打印
2019-12-02 20:51:08.801 [Thread-7] INFO com.taobao.yugong.common.stats.ProgressTracer - {未启动:0, 全量中:0, 已完成:1, 异常数:0}
# 完成的任务
2019-12-02 20:51:08.801 [Thread-7] INFO com.taobao.yugong.common.stats.ProgressTracer - 已完成:[example.user]
2019-12-02 20:51:08.805 [Thread-7] INFO com.alibaba.druid.pool.DruidDataSource - {dataSource-2} closed
2019-12-02 20:51:08.806 [Thread-7] INFO com.alibaba.druid.pool.DruidDataSource - {dataSource-1} closed
2019-12-02 20:51:08.807 [Thread-7] INFO com.taobao.yugong.YugongApp - ## YuGong is down.
```

#### 注意

通过以上日志可以判定任务是否成功启动和结束，如运行当中出现错误会有详细的堆栈信息打印出来，可进一步检查日志来判定问题。

### (4) 检查日志

- 日志记录在 logs 目录下，结构如下：

```
logs
├── example.user # 表名为目录
│   ├── applier.log # 写入日志
│   ├── extractor.log # 抽取数据日志
│   └── table.log # 表操作日志
└── yugong
    └── table.log # 系统日志
positioner_data
└── example_user.dat # 同步进度, 用于断点续传
```

#### 注意

主要观察 table.log 中的运行日志，查看是否有 ERROR 如重新同步数据，需删除相关表的日志和同步进度文件。

## 5.3.4 增量同步

## 1. 增量同步的原理

增量同步需要开启 SQL Server 的 CDC。什么是 SQL Server CDC？CDC 全称 Change Data Capture，设计目的就是用来解决增量数据。CDC 持续读取增量记录，将增量数据发送至消息队列中，以供消费程序解析并写入 TiDB。当数据库表发生变化，Capture process 会从 transaction log 里面获取数据变化，然后将这些数据记录到 Change Table 里面。可以通过特定的 CDC 查询函数将这些变化数据查出来。

## 2. 实操流程

### (1) 开启 SQL Server CDC

```
-- 开启CDC
sys.sp_cdc_enable_db;

-- 开启 example 表的CDC功能
EXEC sys.sp_cdc_enable_table @source_schema = N'dbo', @source_name = N'example', @role_name = NULL;

-- 使用 SQL 查看哪些表开启了 CDC 功能。
-- 通过系统表查询开启CDC的表
SELECT name, is_cdc_enabled FROM sys.databases WHERE is_cdc_enabled = 1;
```

### (2) CDC 开启后，系统会生成一张 `Change Table` 的表，表名为：`cdc.dbo_example_CT`

```
.schema cdc.dbo_example_CT
name          default  nullable   type        length  indexed
-----  -----
__$end_lsn    null     YES        binary      10      NO
__$operation  null     NO         int         4       NO
__$seqval     null     NO        binary      10      NO
__$start_lsn  null     NO        binary      10      YES
__$update_mask null    YES       varbinary   128     NO
id            null    YES        int         4       NO
name          null    YES       varchar(255) 255     NO
```

#### 注意

其中 开头的为系统字段，`id` 和 `name` 为 `example` 表中原始字段。

### (3) 读取 CDC 日志

- 对 `example` 表做一些添删改的操作，而后通过系统函数查询 CDC 记录。

```
-- 定义参数
DECLARE @begin_time datetime, @end_time datetime, @begin_lsn binary(10), @end_lsn binary(10);

-- 设定查询记录的开始和结束时间
SET @begin_time = '2020-03-08 10:00:00.000';
SET @end_time   = '2020-03-08 10:10:00.000';

-- 将时间转换为系统的 lsn
SELECT @begin_lsn = sys.fn_cdc_map_time_to_lsn('smallest greater than', @begin_time);
SELECT @end_lsn = sys.fn_cdc_map_time_to_lsn('largest less than or equal', @end_time);

-- 根据开始和结束的 lsn 查询该表的所有变化
SELECT * FROM cdc.fn_cdc_get_all_changes_dbo_example(@begin_lsn, @end_lsn, 'all');
```

- 查出数据如下：

<code>__\$start_lsn</code>	<code>__\$end_lsn</code>	<code>__\$seqval</code>	<code>__\$operation</code>	<code>__\$update_mask</code>	<code>id</code>	<code>name</code>
0000dede0000019f001a	null	0000dede0000019f0018	2	03	1	AAA
0000dede000001ad0004	null	0000dede000001ad0003	2	03	2	BBB
0000dede000001ba0003	null	0000dede000001ba0002	3	02	2	BBB
0000dede000001ba0003	null	0000dede000001ba0002	4	02	2	CCC
0000dede000001c10003	null	0000dede000001c10002	2	03	3	DDD
0000dede000001cc0005	null	0000dede000001cc0002	1	03	3	DDD

### 注意

`__$operation` 字段代表当前记录所执行的操作。

- 删除
- 插入
- 更新前
- 更新后

### (4) 消费 CDC 日志

确认 CDC 正确开启，且得到了变更数据后，编写一个程序，采用定时任务的方式，设定时间间隔，通过上述 SQL 语句持续读取 Change Table 中的记录，并根据 `__$operation` 将数据转换为对应添删改操作的 SQL 语句写入 TiDB。

这里有几点需要考量：

1. 数据规模，如果变更量较大，那么下游数据库的写入速度会产生瓶颈，则考虑使用 Kafka、RabbitMQ 等消息队列作为缓冲。
2. 使用 Kafka 等消息队列时，考虑消息顺序的严格性，将 Topic 的 Partition 设置为 1。
3. 下游数据库更新异常时的重试机制。
4. 因为全量同步之前 CDC 已经开启，所以增量同步时，可能会存在插入数据时出现主键冲突导致程序异常，推荐使用 `insert ignore` 的方式编写 SQL。

### 5.3.5 检查数据一致性

在 ETL 之后，需要有一个流程来确认数据迁移前后是否一致。虽然理论上不会有差异，但是如果中间有程序异常，或者数据库在迁移过程中发生操作，数据就会不一致。

选择 `yugong` 作为 ETL 工具的一大原因也是因为它提供了多种模式。支持 `CHECK`、`FULL`、`INC`、`AUTO` 四种模式。其中 `CHECK` 模式就是将 `yugong` 作为数据一致性检查工具使用。`yugong` 工作原理是通过 `JDBC` 根据主键范围变化，将数据取出进行批量对比。

有一点需要注意，当表没有主键信息时，`yugong` 默认会使用 SQL Server 的物理地址信息——`physloc`。此时读取性能会大幅下降，所以大表通常建议先建好主键。

在增量同步时，还有另一个方式来实现数据校验功能，再增加一个消费程序，延迟 5 秒消费同一队列，并通过提取主键（或索引）的方式从 TiDB 中查出该条已经写入的数据，将两侧的整行数据做比较（本实践中去除主键后比较），如果有问题会进行尝试重新写入，如出现异常则向相关人员发送报警。

### 5.3.6 迁移注意事项

在迁移过程中需要注意的一些点：

1. 原有 SQL server 语句一定要在程序里检查仔细，尤其是对于一些时间久远且项目比较多的程序，最好是让 DBA 同学在 SQL server 服务请求上开启 DMV，多抓几天的语句，然后根据这些语句再去核对。
2. SQL Server 与 TiDB（MySQL）的字段类型略有出入，需要仔细对比。
3. SQL Server 与 TiDB（MySQL）的语法也不尽相同，在实现之后需要仔细验证，以防出现语义变化，影响业务。

4. 索引机制不同，可以利用迁移的机会重新梳理业务索引。
5. 对于大型系统迁移，演练不可缺少。演练成功标准为单次演练无意外，时间控制在计划内。
6. 通常为了保证数据的顺序性，增量同步时只能有一个程序在消费。所以要多考虑程序健壮性、日志的完善程度以及报警机制，方便技术人员监控和追溯问题。

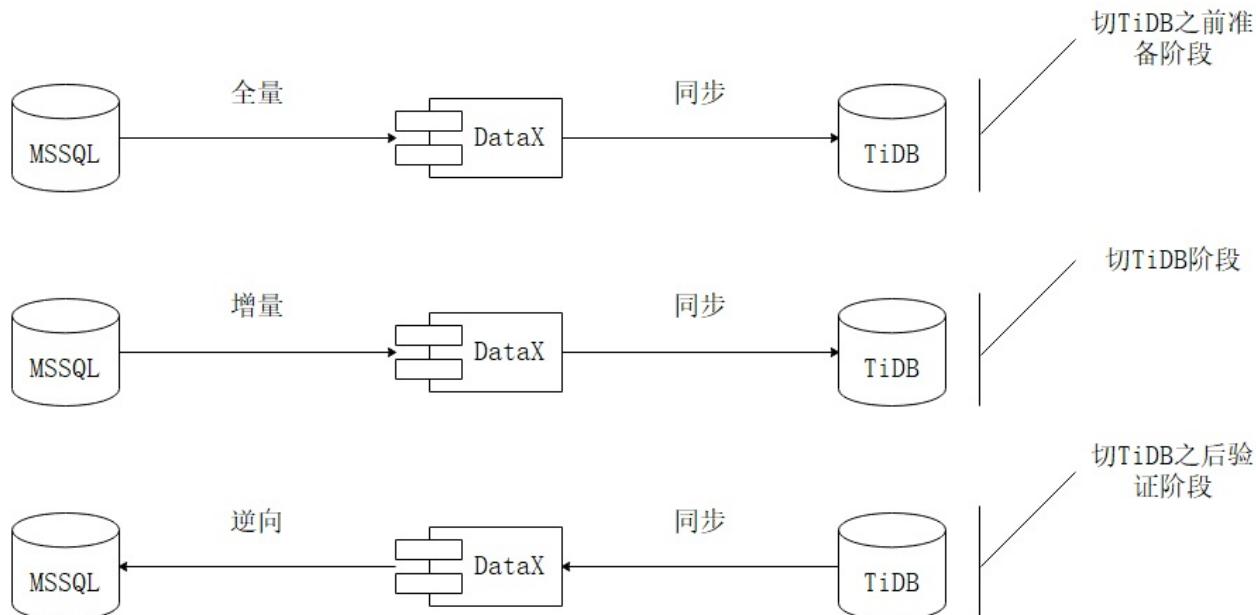
### 5.3.7 总结

这套迁移流程，综合考虑了开发成本、迁移效率和数据一致性而设计。TiDB 兼容 MySQL 的特性，使得在迁移过程中有许多“轮子”可以拿起来就用，让开发人员可以将更多关注放在数据本身。在下一章节中我们还会介绍另外一种 SQL Server 迁移 to TiDB 的方案，相信大家会有新的思考和体会。

DataX 是阿里巴巴集团内被广泛使用的离线数据同步工具/平台，支持包括 MySQL、SQL Server、Oracle、PostgreSQL、HDFS、Hive、HBase、OTS、ODPS 等各种异构数据源之间高效的数据同步功能。DataX 采用了框架 + 插件 的模式。目前已开源，代码托管在 GitHub。

DataX 数据同步效率较高，满足大多数场景下的异构数据库间的数据同步需求。同时其配置灵活，支持字段级别的配置，可以轻松应对因异构数据库迁移到 TiDB 而产生的一些改动。

方案设计如图所示



第一阶段：切换支持 TiDB 的应用上线之前，把 SQL Server 数据库中的全量数据用 DataX 同步到 TiDB 库中。为避免对线上业务产生影响，可以选择备份库，或者在业务低高峰期操作。

第二阶段：把全量同步改为增量同步，利用 UpdateTime 字段（或其它条件，根据实际业务灵活调整）作为同步 Where 条件，进行增量覆盖式同步。t\_sync\_record 表中记录每张表上次增量任务的执行时间。

第三阶段：支持 TiDB 的应用上线以后，增量同步切换读写数据源改为逆向增量同步，将新数据近实时地写回 SQL Server 数据库。一旦上线之后出现需要回退的情况，可随时切回 SQL Server，待修复之后再次上线。

具体操作步骤：

第一步：部署 DataX

下载

```
wget http://datax.opensource.oss-cn-hangzhou.aliyuncs.com/datax.tar.gz
```

解压

```
tar -zvxf datax.tar.gz
```

自检

```
python {YOUR_DATAx_HOME}/bin/datax.py {YOUR_DATAx_HOME}/job/job.json
```

第二步：编写 DataX 数据同步 Job (Json格式)

(1) 全量同步Job

vi full.json

```
{
  "job": {
    "setting": {
      "speed": {
        #数据分片，分片数据可同时进行同步
        "channel": 128
      }
    },
    "content": [
      {
        #SQL Server配置
        "reader": {
          "name": "sqlserverreader",
          "parameter": {
            #数据库用户名
            "username": "${srcUserName}",
            #数据库密码
            "password": "${srcPassword}",
            #需要迁移的列名， * 表示全部列
            "column": ["*"]
          },
          "connection": [
            {
              #需要迁移的表名
              "table": ["${tableName}"],
              #数据库 jdbc 连接
              "jdbcUrl": ["${srcUrl}"]
            }
          ]
        }
      },
      #TiDB配置
      "writer": {
        "name": "tidbwriter",
        "parameter": {
          #数据库用户名
          "username": "${desUserName}",
          #数据库密码
          "password": "${desPassword}",
          #使用 Replace 语法
        }
      }
    ]
  }
}
```

```

        "writeMode": "replace",

        #目标表列名, * 表示全部列

        "column": ["*"],

        "connection": [{

            #数据库 jdbc 连接

            "jdbcUrl": "${desUrl}",

            #目标表名

            "table": ["${tableName}"]

        }],


        #本次迁移开始前执行的sql-作数据迁移日志使用

        "preSql": [

            "replace into t_sync_record(table_name,start_date,end_date) values('@table',now(),null)",

            #本次迁移完成后执行的 sql- 作数据迁移日志使用

            "postSql": [

                "update t_sync_record set end_date=now() where table_name='@table' " ]

        }

    }

}

]
}

```

## (2) 增量同步 Job

vi increase.json

```

{
    "job": {

        "setting": {

            "speed": {

                #数据分片, 分片数据可同时进行同步

                "channel": 128

            }

        },

        "content": [{

            #SQL Server配置

            "reader": {

                "name": "sqlserverreader",

                "parameter": {


```

```

#数据库用户名

"username": "${srcUserName}",

#数据库密码

"password": "${srcPassword}",

#需要迁移的列名, * 表示全部列

"column": ["*"],

"connection": [{

    #需要迁移的表名

    "table": ["${tableName}"],

    #数据库 jdbc 连接

    "jdbcUrl": ["${srcUrl}"]

}],

#抓取一个时间窗口的增量数据

"where": "updateTime >= '${syncTime}'"

}

},

#TiDB配置

"writer": {

    "name": "tidbwriter",

    "parameter": {

        #数据库用户名

        "username": "${desUserName}",

        #数据库密码

        "password": "${desPassword}",

        #使用 Replace 语法

        "writeMode": "replace",

        #目标表列名, * 表示全部列

        "column": ["*"],

        "connection": [{

            #数据库 jdbc 连接

            "jdbcUrl": "${desUrl}",

            #目标表名

            "table": ["${tableName}"]

        }],


        #本次迁移开始前执行的sql-作数据迁移日志使用

        "preSql": [

```

```
        "replace into t_sync_record(table_name,start_date,end_date) values('@table',now(),null")],  
        #本次迁移完成后执行的 sql- 作数据迁移日志使用  
        "postSql": [  
            "update t_sync_record set end_date=now() where table_name='@table' " ]  
        }  
    }]  
}
```

(3) 编写运行DataX Job的Shell执行脚本

```
vi datax_execute_job.sh
```

```

#!/bin/bash

source /etc/profile

srcUrl="Reader Sql Server 地址"

srcUserName="Sql Server 账号"

srcPassword="Sql Server 密码"

desUrl="Writer TiDB 地址"

desUserName="TiDB 账号"

desPassword="TiDB 密码"

# 同步开始时间

defaultSyncUpdateTime="2020-03-03 18:00:00.000"

# 同步周期(秒)

sleepTime="N"

tableName="Table1,Table2, . . ."

# 循环次数标识, -1为一直循环, 其他按输入次数循环

flg=-1

while [ "$flg" -gt 0 -o "$flg" -eq -1 ]

do

    #更新时间设置为上次循序执行的时间

    if [ "" = "$preRunTime" ]; then

        syncTime=$defaultSyncUpdateTime

    else

        syncTime=$preRunTime

    fi

    #记录下本次循环执行时间, 供下次循环使用

    preRunTime=$(date -d +"%Y-%m-%d %T").000";

    echo $syncTime

    echo $preRunTime

    echo $flg

    python {YOUR_DATAx_HOME}/bin/datax.py -p "-DsyncTime='${syncTime}' -DtableName='${tableName}' -DsrcUrl='${srcUrl}' -DsrcUserName='${srcUserName}' -DsrcPassword='${srcPassword}' -DdesUrl='${desUrl}' -DdesUserName='${desUserName}' -DdesPassword='${desPassword}'" {YOUR_DATAx_HOME}/job/increase.json

    if [ -1 -lt "$flg" ]; then

        let "flg-=1"

    fi

    sleep $sleepTime

done

```

#### (4) 执行 Shell 脚本

```
chmod +x datax_excute_job.sh  
nohup ./datax_excute_job.sh > info.file 2>&1 &
```

至此，核心脚本和操作都已完成，可通过修改参数配置，达到自己不同的需求，还可以配合数据对比服务，以期达到将应用程序从 SQL Server 顺利迁移到 TiDB 的目的。

## Db2 到 TiDB (CDC)

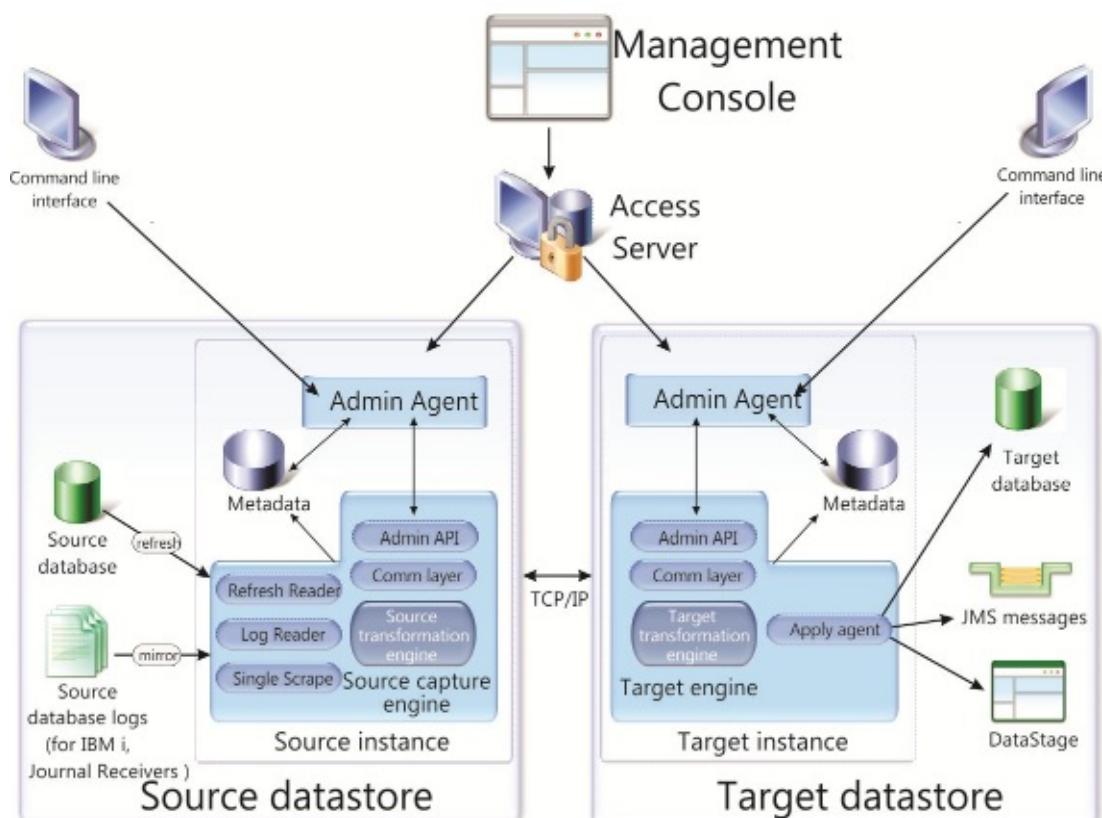
在数据同步场景中，上游数据库可能是任何关系型数据库，异构数据库之间的同步目前还没有一款通用的工具能够很好的适配所有的关系型数据库。异构数据库之间的同步也有比较多的数据同步工具。传统数据库市场中，Db2 有一定的占有率，当这些用户在考虑数据迁移到 TiDB 上时如何选择合适的工具是本章的主要内容。Db2 根据运行平台的不同被分为 Db2 for LUW、Db2 for i (AS/400)、Db2 for z/OS (Mainframe)。当数据源是 Db2 for LUW 时，可以使用 IBM CDC 或 OGG 做同步工具；当数据源是 Db2 for i 或 Db2 for z/OS 时，只能用 IBM CDC 做同步工具。本章节描述的是 Db2 for i 作为数据源时，使用 CDC 同步数据到 TiDB 的案例，可供所有计划使用 IBM CDC 工具同步数据到 TiDB 的用户参考。

IBM 公司为 Db2 的数据同步做了一套完整的工具，初期这款工具被命名为 InfoSphere Data Replication 简称 IIDR，IIDR 有多个引擎，包括 Change data capture (CDC) Replication、Q Replication 和 SQL Replication，在 Db2 for i 同步数据到 TiDB 的场景中，主要用到的是它的 Change data capture (CDC) Replication 引擎，所以文中前面提到的 IBM CDC 等价于 IIDR。

## CDC 简介

- IIDR-CDC 是一种数据库复制解决方案，它捕获源数据库发生的更改，并将其交付给目标数据库或消息队列中。它的表映射关系的配置都是由图化的控制管理平台上完成的。CDC 可以用在对源系统几乎没有任何影响的前提下捕捉数据变更并快速应用到下游。其架构特点如下

其软件架构如下图所示：



- CDC 官方支持列表 参

考：[https://www.ibm.com/support/knowledgecenter/SSTRGZ\\_11.4.0/com.ibm.cdc.doc.sysreq.doc/concepts/supportedsourceandtargets.html](https://www.ibm.com/support/knowledgecenter/SSTRGZ_11.4.0/com.ibm.cdc.doc.sysreq.doc/concepts/supportedsourceandtargets.html)

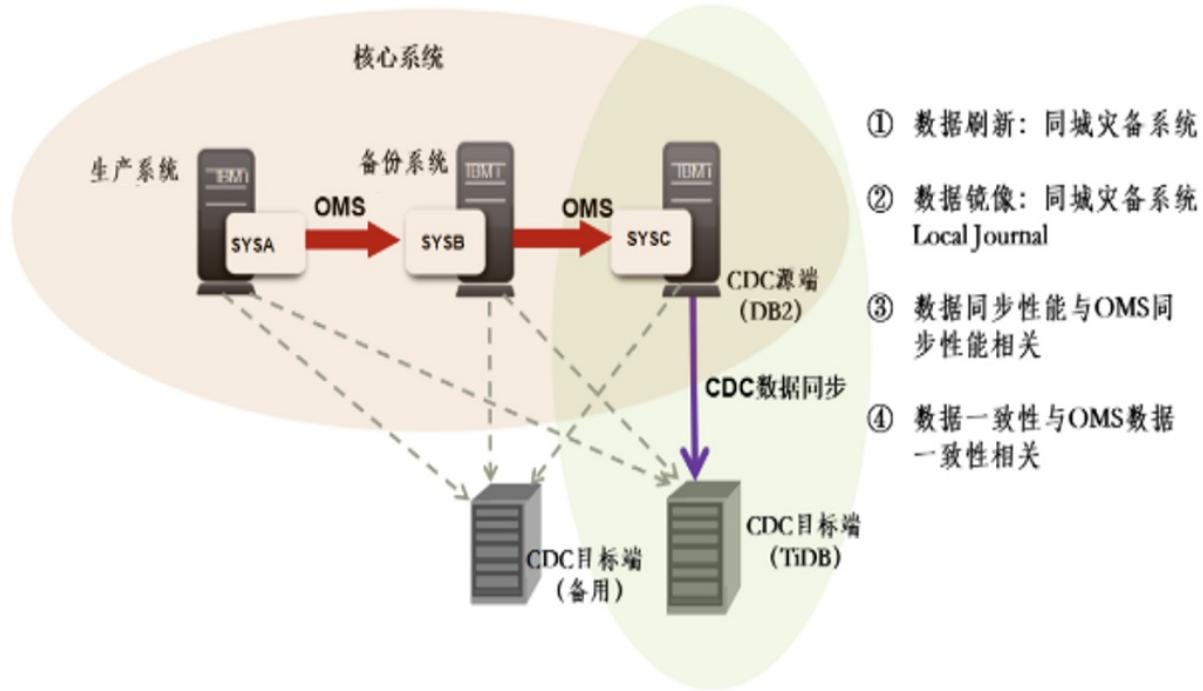
Supported source databases	Supported target databases and middleware applications
IBM??Db2? for Linux, UNIX and Windows (LUW)	IBM?Db2 for Linux, UNIX and Windows (LUW)
IBM?Db2 for i	IBM?Db2 for i
IBM?Db2 for z/OS?	IBM?Db2 for z/OS
IMS	IBM?InfoSphere? DataStage?
Microsoft SQL Server	Microsoft SQL Server
Oracle	CDC Replication Engine for FlexRep
Sybase?1	Netezza?
Informix??1	Oracle
Db2 on Cloud?(formerly dashDB? for Transactions)?2	Sybase?1
VSAM	Informix?1
PostgreSQL	CDC Replication Engine for Event Server?1
Db2 Warehouse on Cloud?(formerly dashDB for Analytics)?3	IBM Cloudant?
Db2 Warehouse?(formerly dashDB Local)?3	Apache? Hadoop
MySQL	Apache Kafka
	Db2 Warehouse on Cloud?(formerly dashDB for Analytics)
	Db2 on Cloud?(formerly dashDB for Transactions)?2
	Db2 Warehouse?(formerly dashDB Local)
	IBM MQ for z/OS (using Classic CDC for z/OS)
	Teradata
	Microsoft Azure SQL Database
	Microsoft Azure SQL Database Managed Instance
	IBM Integrated Analytics System

- 由于在官方列表中并没有列出 TiDB，但是它可以选择 CDC Replication Engine for FlexRep 这种方式支持 JDBC 引擎，因为 TiDB 兼容 MySQL 协议，同时 MySQL 也提供了 JDBC driver，所以这里可以使用它将数据写入到 TiDB 中。
- CDC 在实例配置时可以选择支持 datastage type，选择 FlexRep 后再配置 JDBC driver 驱动后就可以成功向 TiDB 中同步数据。

## 数据同步

下面我们以一个实际案例来介绍如何将 Db2 for i 中的数据同步到 TiDB 上。

### 部署架构



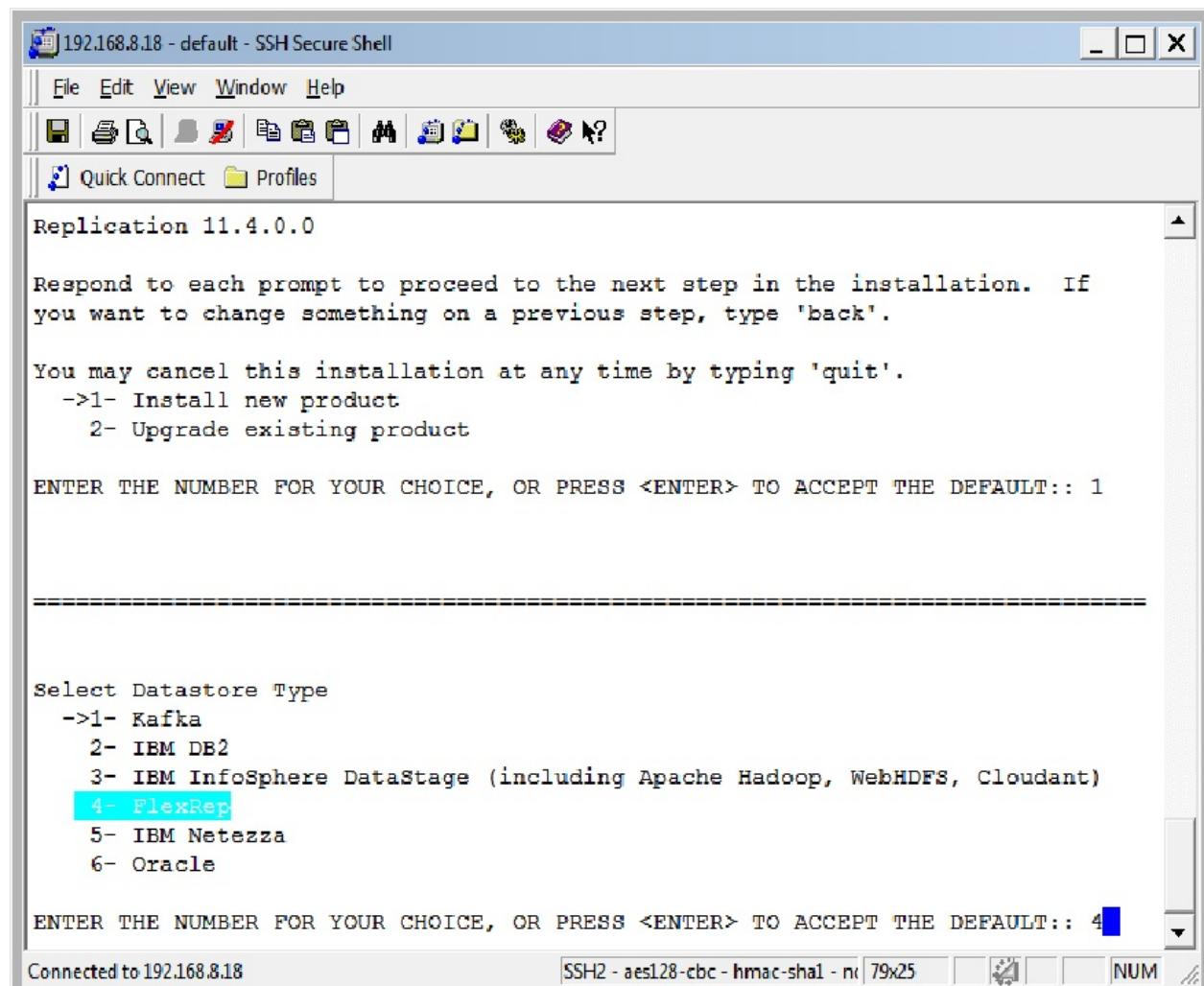
以上图的生产环境架构为例，上游是一个核心系统，使用的是IBM商业平台，生产和备份平台之间使用了 OMS 同步，IIDR 同步的源端为备机数据库，目标端为 TiDB。为保证高可用，在下游的另外一台服务器上部署了 IIDR 软件作为备用节点。

## 关键参数和配置项

关于 IIDR 的部署安装大家可以参考官网，本文不做重点介绍。这里主要说明一下 IIDR 的下游为 TiDB 时，在部署和使用过程中需要注意的点。

- IIDR 安装选项选择 FlexRep

因为 TiDB 并不在 IIDR 官方的支持列表中，所以在下游安装时，我们要选择 **FlexRep** 如下图所示：



- 创建预定需要借助 MySQL 驱动

IIDR 需要借助 MySQL 驱动通过 JDBC 的方式将数据写入 TiDB 中，需要提前下载 MySQL 驱动，并在创建订阅时选择该驱动，如下图所示：

192.168.8.18 - default - SSH Secure Shell

**File Edit View Window Help**

**Quick Connect Profiles**

**NEW INSTANCE: tstcdc >> JDBC DRIVERS**

Please refer to documentation for the list of supported drivers and their associated configuration values/parameters.

**1. Add Driver**

Enter your selection:1

Enter the location of the new JDBC driver file:/etc/mysql-connector-java-5.1.36-bin.jar

Connected to 192.168.8.18      SSH2 - aes128-cbc - hmac-sha1 - nc | 79x25 | NUM

- IIIDR 下游软件参数配置

参数	值
convert_not nullable_column	true
events_max_retain	10000
global_conversion_not_possible_warning	false
global_max_batch_size	25
global_shutdown_after_no_heartbeat_reponse_minute	10
Implicit_transformation_warning	true
jdbc_refresh_commit_after_max_operation	4000
Mirror_commit_after_max_operations	4000
Mirror_global_disk_quota_gb	9223372036854775807
Mirror_interim_commit_thresholds	100
Userexit_max_lob_size_kb	2097151
Mirror_commit_on_transaction_boundary	False

- 数据类型转换对照表

IIIDR 在同步全量数据之前，需要在 TiDB 侧创建好表结构，下表是 Db2 for i 到 TiDB 时表字段对应关系：

Db2字段类型	TiDB字段类型
L	date
T	time
Z	timestamp(6)
A	varchar
P	Decimal
S	decimal
O	varchar

## 使用限制

- CDC 是基于解析 Db2 的日志发现并捕获数据变化，调用的是 Db2 的日志解析的接口，所以只有数据库操作写了事务日志，才能同步到下游 TiDB，类似 load、alter table xxx activate not logged initially 之类的都不会被同步。
- CDC 在安装配置时，在选择 datastage type 时必须选择 FlexRep 才能顺利的将数据同步到 TiDB 中。

## 总结

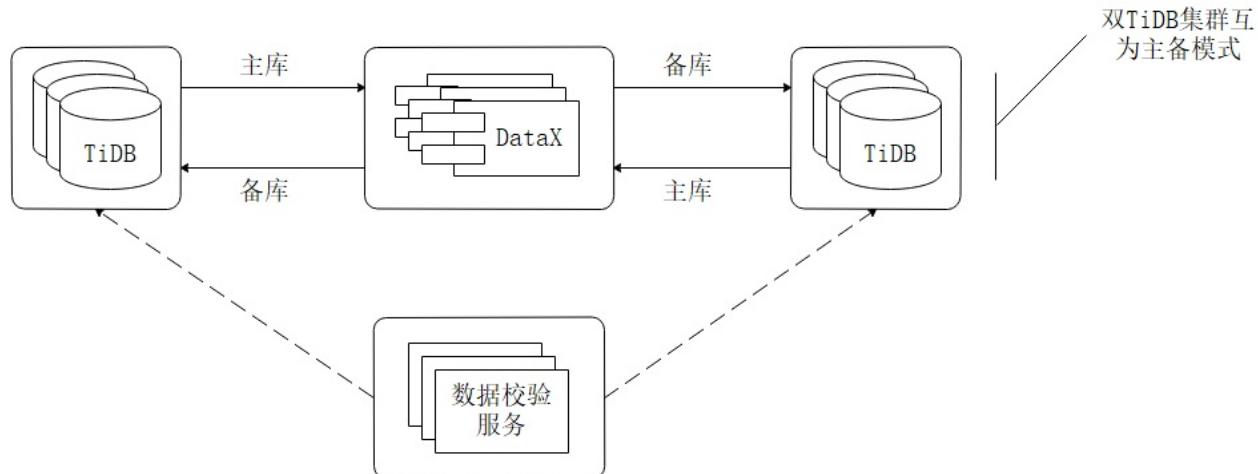
IIDR 是一款能够比较好的将 Db2 for i 的数据同步到 TiDB 的工具，在同步过程中如遇到上述配置还解决不了的问题，请联系 IIDR 官方或者 TiDB 官方，具体问题具体分析解决。

相对于从其它数据库向 TiDB 进行单次迁移切换，TiDB 到 TiDB 的数据复制体系一般是长期的高可用容灾场景，适用于多机房、多数据中心等环境，需要更加稳定可靠。建议增加服务管理和数据比对校验等辅助机制。

DataX 工具实现跨 TiDB 集群的增量数据复制，是在 TiDB 4.0 的 CDC 工具正式发布前比较成熟的一项选择，有利于在数据中心内统一异构平台的数据传输工具。

下面会以双 TiDB 集群互为主备的场景为案例进行操作讲解。从设计上既满足了传统数据库运维的要求，生产数据库拥有容灾备库提高系统健壮性，又使得服务器硬件等资源得到了更充分的利用。

方案设计如图所示



具体操作步骤：

第一步：部署DataX

下载

```
wget http://datax.opensource.oss-cn-hangzhou.aliyuncs.com/datax.tar.gz
```

解压

```
tar -zvxf datax.tar.gz
```

自检

```
python {YOUR_DATAx_HOME}/bin/datax.py {YOUR_DATAx_HOME}/job/job.json
```

第二步：编辑同步Job

vi increase.json

```
{
  "job": {
    "setting": {
      "speed": {
        "channel1": 128 #根据业务情况调整Channel1数
      }
    },
    "content": [
      {
        "table": "t1"
      }
    ]
  }
}
```

```

"reader": {

    "name": "tidbreader",

    "parameter": {

        "username": "${srcUserName}",

        "password": "${srcPassword}",

        "column": ["*"],

        "connection": [{

            "table": ["${tableName}"],

            "jdbcUrl": ["${srcUrl}"]

        }],


        "where": "updateTime >= '${syncTime}'"
    }
},

"writer": {

    "name": "tidbwriter",

    "parameter": {

        "username": "${desUserName}",

        "password": "${desPassword}",

        "writeMode": "replace",

        "column": ["*"],

        "connection": [{

            "jdbcUrl": "${desUrl}",

            "table": ["${tableName}"]

        }],


        "preSql": [
            "replace into t_sync_record(table_name,start_date,end_date) values('@table',now(),null)"
        ],
        "postSql": [
            "update t_sync_record set end_date=now() where table_name='@table' "
        ]
    }
}
}

```

第三步：编写运行DataX Job的Shell执行脚本

vi datax\_execute\_job.sh

```

#!/bin/bash

source /etc/profile

srcUrl="Reader SourceTiDB 地址"

srcUserName="账号"

srcPassword="密码"

desUrl="Writer DestTiDB 地址"

desUserName="账号"

desPassword="密码"

#同步开始时间

defaultSyncUpdateTime="2020-03-03 18:00:00.000"

#同步周期(秒)

sleepTime="N"

tableName="Table1,Table2, . . ."

#循环次数标识, -1为一直循环, 其他按输入次数循环, 可根据需求自定义传参。

flg=-1

while [ "$flg" -gt 0 -o "$flg" -eq -1 ]

do

    #更新时间设置为上次循序执行的时间

    if [ "" = "$preRunTime" ]; then

        syncTime=$defaultSyncUpdateTime

    else

        syncTime=$preRunTime

    fi

    #记录下本次循环执行时间, 供下次循环使用

    preRunTime=$(date -d +"%Y-%m-%d %T").000";

    echo $syncTime

    echo $preRunTime

    echo $flg

    python {YOUR_DATAAX_HOME}/bin/dataax.py -p "-DsyncTime='${syncTime}' -DtableName='${tableName}' -DsrcUrl='${srcUrl}' -DsrcUserName='${srcUserName}' -DsrcPassword='${srcPassword}' -DdesUrl='${desUrl}' -DdesUserName='${desUserName}' -DdesPassword='${desPassword}'" {YOUR_DATAAX_HOME}/job/increase.json

    if [ -1 -lt "$flg" ]; then

        let "flg-=1"

    fi

    sleep $sleepTime

done

```

#### 第四步：执行Shell脚本

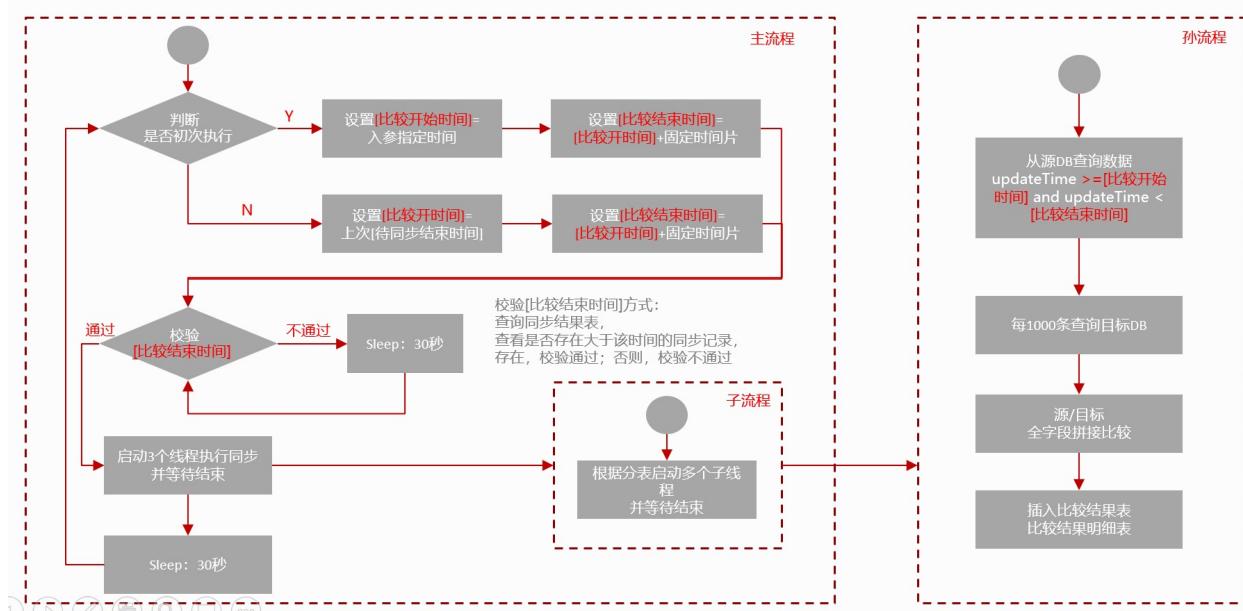
```
chmod +x datax_execute_job.sh
nohup ./datax_execute_job.sh > log.file 2>&1 &
```

DataX Job 的 Shell 执行脚本可以注册在 Supervisor 等服务工具中管理。当双 TiDB 集群需要进行主备切换时，可以做到任务随时启停，正反向同步随时切换。

#### 第五步：数据校验服务

按自定义时间跨度查询两个 TiDB 的数据，设置流式读取两个 TiDB 中表数据，转成字符串进行对比，更高效。在比对过程中，为了提高比对效率，如果发现不一致，记录哪张表的哪个时间片有不一致即可，不需要运算定位到具体某一行。对于发现不一致的记录可以发送报警，人工介入处理，也可以调动同步脚本，重新同步一次，可根据自己的业务灵活选择。

简单的数据校验流程可以参考下图。



用DataX 进行双 TiDB 集群间的数据同步不一定是最好的，但每个方案都有其特定存在的场景和意义。如果双集群间广域网链路质量不太稳定，又或者有特殊的定制需求等原因时，可以考虑本方案。

## 5.7 MongoDB 迁移到 TiDB

### 5.7.1 背景

MongoDB 具有诸如高效字段维护，自动故障转移等优秀特性，这使很多归档类业务可以快速在 MongoDB 上完成搭建，并为业务提供稳定的数据支撑。

但在快消时代的背景下，很多业务量日活数量日益增长，相应的数据量也成倍得增长。之前的基础架构已满足不了现在的需要，所以进行架构重构就是大势所趋。虽然可以使用 `MongoS` 集群方案，通过 `oplog` 解决备份恢复数据一致性的问题，来提供更高吞吐的业务支撑架构，但需要花费更多精力在多个集合关系维护上面。比如像直播业务中会有带货的子业务需求，业务逻辑中会做一些关联更新或者删除的操作。在这种场景中，对大表不但会进行高并发的实时查询和写入操作，还要有少量的更新和删除操作，这都为架构的研发和运维带来了不少麻烦。这类业务场景，在初选方案时候可能会先考虑到使用 MySQL 和分库分表的架构。但是分库分表架构一方面无法满足业务上对于分页查询和非分区健查询的需求等，另外运营部门对用户数据进行聚合分析的需求其他架构来支持。

如果使用 `MongoS` 解决方案，为提高 AP 场景下的查询效率，需要通过 MongoDB 先将数据同步到 `Kafka`，再通过 `ClickHouse` 处理查询请求。由于这个链路较长，导致日常维护成本增加。随着线上的业务越来越具有多样性，现有的工具既不能提供最佳的处理方案，也无法高效的管理数据聚合结构。

因为 MongoDB 到 TiDB 的数据迁移是异构数据迁移，主要是根据实际的业务场景，需要自研一套异构迁移的程序来完成数据的全量、增量同步。所以下文会以迁移流程逻辑为主，尤其是对于数据一致性验证的理论方法做了详细的介绍。

### 5.7.2 TiDB 和 MongoDB 的优势对比

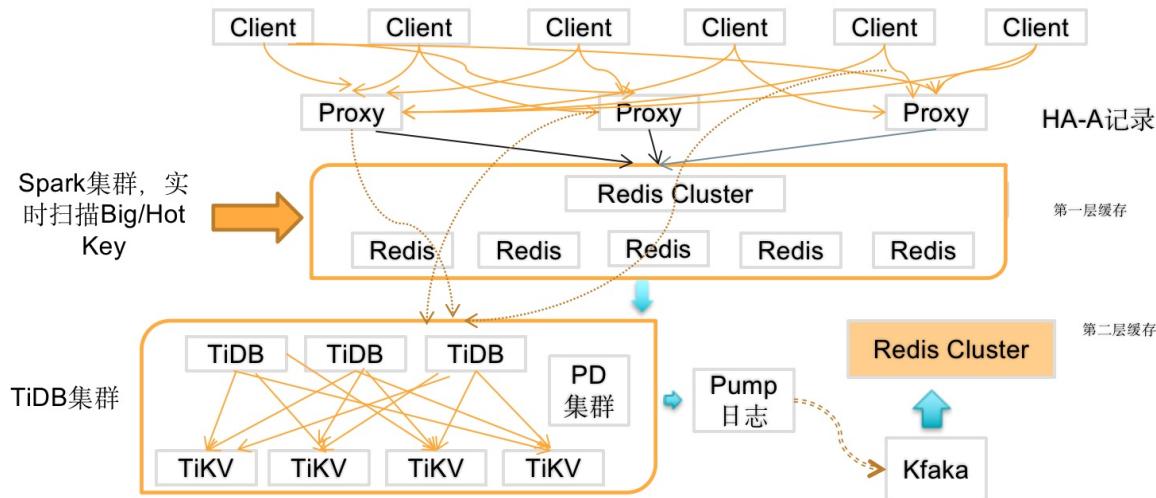
MongoDB	TiDB
1.灵活模式、高可用性、可扩展性； 2.通过 json 文档来实现灵活模式； 3.通过复制集来保证高可用； 4.通过 Sharded cluster 来保证可扩展性；	1.不用分库分表、高可用、弹性拓展； 2.减少分库分表带来的架构复杂性； 3.通过多副本保证高可用 4.通过集群模式来保证可扩展性

### 5.7.3 TiDB 的架构

业务存在多样性，会有高并发查询场景，也有高频查询场景，还会有复杂的聚合、分组的计算查询场景。在高并发的读请求业务对响应延迟极为敏感的场景，需要使用数据缓存层，像 Redis Cluster 这种缓存数据库来处理高并发、高频的热点请求。另外本文介绍的 MongoDB 迁移到 TiDB 的实践中，涉及到数据一致性验证的工作，需要通过 Redis 校验完成，那么先介绍一下完整的 TiDB 架构。

业务的写请求通过多个 Web Service 通过 Proxy 写入到第一层缓存 Redis Cluster，最终写入到 TiDB 集群。因为业务的写入请求可能会有 `Transaction too large` 的大事务，所以业务要先判断写入请求的事务大小，先将超过事务大小限制的写入请求会先进入 Redis Cluster，最后写入到 TiDB 集群，小于事务限制的写入请求直接写入 TiDB 集群。有些在线接近于实时分析，比如在线监控数据信息，使用 TiDB-binlog 将数据通过 `Kafka` 同步到第二层缓存 Redis Cluster。

业务的读请求通过 Proxy 访问 TiDB 集群，首先判断 Redis Cluster 是否 `key-value` 存在，如果有则从 Redis 返回，否则读取 TiDB 集群。对于复杂计算 SQL，例如聚合函数计算、分组查询等，业务可以直接将缓存失效，在 TiDB 集群做计算；另外 Spark 集群提供的高频的实时查询业务会在第一层缓存中，实时扫描 `Big Key` 和 `Hot Key`。主要基于 Redis `LFU` 功能，如果访问某个 `key-value` 大于 5 万次，则将该 `key-value` 通过程序植入基于 Spark 集群，当然有大概 3~5 分钟过期时间，这里可以根据业务的读请求的热点情况。另外如果访问都是一次，可以读取 TiDB，然后结果缓存到 Redis Cluster。



上文描述中，可以发现在架构中是没有引入数据库中间件处理业务请求，是因为数据库中间件成本可能会略高于这个架构，当然这也主要跟业务有关。这套架构中，业务会通过查询 Redis 和 TiDB 中的业务数据，来验证数据一致性，这也使架构变得复杂。另外业内还有其他方式解决数据一致性问题，比如通过 `sleep` 和设置延迟时间来保证读写数据的一致性。

## 5.7.4 数据迁移

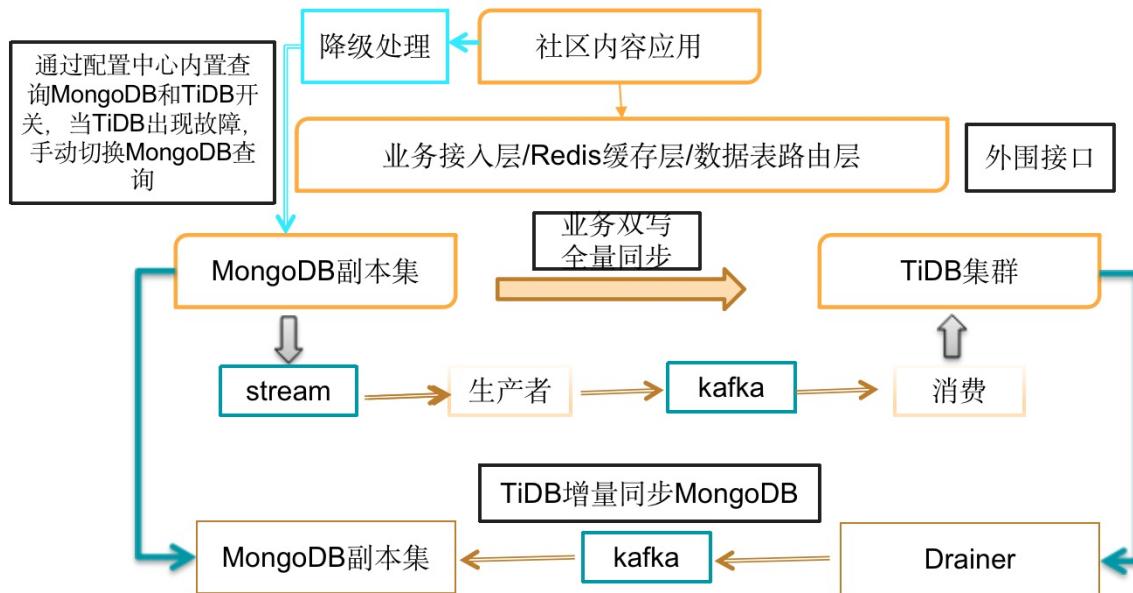
### 1. 迁移前准备

在迁移前要做好迁移流程的梳理，对于一些重要的功能点要确认业务逻辑和技术实现。根据梳理的需求设计迁移流程，确保流程中可以通过手动或者自动程序做到防患于未然，主动出击。以下为在迁移流程设计前，要考虑的几个比较重要问题，并提供一些可行性的方案。

- 数据类型的差异梳理：
  - MongoDB 原有的数据类型属性需要改变，例如 MongoDB 中的 `_id` 要摒弃，使用 TiDB 自增主键 `ID`，另外这里需要考虑到写入热点问题。
    - 业务侧可以考虑使用 `uuid` 或者 `snowflakes` 算法将主键值进行打散；
    - 对于 PK 非整数或没有 PK 的表，TiDB 会使用一个隐式的自增 `rowid`，大量 `INSERT` 时会把数据集中写入单个 Region，造成写入热点。通过设置 `SHARD_ROW_ID_BITS`，可以把 `rowid` 打散写入多个不同的 Region，缓解写入热点问题。但是设置的过大造成 RPC 请求数放大，增加 CPU 和网络开销。
    - 在 v4.0 版本的 TiDB，提供了 `Auto Random Key` 的特性，通过 `Auto Random Key` 可以自动将 `int` 主键生成随机数来避免写入热点问题；
  - MongoDB 和 TiDB 的部分字段类型不一样，MongoDB 时间列写入基本都是时间戳，TiDB 提供多种时间的数据类型：`DATETIME`，`TIMESTAMP`，`TIME`，`YEAR`。
- 数据一致性：全量迁移上十亿的数据的一致性，以及增量同步时变更数据的一致性都要保证；
- 关于时间有序性：要确认业务表是否有更新时间戳的场景。如果有，那么要确认该字段是否有关索引，于此同时如果有删除数据的操作，那么就不能使用更新时间字段作为增量迁移的进度标准；
- 写入性能：线上数据量达到亿级，在全量、增量迁移时要使用批量处理来加快迁移速度；另外通过 Web Service 增加并发线程，因为有幂等性，即回放多次但最终结果还是一致的，所以需要保证表级数据的有序性，也就是要保证一张表同时只有一个线程在进行增量数据的回放；
- 容错能力：制定同步任务异常处理机制，一旦 `watch` 监听任务出现异常，就从异常时间点开始从源端进行增量数据抽取，并进行重试操作流程；
- 断点续传能力：将转换失败的数据 `uid` 记录下来，重试或程序单独处理时方便根据 `uid` 查找对应的数据。

## 2. 迁移方案

迁移方案如图所示，MongoDB 数据通过一套 java 开发的程序来完成全量数据的同步，将一致性校验数据写入 Redis Cluster 中进行全量数据完整性校验。增量同步的数据通过 MongoDB 集群的 Change Stream 同步到 Kafka，最后数据写入到 TiDB 集群中，并进行一致性校验。同步完成后，会将业务读请求切换到 TiDB 集群，TiDB 集群通过 TiDB-binlog 到 Kafka 方式，将业务的增量数据同步回 MongoDB 集群进行一致性校验。此时 TiDB 集群支持读写，MongoDB 集群只支持写入。运维过程中，需要通过查询业务情况、集群状态来监控 TiDB 集群和 MongoDB 集群可用性和状态，如果 TiDB 出现故障，MongoDB 可以作为备库提供服务，管理人员通过手动或者自动程序，将业务请求切换到 MongoDB。下文会有具体的实施方案。

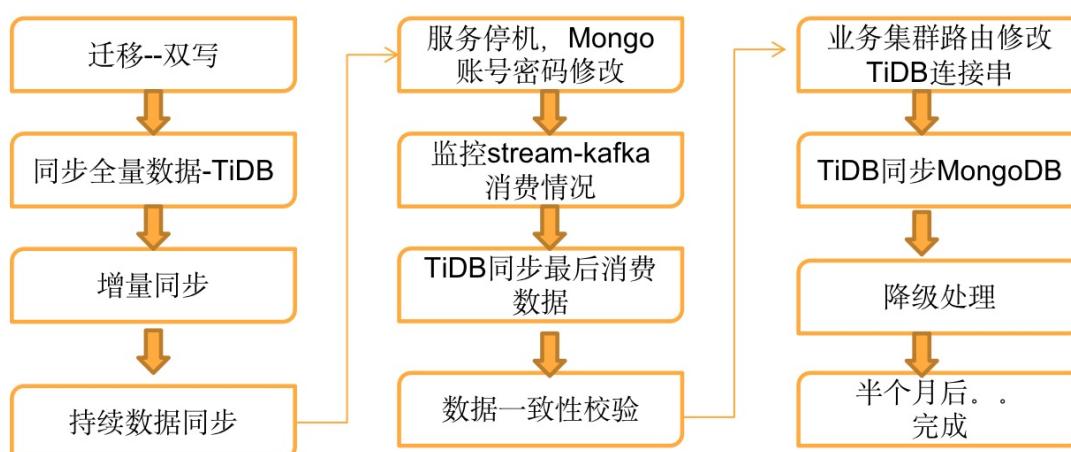


注意

从 MongoDB v3.6 开始提供了 Change Stream 增量功能，可以监控数据的变更情况，为实现数据同步及转换处理提供支撑功能。通过对集合执行 `watch` 命令，获得一个 `MongoCursor` 对象，这样就可以遍历所有的变更，获得被监听对象的实时变更。

## 3. 迁移流程

如图所示，迁移流程主要是 6 个主要部分，我们来分别介绍以下详细的迁移流程。



### (1) 通过代码实现异构迁移的双写

在全量同步过程中，以实时业务为例。在业务逻辑中，一般会获取 `max(uid)` 作为依据进行分页读取数据。所以可以制定这样一个逻辑规则：如果 TiDB 集群新增的业务数据值小于 `max(uid)`，那么开启双写，将在 TiDB 集群和 MongoDB 集群上分别写入一份数据；在迁移时，如果 TiDB 集群的业务数据值小于 `max(uid)` 的数据进行了 DML 操作，那么同时在 TiDB 表中

也进行 DML 操作。

### (2) Redis 记录迁移记录

第一次拉取全量数据，假设以数据维度分片的异构迁移程序的服务，要服务每开启 20 个线程，每个线程读取迁移的数据，并会记录迁移的 uid 到 Redis Cluster，目的是为了线程终止后，不用全量恢复数据，实现断点续传；另外设置 Redis 的 log 检查迁移数据情况、慢查询、错误信息等输出。

```
127.0.0.1:6379> KEYS *
1) "migration:data:st"
127.0.0.1:6379> HGETALL "migration:comminfo:st"
1) "2186200681228325_2382488577771540"
2) "2188513952998466"
3) "2859668199691429_93967041042064455"
4) "2861166835008519"
5) "2487557684037633_2612301467395085"
6) "2488649487696938"
7) "2612301467395085_2657794730464475"
8) "2613352273347597"
9) "2382488577771540_2438002819885096"
10) "2383933609133067"
11) "2708706397606993_2859668199691429"
12) "2709891834446963"
13) "2657794730464475_2708706397606993"
14) "2658901430273111"
15) "2094211406200848_2186200681228325"
16) "2096628556585988"
17) "2438002819885096_2487557684037633"
18) "2439044447011868"
```

### (3) 开始增量迁移

增量数据迁移方案采用 `UpdateTime` 和 `Change Stream` 方案，一方面 `UpdatedTime` 能将 `insert` 和 `update` 的增量数据拉取，另外一方面对于物理删除 `delete` 操作则检测不到，所以使用 MongoDB 通过 `Change Stream` 将增量拉取数据到 Kafka，最后由程序分析 Kafka 数据后同步到 TiDB 集群。

(4) 根据业务访问特点，避开业务高峰数据迁移，将迁移操作实施在低峰时间段。

(5) 数据验证，通过查询表中的数据量总数和业务的验证来保证数据的完整性。

### (6) 业务实现双写

- 关于写请求：在迁移 TiDB 表之前，插入写到 MongoDB 集群和 TiDB 集群，同时源表更新和删除的操作，也会同步到 TiDB 中；在表数据在迁移 TiDB 表之后，源表和 TiDB 表都会进行插入和更新操作。
- 关于读请求：在迁移 TiDB 表之前，业务查询会优先查询 MongoDB 集群；表数据在迁移 TiDB 表之后，业务查询会优先查询 TiDB 集群。
- 最后读写请求可以在 TiDB 集群完成，此时停止双写操作。增量数据通过 TiDB-binlog 经过 Kafka 同步到 MongoDB，可以设计一个降级处理机制，为业务准备逃生通道。

## 5.7.5 关于数据一致性的校验机制

### 1. 增量同步与校验机制

使用 MongoDB 的 `Change Stream` 根据时间拉取数据到 Kafka，同步到 TiDB 集群。同时 Redis 也会记录以 `uid` 的数据维度的迁移范围数据，可以通过校验 Kafka 的 `uid` 和 Redis 的 `uid` 进行比对是否一致。

```

127.0.0.1:6379> KEYS *
1) "migration:data:st"
127.0.0.1:6379> HGETALL "migration:comminfo:st"
1) "2186200681228325_2382488577771540"
2) "2188513952998466"
3) "2859668199691429_93967041042064455"
4) "2861166835008519"
5) "2487557684037633_2612301467395085"
6) "2488649487696938"
7) "2612301467395085_2657794730464475"
8) "2613352273347597"
9) "2382488577771540_2438002819885096"
10) "2383933609133067"
11) "2708706397606993_2859668199691429"
12) "2709891834446963"
13) "2657794730464475_2708706397606993"
14) "2658901430273111"
15) "2094211406200848_2186200681228325"
16) "2096628556585988"
17) "2438002819885096_2487557684037633"
18) "2439044447011868"

```

## 2. MongoDB metadata 的一致性验证

`Metadata` 作为 MongoDB 中最重要的表，虽然有 TiDB 集群的数据复制来保证数据同步，也最好双重确认来保障服务可用性。可以在 MongoDB 中开发了一个脚本来批量对比两个 MongoDB 数据集的 `Metadata` 表，通过扫描 `Metadata` 表所有的键值和时间戳来发现差异。在初期确实发现了差异，也依此来修正了 TiDB 集群的数据复制的配置。

## 3. 新 TiDB 集群的数据表的可用性验证

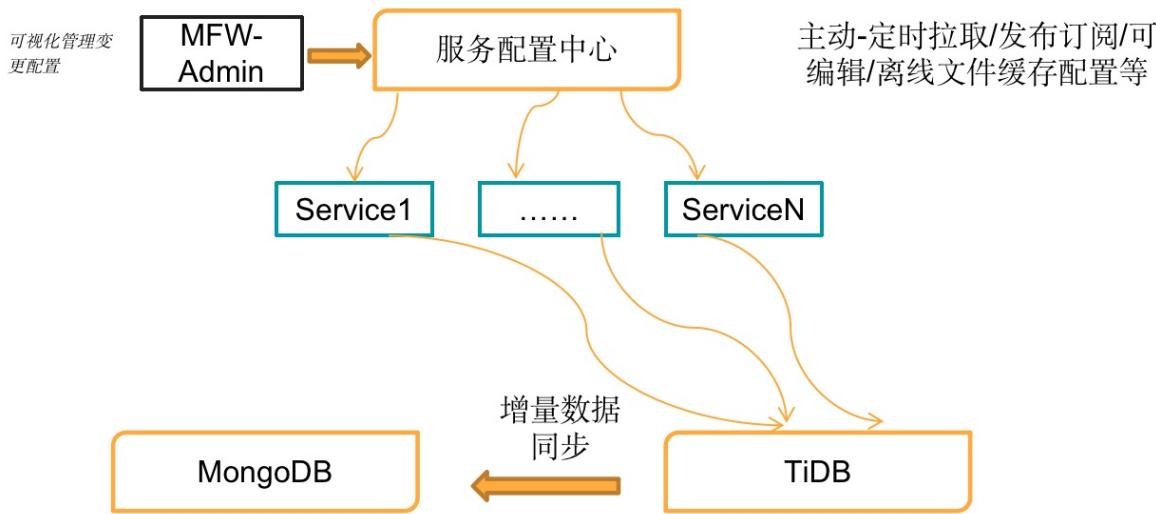
为了验证新集群 TiDB 集群的数据可用性，需要启动了一个测试的 MongoDB 实例用以模拟兼容 TiDB 集群的查询。该测试实例不直接对用户服务，而是通过回放 SQL 方式来进行可用性测试。回放测试会自动验证查询结果是否正常返回。这种测试方式弥补了回归测试用例覆盖范围的不足，通过测试可以确实发现隐藏的问题并进行了修复。当然在生产环境的切换后，如果未发现问题那是最好的。

### 5.7.6 回退方案 - 降级处理

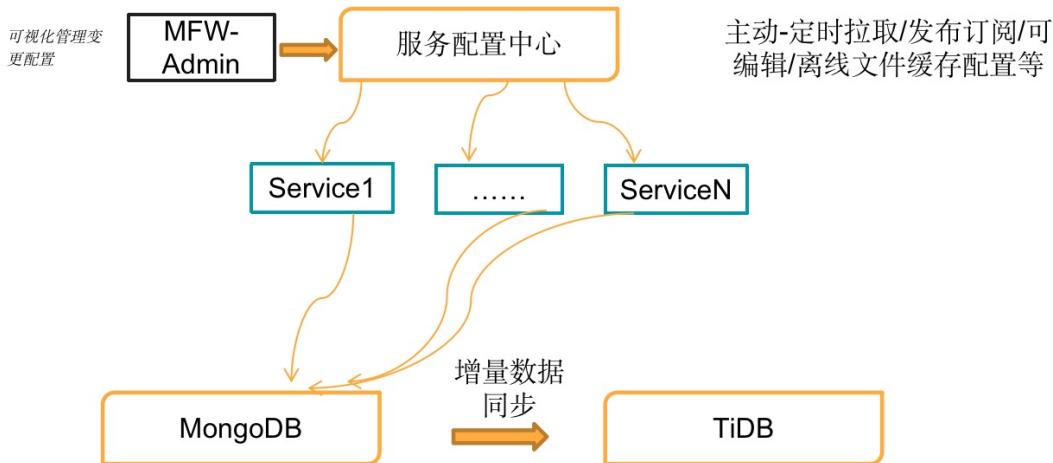
正常情况下，建议为业务准备逃生通道。逃生通道很重要，如果切换读写请求到 TiDB 集群后，可能因为业务逻辑兼容性或者其他的服务异常情况，需要通过切换回 MongoDB 保证业务的可用性。我们在这里可以设置一个降级处理逻辑，要设计触发器和制定完备的处理预案。

#### 1. 降级处理流程

如下图所示，运维平台最好有一套服务配置中心系统，可主动的定时拉取、发布订阅，可编辑、离线缓存配置文件等功能。从流程图中，可以看到正常的业务读写请求会在 TiDB 集群，TiDB 集群会通过 TiDB-binlog 的下游 Kafka 模式将数据同步到 MongoDB 的集群，此时 MongoDB 作为备库在整个架构里。



如当遇到 TiDB 集群业务不可以访问的情况的突发情况，需要通过人为手段恢复。通过手动变更配置信息，通过服务配置中心系统生效配置请求处理，将 Web Service 请求全部切换到 MongoDB，同时 MongoDB 会通过 Change Stream，下游配置 Kafka 模式将增量数据同步回 TiDB 集群。



### 1. 降级处理流程的触发器设计

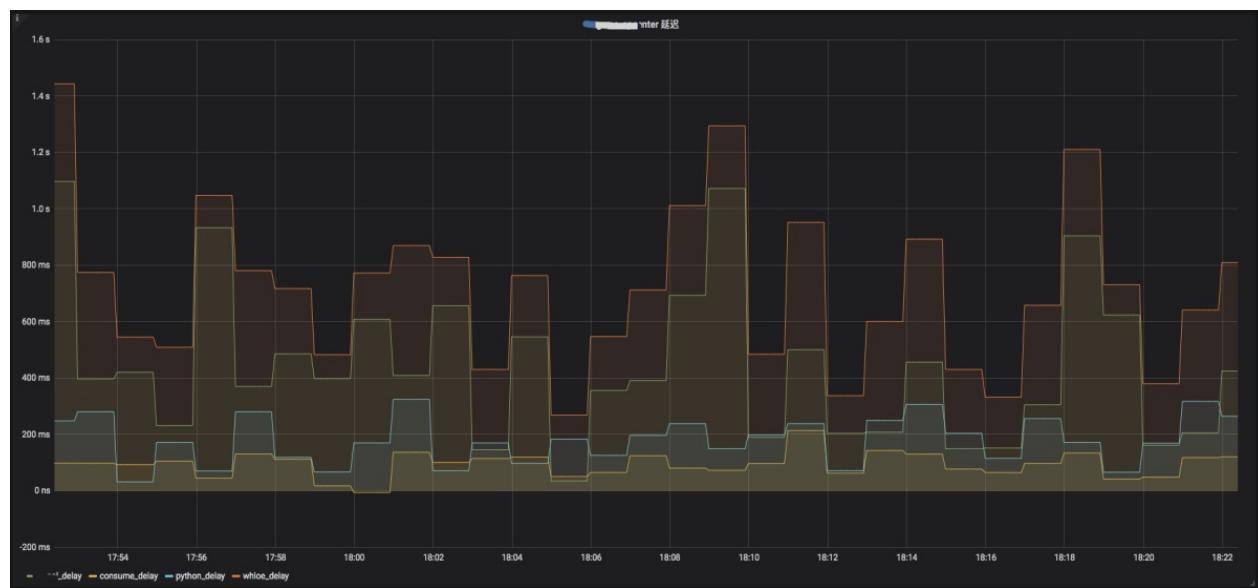
在业务层可以设计“熔断机制”，通过监控 RPC 调用失败次数占比来确认业务的请求是否正常处理，一旦触发“异常统计条件”，直接熔断服务。在业务侧直接返回报错，不再 RPC 调用远端服务。

异常统计条件：指定的时间窗口内，RPC 调用失败次数的占比，超过设定的阈值，就不再 RPC 调用，直接返回“降级逻辑”。

```
# 熔断参数配置：熔断器的参数
circuitBreaker.requestVolumeThreshold : //滑动窗口的大小，默认为 20
circuitBreaker.errorThresholdPercentage : //错误率，默认 50%
circuitBreaker.sleepWindowInMilliseconds : //过多长时间，熔断器再次检测是否开启，默认为 5000，即 5s 钟
# 配置含义
每当 20 个请求中，有 50% 失败时，熔断器就会断开。此时，再调用此服务，将不再调远程服务，直接返回失败。5s 后，重新检测该触发条件，判断是否熔断器连接，或者继续保持断开。
```

降级操作是配合“熔断机制”，熔断后，Web Service 不再调用远端服务器的 RPC 接口，而采用本地的回退机制，返回一个“备用方案”和“默认取值”。

这个机制相比直接挂掉业务要好一些，但也要看业务场景。接下来业务隔离重试，运维同学要通过配置中心系统内置查询 MongoDB 和 TiDB 的开关，将 TiDB 手动切换 MongoDB 查询。切换后，因为还有增量数据同步到 TiDB 集群，可以通过 Kafka 监控来确认数据同步的延迟情况。



### 5.7.7 迁移后的 SQL 优化

MongoDB 的查询也基本都是单表，通过走索引方式查询。针对每一条改造后的 SQL 都要进行了优化。比如可以使用精确的命中最优的索引，从而实现了在几十亿数据量。在类似迁移案例的用户反馈中，优化前的 TP 业务 99% 的响应时间在 15ms，99.9% 的响应时间在 700ms 内，这其中有很多的查询请求是都是 sum、group by 的形式的。经过对于查询优化，通过调整聚合计算的并发参数或者索引优化。优化后，TP 业务 99% 的响应时间在 15ms，99.9% 的响应时间在 48ms；具体的优化建议可以查阅 SQL 优化的官方文档。以下为用户反馈的优化前后的性能对比。

- 优化前



- 优化后



### 5.7.8 总结

MongoDB 迁移到 TiDB 链路比较长，使用了 MongoDB 的 change stream 和 TiDB 的 TiDB-binlog，均通过 Kafka 实现增量的数据同步。另外比较重要的数据一致性校验，在全量同步、增量同步过程中，主要通过业务表的唯一键和 Updatetime，通过 Redis、MongoDB、TiDB 的业务数据比对来完成。其次就是在整个链路中，为业务准备逃生通道，在任意场景中，都保证了

业务的可用性以及可控的停机时间。因为 MongoDB 和 TiDB 在数据存储格式的差异，也使迁移后的读写请求的优化不可避免。所以建议阅读本书涉及性能优化的章节，为业务的性能助力。

## 第 6 章 业务适配最佳实践

TiDB作为一个HTAP数据库，结合上层应用能够承载不同的应用负载(OLTP、OLAP、混合负载)，本章将会通过一些最佳实践案例来向读者展示在不同的场景下如何正确的使用TiDB分布式数据库。

- [TiDB 中事务限制及应对方案](#)
- [一种高效分页批处理方案](#)
- [TiDB + TiSpark 跑批最佳实践](#)
- [TiDB 在企业数据分级存储中的应用实践](#)
- [TiDB 与 HBase、ES、Druid 的数据交互实战](#)
- [TiDB 与可视化展现 Saiku、Grafana 的集成应用](#)
- [TiKV 常见配置优化](#)

## 6.1 业务开发最佳实践

在使用 TiDB 进行业务开发的时候，由于其自身的特点，有许多跟传统单机数据库不同的地方。本章节结合用户实际使用过程中的经验，总结了一些在使用 TiDB 过程中最佳实践。

- [乐观锁模式下的事务最佳实践](#)
- [TiDB 中事务限制及应对方案](#)
- [高并发的唯一序列号生成方案](#)
- [一种高效分页批处理方案](#)
- [通过 hint 调整执行计划](#)
- [SQL 调优案例](#)
- [TiDB + TiSpark 跑批最佳实践](#)

## 6.1.1 乐观锁模式下的事务最佳实践

在 3.0.8 之前，TiDB 的默认事务模式是乐观事务，TiDB 乐观事务存在以下优点：

- 基于单实例事务实现了跨节点事务
- 去中心化的锁管理

缺点如下：

- 两阶段提交，网络交互多。
- 需要一个中心化的版本管理服务。
- 事务在 commit 之前，数据写在内存里，数据过大内存就会暴涨

基于以上缺点，我们有了后面的建议。

### 6.1.1.1 减少乐观锁事务冲突

事务的冲突，主要分两种：

- 读写冲突：存在并发的事务，部分事务对相同的 Key 读，部分事务对相同的 Key 进行写。
- 写写冲突：存在并发的事务，同时对相同的 Key 进行写入。

在 TiDB 的乐观锁机制中，因为是在客户端对事务 commit 时，才会触发两阶段提交，检测是否存在写写冲突。所以，在乐观锁中，存在写写冲突时，很容易在事务提交时暴露，因而更容易被用户感知。

频繁乐观锁事务冲突会对应用的整体性能造成很大影响

1. 当前应用和 TiDB 需要额外的时间和资源重试失败的事务
2. 微服务环境下，应用本身常是上下游服务间柔性事务（如TCC）的一部分。这时失败的 TiDB 事务会触发应用层的回滚和重试，从而对整个服务链路带来压力

业务上并发修改的场景(如账本, 余额, 秒杀等)，需要改造业务，减少乐观锁事务冲突，常见方案有

1. 业务要求同步执行时，使用 TiDB v3 引入的悲观锁，或者分布式锁(如 redis)，将业务串行化
2. 业务允许异步执行时，使用消息队列串行化请求，消息队列之前可以用缓存限流。当使用消息队列写入 TiDB 时，将同一行的读写分配到同一个分区(例如 kafka 的 partition)，这样不同消费者之间就不会对同一行并发读写
3. 将大事务拆解成多个小事务以减少单个事务的运行时间。需要注意的是如果事务粒度过细，事务网络开销也会造成性能问题，需要根据业务场景压测的结果得到最优的事务大小

### 6.1.1.2 控制事务大小

TiDB 两阶段提交的网络开销相对较大，因此建议将多个单行事务(100~500 行)打包成一个多行事务发送

当事务过大时，会有以下问题

1. TiDB 内存使用量过大甚至 OOM
2. 乐观锁事务冲突可能性变大。对大事务的不断重试更是性能上的恶性循环
3. 第二阶段提交时耗时过长

因此目前 TiDB 对大事务有如下限制

1. 单个事务包含的 SQL 语句不超过 5000 条（默认）
2. 每个键值对不超过 6MB
3. 键值对的总数不超过 300,000
4. 键值对的总大小不超过 100MB

### 6.1.1.3 冲突预检

检测数据是否存在写写冲突是一个很重的操作，这个操作在 prewrite 时 TiKV 中具体执行。为了优化这一块性能，TiDB 集群会在内存里面进行一次冲突预检测。

主要在两个模块进行：

- TiDB 层，如果在 TiDB 实例本身发现存在写写冲突，那么第一个写入发出去后，后面的写入就已经能清楚的知道自己冲突了，没必要再往下层 TiKV 发送请求去检测冲突。
- TiKV 层，主要发生在 prewrite 阶段。因为 TiDB 集群是一个分布式系统，TiDB 实例本身无状态，实例之间无法感知到彼此的存在，所以无法确认自己的写入与别的 TiDB 实例是否存在冲突，所以会在 TiKV 这一层检测具体的数据是否有冲突。

其中 TiDB 层的冲突检测通过下面参数控制：

```
txn-local-latches 事务内存锁相关配置，当本地事务冲突比较多时建议开启。 enable 开启 默认值：false
capacity Hash 对应的 slot 数，会自动向上调整为 2 的指数倍。每个 slot 占 32 Bytes 内存。当写入数据的范围比较广时（如导数据），设置过小会导致变慢，性能下降。 默认值：1024000
```

这里 capacity 的配置会影响到冲突判断的正确性。在实现冲突检测时，真正存下来的是每个 key 的 hash 值，有 hash 算法就有误判的概率，这里我们通过 capacity 来控制 hash 取模的值：

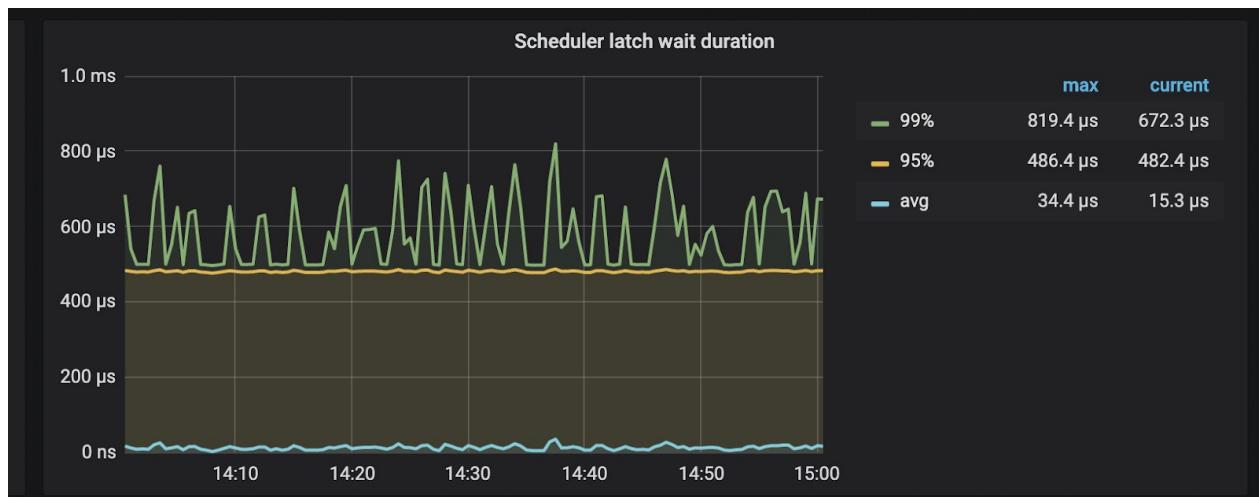
- capacity 值越小，占用内存小，误判概率越大。
- capacity 值越大，占用内存大，误判概率越小。

如果业务场景能够预判断写入不存在冲突，建议关闭 TiDB 层的冲突检测。

TiKV 内存中的冲突检测也有一套类似的机制。不同的是，TiKV 的检测是必须进行的，只提供了一个 hash 取模值的配置项：

```
scheduler-concurrency scheduler 内置一个内存锁机制，防止同时对一个 key 进行操作。每个 key hash 到不同的槽。 默认值：2048000
```

此外，TiKV 提供了监控查看具体消耗在 latch 等待的时间：



如果发现这个 wait duration 特别高，说明耗在等待锁的请求上比较久，如果不存在底层写入慢问题的话，基本上可以判断这段时间内冲突比较多。

#### 6.1.1.4 谨慎使用tidb的乐观锁重试机制

由于乐观锁是在 commit 阶段检测事务冲突，在冲突比较大的时候，Commit 很容易出现失败，而悲观锁模式数据库如 MySQL 的冲突检测在 SQL 执行过程中执行，所以 commit 时很难出现异常。为了解决这种行为不一致的问题，TiDB 提供了重试机制，由以下两个参数控制：

- `tidb_disable_txn_auto_retry`：这个参数控制是否自动重试，默认为 on，即不重试。
- `tidb_retry_limit`：用来控制重试次数，注意只有第一个参数启用时该参数才会生效

重试的步骤如下：

1. 重新获取 start\_ts
2. 对带写入的 SQL 进行重放
3. 两阶段提交

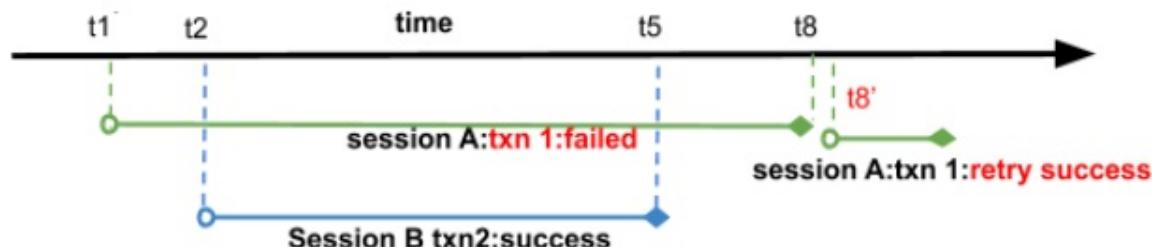
打开乐观锁重试时 (`tidb_disable_txn_auto_retry = off`) , 如果事务写入的执行条件依赖于这个事务中先前读到的结果，并发执行就会有 lost update 问题。此时不要打开 TiDB 乐观锁的重试机制。需要注意的是，如果是一些内部原因引起的重试不需要新的 tso, 例如网络抖动，平衡 region 等原因引起的重试，TiDB 会自动的，安全的重试，即使`tidb_disable_txn_auto_retry = off`

打开了重试后，我们来看下面的例子：

时间	Session A	Session B											
t1	MySQL [test]> begin; <b>Query OK, 0 rows affected (0.00 sec)</b>												
t2		MySQL [test]> begin; <b>Query OK, 0 rows affected (0.00 sec)</b>											
t3		MySQL [test]> update tidb set status=0 where id=1; <b>Query OK, 1 row affected (0.01 sec)</b> <b>Rows matched: 1</b> <b>Changed: 1</b> <b>Warnings: 0</b>											
t4	MySQL [test]> select from tidb where id=1; **+-----+-----+-- -----+ \	id \	name \	status \	+---- ---+-- ---+ +---- ---+ \	1 \	tikv \	1 \		+---- ---+-- ---+ +---- ---+ 1 row in set (0.01 sec)*			
t5		commit; <b>Query OK, 0 rows affected (0.01 sec)</b>											
t6	update tidb set name='pd' where id =1 and status=1; <b>Query OK, 1 row affected (0.00 sec)</b> <b>Rows matched: 1</b> <b>Changed: 1</b> <b>Warnings: 0</b>												
t7	MySQL [test]> select from tidb where id=1; **+-----+-----+--	id \	name \	status \	+---- ---+-- ---+ +---- ---+ 1 \	\	pd \	1 \	+---- ---+-- ---+ +---- ---+				

									1 row in set (0.01 sec)*
t8	MySQL [test]> commit; <b>Query OK, 0 rows affected (0.01 sec)</b>								
t9	MySQL [test]> select from tidb where id=1; **+-----+-----+-- -----+ \\	id \\	name \\	status \\	+---- ---+-- --- +---- ---+ \\	1 \\	tikv \\	0 \\	+---- ---+-- --- +---- ---+ 1 row in set (0.00 sec)*

我们来详细分析以下这个 case:



- 如图，在session B 在 t2 开始事务 2，t5 提交成功。session A 的事务 1 在事务 2 之前开始，在事务2 提交完成后提交。
- 事务 1、事务 2 会同时去更新同一行数据。
- session A 提交事务 1 时，发现冲突，tidb 内部重试事务 1
  - 重试时，重新取得新的 start\_ts 为 t8'
  - 回放更新语句 update tidb set name='pd' where id =1 and status=1
    - 发现当前版本 t8' 下并不存在符合条件的语句，不需要更新
    - 没有数据更新，返回上层成功
- tidb 认为事务 1 重试成功，返回客户端成功。
- session A 认为事务执行成功，查询结果，在不存在其他更新的情况下，发现数据与预想的不一致。

这里我们可以看到，对于重试事务，如果本身事务中更新语句需要依赖查询结果时，因为重试时会重新取版本号作为 start\_ts，因而无法保证事务原本的 ReadRepeatable 隔离型，结果与预测可能出现不一致。

综上所述，如果存在依赖查询结果来更新 SQL 语句的事务，建议不要打开TiDB 乐观锁的重试机制。

在 TiDB 乐观事务模型下有一些缺点，需要应用，架构层进行改造。同时为了克服这些缺点，满足更加严苛的场景，TiDB 实现了悲观事务，可以参考悲观事务章节。

## 4.6 TiDB 中事务限制及应对方案

在 2.1 及之前的 TiDB 版本中，对于事务的限制是和其他关系型数据库而言比较特殊的地方，很多用户在使用过程中总是会感觉比较困惑，本文针对事务限制做一些详细的说明，希望能够帮助大家理解。

### 4.6.1 官方定义

&gt;

由于 TiDB 分布式两阶段提交的要求，修改数据的大事务可能会出现一些问题。因此，TiDB 特意对事务大小设置了一些限制以减少这种影响：

每个键值对不超过 6MB

键值对的总数不超过 300,000

键值对的总大小不超过 100MB

单个事务包含的 SQL 语句不超过 5000 条（默认）

详见 [PingCAP 官方文档 - 大事务](#)

键值对应该比较容易理解，毕竟 TiDB 底层存储选用的是 RocksDB 引擎，一种基于 Key-Value 的存储结构。而每个键值对的大小和总大小限制分别是 6MB 和 100MB，这个应该也比较容易理解。关键在于每个事务包含键值对的总数不超过 30W，这个经常会引起一些误解，下面做一些详细说明。

### 4.6.2 如何理解 30W

很多人第一眼看上去，以为是一个事务涉及的行数不能超过 30W，但其实不是这样的，首先需要了解 TiKV 是如何将结构化数据转化为 Key-Value 结构存储的。

对于 Key-Value 结构的数据，结构如下：

Key	Value	Flag

- **Insert**操作

插入数据时，TiKV 的处理包含以下几个步骤：

(1) 插入数据本身

Key: PK + TSO	Value: Fields	Flag: Put

(2) 插入唯一索引

Key: Index (UK) + TSO	Value: PK	Flag: Put

(3) 插入普通索引

Key: Index + PK + TSO	Value: Null	Flag: Put

综上，当执行 Insert 事务时，30W 限制需要除以  $(1 + \text{所有索引的数量 (包含唯一索引)})$ 。

- **Delete**操作

下面考虑当删除一条数据时，TiKV 是如何处理的。首先需要明确，RocksDB 引擎所有的操作都是新增，所以删除也是插入，相当于插入了一条 Flag = Del 的记录。具体步骤如下：

(1) 插入数据本身的删除标记

<b>Key: PK + TSO</b>	<b>Value: Null</b>	<b>Flag: Del</b>
----------------------	--------------------	------------------

(2) 插入唯一索引的删除标记

<b>Key: Index (UK) + TSO</b>	<b>Value: Null</b>	<b>Flag: Del</b>
------------------------------	--------------------	------------------

(3) 插入普通索引的删除标记

<b>Key: Index + PK + TSO</b>	<b>Value: Null</b>	<b>Flag: Del</b>
------------------------------	--------------------	------------------

综上，当执行 Delete 事务时，30W 限制需要除以 (1 + 所有索引的数量 (包含唯一索引))。

- **Update**操作

首先，我们看看如果更新的是非主键且无索引字段的情况。这种情况，只需要修改记录本身的内容即可，也就是下面一步：

(1) 插入数据本身即可

<b>Key: PK + TSO</b>	<b>Value: Fields</b>	<b>Flag: Put</b>
----------------------	----------------------	------------------

综上，非主键且无索引字段更新，30W 限制就是 30W。

其次，来看更新的是非主键，但包含索引的字段情况。

(1) 数据本身

<b>Key: PK + TSO</b>	<b>Value: Fields</b>	<b>Flag: Put</b>
----------------------	----------------------	------------------

(2) 如果更新字段上有唯一索引

<b>Key: Index (UK) + TSO</b>	<b>Value: Null</b>	<b>Flag: Del</b>
<b>Key: Index (UK) + TSO</b>	<b>Value: PK</b>	<b>Flag: Put</b>

(3) 如果更新字段上有普通索引

<b>Key: Index + PK + TSO</b>	<b>Value: Null</b>	<b>Flag: Del</b>
<b>Key: Index + PK + TSO</b>	<b>Value: Null</b>	<b>Flag: Put</b>

综上，非主键但索引相关字段的更新，30W 限制需要除以 (1 + 字段涉及索引数量 \* 2)。

最后来看当更新的是主键字段的情况。

从上面的插入描述中可以看出，无论是数据本身，还是索引，都包含了 PK，所以主键更新会触发所有的 Key 更新，具体如下：

(1) 数据本身

<b>Key: PK + TSO</b>	<b>Value: Null</b>	<b>Flag: Del</b>
----------------------	--------------------	------------------

Key: PK + TSO	Value: Fields	Flag: Put

(2) 所有的唯一索引

Key: Index (UK) + TSO	Value: PK	Flag: Put

(3) 所有的普通索引

Key: Index + PK + TSO	Value: Null	Flag: Del

Key: Index + PK + TSO	Value: Null	Flag: Put

综上，主键字段的更新，30W 限制需要除以  $((1 + \text{普通索引数量}) * 2 + \text{唯一索引数量})$ ，Update 主键的时候，唯一索引当做 1 个 KV，普通索引和主键当做 2 个 KV (唯一索引在对应的 Key-Value 中，Key 是 UK 的值，Value 是 PK，Update PK 的时候，Key 值不变，所以只需一次 Put kv 操作；其他的，比如普通索引，Key 里面就存了 PK 的值，这样 Update 的时候记录的 Del 是一个 kv，Put 是一个新的 kv，所以当做两次处理)。

### 4.6.3 30W 键值对的转换

总结如下：|操作|键值对转换公式| |----:|----:| | Insert |  $30W/(1+\text{Idx\_Count})$  || Delete |  $30W/(1+\text{Idx\_Count})$  || Update\_On\_PK |  $30W/((1+\text{Non\_UK})*2+\text{UK}^1)$  || Update\_non\_PK |  $30W/(1+\text{Involved\_Idx\_Count}^2)$  |

具体案例：

```
CREATE TABLE t1 (
    id int(11) NOT NULL AUTO_INCREMENT,
    name char(10) CHARSET utf8mb4 COLLATE utf8mb4_bin DEFAULT Null,
    age int(11) DEFAULT Null,
    PRIMARY KEY (id),
    Key idx_name (name)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

以上面的简单表结构为例，该表有自增主键，外加 1 个普通索引，那么上面的事务限制对应的记录数为：

操作	键值对转换公式	最大操作行数
Insert	$30W/(1+1)$	15W
Delete	$30W/(1+1)$	15W
Update_On_id	$30W/((1+1)^2 + 0)$	7.5W
Update_On_name	$30W/(1+1^2)$	10W
Update_On_age	$30W/(1+0^2)$	30W

对于 TiDB 来说，有一个特殊之处，就是当主键是非 int 类型时，会有一个隐藏 int 类型主键，同时，本身定义的这个主键变成了唯一索引。所以，修改下上面表定义为如下：

```
CREATE TABLE t1 (
    id varchar(11) NOT Null,
    name char(10) CHARSET utf8mb4 COLLATE utf8mb4_bin DEFAULT Null,
```

```

age int(11) DEFAULT Null,
PRIMARY Key ( id ),
Key idx_name ( name )
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin

```

那么，该表有一个隐藏主键，外加 1 个唯一索引（用户定义的主键），外加 1 个普通索引，那么上面的事务限制对应的记录数为：

操作	键值对转换公式	最大操作行数
Insert	30W/(1+2)	10W
Delete	30W/(1+2)	10W
Update_On_id	30W/(1+1*2)	10W
Update_On_name	30W/(1+1*2)	10W
Update_On_age	30W/(1+0*2)	30W

#### 4.6.4 事务的其他限制

除了上述限制以外，TiDB 中对于事务还有另外一个限制

(1) 参数 stmt-count-limit，默认值是 5000。

StmtCountLimit limits the max count of statement inside a transaction.

也就是一个事务里面，默认最多包含 5000 条 SQL statement，在不超过上面几个限制的前提下，这个参数可以修改 TiDB 的配置文件进行调整。

(2) 另外在某些场景下，例如执行 Insert Into Select 的时候，可能会遇到下面的报错

ERROR 1105 (HY000): BatchInsert failed with error: [try again later]: con:3877 **txn takes too much time**, start: 405023027269206017, commit: 405023312534306817

这个主要是有一个隐藏参数，max-txn-time-use，默认值是 gc\_life\_time - 10s，也就是 590。

具体参考 PingCAP GitHub 上的文档：<https://github.com/pingcap/TiDB/blob/master/config/config.toml.example#L240>

# The max time a Txn may use (in seconds) from its startTS to commitTS. # We use it to guarantee GC worker will not influence any active txn. Please make sure that this# Value is less than gc\_life\_time - 10s.

所以我们要尽量保证一个事务在这个 gc\_life\_time - 10s 的时间内完成，也可以通过调整 gc 时间 + 修改这个参数来避免这个问题，可能 TiDB 的配置文件中没有放出这个参数，可以手动编辑加入这个值。当然，更好的办法应该是开启 Batch 功能来规避单个事务过大的问题。

#### 4.6.5 如何绕开大事务的限制

官方提供内部 Batch 的方法来绕过大事务的限制，分别由三个参数来控制：

- tidb\_batch\_insert

作用域: SESSION 默认值: 0 这个变量用来设置是否自动切分插入数据。仅在 Autocommit 开启时有效。当插入大量数据时，可以将其设置为 True，这样插入数据会被自动切分为多个 Batch，每个 Batch 使用一个单独的事务进行插入。

- tidb\_batch\_delete

作用域: SESSION 默认值: 0 这个变量用来设置是否自动切分待删除的数据。仅在 Autocommit 开启时有效。当删除大量数据时，可以将其设置为 True，这样待删除数据会被自动切分为多个 Batch，每个 Batch 使用一个单独的事务进行删除。

- `tidb_dml_batch_size`

作用域: SESSION 默认值: 20000 这个变量用来设置自动切分插入 / 待删除数据的的 Batch 大小。仅在 `tidb_batch_insert` 或 `tidb_batch_delete` 开启时有效。需要注意的是，当单行数据大小很大时，20k 行总数据量有可能会超过单个事务大小限制，因此在这种情况下，用户应当将该参数设置为一个较小的合适值。

针对 Update 场景，官方还是建议通过 `limit` 的方式来循环操作，目前并未提供内部 Batch Update 的参数开关。

需要注意的是，开启了 Batch 功能之后，大事务的完整性就没法保证，只能保证每个批次的事务完整性。当然，数据库的最佳实践依然是由程序或 DBA 来控制事务的大小，尤其是针对分布式数据库，建议每个 Batch 控制在 100 条左右，高并发的写入，同时避免热点现象，才能发挥 TiDB 分布式的优势。

## 6.1.3 高并发的唯一序列号生成方案

应用程序通常使用唯一标识符来确认一行记录。传统解决方案普遍依赖数据库的序列对象（Sequence）来生成唯一标识符中的数字部分。TiDB 4.0 正式支持序列。但是，互联网应用需要处理的数据量巨大且往往呈现爆发式增长，使得应用程序必须在短时间内为大量数据和消息生成唯一标识符。这种场景下，即使有了序列功能，数据库也可能会由于高并发的序列分配请求而出现性能瓶颈。

本节将介绍两种高性能的序列号生成方案。

### 方案一：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

<code>  sign  </code>	<code>delta seconds</code>	<code>  worker node id  </code>	<code>sequence  </code>
<code>1bit</code>	<code>28bits</code>	<code>22bits</code>	<code>13bits</code>

- sign：长度固定为 1 位。固定为 0，表示生成的 ID 始终为正数。
- delta seconds：默认 28 位。当前时间，表示为相对于某个预设时间基点（默认 "2016-05-20"）的增量值，单位为秒。28 位最多可支持约 8.7 年。
- worker node id：默认 22 位。表示机器 id，通常在应用程序进程启动时从一个集中式的 ID 生成器取得。常见的集中式 ID 生成器是数据库自增列或者 Zookeeper。默认分配策略为用后即弃，进程重启时会重新获取一个新的 worker node id，22 位最多可支持约 420 万次启动。
- sequence：默认 13 位。表示每秒的并发序列，13 位可支持每秒 8192 个并发。

使用类 Snowflake 方案时需要注意几个问题：

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

### 方案二：号段分配方案

号段分配方案可以理解为从数据库批量获取自增 ID。本方案需要一张序列号生成表，每行记录表示一个序列对象。表定义示例如下：`| 字段名 | 字段类型 | 字段说明 || :---- | :----- | :----- || SEQ_NAME | varchar(128) | 序列名称，用来区分不同业务 || MAX_ID | bigint(20) | 当前序列已被分配出去的最大值 || STEP | int(11) | 步长，表示每次分配的号段长度 |`

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

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



## 6.1.4 一种高效分页批处理方案

常规的分页更新 SQL 一般使用主键或者唯一索引进行排序，这样能避免相邻两页之间出现空隙或重叠；再配合 MySQL `limit` 语法中非常好用的 `offset` 功能按固定行数拆分页面，然后把页面包装进独立的事务中，从而实现灵活的分页更新。

```
begin;
update sbtest1 set pad='new_value' where id in (select id from sbtest1 order by id limit 0,10000);
commit;
begin;
update sbtest1 set pad='new_value' where id in (select id from sbtest1 order by id limit 10000,10000);
commit;
begin;
update sbtest1 set pad='new_value' where id in (select id from sbtest1 order by id limit 20000,10000);
commit;
```

如上述 SQL 所示，该方案逻辑清晰，代码也易于编写。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，相应地扫描数据过程中对 TiKV 的压力也会线性增长。这导致整体处理效率偏低，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，甚至引发性能问题，影响线上业务。

本节将介绍一种改进方案。

这里我们假定的业务需求是，要在一小时内完成 200 万行数据的并发处理。下面我们来初始化一张表 `tmp_loan`，表结构如下所示；该表初始状态即包含约 200 万行数据。

```
MySQL [demo]> desc tmp_loan;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| serialno | int(11) | NO | PRI | NULL |       |
| name | varchar(40) | NO |     |       |       |
| businesssum | int(10) | NO |     | 0 |       |
+-----+-----+-----+-----+-----+
MySQL [demo]> select count(1) from tmp_loan;
+-----+
| count(1) |
+-----+
| 1998985 |
+-----+
MySQL [demo]> select * from tmp_loan limit 10;
+-----+-----+-----+
| serialno | name | businesssum |
+-----+-----+-----+
| 200000000 | 华碧波 | 10000 |
| 200000001 | 陶南 | 10000 |
| 200000002 | 何谷 | 10000 |
| 200000003 | 曹念 | 10000 |
| 200000004 | 潘旋千 | 10000 |
| 200000005 | 魏柔 | 10000 |
| 200000006 | 公羊 | 10000 |
| 200000007 | 司马 | 10000 |
| 200000008 | 陶之 | 10000 |
| 200000009 | 严香 | 10000 |
+-----+-----+-----+
```

改进方案的基本思路是，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。下面的代码演示了具体的做法：

```
MySQL [demo]> select min(t.serialno) as start_key, max(t.serialno) as end_key, count(*) as page_size from ( select *
, row_number () over (order by serialno) as row_num from tmp_loan ) t group by floor((t.row_num - 1) / 50000) order by start_key;
+-----+-----+-----+
| start_key | end_key | page_size |
+-----+-----+-----+
| 200000000 | 200050001 |      50000 |
| 200050002 | 200100007 |      50000 |
| 200100008 | 200150008 |      50000 |
| 200150009 | 200200013 |      50000 |
| 200200014 | 200250017 |      50000 |
| ..... | ..... | ..... |
| 201900019 | 201950018 |      50000 |
| 201950019 | 201999003 |     48985 |
+-----+-----+-----+
40 rows in set (1.51 sec)
```

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

```
MySQL [demo]> select serialno from tmp_loan where serialno between 200050002 and 200100007;
+-----+
| serialno |
+-----+
| 200050002 |
| 200050003 |
| 200050004 |
| 200050005 |
| 200050006 |
| ..... |
+-----+
50000 rows in set (0.070 sec)
```

当我们需要批量修改数据时，也可以借助上面计算好的分片信息，实现高效数据更新。

```
MySQL [demo]> update tmp_loan set businesssum = 6666 where serialno between 200000000 and 200050001;
Query OK, 50000 rows affected (0.89 sec)
Rows matched: 50000  Changed: 50000  Warnings: 0

MySQL [demo]> select * from tmp_loan order by serialno limit 10;
+-----+-----+-----+
| serialno | name    | businesssum |
+-----+-----+-----+
| 200000000 | 华碧波   |      6666 |
| 200000001 | 陶南    |      6666 |
| 200000002 | 何谷    |      6666 |
| 200000003 | 曹念    |      6666 |
| 200000004 | 潘旋千  |      6666 |
| 200000005 | 魏柔    |      6666 |
| 200000006 | 公羊    |      6666 |
| 200000007 | 司马    |      6666 |
| 200000008 | 陶之    |      6666 |
| 200000009 | 严香    |      6666 |
+-----+-----+-----+
```

总体而言，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。

## 6.1.5 使用 Hint 绑定执行计划

当优化器选择了不当的执行计划的时候，需要使用 hint 进行执行计划的绑定。TiDB 兼容了 MySQL 的 USE INDEX, FORCE INDEX, IGNORE INDEX 语法，同时开发了 TiDB 自身的 Optimizer Hints 语法，它基于 MySQL 5.7 中介绍的类似 comment 的语法，例如 `/+ TIDB_XX(t1, t2) /`。TiDB 目前支持的 hint 语法列表：

Hint	功能说明
USE INDEX	Index Hint: Choose Index
FORCE INDEX	Index Hint: Choose Index
IGNORE INDEX	Index Hint: Ignore Index
<code>/+ TIDB_INLJ(t) /</code>	Join Hint: Nested Index Lookup Join
<code>/+ TIDB_HJ(t) /</code>	Join Hint: Hash Join
<code>/+ TIDB_SMJ(t) /</code>	Join Hint: Merge Join
<code>/+ MAX_EXECUTION_TIME(num) /</code>	Execution Time Limit

### 6.1.5.1 USE INDEX, FORCE INDEX, IGNORE INDEX

与 MySQL 类似，不合适的查询计划是慢查询的常见原因，这时就要用 USE INDEX 指定查询用的索引，例如下面例子 USE/FORCE INDEX 使得原本全表扫描的 SQL 变成了通过索引扫描。

```
mysql> explain select * from t;
+-----+-----+-----+-----+
| id      | estRows | task      | access object | operator info      |
+-----+-----+-----+-----+
| TableReader_5    | 8193.00 | root      |                | data:TableFullScan_4 |
| └─TableFullScan_4 | 8193.00 | cop[tikv] | table:t      | keep order:false   |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> explain select * from t use index(idx_1);
+-----+-----+-----+-----+
| id      | estRows | task      | access object      | operator info      |
+-----+-----+-----+-----+
| IndexLookUp_6  | 8193.00 | root      |                  |                   |
| ├─IndexFullScan_4(Build) | 8193.00 | cop[tikv] | table:t, index:idx_1(a) | keep order:false |
| └─TableRowIDScan_5(Probe) | 8193.00 | cop[tikv] | table:t          | keep order:false |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> explain select * from t force index(idx_1);
+-----+-----+-----+-----+
| id      | estRows | task      | access object      | operator info      |
+-----+-----+-----+-----+
| IndexLookUp_6  | 8193.00 | root      |                  |                   |
| ├─IndexFullScan_4(Build) | 8193.00 | cop[tikv] | table:t, index:idx_1(a) | keep order:false |
| └─TableRowIDScan_5(Probe) | 8193.00 | cop[tikv] | table:t          | keep order:false |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

下面的例子 IGNORE INDEX 使得原本走索引的 SQL 变成了全表扫描

```
mysql> explain select a from t where a=2;
+-----+-----+-----+-----+
| id      | estRows | task      | access object          | operator info      |
+-----+-----+-----+-----+
| IndexReader_6    | 1.00    | root      |                   | index:IndexRangeScan_5 |
| └─IndexRangeScan_5 | 1.00    | cop[tikv] | table:t, index:idx_1(a) | range:[2,2], keep order:false |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> explain select a from t ignore index(idx_1) where a=2 ;
+-----+-----+-----+-----+
| id      | estRows | task      | access object          | operator info      |
+-----+-----+-----+-----+
| TableReader_7   | 1.00    | root      |                   | data:Selection_6 |
| └─Selection_6    | 1.00    | cop[tikv] | eq(test.t.a, 2)       |                  |
|   └─TableFullScan_5 | 8193.00 | cop[tikv] | table:t              | keep order:false |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

和 MySQL 不同的是，目前 TiDB 并没有对 USE INDEX 和 FORCE INDEX 做区分

当表上有多个索引时，建议使用 USE INDEX。TiDB 的表都比较大，`analyze table` 会对集群性能造成较大影响，因此无法频繁更新统计信息。这时就要用 USE INDEX 保证查询计划的正确性

### 6.1.5.2 MAX\_EXECUTION\_TIME(N)

在 SELECT 语句中可以使用 `MAX_EXECUTION_TIME(N)`，它会限制语句的执行时间不能超过 N 毫秒，否则服务器会终止这条语句的执行。

例如，下面例子设置了 1 秒超时

```
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM t1
```

另外，环境变量 `MAX_EXECUTION_TIME` 也会对语句执行时间进行限制。

对于高可用和时间敏感的业务，建议使用 `MAX_EXECUTION_TIME`，以免错误的查询计划或 bug 影响整个 TiDB 集群的性能甚至稳定性。OLTP 业务查询超时一般不超过 5 秒。

需要注意的是，MySQL jdbc 的查询超时设置对 TiDB 不起作用。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` 保证查询超时的效果

### 6.1.5.3 JOIN HINT

TiDB 目前表 Join 的方式有 Sort Merge Join，Index Nested Loop Join，Hash Join，具体的每个 join 方式的实现细节可以参考 [TiDB 源码阅读系列](#)

语法：

#### 1. TIDB\_SMJ(t1, t2)

```
SELECT /*+ TIDB_SMJ(t1, t2) */ * from t1, t2 where t1.id = t2.id;
```

提示优化器使用 Sort Merge Join 算法，简单来说，就是将 Join 的两个表，首先根据连接属性进行排序，然后进行一次扫描归并，进而就可以得出最后的结果，这个算法通常会占用更少的内存，但执行时间会更久。当数据量太大，或系统内存不足时，建议尝试使用。

#### 2. TIDB\_INLJ(t1, t2)

```
SELECT /*+ TIDB_INLJ(t1, t2) */ * from t1, t2 where t1.id = t2.id;
```

提示优化器使用 Index Nested Loop Join 算法，Index Look Up Join 会读取外表的数据，并对内表进行主键或索引键查询，这个算法可能会在某些场景更快，消耗更少系统资源，有的场景会更慢，消耗更多系统资源。对于外表经过 WHERE 条件过滤后结果集较小（小于 1 万行）的场景，可以尝试使用。TIDB\_INLJ() 中的参数是建立查询计划时，内表的候选表。即 TIDB\_INLJ(t1) 只会考虑使用 t1 作为内表构建查询计划

### 3. TIDB\_HJ(t1, t2)

```
SELECT /*+ TIDB_HJ(t1, t2) */ * from t1, t2 where t1.id = t2.id;
```

提示优化器使用 Hash Join 算法，简单来说，t1 表和 t2 表的 Hash Join 需要我们选择一个 Inner 表来构造哈希表，然后对 Outer 表的每一行数据都去这个哈希表中查找是否有匹配的数据这个算法多线程并发执行，执行速度较快，但会消耗较多内存。

另外其他的 hint 语法也在开发中如 `/+ TIDB_STREAMAGG()`，`/+ TIDB_HASHAGG()` 等。

使用 Hint 通常是在执行计划发生变化的时候，通过修改 SQL 语句调整执行计划行为，但有的时候需要在不修改 SQL 语句的情况下干预执行计划的选择。[执行计划绑定](#)提供了一系列功能使得可以在不修改 SQL 语句的情况下选择指定的执行计划。

## 6.2 SQL 调优案例

本节主要汇集了一些用户常见的 SQL 优化案例，关于 SQL 调优原理的介绍见第三部分第一章。

注：以下语句及结果基本为当时实际环境所记录的情况，因为版本更新原因，可能和现有格式略有差别，如 count 等价于现在的 estRows。

### 案例1 Delete 涉及数据量过大导致 OOM

```
MySQL [db_stat]> explain delete from t_stat where imp_date<='20200202';
+-----+-----+-----+
| id      | count      | task | operator info
+-----+-----+-----+
| TableReader_6    | 220895815.00 | root | data:Selection_5
|  \-Selection_5   | 220895815.00 | cop   | le(db_stat.t_stat.imp_date, "20200202")
|    \-TableScan_4  | 220895815.00 | cop   | table:t_stat, range:[-inf,+inf], keep order:false
+-----+-----+-----+
3 rows in set (0.00 sec)

MySQL [db_stat]> select count(*)  from t_stat where imp_date<='20200202';
+-----+
| count(*) |
+-----+
| 184340473 |
+-----+
1 row in set (17.88 sec)
```

#### 背景

- 大批量清理数据时系统资源消耗高，在 TiDB 节点内存不足时可能导致 OOM

#### 分析

- imp\_date 字段上虽然有索引，但是扫描的时间范围过大，无论优化器选择 IndexScan 还是 Table Scan，TiDB 都要向 TiKV Coprocessor 请求读取大量的数据

#### 影响

- TiKV 节点 Coprocessor CPU 使用率快速上涨
- 执行 Delete 操作的 TiDB 节点内存占用快速上涨，因为要将大批量数据加载到 TiDB 内存

#### 建议

- 删除数据时，缩小数据筛选范围，或者加上 limit N 每次删除一批数据
- 建议使用 3.0 版本的 Range 分区表，按照分区快速删除

### 案例2 执行计划不稳定导致查询延迟增加

```

MySQL [db_stat]> explain SELECT * FROM `tbl_article_check_result` `t` WHERE (articleid = '20190925A0PYT800') ORDER BY checkTime desc LIMIT 100 ;
+-----+-----+-----+
| id      | count   | task  | operator info
+-----+-----+-----+
| Projection_7    | 100.00  | root  | db_stat.t.type, db_stat.t.articleid, db_stat.t.docid, db_stat.t.version, db_stat.t.checkid, db_stat.t.checkstatus, db_stat.t.seclevel, db_stat.t.t1checkstatus, db_stat.t.t2checkstatus, db_stat.t.mdaichannel, db_stat.t.mdaisubchannel, db_stat.t.checkuser, db_stat.t.checktime, db_stat.t.addtime, db_stat.t.havegot, db_stat.t.checkcode
| └─Limit_12       | 100.00  | root  | offset:0, count:100
|
|   └─IndexLookUp_34 | 100.00  | root  |
|
|     └─IndexScan_31 | 30755.49 | cop   | table:t, index:checkTime, range:[NULL,+inf], keep order:true, desc
|
|       └─Selection_33 | 100.00  | cop   | eq(db_dayu_1.t.articleid, "20190925A0PYT800")
|
|         └─TableScan_32 | 30755.49 | cop   | table:tbl_article_check_result, keep order:false
+
+-----+-----+-----+
6 rows in set (0.00 sec)

```

## 背景

- articleid 和 checkTime 字段上分别建有单列索引，正常情况下走 articleid 上的索引比较快，偶尔执行计划不稳定时走 checkTime 上的索引，导致查询延迟达到分钟级别

## 分析

- LIMIT 100 限定了获取 100 条记录，如果 checkTime 和 articleid 列之间的相关度不高，在独立性假设失效时，优化器估算走 checkTime 上的索引并满足 articleid 条件时扫描的行数，可能比走 articleid 上的索引扫描的行数更少

## 影响

- 业务响应延迟不稳定，监控 Duration 偶尔出现抖动

## 建议

- 手动 analyze table，配合 crontab 定期 analyze，维持统计信息准确度
- 自动 auto analyze，调低 analyze ratio 阈值，提高收集频次，并设置运行时间窗口
  - set global tidb\_auto\_analyze\_ratio=0.2;
  - set global tidb\_auto\_analyze\_start\_time='00:00 +0800';
  - set global tidb\_auto\_analyze\_end\_time='06:00 +0800';
- 业务修改 SQL，使用 force index 固定使用 articleid 列上的索引
- 3.0 版本下，业务可以不用修改 SQL，使用 create binding 创建 force index 的绑定 SQL

- 4.0 版本支持 SQL Plan Management (见第三部分 1.3 节)，可以避免执行计划不稳定导致的性能下降

### 案例3 查询字段与值的数据类型不匹配

```
MySQL [db_stat]> explain select * from t_like_list where person_id=1535538061143263;
+-----+-----+-----+
| id      | count   | task | operator info
+-----+-----+-----+
| Selection_5    | 1430690.40 | root | eq(cast(db_stat.t_like_list.person_id), 1.535538061143263e+15)
|           |
| └TableReader_7 | 1788363.00 | root | data:TableScan_6
|           |
|   └TableScan_6 | 1788363.00 | cop  | table:t_like_list, range:[-inf,+inf], keep order:false
|           |
+-----+-----+-----+
-----+
3 rows in set (0.00 sec)
```

#### 背景

- person\_id 列上建有索引且选择性较好，但执行计划没有按预期走 IndexScan

#### 分析

- person\_id 是字符串类型，但是存储的值都是数字，业务认为可以直接赋值；而优化器需要在字段上做 cast 类型转换，导致无法使用索引

#### 建议

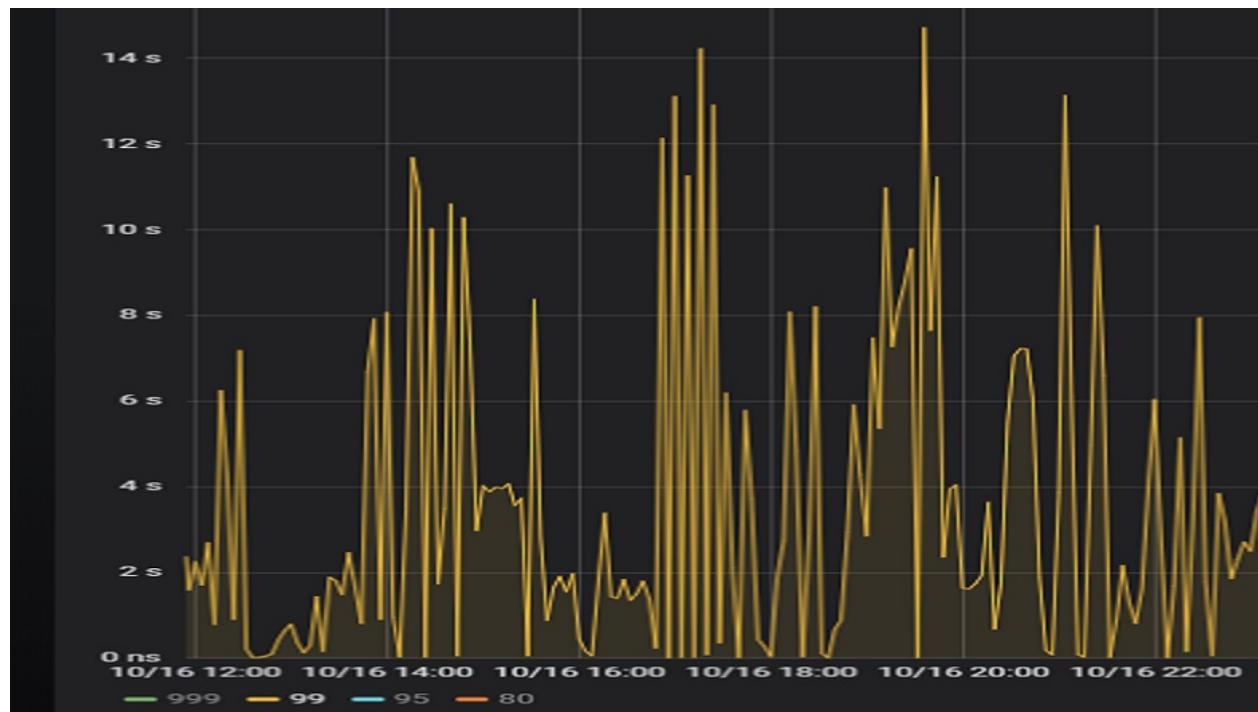
- where 条件的值加上引号，之后执行计划使用了索引

```
MySQL [db_stat]> explain select * from table:t_like_list where person_id='1535538061143263';
+-----+-----+-----+
| id      | count   | task | operator info
|           |
+-----+-----+-----+
| IndexLookUp_10 | 0.00 | root |
|           |
| └IndexScan_8   | 0.00 | cop  | table:t_like_list, index:person_id, range:["1535538061143263", "1535538061143263"], keep order:false
|           |
| └TableScan_9   | 0.00 | cop  | table:t_like_list, keep order:false
|           |
+-----+-----+-----+
-----+
3 rows in set (0.00 sec)
```

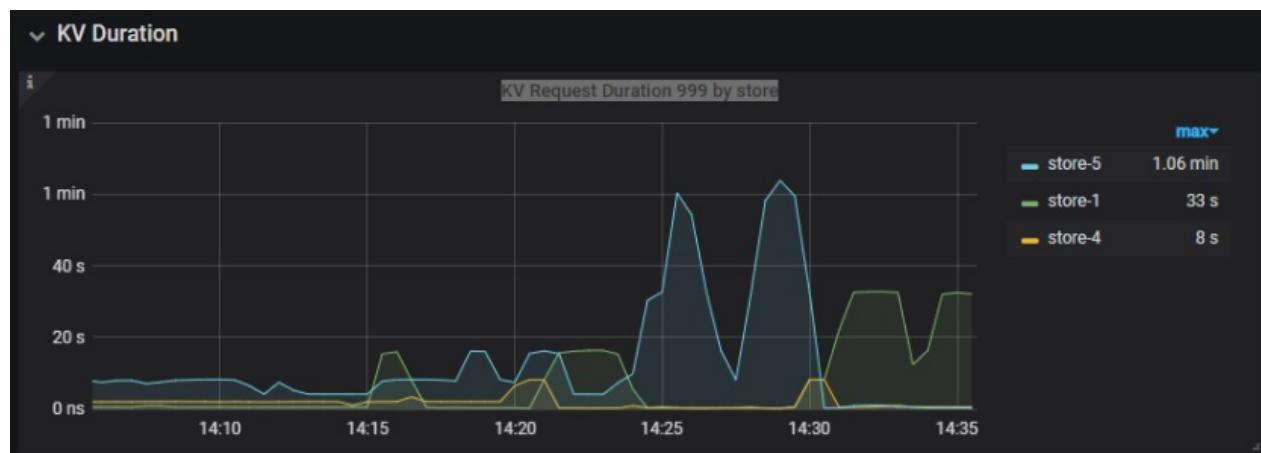
### 案例4 读热点导致 SQL 延迟增加

#### 背景

某个数据量 600G 左右、读多写少的 TiDB 集群，某段时间发现 TiDB 监控的 Query Summary - Duration 指标显著增加，p99 如下图。



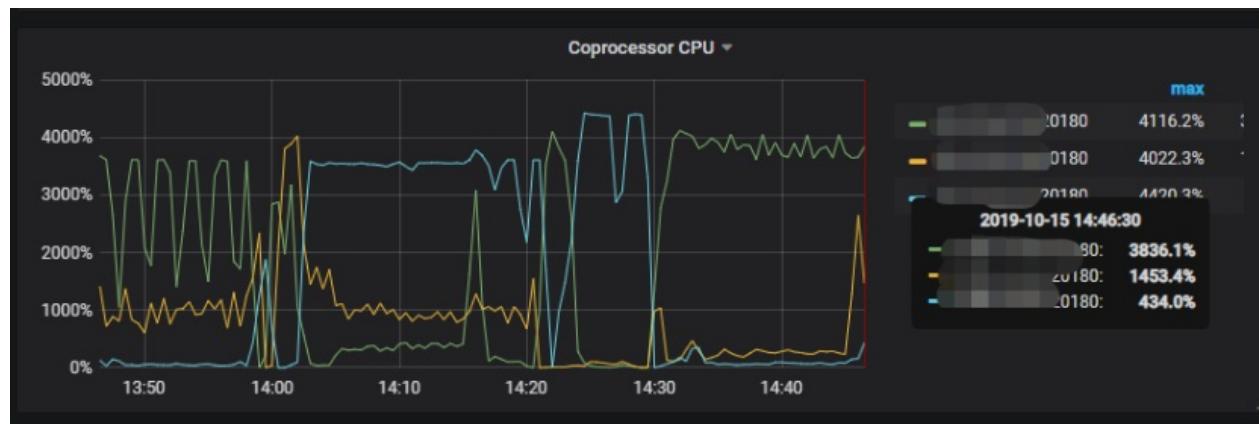
查看 TiDB 监控下的 KV Duration 明显升高，其中 KV Request Duration 999 by store 监控看到多个 TiKV 节点 Duration 均有上涨。



查看 TiKV 监控 Coprocessor Overview



查看监控 Coprocessor CPU



发现 Coprocessor CPU 线程池几乎打满。下面开始分析日志，调查 Duration 和 Coprocessor CPU 升高的原因。

### 慢查询日志分析

使用 pt-query-digest 工具分析 TiDB 慢查询日志。

```
./pt-query-digest tidb_slow_query.log > result
```

分析慢日志解析出来的 TopSQL 发现 Process keys 和 Process time 并不是线性相关，Process keys 数量多的 SQL 的 Process time 处理时间不一定更长，如下面 SQL 的 Process keys 为 22.09M，Process time 为 51s。

```
# Query 26: 0.08 QPS, 0.34x concurrency, ID 0x2AA5E845384665393DC48BB36364AC51 at byte 140924321
# This item is included in the report because it matches --outliers.
# Scores: V/M = 4.20
# Time range: 2019-10-15T12:14:48 to 2019-10-15T14:45:55
# Attribute      pct     total      min      max      avg     95%    stddev   median
# ======      ===     ===     ===     ===     ===     ===     ===     ===     ===
# Count          0      739
# Exec time     0     3039s     300ms     28s      4s     12s      4s      3s
# Query size    0    22.37k      31       31      31      31      0      31
# Conn ID       0    559.72k      15     7.32k    775.58    4.71k    1.32k   299.03
# Cop proc avg  0     50.76     0.04     0.14     0.07     0.09     0.01     0.07
# Cop proc max  0     50.76     0.04     0.14     0.07     0.09     0.01     0.07
# Cop proc p90  0     50.76     0.04     0.14     0.07     0.09     0.01     0.07
# Cop wait avg  0     2.30k      0     24.03     3.19     8.91     3.15     2.39
# Cop wait max  0     2.30k      0     24.03     3.19     8.91     3.15     2.39
# Cop wait p90  0     2.30k      0     24.03     3.19     8.91     3.15     2.39
# Mem max        0   699.31M   968.95k   969.37k   969.01k   961.27k      0   961.27k
# Num cop task  0      739       1       1       1       1       0       1
# Process keys  0    22.09M     157    32.17k    30.60k    31.59k    6.77k   31.59k
# Process time  0      51s      35ms    138ms    69ms     91ms     15ms    68ms
# Request coun  0      739       1       1       1       1       0       1
# Total keys   0    22.43M     197    32.88k    31.08k    31.59k    6.76k   31.59k
# Txn start ts 0   263.99E   365.80P   365.81P   365.80P   1.25P      0   1.25P
# Wait time    0    2355s     1ms     24s      3s      9s      3s      3s
# String:
# Cop proc add 0.245:20171 (392/53%)... 2 more
# Cop wait add 0.245:20171 (392/53%)... 2 more
# DB
# Digest      769ecfd1640a6a79558c26dale5b64a2b68d65919b23...
# Is internal false
# Stats        grp:411858... (100/13%), grp:411859... (95/12%)... 20 more
# Succ         true
```

下面 SQL 的 Process keys 为 12.68M，但是 Process time 高达 142353s。

```

# Query 4: 0.36 QPS, 14.84x concurrency, ID 0xF0ECDD453DA544E79C7D0E42506FF4BE at byte 163086266
# This item is included in the report because it matches --limit.
# Scores: V/M = 13.91
# Time range: 2019-10-15T12:13:52 to 2019-10-15T15:20:49
# Attribute      pct    total      min      max      avg     95%   stddev   median
# ====== ====== ====== ====== ====== ====== ====== ====== ======
# Count          1    3999
# Exec time     11 166406s      3s    170s     42s     80s     24s     39s
# Query size    1 515.50k     132     132     132     132      0     132
# Conn ID       0   2.22M      10    7.32k   583.33    2.06k   1.03k   258.32
# Cop proc avg  32   2.53k     0.17    1.34     0.65     0.99     0.17     0.61
# Cop proc max  35   7.15k     0.45    3.60     1.83     2.76     0.49     1.70
# Cop proc p90  34   6.44k     0.34    3.32     1.65     2.63     0.46     1.62
# Cop wait avg  1   8.10k     0.00    8.48     2.07     4.50     1.41     1.96
# Cop wait max  3   30.67k     0.00   36.06     7.85   16.81     5.21     6.98
# Cop wait p90  1   18.28k     0.00   23.68     4.68     9.83     3.04     4.50
# Mem max       39 119.23G   30.27M   30.54M   30.53M   29.99M      0   29.99M
# Num cop task 18 214.79k     51     55   55.00     54.21     0.08   54.21
# Process keys  0   12.68M   3.18k     3.36k   3.25k     3.19k   57.63   3.19k
# Process time  61 142353s     9s     74s     36s     54s     10s     33s
# Request count 18 214.79k     51     55   55.00     54.21     0.08   54.21
# Total keys    0   13.00M   3.23k     3.50k   3.33k     3.35k   69.99   3.19k
# Txn start ts  1   1.40Z 365.80P 365.81P 365.80P   1.25P      0   1.25P
# Wait time     24 456039s     6ms    466s    114s    246s     78s    107s
# String:
# Cop proc add [REDACTED] .245:20171 (1814/45%)... 2 more
# Cop wait add [REDACTED] .245:20171 (1835/45%)... 2 more
# DB
# Digest        473bf876de8bb5210bf4b300a6b3d94b0c0210balale...
# Is internal   false
# Stats          tpl:411850... (202/5%), tpl:411850... (192/4%)... 196 more
# Succ          true

```

过滤 Process time 较多的 SQL，发现 3 个典型的 slow query，分析具体的执行计划。

- SQL1

```
select a.a_id, a.b_id, uqm.p_id from a join hsq on a.b_id=hsq.id join uqm on a.a_id=uqm.id;
```

```

id count task operator info execution info memory
Projection_9 1878.75 root
└─IndexJoin_13 1878.75 root inner join, inner:TableReader_12, outer key:[REDACTED], inner key:[REDACTED]
  └─IndexJoin_31 1878.75 root inner join, inner:IndexLookUp_30, outer key:[REDACTED], inner key:[REDACTED]
    └─TableReader_44 1503.00 root data:TableScan_43 time:1.963267ms, loops:3, rows:1503 16.3544921875 KB
      └─TableScan_43 1503.00 cop table:tpl, range:[0,+inf], keep order:false time:1ms, loops:6, rows:1503 N/A
      └─IndexLookUp_30 284.10 root time:1m35.454370598s, loops:422, rows:429136 100.359375 KB
        └─IndexScan_28 284.10 cop table:[REDACTED], index:[REDACTED], range: decided by [eq("
          [REDACTED]), keep order:false proc max:470ms, min:36ms, p80:470ms, p95:470ms, rows:429136, iters:428, tasks:2 N/A
          ], keep order:false proc max:470ms, min:36ms, p80:470ms, p95:470ms, rows:429136, iters:428, tasks:2 N/A
        └─TableScan_29 284.10 cop table:[REDACTED], keep order:false proc max:1.695s, min:30ms, p80:1.291s, p95:1.525s, rows:429136,
          iters:543, tasks:27 N/A
      └─TableReader_12 1.00 root data:TableScan_11 time:1m9.184965119s, loops:100, rows:64302 N/A
        └─TableScan_11 1.00 cop table:[REDACTED], range: decided by [REDACTED], keep order:false proc max:702ms,
          min:4ms, p80:490ms, p95:627ms, rows:64302, iters:165, tasks:25 N/A

```

- SQL2

```
select distinct g.abc, g.def, g.ghi, h.abcd, hi.jq from ggg g left join ggg_host gh on g.id = gh.ggg_id left join host h on gh.a_id = h.id left join a_jq hi on h.id = hi.hid where h.abcd is not null and h.abcd <> '' and hi.jq is not null and hi.jq <> '';
```

```

id count task operator info execution info memory
HashAgg_13 80296.47 root group by:dbl.g.def, dbl.g.abc, dbl.g.ghi, dbl.h.abcd, dbl.h.i.jq, funcs:firstrow(dbl.g.abc),
firstrow(dbl.g.def), firstrow(dbl.g.ghi), firstrow(dbl.h.abcd), firstrow(dbl.h.i.jq) time:8.646954728s, loops:347, rows:354274 N/A
└─IndexJoin_18 80296.47 root inner join, inner:IndexReader_17, outer key:dbl.h.i.hid, inner key:dbl.h.i.hid time:7.613316688s,
loops:507, rows:517734 257.38256072998047 MB
  └─IndexJoin_47 41156.25 root inner join, inner:TableReader_46, outer key:dbl.g.ha_id, inner key:dbl.h.i.hid time:2.76087231s,
loops:656, rows:669536 438.8576126098633 MB
  └─IndexJoin_54 41156.25 root inner join, inner:IndexReader_53, outer key:dbl.g.id, inner key:dbl.gh.ggg_id
    time:1.69100157s, loops:1189, rows:1215334 27.39978790283203 MB
    └─TableReader_66 32925.00 root data:TableScan_65 time:75.325365ms, loops:35, rows:32925 4.20347785949707 MB
      └─TableScan_65 32925.00 cop table:g, range:[0,+inf], keep order:false time:52ms, loops:37, rows:32925 N/A
      └─IndexReader_53 0.99 root index:Selection_52 time:1.898590556s, loops:1202, rows:1215334 15.298088073730469 MB
        └─Selection_52 0.99 cop not(isnull(dbl.gh.ggg_id)) proc max:664ms, min:29ms, p80:317ms, p95:664ms, rows:1215334,
          iters:1218, tasks:10 N/A
        └─IndexScan_51 0.99 cop table:gh, index:ggg_id, a_id, range: decided by [eq(dbl.gh.ggg_id, dbl.g.id)
          not(isnull(dbl.gh.a_id))], keep order:false proc max:640ms, min:29ms, p80:316ms, p95:640ms, rows:1215334, iters:1218, tasks:10 N/A
      └─TableReader_46 0.80 root data:Selection_45 time:3.441797722s, loops:732, rows:666902 N/A
        └─Selection_45 0.80 cop ne(dbl.h.abcd, ""), not(isnull(dbl.h.abcd)), not(isnull(dbl.h.abcd)) proc max:206ms, min:0s,
          p80:10ms, p95:30ms, rows:666902, iters:4012, tasks:1487 N/A
        └─TableScan_44 1.00 cop table:h, range: decided by [dbl.g.ha_id], keep order:false proc max:204ms, min:0s, p80:10ms,
          p95:30ms, rows:668753, iters:4012, tasks:1487 N/A
      └─IndexReader_17 1.20 root index:Selection_16 time:20.824781527s, loops:545, rows:510094 144.6767578125 KB
        └─Selection_16 1.20 cop not(isnull(dbl.hi.hid)) proc max:876ms, min:4ms, p80:728ms, p95:789ms, rows:510094, iters:781, tasks:93
          N/A
        └─IndexScan_15 1.20 cop table:hi, index:hid, jq, source, status, range: decided by [eq(dbl.hi.hid, dbl.h.i.hid)
          not(isnull(dbl.hi.jq)) ne(dbl.hi.jq, )], keep order:false proc max:875ms, min:4ms, p80:728ms, p95:789ms, rows:510094, iters:781,
          tasks:93 N/A

```

- SQL3

```
select tb1.mt, tb2.name from tb2 left join tb1 on tb2.mtId=tb1.id where tb2.type=0 and (tb1.mt is not null and tb1.mt != '') and (tb2.name is not null or tb2.name != '');
```

```
id count task operator info execution info memory
Projection_6 121393.07 root dbl.tbl.mt, dbl.tb2.name time:713.605177ms, loops:14, rows:10753 N/A
└─HashLeftJoin_11 121393.07 root inner join, inner:TableReader_26, equal:[eq(dbl.tb2.mtid, dbl.tbl.id)] time:713.544424ms,
  loops:14, rows:10753 2.650096893310547 MB
  └─IndexLookup_23 121393.07 root time:621.192752ms, loops:120, rows:121269 2.6519317626953125 MB
    └─IndexScan_20 121393.07 cop table:tb2, index:type, range:[0,0], keep order:false time:69ms, loops:123, rows:121269 N/A
      └─Selection_22 121393.07 cop or(not(isnull(dbl.tb2.name)), ne(dbl.tb2.name, "")) proc max:390ms, min:9ms, p80:368ms, p95:390ms,
        rows:121269, iters:166, tasks:10 N/A
        └─TableScan_21 121393.07 cop table:tb2, keep order:false proc max:387ms, min:8ms, p80:366ms, p95:387ms, rows:121269,
          iters:166, tasks:10 N/A
    └─TableReader_26 41200.00 root data:Selection_25 time:171.796012ms, loops:42, rows:41200 2.2558584213256836 MB
      └─Selection_25 41200.00 cop ne(dbl.tbl.mt, ""), not(isnull(dbl.tbl.mt)) proc max:156ms, min:11ms, p80:156ms, p95:156ms,
        rows:41200, iters:53, tasks:3 N/A
        └─TableScan_24 41200.00 cop table:tbl, range:[-inf,+inf], keep order:false proc max:152ms, min:11ms, p80:152ms, p95:152ms,
          rows:41200, iters:53, tasks:3 N/A
```

分析执行计划未发现异常，查看相关表的统计信息也都没有过期，继续分析 TiDB 和 TiKV 日志。

### 常规日志分析

查看 TiKV 日志中标记为 [slow-query] 的日志行中的 region 分布情况。

```
more tikv.log.2019-10-16-06\:28\:13 |grep slow-query |awk -F '[' '{print $1}' | awk '{print $6}' | sort | uniq -c | sort -n
```

找到访问频率最大的 3 个 region。

```
73 29452
140 33324
757 66625
```

这些 region 的访问次数远远高于其它 region，之后定位这些 region 所属的表名。首先查看 [slow-query] 所在行记录的 table\_id 和 start\_ts，然后查询 TiDB 日志获取表名，比如 table\_id 为 1318，start\_ts 为 411837294180565013，使用如下命令过滤，发现是上述慢查询 SQL 涉及的表。

```
more tidb-2019-10-14T16-40-51.728.log | grep '"/[1318/]"' |grep 411837294180565013
```

### 解决

对这些 region 做 split 操作，以 region 66625 为例，命令如下（需要将 x.x.x.x 替换为实际的 pd 地址）。

```
pd-ctl -u http://x.x.x.x:2379 operator add split-region 66625
```

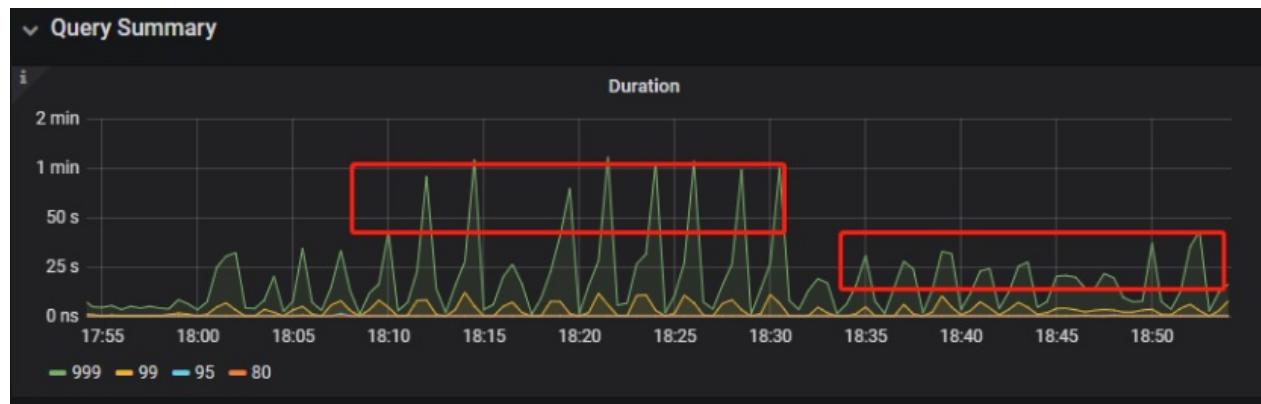
### 操作后查看 PD 日志

```
[2019/10/16 18:22:56.223 +08:00] [INFO] [operator_controller.go:99] ["operator finish"] [region-id=30796] [operator="\"admin-split-region (kind:admin, region:66625(1668,3), createAt:2019-10-16 18:22:55.888064898 +0800 CST m=+110918.823762963, startAt:2019-10-16 18:22:55.888223469 +0800 CST m=+110918.823921524, currentStep:1, steps:[split region with policy SCAN]) finished\""]
```

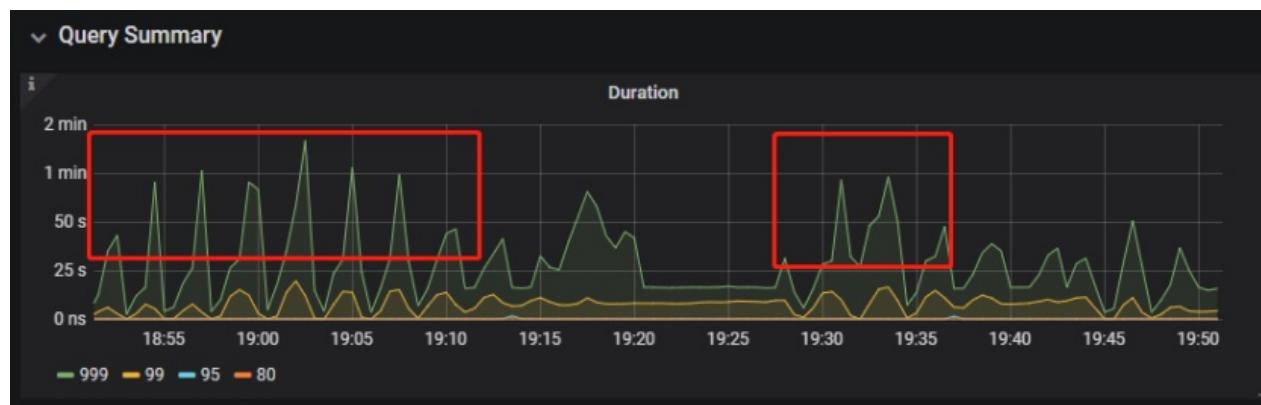
日志显示 region 已经分裂完成，之后查看该 region 相关的 slow-query。

```
more tikv.log.2019-10-16-06\:28\:13 |grep slow-query | grep 66625
```

观察一段时间后确认 66625 不再是热点 region，继续处理其它热点 region。所有热点 region 处理完成后，监控 Query Summary - Duration 显著降低。



Duration 稳定了保持一段时间，18:55 之后仍然有较高的 Duration 出现。



观察压力较重的 tikv，移走热点 region 的 leader。

```
pdctl -u http://x.x.x.x:2379 operator add transfer-leader 1 2 //把 region1 的 leader 调度到 store2
```

leader 迁走之后，原 TiKV 节点的 Duration 立刻下降，但是迁移到新 TiKV 节点的 Duration 随之上升。



之后多次对热点 region 进行 split 操作，最终 Duration 明显下降并恢复稳定。



### 案例总结

对于分布式数据库的读热点问题，有时难以通过优化 SQL 的方式解决，需要分析整个 TiDB 集群的监控和日志来定位原因。严重的读热点可能导致部分 TiKV 达到资源瓶颈，这种短板效应限制了整个集群性能的充分发挥，通过分裂 region 的方式可以将热点 region 分散到更多的 TiKV 节点上，让每个 TiKV 的负载尽可能达到均衡，缓解读热点对 SQL 查询性能的影响。更多热点问题的处理思路可以参考第四部分 7.2 节。

## 案例5 SQL 执行计划不准

### 背景

- SQL 执行时间突然变长

### 分析

- SQL 语句

```
select count(*)
from todbs.bus_jijin_trade_record a, todbs.bus_jijin_info b
where a.fund_code=b.fund_code and a.type in ('PURCHASE', 'APPLY')
and a.status='CANCEL_SUCCESS' and a.pay_confirm_status = 1
and a.cancel_app_no is not null and a.id >= 177045000
and a.updated_at > date_sub(now(), interval 48 hour) ;
```

执行结果，需要 1 分 3.7s：

```
mysql> select count(*)
-> from todbs.bus_jijin_trade_record a, todbs.bus_jijin_info b
-> where a.fund_code=b.fund_code and a.type in ('PURCHASE', 'APPLY')
-> and a.status='CANCEL_SUCCESS' and a.pay_confirm_status = 1
-> and a.cancel_app_no is not null and a.id >= 177045000
-> and a.updated_at > date_sub(now(), interval 48 hour) ;
+-----+
| count(*) |
+-----+
|      708 |
+-----+
1  row in set (1 min 3.77 sec)
```

- 索引信息

表名	数据行	索引名
bus_jijin_trade_record	176384036	PRIMARY KEY ( ID ) KEY idx_bus_jijin_trade_record_upd ( UPDATED_AT )
bus_jijin_info	6442	PRIMARY KEY ( ID )

- 查看执行计划

```

mysql> explain
-> select count(*)
-> from todbs.bus_jijin_trade_record a, todbs.bus_jijin_info b
-> where a.fund_code=b.fund_code and a.type in ('PURCHASE','APPLY')
-> and a.status='CANCEL_SUCCESS' and a.pay_confirm_status = 1
-> and a.cancel_app_no is not null and a.id >= 177045000
-> and a.updated_at > date_sub(now(), interval 48 hour) ;
+-----+-----+-----+
| id      | count | task | operator info
+-----+-----+-----+
| StreamAgg_13 | 1.00 | root | funcs:count(1)
|   |
|   | HashRightJoin_27 | 421.12 | root | inner join, inner:TableReader_18, equal:[eq(a.fund_code, b.fund_code)]
|   |   | TableReader_18 | 421.12 | root | data:Selection_17
|   |   |   | Selection_17 | 421.12 | cop | eq(a.pay_confirm_status, 1), eq(a.status, "CANCEL_SUCCESS"), gt(a.updated_at, 2020-03-03 22:31:08), in(a.type, "PURCHASE", "APPLY"), not(isnull(a.cancel_app_no))
|   |   |   | TableScan_16 | 145920790.55 | cop | table:a, range:[177045000,+inf], keep order:false
|   |   |   | TableReader_37 | 6442.00 | root | data:TableScan_36
|   |   |   | TableScan_36 | 6442.00 | cop | table:b, range:[-inf,+inf], keep order:false
+-----+-----+-----+

```

TableScan\_16, TableScan\_36 : 表示在 TiKV 端分别对表 a 和 b 的数据进行扫描，其中 TableScan\_16 扫描了 1.46 亿的行数；

Selection\_17 : 表示满足表 a 后面 where 条件的数据；

TableReader\_37 : 由于表 b 没有独立的附加条件，所以直接将这部分数据返回给 TiDB；

TableReader\_18 : 将各个 coprocessor 满足 a 表条件的结果返回给 TiDB；

HashRightJoin\_27 : 将 TableReader\_37 和 TableReader\_18 上的结果进行 hash join；

StreamAgg\_13 : 进一步统计所有行数，将数据返回给客户端；

可以看到语句中 a 表(bus\_jijin\_trade\_record)的条件 id >= 177045000 , 和 updated\_at > date\_sub(now(), interval 48 hour)上，这两个列分别都有索引，但是 TiDB 还是选择了全表扫描。

按照上面两个条件分别查询数据分区情况

```

mysql> SELECT COUNT(*) FROM todbs.bus_jijin_trade_record WHERE id >= 177045000 ;
+-----+
| COUNT(*) |
+-----+
| 145917327 |
+-----+
1 row in set (16.86 sec)

mysql> SELECT COUNT(*) FROM todbs.bus_jijin_trade_record WHERE updated_at > date_sub(now(), interval 48 hour) ;
+-----+
| COUNT(*) |
+-----+
| 713682 |
+-----+

```

可以看到，表 bus\_jijin\_trade\_record 有 1.7 亿的数据量，应该走 updated\_at 字段上的索引。

使用强制 hint 进行执行，6.27 秒就执行完成了，速度从之前 63s 到现在的 6.3s，提升了 10 倍。

```
mysql> select count(*)
-> from tod.s.bus_jijin_trade_record a  use index(idx_bus_jijin_trade_record_upt), tod.s.bus_jijin_info b
-> where a.fund_code=b.fund_code and a.type in ('PURCHASE','APPLY')
-> and a.status='CANCEL_SUCCESS' and a.pay_confirm_status = 1
-> and a.cancel_app_no is not null and a.id >= 177045000
-> and a.updated_at > date_sub(now(), interval 48 hour) ;
+-----+
| count(*) |
+-----+
|    709 |
+-----+
1 row in set (6.27 sec)
```

## 强制 hint 后的执行计划

```
mysql> explain
-> select count(*)
-> from tod.s.bus_jijin_trade_record a  use index(idx_bus_jijin_trade_record_upt), tod.s.bus_jijin_info b
-> where a.fund_code=b.fund_code and a.type in ('PURCHASE','APPLY')
-> and a.status='CANCEL_SUCCESS' and a.pay_confirm_status = 1
-> and a.cancel_app_no is not null and a.id >= 177045000
-> and a.updated_at > date_sub(now(), interval 48 hour) ;
+-----+-----+-----+
| id      | count   | task  | operator info
+-----+-----+-----+
| StreamAgg_13 | 1.00    | root  | funcs:count(1)
|           |          |       |
| └HashRightJoin_24 | 421.12   | root  | inner join, inner:IndexLookUp_20, equal:[eq(a.fund_code, b.fun
d_code)]
|   └IndexLookUp_20 | 421.12   | root  |
|   |   └Selection_18 | 146027634.83 | cop  | ge(a.id, 177045000)
|   |   |   └IndexScan_16 | 176388219.00 | cop  | table:a, index:UPDATED_AT, range:(2020-03-03 23:05:30,+inf], k
eep order:false
|   |   |   └Selection_19 | 421.12   | cop  | eq(a.pay_confirm_status, 1), eq(a.status, "CANCEL_SUCCESS"), i
n(a.type, "PURCHASE", "APPLY"), not(isnull(a.cancel_app_no)) |
|   |   └TableScan_17 | 146027634.83 | cop  | table:bus_jijin_trade_record, keep order:false
|   └TableReader_31 | 6442.00  | root  | data:TableScan_30
|   └TableScan_30 | 6442.00  | cop  | table:b, range:[-inf,+inf], keep order:false
+-----+-----+-----+
```

使用 hint 后的执行计划，预估 updated\_at 上的索引会扫描 176388219，索引选择了全表扫描，可以判定是由于错误的统计信息导致执行计划有问题。

查看表 bus\_jijin\_trade\_record 上的统计信息。

```
mysql> show stats_meta where table_name like 'bus_jijin_trade_record' and db_name like 'tods';
+-----+-----+-----+-----+
| Db_name | Table_name        | Update_time      | Modify_count | Row_count |
+-----+-----+-----+-----+
| tod.s | bus_jijin_trade_record | 2020-03-05 22:04:21 | 10652939 | 176381997 |
+-----+-----+-----+-----+

mysql> show stats_healthy where table_name like 'bus_jijin_trade_record' and db_name like 'tods';
+-----+-----+
| Db_name | Table_name        | Healthy |
+-----+-----+
| tod.s | bus_jijin_trade_record | 93 |
+-----+-----+
```

根据统计信息，表 bus\_jijin\_trade\_record 有 176381997，修改的行数有 10652939，该表的健康度为：(176381997-10652939)/176381997 \*100=93。

### 解决

- 重新收集统计信息

```
mysql> set tidb_build_stats_concurrency=10;
Query OK, 0 rows affected (0.00 sec)

#调整收集统计信息的并发度，以便快速对统计信息进行收集
mysql> analyze table todbs.bus_jijin_trade_record;
Query OK, 0 rows affected (3 min 48.74 sec)
```

- 查看没有使用 hint 语句的执行计划

```
mysql> explain select count(*)
-> from todbs.bus_jijin_trade_record a, todbs.bus_jijin_info b
-> where a.fund_code=b.fund_code and a.type in ('PURCHASE','APPLY')
-> and a.status='CANCEL_SUCCESS' and a.pay_confirm_status = 1
-> and a.cancel_app_no is not null and a.id >= 177045000
-> and a.updated_at > date_sub(now(), interval 48 hour) ;;
+-----+-----+-----+
| id | count | task | operator info
+-----+-----+-----+
| StreamAgg_13 | 1.00 | root | funcs:count(1)
|   |
|   | \HashRightJoin_27 | 1.99 | root | inner join, inner:IndexLookUp_23, equal:[eq(a.fund_code, b.fund_code)]
|   |   | IndexLookUp_23 | 1.99 | root |
|   |   |   |
|   |   |   | Selection_21 | 626859.65 | cop | ge(a.id, 177045000)
|   |   |   |   |
|   |   |   |   | IndexScan_19 | 757743.08 | cop | table:a, index:UPDATED_AT, range:(2020-03-03 23:28:14,+inf], keep_order:false
|   |   |   |   |   |
|   |   |   |   | Selection_22 | 1.99 | cop | eq(a.pay_confirm_status, 1), eq(a.status, "CANCEL_SUCCESS"), in(a.type, "PURCHASE", "APPLY"), not(isnull(a.cancel_app_no)) |
|   |   |   |   |   | TableScan_20 | 626859.65 | cop | table:bus_jijin_trade_record, keep_order:false
|   |   |   |   |   |
|   |   |   |   | TableReader_37 | 6442.00 | root | data:TableScan_36
|   |   |   |   |
|   |   |   |   | TableScan_36 | 6442.00 | cop | table:b, range:[-inf,+inf], keep_order:false
|   |   |
+-----+-----+-----+
9 rows in set (0.00 sec)
```

可以看到，收集完统计信息后，现在的执行计划走了索引扫描，与手动添加 hint 的行为一致，且扫描的行数 757743 符合预期。

此时执行时间变为 1.69s，在执行计划没变的情况下，应该是由于缓存命中率上升带来的提升。

```
mysql> select count(*)
-> from todbs.bus_jijin_trade_record a, todbs.bus_jijin_info b
-> where a.fund_code=b.fund_code and a.type in ('PURCHASE','APPLY')
-> and a.status='CANCEL_SUCCESS' and a.pay_confirm_status = 1
-> and a.cancel_app_no is not null and a.id >= 177045000
-> and a.updated_at > date_sub(now(), interval 48 hour) ;
+-----+
| count(*) |
+-----+
|    712 |
+-----+
1 row in set (1.69 sec)
```

### 案例总结

可以看出该 SQL 执行效率变差是由于统计信息不准确造成的，在通过收集统计信息之后得到了正确的执行计划。

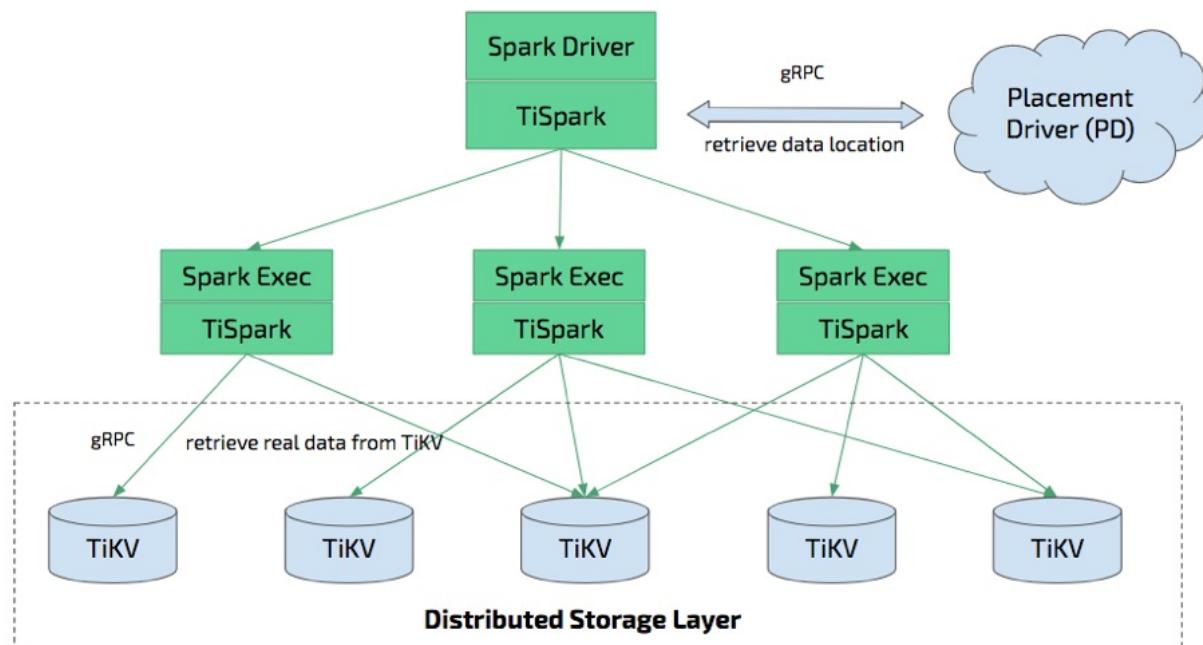
从最终结果 712 行记录来看，创建联合索引可以更大的降低扫描数据的量，更进一步提升性能。在性能已经满足业务要求情况下，联合索引会有额外的成本，留待以后尝试。

## 6.3 TiDB + TiSpark 跑批最佳实践

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。在借助 Spark 平台在计算及生态等方面优势的同时也融合了 TiKV 分布式集群的优势，为大数据环境下的批量任务提供了一种解决方案。在本节将介绍如何在已有的 TiDB 集群基础上引入 TiSpark 进行批量任务开发。本节假设你对 Spark 有基本认知。你可以参阅 [Apache Spark 官网](#) 了解 Spark 相关信息。

### 6.3.1 TiSpark 概述

TiSpark 是将 Spark SQL 直接运行在 TiDB 存储引擎 TiKV 上的 OLAP 解决方案。TiSpark 架构图如下：



- TiSpark 深度整合了 Spark Catalyst 引擎，可以对计算提供精确的控制，使 Spark 能够高效的读取 TiKV 中的数据，提供索引支持以实现高速的点查；
- 通过多种计算下推减少 Spark SQL 需要处理的数据大小，以加速查询；利用 TiDB 内建的统计信息选择更优的查询计划。
- 从数据集群的角度看，TiSpark+TiDB 可以让用户无需进行脆弱和难以维护的 ETL，直接在同一个平台进行事务和分析两种工作，简化了系统架构和运维。
- 除此之外，用户借助 TiSpark 项目可以在 TiDB 上使用 Spark 生态圈提供的多种工具进行数据处理。例如使用 TiSpark 进行数据分析和 ETL；使用 TiKV 作为机器学习的数据源；借助调度系统产生定时报表等等。

### 6.3.2 环境准备

#### (1) TiSpark 依赖包

当前，TiSpark 2.1.8 是最新的稳定版本，官方强烈建议使用。它与 Spark 2.3.0+ 和 Spark 2.4.0+ 兼容。它还与 TiDB-2.x 和 TiDB-3.x 兼容。可以前往 TiSpark 在 GitHub 上的[首页](#)查看更详细的版本兼容情况并下载。

## (2) Spark

根据现有 TiDB 版本确定兼容的 TiSpark 依赖以后，便可以从 Spark 官网下载支持的版本，这里推荐[下载](#)自带 Hadoop 环境的预编译版。

## (3) JDK

TiSpark 需要 JDK 1.8+ 以及 Scala 2.11 (Spark2.0+ 默认 Scala 版本)。

### 6.3.3 TiSpark 集群部署配置

TiSpark 可以在 YARN，Mesos，Standalone 等任意 Spark 模式下运行。这里使用 Spark Standalone 方式部署。关于 Spark Standalone 的具体配置方式请参考官方说明，下面给出的是与 TiSpark 相关的配置范例。

#### (1) spark-env.sh 配置

```
SPARK_EXECUTOR_MEMORY=10g
SPARK_EXECUTOR_CORES=5
SPARK_WORKER_MEMORY=40g
SPARK_WORKER_CORES=20
```

#### (2) spark-defaults.conf 配置

```
spark.sql.extensions org.apache.spark.sql.TiExtensions
spark.tispark.pd.addresses 127.0.0.1:2379
```

其中 PD 格式为 地址:端口号，多个 PD 使用逗号间隔。

## (3) 部署 TiSpark

将 TiSpark 组件部署到 Spark 集群有两种方式，如果不想重启现有集群，可以使用 Spark 的 --jars 参数将 TiSpark 作为依赖引入：

```
sh spark-shell --jars $TISPARK_FOLDER/tispark-${name_with_version}.jar
```

如果想将 TiSpark 作为默认组件部署，只需要将 TiSpark 的 jar 包放进 Spark 集群每个节点的 jars 路径并重启 Spark 集群：

```
 ${SPARK_HOME}/jars
```

## (4) 启动 TiSpark 集群

在选中的 Spark Master 节点执行如下命令：

```
cd ${SPARK_HOME}
./sbin/start-all.sh
```

命令执行以后，控制台会输出 master 和 slave 的启动信息并指出相应 log 文件。检查 log 文件确认集群各个节点是否启动成功。可以打开 <http://spark-master-hostname:8080> 查看集群信息（Spark-Master 默认的 web 端口号）。

## 6.3.4 使用范例

假设你已经按照上面的步骤成功部署并启动了 TiSpark 集群，下面分别介绍使用 Spark-Shell 和 Spark-Submit 两种方式来作 OLAP 分析。

### (1) Spark-Shell

如果你的 TiSpark 版本是 2.0 以上，那么在 Spark-Shell 中你可以直接调用 Spark SQL 与 TiDB 数据库交互：

```
spark.sql("use test")
spark.sql("select count(*) from user").show
```

TiSpark 版本在 2.0 以前的需要在之前执行如下命令：

```
import org.apache.spark.sql.TiContext
val ti = new TiContext(spark, List("127.0.0.1:2379"))
ti.tidbMapDatabase("test")
```

之后便可以像上面那样调用 Spark SQL，不过建议尽量使用 2.0 以上版本的 TiSpark。

### (2) Spark-Submit

在实际开发中，Spark-Shell 多用于测试，更多时候需要将代码打包后使用 Spark-Submit 命令提交到 TiSpark 集群。

如果你的工程是采用 Maven 构建的，需要在 POM 文件中引入 Spark 及 TiSpark 依赖，由于这些依赖在 Spark 集群上已经存在，需要将他们的依赖范围设置为 Provided。

```
<dependencies>
<dependency>
  <groupId>com.pingcap.tispark</groupId>
  <artifactId>tispark-core</artifactId>
  <version>2.1.8-spark_${spark.version}</version>
  <scope>provided<scope>
</dependency>

<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.11</artifactId>
  <version>${spark.version}</version>
  <scope>provided<scope>
</dependency>

<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>${spark.version}</version>
  <scope>provided<scope>
</dependency>
</dependencies>
```

配置好必要的依赖以后，便可以使用如下代码初始化一个 SparkSession 对象，Spark 提供了面向多种语言的 API，如 Scala、Python、Java 等，本节均以 Java 为例。

```

SparkSession sc = SparkSession
    .builder()
    .appName("TiSpark example")
    .master("local")
    .config("spark.sql.extensions", "org.apache.spark.sql.TiExtensions")
    .config("spark.tispark.pd.addresses", "127.0.0.1:2379")
    .getOrCreate();

```

上面的代码支持你在本地调试你的 TiSpark 程序，如果你要打包到 Spark 集群运行，那么指定 master 和 TiSpark 的两个配置项都是不需要的。接下来我们结合一个常见的场景展示一个批量任务案例：每天有一些交易文件需要落表，它包含交易双方和金额信息，之后需要基于该表进行分析。在这个案例中可以拆解出两个最基本的批量任务。

一个任务负责解析每天的文件并将数据写入目标表，目前 TiSpark 不支持直接将数据写入 TiDB，但可以采用 Spark 原生 JDBC 方式写入，Java 案例代码如下：

```

sc.read().schema(getStructType())
    .option("delimiter",true)
    .option("header",true)
    .csv(filePath)
    .withColumn("input_time", functions.current_timestamp())
    .repartition(100)
    .write().mode(SaveMode.Append)
    .jdbc(url,tableName,connProperties);

public StructType getStructType(){
    List<StructField> fields=new ArrayList<>();
    StructField transferAccount = DataTypes.createStructField("transferAccount",DataTypes.StringType,false);
    StructField receiveAccount = DataTypes.createStructField("receiveAccount",DataTypes.StringType,false);
    StructField amount = DataTypes.createStructField("amount",DataTypes.createDecimalType(10,2),false);
    fields.add(transferAccount);
    fields.add(receiveAccount);
    fields.add(amount);
    return DataTypes.createStructType(fields);
}

```

在这段代码中为顺利解析文件而定义了被解析文件的格式和字段属性，随后为数据集添加了一个标志入库时间的列，最后将数据集以追加的方式写入到目标表。关于写入 TiDB，有以下几点需要注意：

1. 为了获得更好的写入效率并充分利用 Spark 集群资源，在写入之前最好使用 repartition 算子对数据进行再分区，充分发挥 Spark 集群的计算能力，具体数值要根据任务数据量和集群资源来决定。
2. 要想让数据进行批量写入，TiDB 连接串中需要跟上 rewriteBatchedStatements 参数并将其设置为 True，然后通过 JDBCOptions.JDBC\_BATCH\_INSERT\_SIZE 参数去控制批量写入的大小，官方推荐的大小为 150。
3. 对于大量数据写入，推荐将事务隔离级别参数 isolationLevel 设置为 NONE。

另一个任务从数据库读取数据进行分析，一段简单的 Java 分析代码如下：

```

sc.sqlContext()
    .udf()
    .register("convert",newConvertUDF(),DataTypes.createDecimalType(10,2));
sc.sql("select transferAccount,receiveAccount,convert(amount) from tableName where receiveAccount='0001'");

```

上面的代码从表中筛选出账户号为 0001 的所有入账信息，其中 convert 是一个提供单位转换功能的自定义 UDF 函数，他在被注册到 Job 后可以直接在 Spark SQL 中使用。Spark SQL 支持各种自定义 UDF 或 UDAF 函数，合理利用这个特性可以使你的 SQL 更加强大。TiSpark 支持常见的各种 SQL 语法，如连接和子查询，不过目前 TiSpark 暂时不支持 update 和 delete 的操作，后续会考虑支持这两个操作。

在开发完毕并将代码打包后，使用 Spark-Submit 命令将任务提交到 Spark 集群：

```
cd ${SPARK_HOME}
./bin/spark-submit \
--class Analyze \
--master spark://127.0.0.1:7077 \
/home/tispark/TiSparkExample.jar
```

关于 Spark-Submit 命令的更多参数，请参考 [Spark 官网](#)。

## 6.3.5 小结

本节介绍了如何在现有 TiDB 集群基础上部署和配置 TiSpark 集群，并通过一些简单案例展示了如何使用 TiSpark 组件进行批量任务开发。你应该发现了，使用 TiSpark 开发与原生 Spark 相比没有什么差别，如果你本就熟悉 Spark 的话会很快上手，这正如 TiSpark 官方所说的那样，TiSpark 只是 Spark 之上的一个薄层。

如果你想使用 TiSpark 支撑批量任务的话，目前为止还是不够完善的，至少还有两个方面的问题需要解决。

首先在案例中是通过手动执行 Spark-Submit 命令提交任务的，这种方式只能用于调试不能用于生产，它也无法帮助你管理和监控批量任务，你需要引入类似 Oozie 和 Azkaban 这样的批量工作流任务调度器，他们能帮助你管理批量任务并轻松做到定时运行或是组合多个任务这样的高级功能。如果有条件可以选择自主开发，毕竟适合自己的才是最好的。

另一个问题是，使用 Spark API 进行开发虽然简单但也无法适应各种类型的批量任务，而 Spark 任务参数众多，有时需要根据每个任务的具体情况随时做出调整，这就使得你必须有某种参数解析机制来保证在批量任务运行时动态调整。所以你还需要在 Spark API 基础上设计一套适合你业务需求的批量任务代码规范，这样你就能保证批量任务的规范性和可靠性，同时也能减少冗余代码达到简化开发的目的。

## 6.4 分区表实践及问题处理

随着数据表存储的数据量越来越大，有时会超过亿或者百亿级别，对于这些大表的历史数据删除需求时，特别的不方便，如果采用 DELETE 语句来删除的话，一是可能会线上正常业务造成影响，二是删除的速度太慢，针对这些问题，分区表应运而生，大家比较熟悉的就是基于时间维度的 Range 分区表，通过对分区的 DDL 操作来快速的删除数据，提高了处理数据的效率，避免了对线上业务的影响。

## 6.4.1 TiDB 分区表简介

TiDB 是从 2.1 版本开始支持分区表，3.0 版本开始成熟使用，最新的 4.0 版本做了一些 Bug 修复和分区裁剪方面的增强和优化。

### 6.4.1.1 分区类型

当前支持的类型包括 Range 分区和 Hash 分区，不支持 MySQL 的 List 分区和 Key 分区。

Range 分区是指将数据行按分区表达式计算的值都落在给定的范围内。在 Range 分区中，你必须为每个分区指定值的范围，并且不能有重叠，通过使用 VALUES LESS THAN 操作进行定义。目前只支持单列的 Range 分区表。

Hash 分区主要用于保证数据均匀地分散到一定数量的分区里面。在 Hash 分区中，你只需要指定分区的数量。使用 Hash 分区时，需要在 CREATE TABLE 后面添加 PARTITION BY HASH (expr) PARTITIONS num，其中：expr 是一个返回整数的表达式，它可以是一个列名，但这一列的类型必须整数类型；num 是一个正整数，表示将表划分为多少个分区。

### 6.4.1.2 约束和限制

#### 1. 建表限制

建立主键和唯一键时必须包含分区表达式中用到的所有列，以 Range 分区举例说明：

```
CREATE TABLE employees_attendance (
    id INT NOT NULL AUTO_INCREMENT,
    uid INT NOT NULL,
    name VARCHAR(25) NOT NULL,
    login_date date NOT NULL,
    create_time timestamp NOT NULL COMMENT '打卡时间',
    type tinyint NOT NULL DEFAULT '0' COMMENT '0:上班, 1:下班',
    PRIMARY KEY (`id`, `login_date`),
    UNIQUE KEY `idx_attendance` (`uid`, `login_date`, `type`)
)
PARTITION BY RANGE COLUMNS(login_date) (
    PARTITION p20200306 VALUES LESS THAN ('20200307'),
    PARTITION p20200307 VALUES LESS THAN ('20200308'),
    PARTITION pmax VALUES LESS THAN MAXVALUE
);
```

上表中主键和唯一键中都包括了分区 login\_date 字段，否则就会报如下错误：

```
> ERROR 1503 (HY000): A (PRIMARY KEY/UNIQUE INDEX) must include all columns in the table's partitioning function
```

#### 2. 分区管理和使用方面的限制

只要底层实现可能会涉及数据挪动的操作，TiDB 目前都不支持。包括但不限于：调整 Hash 分区表的分区数量；修改 Range 分区表的范围；合并分区；交换分区等。

使用方面的限制主要指 Load Data 不支持指定分区 load，例如：

```
load local data infile "xxx" into t partition (p1);
```

### 6.4.1.3 TiDB 4.0 对分区表的优化

TiDB 4.0 版本对分区表进行了较多的 Bug 修复、功能增强和性能提升，主要有以下几个方面：

- 稳定性提升，修复了很多 Bug

- 易用性提升，譬如支持 `INFORMATION_SCHEMA.PARTITION` 表，运维人员一般都会基于这个表来获取分区表信息，然后再创建新分区和删除老分区
- 性能提升，主要有两方面：
  - 分区裁剪的优化，所谓分区裁剪就是不需要扫描那些匹配不上的分区
  - 点查优化，下面分别对 Range 和 Hash 分区表做的具体优化进行说明：
    - 对于 Range Partition，在当前的 Expression 框架下，做分区裁剪不太高效，以前做裁剪的计算过程会生成很多中间表达式，计算效率低。现在基本绕开了 Expression，只有常量比较操作可以直接基于 int 比较，以前每个 Partition 都会 Constant Propagate，现在只 Constant Propagate 一次，只把符合 Pattern 的 Expr 选出来，从而提升 Range Partition 的性能。
    - 4.0 版本之前的 Hash Partition 是转成 Range 来实现的，主要的问题是表达式计算的开销很大，而且随着 Partition 的增多，开销会线性增长。在经过优化之后，Hash Partition 会根据给出的查询条件直接对分区表达式进行求值，而不是转化成 Range Partition，这样只经过一次表达式求值就可以算出分区。同时，优化之后的 Hash Partition 也支持了 PointGet 查询计划，对于只有一列且包含在唯一索引中的 Hash Partition 表达式，例如`partition by hash(id)` (`id` 是唯一索引中的一列)，会使用 PointGet 作为查询计划。优化后的 Hash Partition 只支持非常简单的表达式计算，最好只用一列作为 Hash Partition 的表达式，可以减少表达式计算的开销，从而提升性能。

## 6.4.2 TiDB 分区表使用场景

对于业务来讲，在大数据量(至少过千万)，并且表数据选择性查询，使用分区表的场景如下：

- Range 分区可以用于解决业务中大量删除带来的性能问题，支持快速删除分区，TiDB 4.0 虽然事务限制放开，但是 DELETE 数据还是没有直接 DDL 这种直接删除底层的 SST 文件来更快速。
- 使用 Hash 分区表来一定程度上解决写热点问题：Hash 分区则可以用于大量写入场景下的数据打散。
- 对于业务透明，避免了分库分表，还是操作一张表，只不过通过分区表的方式将数据分布到不同分区，也能在一定程度上保证 SQL 性能。
- 运维管理便利，可以单独维护具体分区。
- 典型场景举例：
  - 商业数据分析场景：经常有一些按时间实时写入的广告的点击/曝光日志，用户账户实时消费报表，指标数据实时监控表。因为数据量较大，TiDB 中可以选择保留一个月或者半年，业务程序经常会访问当天分时数据统计，同比上个月的当天，以及去年当天的数据，这个时候只需要根据时间字段创建 Range 分区表，对于以上的需求，只需要访问 3 个分区就可以快速实现统计，因为分区裁剪功能避免了扫描其他分区数据。
  - 审核业务：审核日志表这些都可以按天做 Range 分区。审核 Log 表数据用来统计分析每个审核人员的工作量，可以用作绩效等方面的参考。
  - 基础信息业务：比如一些账户表。由于用户信息太庞大了，并没有明显可以分区的特征字段。可以根据表中 BIGINT 类型的 UID 字段做 Hash 分区，结合分区时配置的分区数量，这样可以把 UID 打散到不同的分区，Hash 分区只能针对整数进行 Hash，从而提高查询的性能。

## 6.4.3 TiDB 分区表最佳实践

- 当业务写入数据有问题，想清理某个分区数据时，不用批量的 DELETE 数据，可以通过 TRUNCATE 命令直接清理分区：

```
ALTER TABLE employees_attendance TRUNCATE PARTITION p20200306;
```

- TiDB 表的统计信息可能不准确，然而 SQL 会因为统计信息准确而选错索引导致 SQL 性能问题，但是如果整个表太大，收集全表的时间太长，解决不了当时的慢 SQL 问题，比如：对于一些按天/周/月度销售额度数据。此时，可以只对 SQL 中用到的分区进行统计信息收集：

```
ALTER TABLE employees_attendance ANALYZE PARTITION p20200306;
```

- 可以使用分区表函数来简化运维，有的程序会用时间戳来存储时间，DBA 再创建新分区时还需要将日期转换时间戳，然后再建立分区。为了避免麻烦，可以使用 TiDB 支持的函数 UNIX\_TIMESTAMP 来搞定：

```
ALTER TABLE ADD PARTITION p20200306 VALUES LESS THAN (UNIX_TIMESTAMP('2020-03-07'))
```

关于 TiDB 支持的可以用于分区表达式的函数，详情见[函数](#)

- Range 分区中 MAXVALUE 表示一个比所有整数都大的整数。避免 Data Maintenance 脚本问题导致的分区没有创建从而影响业务写入，详情可见上面例子。
- Hash 分区使用：最高效的 Hash 函数是作用在单列上，并且函数的单调性是跟列的值是一样递增或者递减的，因为这种情况可以像 Range 分区一样裁剪。不推荐 Hash 分区在表达式中涉及多列。
- 分区表有损调整字段 (TiDB 默认不支持有损更新)，可以通过创建新字段列，将原列的值 UPDATE 到新列，然后 DROP 原字段的方式实现。

## 6.4.4 TiDB 分区表问题处理

1. TiDB 4.0 Fixed 了不少普遍且重要的 Bug , 如果遇到以下问题 , 建议升级到最新的 GA 版本。

- 唯一索引不能创建 [详情](#)。
- 在显式事务中查询对 Table Partition 的查询包含谓词时 , 查询结果不正确的问题 [详情](#) :

```
create table t (a int) partition by hash(a) partitions 4;
begin
insert into t values (0),(1);
select * from t where a>0;
+---+
| a |
+---+
| 1 |
| 0 | -- Bug: filter is a>0
+---+
```

- INSERT ... ON DUPLICATE 语句作用在 Table Partition 时执行失败报错的问题 [详情](#) :

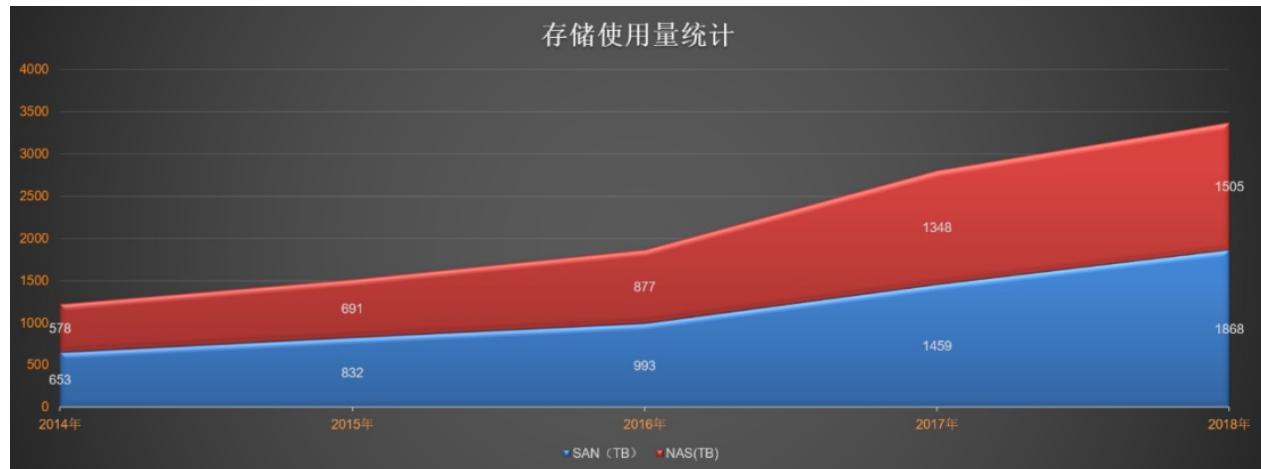
```
create table t1 (a int,b int,primary key(a,b))
partition by range(a)
(partition p0 values less than (100),
partition p1 values less than (1000)
);
insert into t1 set a=1,b=1;
insert into t1 set a=1,b=1 on duplicate key update a=1,b=1;
ERROR 1105:can not be duplicated row, due to old row not found. handle 1 not found
```

- 分区表统计信息收集 Bug , 比如新增一个分区后 , 收集统计信息没有获取这个新分区的元数据 , 导致没有收集 [详情](#)。
- 分区表 LIMIT 没有下推 , 导致千兆网卡跑满 , 引发 TiDB Server OOM Kill [详情](#)。
- OR 查询引发的全表扫描。对每个 Partition 做裁剪的时候 , 会提取带有 Partition Column 的条件 , 但是 CheckScalarFunction 在判断 OR 条件时候 , 如果两个参数有一个不包含 Partition Column , 就会把整个条件扔掉。导致 TiDB 认为每个分区都没有需要裁剪的条件 , 从而选择全表扫描 [详情](#)。

## 6.5 TiDB在企业数据分级存储中的应用

### 6.5.1 数据爆炸增长

随着企业信息化建设工作的深入，企业数据的增长呈现爆炸性的趋势。以某个省电力公司为例，在最近几年里的用于数据存储的设备上线也呈加速增长的态势。



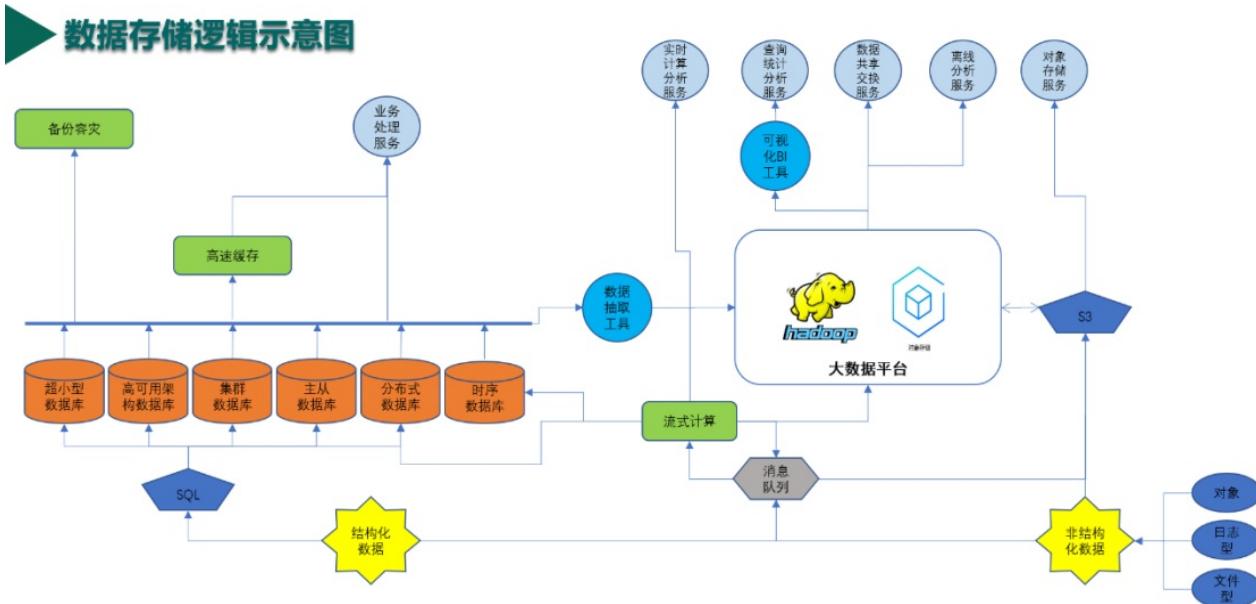
2014 年该企业的 SAN 存储容量与 NAS 存储容量加载一起刚刚突破 1PB，到 2018 年，就已经翻了 3 倍了。从 2017 年开始，出现了更为高速增长的趋势。这些企业数据包括企业管理业务中产生的数据，面向设备采集的数据，也包括企业从外部购买的数据。这些数据大多数都存储在各种数据库系统种。在某省电力公司的已经建设了 180 多套数据库系统，用于存储各种各样的数据，而为了进一步支撑业务的开展，还需要再建设 100 多套数据库系统。这些用传统方式烟囱式建设的数据库系统带来的管理难度已经让运维部门叫苦不迭。

另外一个方面，由于各种数据与应用系统的的建设缺乏统一规划，这 180 多套数据库系统包括了 Oracle、DB2、MySQL、GBASE 8a、Mongodb、达梦、HIVE、HBASE 等，再按数据库的版本统计，已经高达 20 多种。纷乱复杂的数据库种类，不同数据库版本之间的运维与建设差异，更是让运维部门深感头疼。

### 6.5.2 规划统一数据存储的顶层方案

为了在一个新的建设高峰到来之前先厘清现状，做好顶层设计，企业决定进行数据分层存储的顶层设计。首先把家底梳理清楚，然后进行整理分析，确定数据分类分级的标准，然后针对不同类别的数据制定统一的数据存储规范，采用统一的数据存储架构，减少数据库的种类，统一数据库的版本。

经过对企业数据的现状进行梳理，我们发现目前的企业数据存储逻辑架构是这样的：



其中结构化数据主要存储在关系型数据库中，而非结构化数据主要存储在大数据平台和对象存储中。非结构化数据的存储与处理起步较晚，因此规划基本合理，平台也比较统一。最大的问题是结构化数据，不同的业务需求与建设要求并没有在一个统一的顶层框架中考虑，因此不同规模，不同种类的数据库系统在不同的阶段被建设起来。很多数据库系统高峰期的并发访问量不超过 5 个，造成了绝大的资源浪费。另外一个极端是，有些集中式数据库系统虽然已经使用了最为豪华的硬件配置，其处理能力已经严重不足，而企业很快将面临硬件扩无可扩的窘境。

为了解决目前企业的困境，同时为今后几年泛在电力物联网建设所面临的几十 PB 的设备数据接入做好准备，企业决定颠覆原有的数据存储架构，按照数据分级存储的总体远哲，规划统一数据存储的顶层方案。

新的顶层设计从两个方面进行考虑：

- 首先根据系统的特点不同，支持多种分级承载的方案，通过存储分层，为应用提供全面的支撑。
- 其次规范数据库的种类与版本，建立数据库标准化编排能力，逐步实现数据库部署自动化，最终实现全栈自动化编排。

对于传统业务应用系统，数据分为在线数据、历史数据、归档数据等，根据访问特性不同采用不同档次的存储模式，存储方式主要以传统大型数据库系统为主；

- 对于泛在电力物联网应用，数据分为实时在线数据、热在线数据、温在线数据、本地历史数据、分布式历史数据等，不同的数据采用不同存储形式，以分布式方式存储为主，部分明细数据存储于边缘侧;同时考虑到数据大多为结构化数据，主要存储于分布式关系型数据库中。
- 对于互联网特征应用，学习互联网企业的经验，在线数据存储在内存缓冲层，热数据存储于高性能存储设备，冷数据存储于廉价海量分布式存储，数据库采用分布式关系型数据库和分布式内存数据库。

### 6.5.3 关系型数据库规划

在经过重新规划梳理的数据存储架构中，关系型数据库仍然占据较为重要的位置，不过数据库的种类经过整理后将会缩小为 Oracle、MySQL、某种分布式关系型数据库三种。Oracle 数据库仍然承载传统的核心管理类应用系统，MySQL 数据库目前已经企业在业内大规模使用，主要是在云上的小型数据库，分布式关系型数据库的选型是下一步选型的要点。在分布式关系型数据库选型时，根据企业数据分级存储顶层设计的要求，需要备选数据库满足以下的要求：

- (1) 支持 ACID 事务：对于关系型数据库来说 ACID 是基本特性，但是在支持高并发的分布式场景中，在提升扩展性与系统性能的同时保证数据完整性与正确性成为了评估一款分布式数据库系统的重要指标之一。
- (2) 水平弹性扩展：集中式数据库的架构使得数据库成为了整个系统的瓶颈，已经逐渐无法满足海量数据对存储和计算能力的巨大需求。企业选择的分布式数据库系统不仅仅你要求能够做横向的水平弹性扩展，而且需要扩展时对现有生产业务的影响最小化。

(3) 高可用性：首先，分布式数据库系统故障的恢复成本要远远高于普通的数据库系统，因此分布式数据库系统必须支持多副本自动容错；其次，为了进一步提高核心数据的可靠性，对于分布式数据库中的部分高可靠性要求的数据，还需要建立跨机房远程灾备，因此备选的分布式数据库系统必须支持远程增量复制，并且可以选择性复制部分数据。

(4) 为 OLTP 和 OLAP 场景提供一站式的解决方案：随着泛在电力物联网业务的发展，大量的数据需要多次使用，这些数据的量级是 PB 级的，因此通过搬数据的方式建立 OLTP 与 OLAP 两套数据库来满足不同的计算要求的成本太高，分布式关系型数据库必须支持 OLTP/OLAP 混合计算。

(5) 为了减少应用迁移的工作量与今后运维的工作量，备选的分布式数据库系统最好能与 MySQL 具有较好的兼容性。

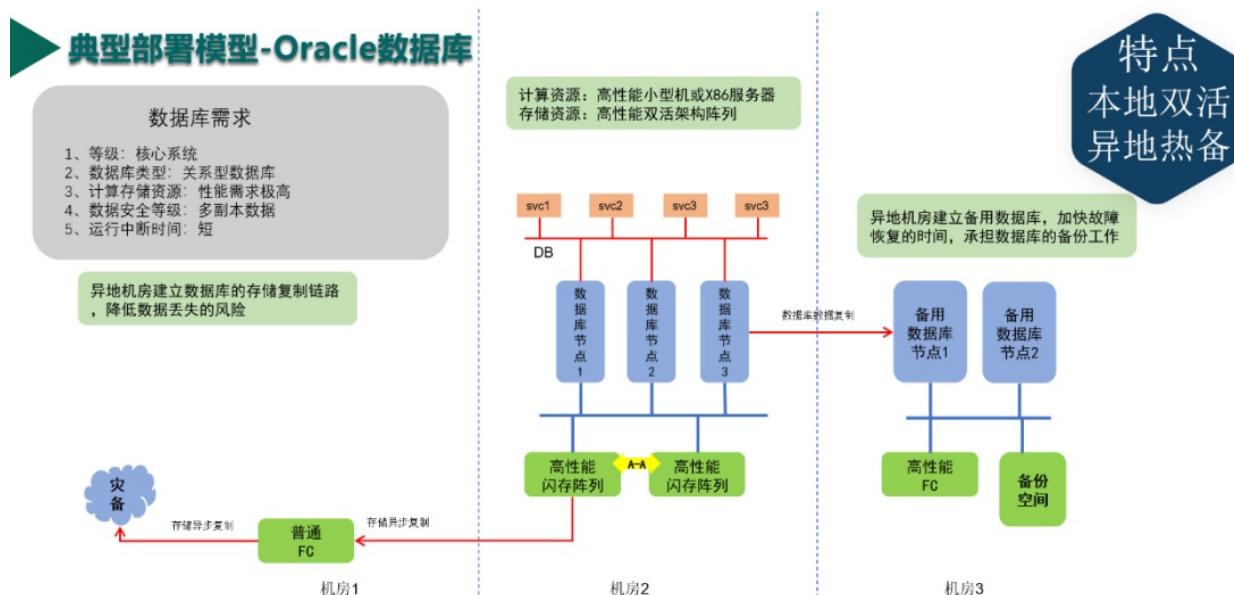
经过对国内外的数十种数据库的比对分析，最终 TiDB 脱颖而出，作为分布式关系型数据库的首选。主要用于作为泛在电力物联网的核心支撑数据库系统。其应用模式分为三种模式：

(1) 泛在电力物联网数据采集大型分布式数据库系统：部署大规模的数据库集群，作为海量物联网数据的采集、存储、共享、分析的核心数据库系统；

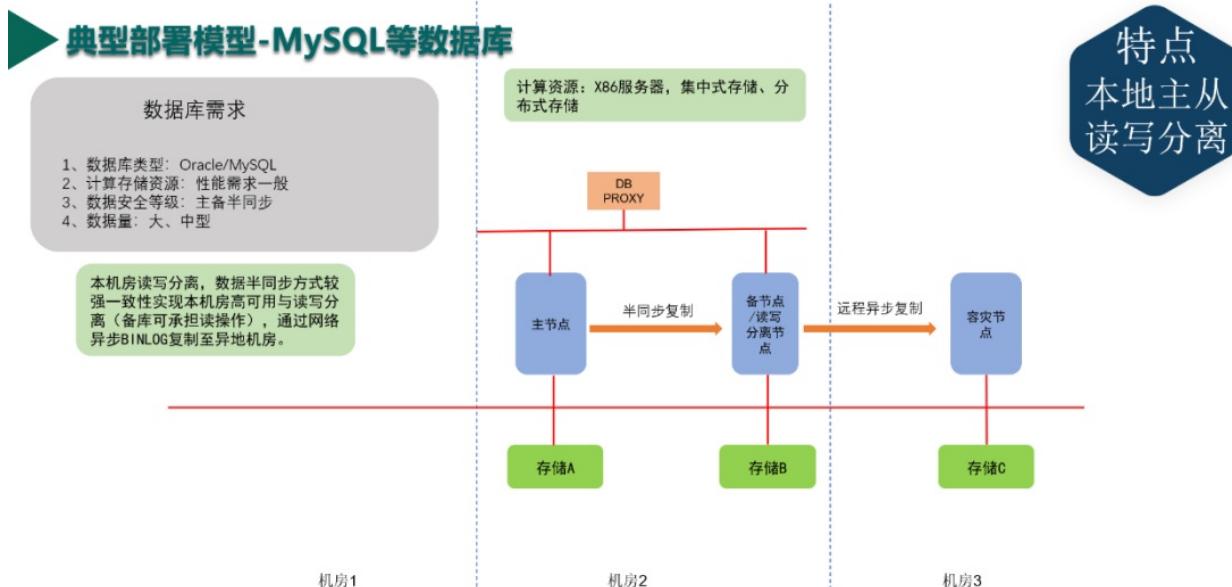
(2) 多租户数据库云服务：建立统一的数据库云服务，为企业各种小型数据库应用提供统一的服务；

(3) 边缘侧数据中心的主数据库：在泛在电力物联网建设过程中，各类变电站、配电网中将会建设大量的小型数据中心，这些小型数据中心用于承担边缘侧数据的存储与计算任务，其数据库要求支撑海量计算、高可用、易扩展、易维护。

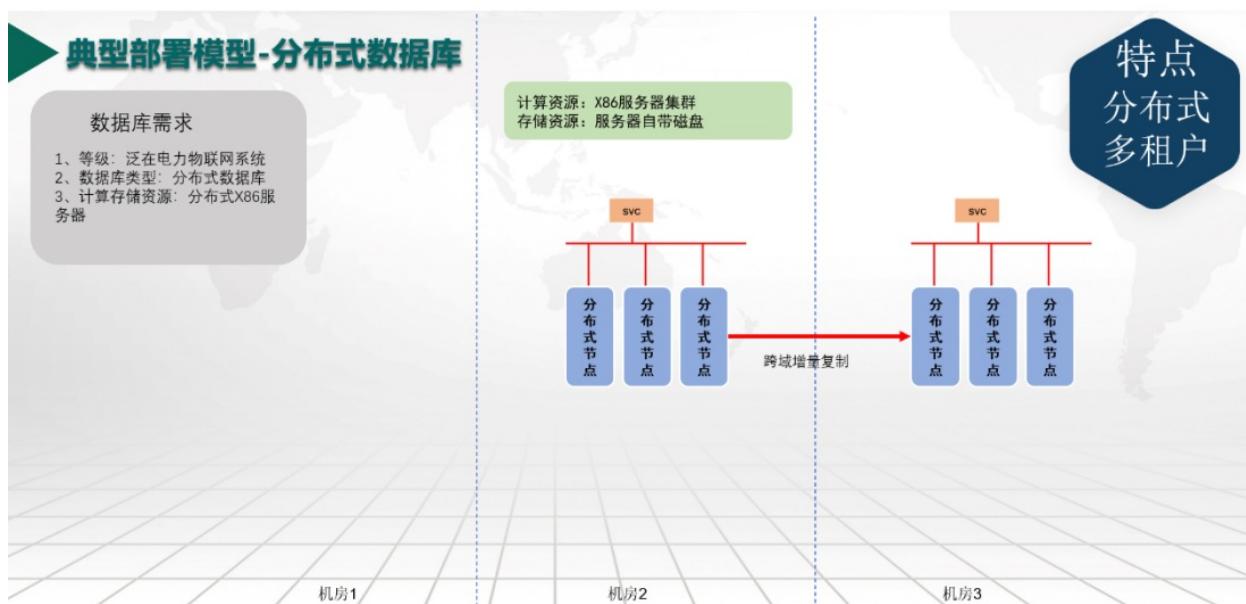
在数据分级存储顶层设计原则指导下优化后的关系型应用系统，其部署架构被统一成三种模式。



对于传统的核心业务系统，暂时无法进行云化改造的，进一步加强其系统安全性，提升高可用等级，确保核心业务稳定运行。



对于重要性次一级的系统，可以采用 MySQL 或者 Oracle（尽可能选择 MySQL），采用主从复制的方式实现高可用，这类数据库系统可以部署在云平台上，也可以部署在物理机组成的裸金属云上，由数据库运管平台统一纳管。



对于泛在电力物联网应用、边缘侧三站合一数据中心应用等应用场景，采用 TiDB 分布式数据库系统，采用多租户的方式，为各类应用系统提供数据库服务。

## 6.5.4 总结

经过对数据存储模式的梳理，在数据分级存储的顶层架构指导下，企业数据存储架构更为合理，数据库种类得到了有效的控制，数据库部署架构更为规范。TiDB 分布式数据库的引入填补了以往海量数据处理与时序数据处理方面的空白，使数据处理的范围与能力得到了极大的扩展。在新的数据存储架构下，企业将可以从容面对未来数据高速增长与数据处理需求不断提升的需求，为建设信息化企业提供了有力的保障。

## 6.6 TiDB 与 HBase、ES、Druid 的数据交互实战

大数据生态组件可谓是百花齐放，百家争鸣，由于时间和技术的原因，企业多会部署多类大数据基础组件，上了 TiDB 之后，如何实现 TiDB 与现有大数据生态的数据库进行互联互通呢？大家可能会有不同的方案来实现，本章主要讨论如何使用利用现有的开源工具高效低成本的实现 TiDB 与现有大数据组件比如 Hbase、ES、Druid 进行数据交互。实现不同类型的数据库之间的数据互联互通，从平台和工具层次看主要解决方案是使用 ETL 工具。这里 ETL 我们主要使用 Kettle。主要介绍如何使用 Kettle 实现 TiDB 与 HBase、ES、Druid 之间实现数据交互。

### 6.6.1. ES 数据导入 TiDB

主要实现将 ES 的数据以数据流的方式抽取到内存，经过一系列处理之后，以流的方式保存到 TiDB，主要使用到的 Kettle 的组件为：tableinput、Elasticsearch Query（扩展插件）。

下面将在 ES 的表 `bmsshakehandsinfo_202003` 每天定时导入 TiDB 的数据库中的表 `Temp_ETL_Handshake`。

#### 1. ES 的查询设置

如下图 1 所示，通过 ES 查询组件设置需要查询的 SQL。从 ES 数据库获取数据是采用分批的方式获取的。可以通过界面设置每次获取数据的条数大小。其中 SQL 查询可以通过采用 Kettle 变量的形式进行参数的动态传递 如下图 2 所示。

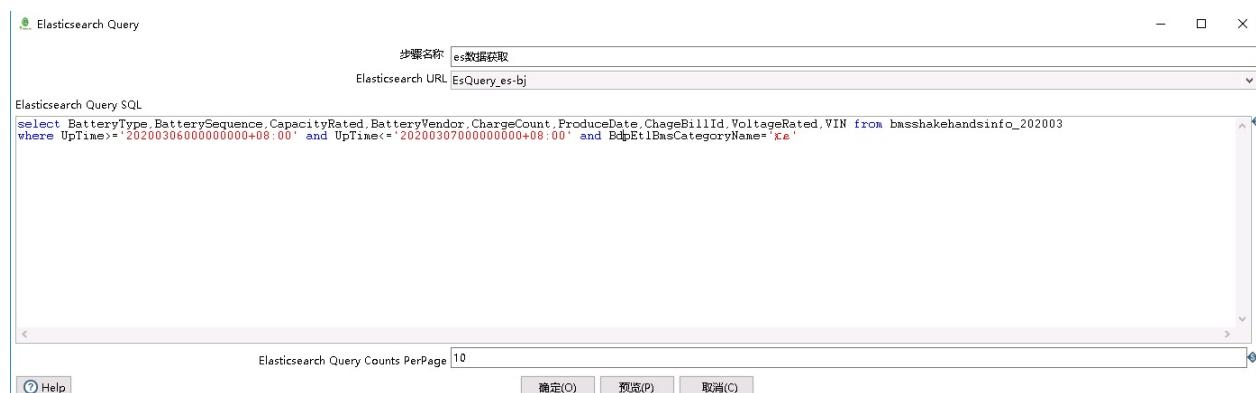


图1

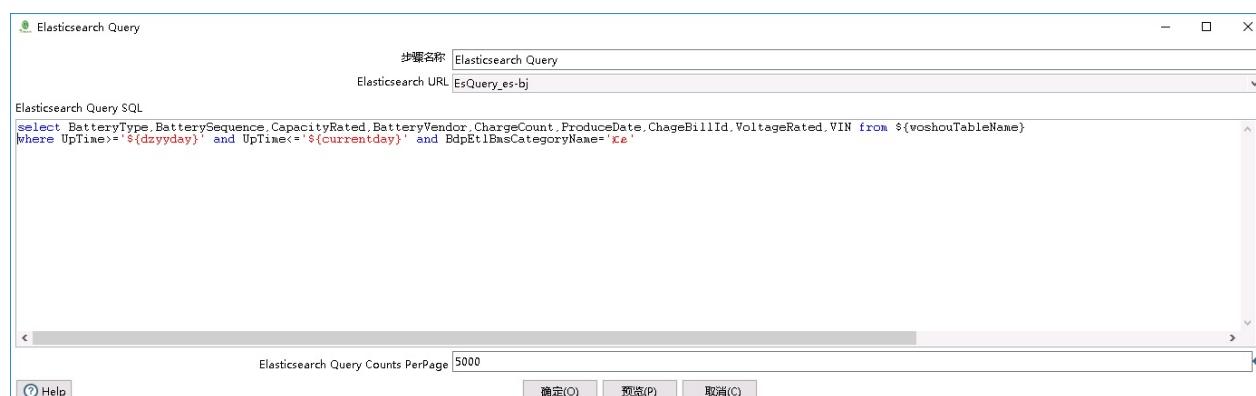


图2

#### 2. TiDB 的查询组件设置

将数据写入 TiDB，主要采用 Kettle 内置的 `tableout` 插件相关的设置如下。

##### 2.1 数据源设置

Kettle 里面设置 TiDB 的数据源直接采用 Kettle 内置的 MySQL 数据驱动即可。如下图 3 所示

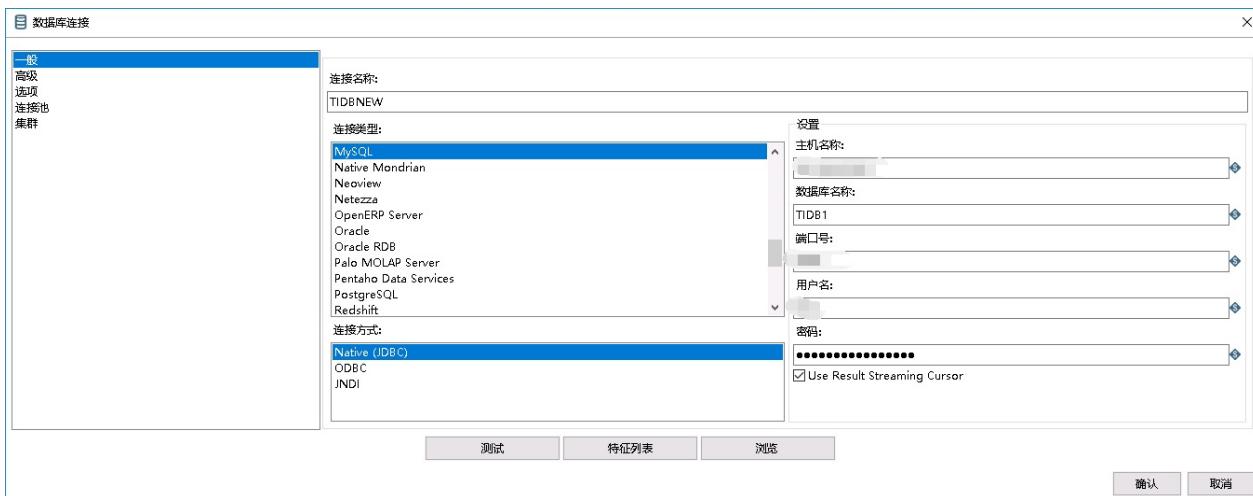


图3

## 2.2 TiDB 数据保存设置

TiDB 的数据保存设置界面如图 4 所示，设置对应的数据库、数据库表以及每次提交的数据量。相关参数注意事项如下：

- 提交的数据量

1. 需要结合 TiDB 的事务大小对应处理，一般建议每次处理的条数在 60-100. (可以根据硬件资源情况和事务设置进行调整)
2. 设置条数的同时，可以通过设置此插件运行时复制的个数进行性能提升，充分利用 TiDB 的高并发特性

- 忽略插入错误

1. 对于插入过程中出现的错误，可以选择插入过程中忽略或者默认停止
2. 如果插入过程中针对重复数据进行更新，可以采用 Kettle 的插入/更新步骤，由于此插件没有实现 TiDB 支持的 on duplicate 语法，如果需要性能较高，可以通过开发插件的方式，进一步提升插入的性能



图4

## 3. 数据处理流程设置

通过 Kettle 的转换流程，将上述步骤设置的 ES 的数据抽取、TiDB 的数据导入以流程的方式进行数据流的处理，当然中间可以进行复杂的清洗和转换设置，这里不做重点描述。数据流设置界面图 5 所示，其中 60 代表数据流程执行时，同时运行 60 个插入的实例，向 TiDB 进行高速的写入数据。



图5

通过 Kettle 的定时器，可以设置流程的执行时间。如图 6 所示。

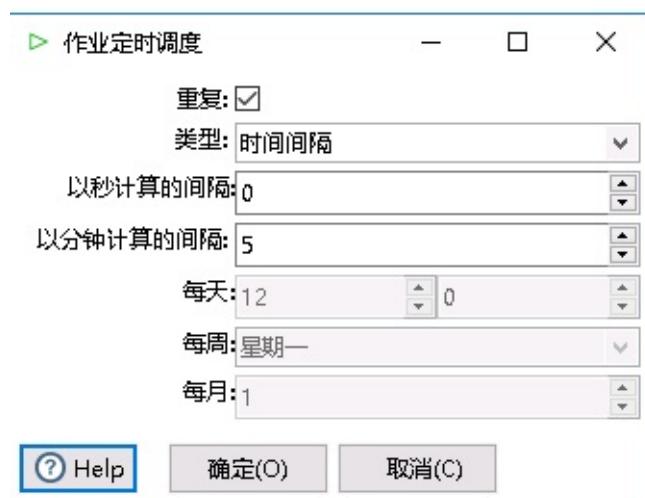


图6

以上完整的定义了通过 Kettle 工具，按需定时的将数据从 ES 以流的方式导入到 TiDB。

### 6.6.2 Hbase 数据导入 TiDB

Hbase 的数据导入 TiDB，主要使用的步骤为 Kettle 内置的 HbaseInput、TableOutPut。其中数据导入到 TiDB 的 TableOutPut 同 ES 导入 TiDB 类似，不在赘述。

数据源设置是直接采用 Kettle 的 HBase input 组件,如图 7,选择 Hbase 集群链接、获取对应的表名和 mapping 列表。

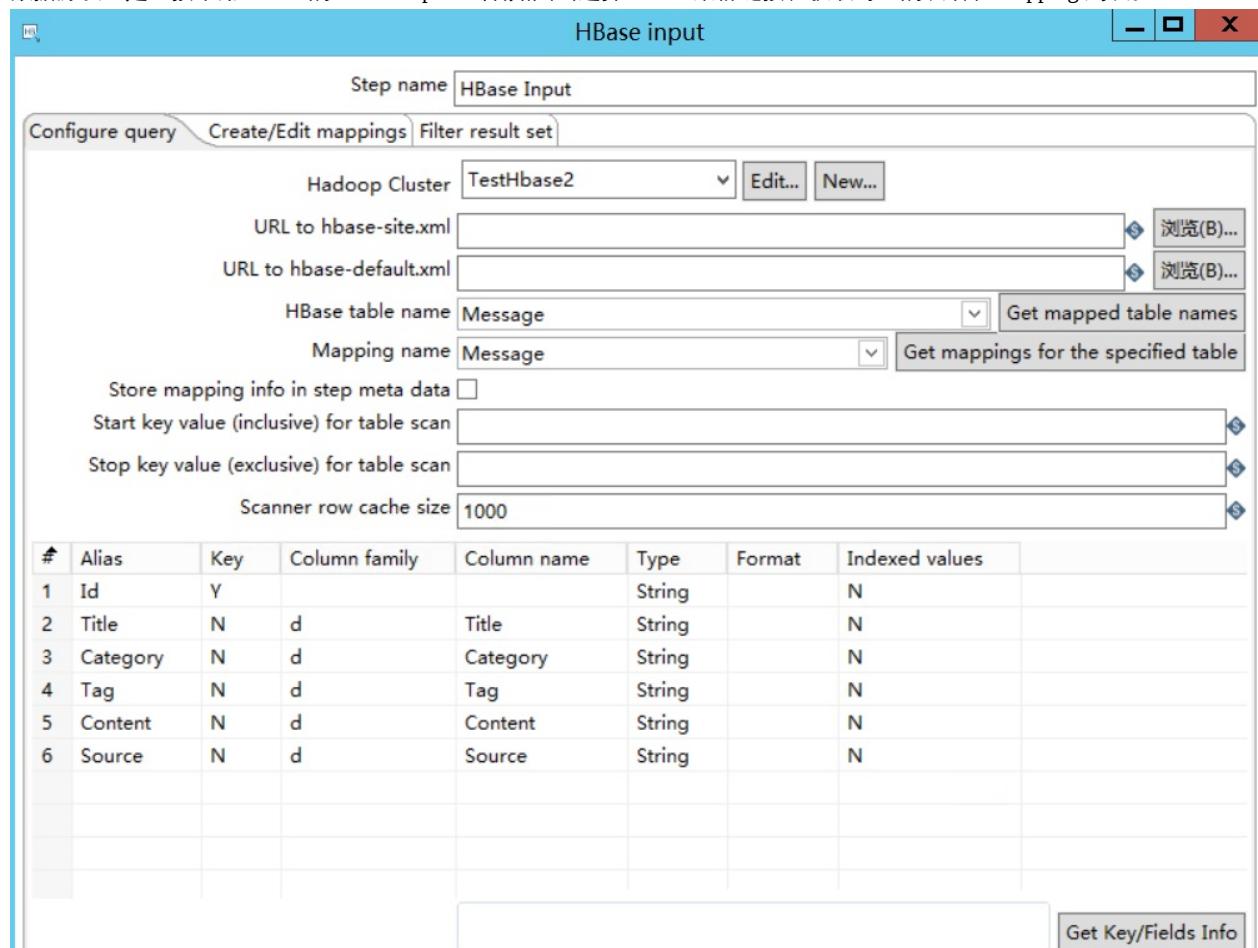


图7

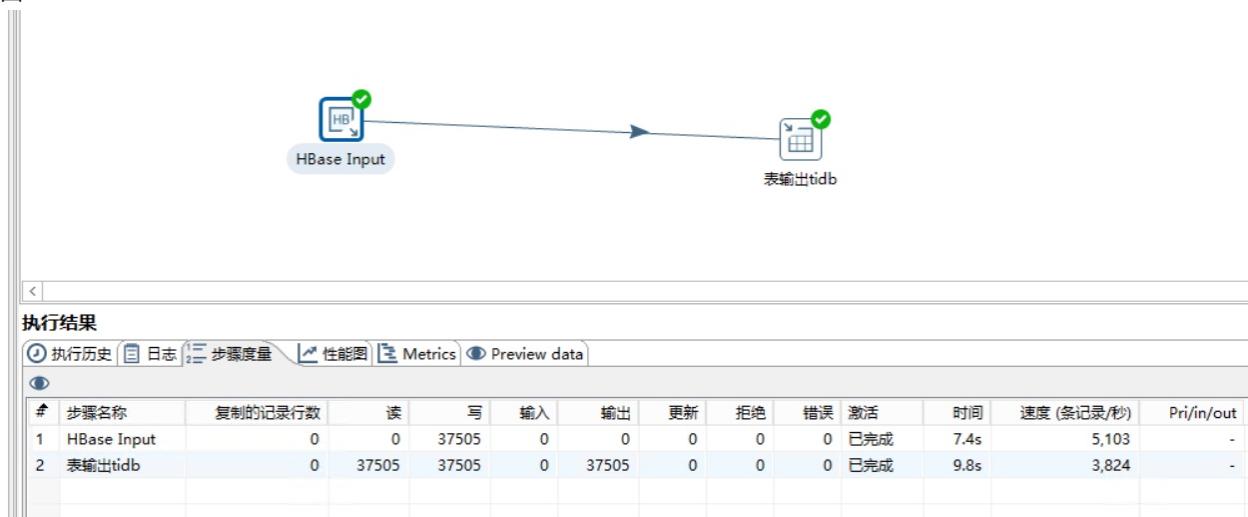


图8

### 6.6.3 Druid 数据导入 TiDB

Druid 的数据导入 TiDB ,主要使用的步骤为 Druid 查询组件 (扩展开发) 、TableOutPut。其中数据导入到 TiDB 的 TableOutPut 同 ES 导入 TiDB 类似 ,不在赘述。Druid 的查询组件设置如下 ,通过该组件可以通过设置 Druid SQL 的方式进行数据源的抽取。



图9

## 6.7 TiDB 与大数据可视化组件的集成应用

TiDB 作为一款 HTAP 数据库，具备海量数据存储的能力，非常适合构建实时数据仓库分析系统。本文主要介绍将 TiDB 作为核心数据存储与大数据可视化领域的 Grafana 和 Saiku 的集成。

### 6.7.1. TiDB 与 Grafana 的集成

Grafana 是一个跨平台的开源的度量分析和可视化工具，可使用内置的 MySQL 的插件连接 TiDB 数据库。

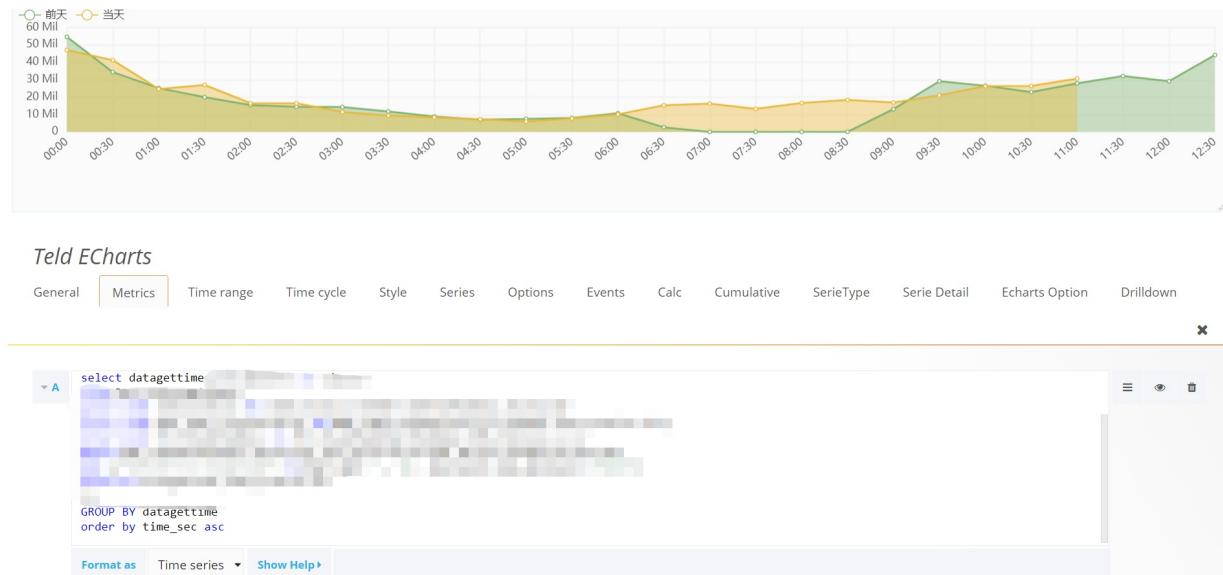


图 1

如图 1 所示为 Grafana 面板的设计界面。通过该界面可以通过 SQL 的方式，将数据通过 Grafana 组件进行展示，用于大数据的可视化展现。

注意：

1. 基于 Grafana 的展现，最好在创建 TiDB 数据表的时候，包含时间列，这样能够基于时间进行展示分析
2. 针对 Grafana 的报表，控制查询范围，另外针对性能要求较高的场景，可以针对性的优化增加索引

### 6.7.2 TiDB 与 Saiku 的集成

Saiku 提供了一个对用户友好的基于 Web 的分析解决方案，可让用户通过创建和共享报告来快速轻松地分析公司数据。该解决方案可连接到一系列 OLAP 数据库，且部署方案高效简洁。通过 TiDB + Saiku 构架可以搭建成一个 BI 分析平台，使业务分析人员可以进行动态多维分析。同时后期基于 TiDB 的 TiFlash 列存储，TiDB 在 OLAP 方面的能力得到大幅提升。

多维分析平台的搭建分为两大部分：

第一部分是 Saiku 集群

第二部分 TiDB 集群：TiDB 作为 Saiku 查询的数据源，通过 Saiku 的业务 SQL 直接从 TiDB 中进行数据查询。

#### 6.7.2.1. 多维报表的 Schema 定义

按照 Saiku 定义 Schema 的通用规则，可以直接定义跑在 TiDB 上的 Schema。无需特殊处理。同时可以充分发挥在大数据下的多表关联优势，降低存储和数据维护成本，对关系型数据库常用的星星模型有比较好的支撑。定义的 Schema 实例如下

```

▼<Schema name="001充电日期">
  ▼<Cube name="001充电日期" caption="001充电日期" cache="true" enabled="true">
    <Table name="kms_dt" alias="" />
    <!-- 001充电日期-年 (字段:RYear) -->
    ▼<Dimension type="StandardDimension" name="001充电日期-年" foreignKey="bizdate">
      ▼<Hierarchy name="" hasAll="true" primaryKey="tday" allLevelName="全部" allMemberCaption="全部">
        <Table name="kms_dt" />
        <Level name="充电日期-年" column="tyear" type="String" uniqueMembers="true" levelType="Regular"> </Level>
      </Hierarchy>
    </Dimension>
    <!-- 002充电日期-月 (字段:RYearMonth) -->
    ▼<Dimension type="StandardDimension" name="002充电日期-月" foreignKey="bizdate">
      ▼<Hierarchy name="" hasAll="true" primaryKey="tday" allLevelName="全部" allMemberCaption="全部">
        <Table name="kms_dt" />
        <Level name="充电日期-月" column="ymonth" type="String" uniqueMembers="true" levelType="Regular"> </Level>
      </Hierarchy>
    </Dimension>
    <!-- 003充电日期-周 (字段:周描述) 先把周注释掉，重新修改去除UPPER()逻辑后，再显示该维度。(通过修改sensitive配置，已成功去除Upper)
    -->
    ▼<Dimension type="StandardDimension" name="003充电日期-周" foreignKey="bizdate">
      ▼<Hierarchy name="" hasAll="true" primaryKey="tday" allLevelName="全部" allMemberCaption="全部">
        <Table name="kms_dt" />
        <Level name="周" column="seqdesc" type="String" uniqueMembers="true" levelType="Regular"> </Level>
      </Hierarchy>
    </Dimension>
    <!-- 004充电日期-日 (字段:RBizDate) -->
    ▼<Dimension type="StandardDimension" name="004充电日期-日" foreignKey="bizdate">
      ▼<Hierarchy name="" hasAll="true" primaryKey="tday" allLevelName="全部" allMemberCaption="全部">
        <Table name="kms_dt" />
        <Level name="充电日期-日" column="tday" type="String" uniqueMembers="true" levelType="Regular"> </Level>
      </Hierarchy>
    </Dimension>
    <!-- 005启动时段(字段:DHour) -->
    ▼<Dimension type="StandardDimension" name="005启动时段">
      ▼<Hierarchy name="" hasAll="true" allLevelName="全部" allMemberCaption="全部">
        <Level name="启动时段" column="dhour" type="String" uniqueMembers="true" levelType="Regular"> </Level>
      </Hierarchy>
    </Dimension>
    <!-- 011省 (字段:省名称) -->
  
```

图 2

如上图是一个星星模型的 Schema，维表 kms\_dt 通过主键 tday 和主表的 bizdate 进行关联构成星星模型。

注意：

在配置 Schema 时，表和字段的大小写要和 TiDB 中的表一致，否则定义的 Schema 在 Saiku 中报错。

### 6.7.2.2 TiDB 的数据源定义

通过 Saiku 的管理控制台界面，进行数据源的定义，如图 1-3 所示，相关参数定义如下 URL: TiDB 的拦截地址，如果有负载均衡，可以设置负载均衡的的地址 Schema: 多维分析报表对应的 schema jdbcdriver：由于 TiDB 与 MySQL 良好的集成性，可以直接使用 MySQL 的驱动 jdbc Driver:com.mysql.jdbc.Driver 账户和密码：TiDB 的数据库访问密码

**Create Data Source**

**Name:** [Redacted]

**Connection Type:** Mondrian

**URL:** jdbc:mysql://[Redacted].4000/tidb1

**Schema:** /datasources/[Redacted].xml

**Jdbc Driver:** com.mysql.jdbc.Driver

**Username:** [Redacted]

**Password:** [Redacted]

图 3

## 2.3 多维报表的运行

在 Saiku 界面新建对应多维分析报表分析，在选择多维数据中找到需要定义的数据源，定义上对应的指标和维度作为默认的多维分析界面。分析人员可以根据需求指定自己的私有报表分析。定义完成的报表后通过报表管理界面直接运行，并进行交互式的分析。运行界面实例如下：

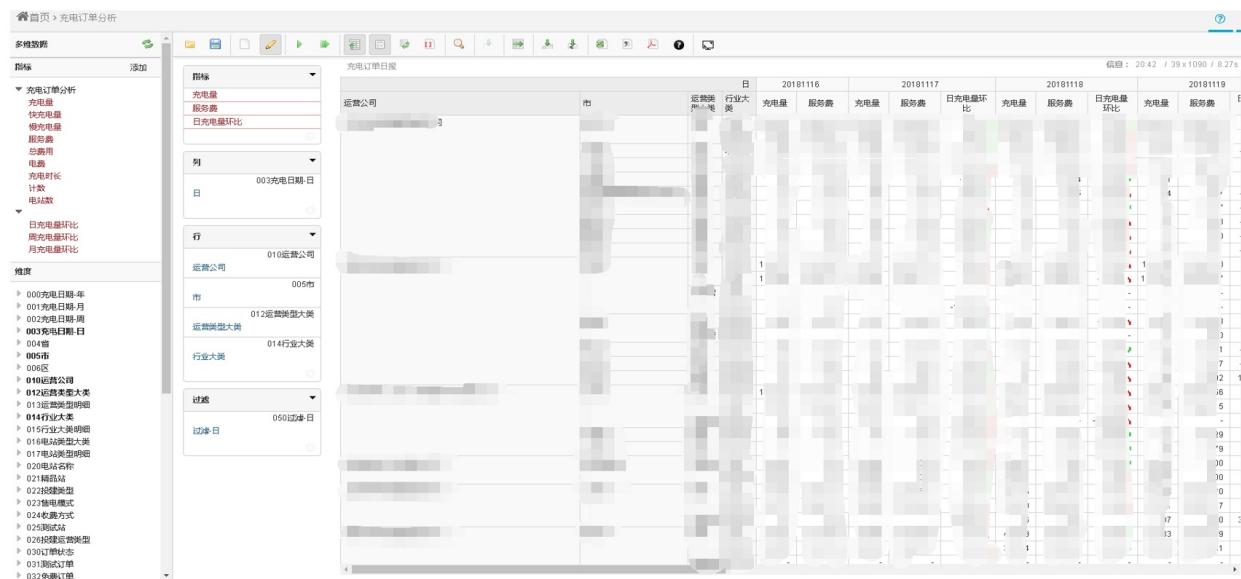


图 4

注意：

1. 由于 Saiku 默认的星星模型的关联方式是内连接，因此在数据的关联上要注意数据的准确性

2. 由于 Saiku 的查询 SQL 是根据用户自定义的，因此会出现查询 SQL 无法直接使用现有数据库表索引或者语法不够优化等情况。可以通过修改 Saiku 工程的 Mondrian 代码方式，针对 TiDB 数据库进行针对性的修改。同时通过分析 TiDB 的日志表，针对性的对报表增加索引，提升查询速度。

## 第7章 常见问题处理思路

用户在使用 TiDB 数据库的时候，会遇到很多问题，对于问题的处理思路，本章从整理，以及几个案例来进行描述。其中 7.1 oncall-map 是汇集了众多用户案例总结的一些问题通用处理手段。7.2、7.3、7.4 这三个章节详细描述了其中 3 个最常见的问题的场景及处理思路。

目录：

- 7.1 Oncall 地图
- 7.2 热点问题处理思路
- 7.3 TiKV is busy 处理思路
- 7.4 TiDB OOM 的常见原因
- 7.5 TiKV 磁盘空间占用与回收的常见问题

## 7.1 Oncall 地图

### 1. TiDB 集群问题导图的起源

在使用 TiDB 的过程中，用户总会遇到不同的问题，解决这些问题的通常做法是根据报错或者现象，去 TiDB 官网查询文档以及 FAQ，或者在 askTUG 网站上发帖子寻求帮助。但 PingCAP 官方经常处理用户各种各样的问题，积累下来了很多处理问题的经验，那么有没有一种方式，可以让用户自助的根据报错及现象，利用官方内部积累的经验，能快速定位和解决一些常见问题的方法呢？并且针对各个模块的问题用户有没有方法快速有一个系统化的了解？并且以上需要有很低的学习成本，容易理解。

答案是肯定的，为了达到上面提到的目标，PingCAP 把一些常见问题的现象、原因、解决办法以及涉及到的版本归纳总结到一张思维导图中，我们叫做 diagnose-map。用户可以参考导图来快速定位和解决自己遇到的问题。其中包括了 7 大系列的几十中常见问题，并且还在逐步补充中。

### 2. 获取问题导图的方式

可以在 <https://github.com/pingcap/tidb-map/blob/master/maps/diagnose-map.png> 来下载 TiDB 集群问题导图。该 png 图片是通过思维导图工具把 markdown 格式的 <https://github.com/pingcap/tidb-map/blob/master/maps/diagnose-map.md> 转换而来的。

### 3. 问题导图的使用

TiDB 集群问题导图是一个思维导图，汇集了各个模块常见的一些问题。该导图中把各个模块的问题进行了分类，比如把引起 TiKV OOM 问题的一些潜在原因以及解决方案放在一起，把导致 PD 选举问题的一些潜在原因和解决办法放在一起等等。另外也把两种常见的现象，服务不可用和 latency 明显增高作为两个单独的分支，用户可以从这两个分支出发寻找潜在的问题。举个例子，比如客户端收到 region is unavailable 错误，1.1.1 解释了导致该错误的原理是怎样的，然后列举了 4 种可能导致该问题的原因，用户可以根据自己集群的现象对号入座，按照流程来分析和解决自己遇到的问题。

另外可以看到一些类似 ONCALL-958 的东西，是该问题的处理案例，经过抽象后发布在该项目中 <https://github.com/pingcap/tidb-map/tree/master/maps/diagnose-case-study>。

### 4. 问题反馈

由于 TiDB 一直在快速迭代中，该导图不可能把所有可能的问题都提前列出来，只能是一点点完善，如果在该导图中找不到答案的问题可以搜索官方文档或者在 askTUG 上发帖子询问。

另外在使用 TiDB 集群问题导图的过程中如果有遇到任何问题，或者对于改进该导图有自己的建议，或者导图中一些信息是错误的，欢迎给 <https://github.com/pingcap/tidb-map> 提 issue。

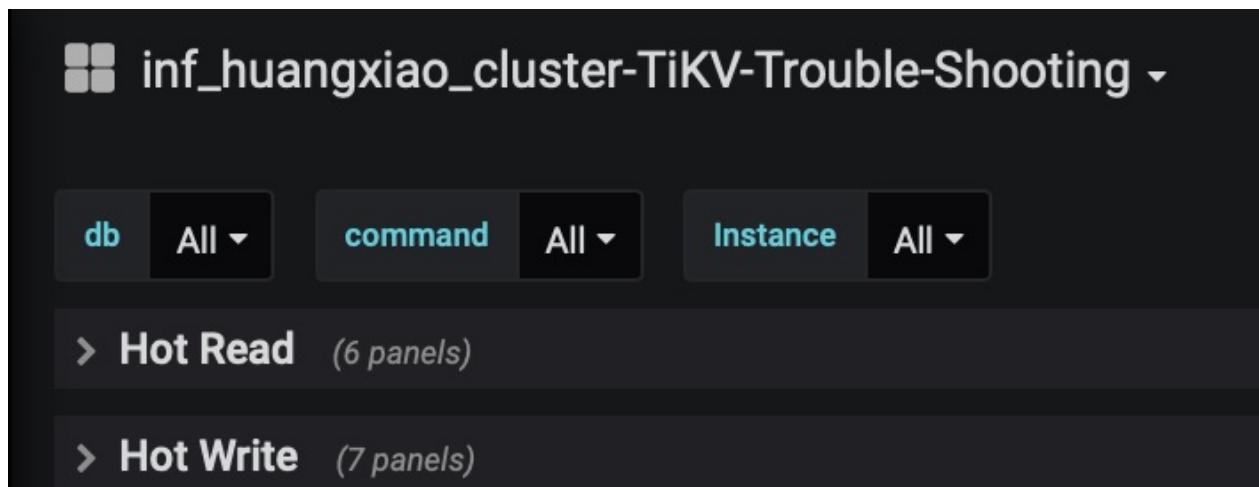
## 7.2 热点问题处理思路

众所周知，在分布式数据库中，除了本身的基础性能外，最重要的就是充分利用所有节点能力，避免让单个节点成为瓶颈。但随着业务场景的复杂性，各节点上的数据读写访问热度，总是无法保证均衡的，热点现象就此产生。严重的热点问题，会导致单个节点成为资源瓶颈，进而影响整个系统的吞吐能力。如果不能很好的解决热点问题，数据库的水平扩展能力及稳定性也就无法得到保障。TiDB 作为一个分布式数据库，虽然会自动且动态的进行数据的重新分布以到达尽可能的均衡，但是有时候由于业务特性或者业务负载的突变，仍然会产生热点，这时候往往就会出现性能瓶颈。

TiDB 是一个分布式的数据库，在表结构设计的时候需要考虑的事情和传统的单机数据库有所区别，需要开发者能够带着「这个表的数据会分散在不同的机器上」这个前提，才能做更好的设计。

### 7.2.1 确认热点问题

初步怀疑集群存在热点问题时，可以通过 TiDB 3.0 Grafana 提供的 TiKV-Trouble-Shooting 的 Dashboard 的 Hot Read 和 Hot Write 面板来快速确认是否存在读热点或者写热点。



1. Hot Read 面板聚集了读取热点相关的核心指标：

- CPU：每个 TiKV 节点的 CPU 使用率
- QPS：每个 TiKV 实例上各种命令的 QPS
- Storage ReadPool CPU：Readpool 线程的 CPU 使用率
- Coprocessor CPU：coprocessor 线程的 CPU 使用率
- gRPC poll CPU：gRPC 线程的 CPU 使用率，通常应低于 80%
- IO utilization：每个 TiKV 实例 IO 的使用率

2. Hot Write 面板聚集了写热点相关的核心指标：

- CPU：每个 TiKV 节点的 CPU 使用率
- QPS：每个 TiKV 实例上各种命令的 QPS
- gRPC poll CPU：gRPC 线程的 CPU 使用率，通常应低于 80%
- IO utilization：每个 TiKV 实例 IO 的使用率
- Raft store CPU：raftstore 线程的 CPU 使用率，通常应低于 80%
- Async apply CPU：async apply 线程的 CPU 使用率，通常应低于 90%
- Scheduler CPU：scheduler 线程的 CPU 使用率，通常应低于 80%

通过观察 Hot Read 和 Hot Write 面板中是否存在个别 TiKV 节点的指标明显高于其他节点，可以快速判断集群是否存在读热点或者写热点。

### 7.2.2 定位热点表 / 索引

确定是个别 TiKV 实例的热点问题后，需要进一步确认是哪张表的热点，是否是索引热点，是读还是写。从 TiDB 3.0 开始，推荐在热点现场通过 SQL 查询 information\_schema.TIDB\_HOT\_REGIONS 表定位热点表/索引：

```
-- TYPE 用于过滤热点的类型, read 表示读热点, write 表示写热点
SQL> select * from information_schema.TIDB_HOT_REGIONS where type = 'read'\G
***** 1. row *****
    TABLE_ID: 21
    INDEX_ID: NULL
    DB_NAME: mysql
    TABLE_NAME: stats_histograms
    INDEX_NAME: NULL
    REGION_ID: 44
    TYPE: read
    MAX_HOT_DEGREE: 17
    REGION_COUNT: 0
    FLOW_BYTES: 248548
1 row in set (0.02 sec)
```

或者通过 [pd-ctl](#) 用于查询读、写流量最大的 Region：

```
$ pd-ctl -u http://{pd}:2379 -d region topread [limit]
$ pd-ctl -u http://{pd}:2379 -d region topwrite [limit]
```

上面的命令中，输出信息中最核心的内容就是 region\_id：

```
$ pd-ctl -u http://{pd}:2379 -d region topread 1
{
  "as_peer": null,
  "as_leader": {
    "1": {
      "total_flow_bytes": 248535,
      "regions_count": 1,
      "statistics": [
        {
          "region_id": 44,
          "flow_bytes": 248548,
          "hot_degree": 15, -- 每分钟统计 1 次, 如果是热点, degree+1
          "last_update_time": "2020-03-07T16:14:51.186168306+08:00",
          "AntiCount": 1,
          "Version": 11,
          "Stats": null
        }
      ]
    }
  }
}
```

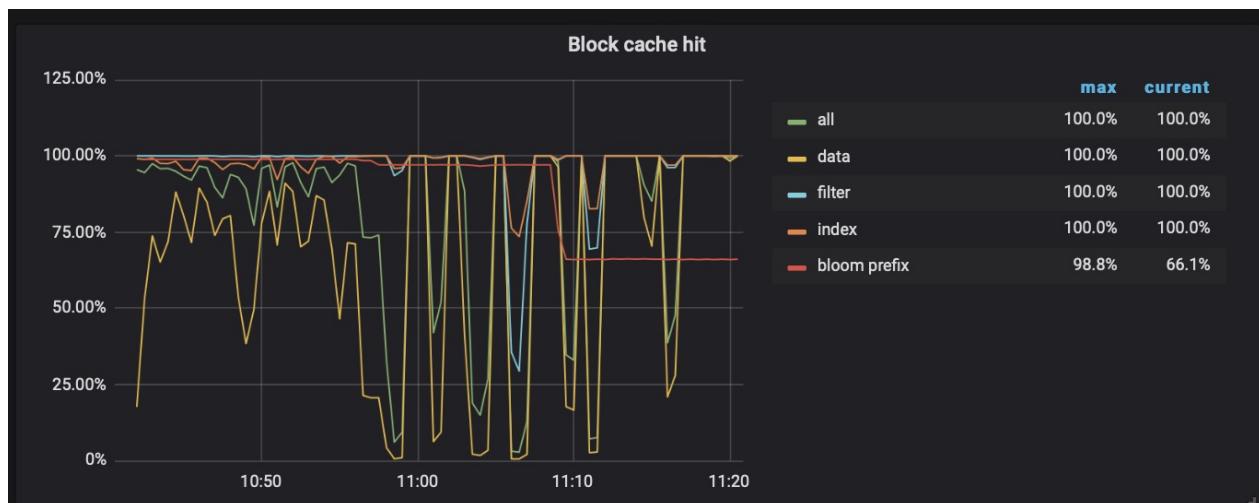
再从 Region ID 定位到表或索引：

```
$ curl http://{TiDBIP}:10080/regions/{RegionId}
{
  "region_id": 44,
  "start_key": "dIAAAAAAAAAB",
  "end_key": "dIAAAAAAAAAX",
  "frames": [
    {
      "db_name": "mysql",
      "table_name": "stats_histograms",
      "table_id": 21,
      "is_record": false,
      "index_name": "tbl",
      "index_id": 1
    },
    {
      "db_name": "mysql",
      "table_name": "stats_histograms",
      "table_id": 21,
      "is_record": true
    }
  ]
}
```

最后根据热点表表结构和业务沟通改造方案。如果热点现场已过，可以通过业务监控提取问题表和问题 SQL。未来 TiDB 4.0 即将提供的 Key Visualizer 功能，直观展示整个数据库的不同位置数据访问频度和流量，快速定位热点，具体可参考本书的“识别集群热点和业务模式”章节。

### 7.2.3 读热点解决方案

TiDB 读取热点产生的原因通常是 TiKV 的 BlockCache 命中率下降或者是小表的并发读取过大。检查 TiKV-Details Dashboard 里 RocksDB KV 面板里的 Block cache hit 命中率，如果发现命中率出现大幅度下降或抖动，基本可以定位为慢 SQL 问题；否则倾向于怀疑是小表的大量并发读取导致的。



1. BlockCache 命中率下降 BlockCache 的命中率下降或者抖动时可能是存在全表扫描的 SQL、执行计划选择不正确的 SQL、存在大量 count(\*) 操作的 SQL 等等。可以参照本书 “快速定位慢 SQL” 的章节来定位，通过添加索引或者优化 SQL 语句执行计划、增加 SQL\_NO\_CACHE 的 Hints 等手段来进行优化。
2. Region 并发读取过高 有些表数据量不大、包含的 Region 数量较少，但业务查询频繁的命中个别 Region 最终导致单个 TiKV 节点性能达到极限。目前可以通过改造小表为 hash 分区表来保证数据均匀地分散到一定数量的分区来解决。从 SHOW TABLE REGIONS 命令可以观察到新建的 hash 分区表已经提前创建了4个分区：

```

SQL> CREATE TABLE t1(
-> id INT NOT NULL,
-> name VARCHAR(30),
-> hired DATE NOT NULL DEFAULT '1970-01-01',
-> separated DATE NOT NULL DEFAULT '9999-12-31',
-> store_id INT
-> )
-> PARTITION BY HASH(store_id)
-> PARTITIONS 4;
Query OK, 0 rows affected (1.29 sec)
SQL> show table t1 regions\G
***** 1. row *****
REGION_ID: 840
START_KEY: t_194_
END_KEY: t_195_
LEADER_ID: 843
LEADER_STORE_ID: 9
PEERS: 841, 842, 843
SCATTERING: 0
WRITTEN_BYTES: 35
READ_BYTES: 0
APPROXIMATE_SIZE(MB): 1
APPROXIMATE_KEYS: 0
***** 2. row *****
REGION_ID: 844
START_KEY: t_195_
END_KEY: t_196_
LEADER_ID: 847
LEADER_STORE_ID: 9
PEERS: 845, 846, 847
SCATTERING: 0
WRITTEN_BYTES: 35
READ_BYTES: 0
APPROXIMATE_SIZE(MB): 1
APPROXIMATE_KEYS: 0
***** 3. row *****
REGION_ID: 848
START_KEY: t_196_
END_KEY: t_197_
LEADER_ID: 851
LEADER_STORE_ID: 9
PEERS: 849, 850, 851
SCATTERING: 0
WRITTEN_BYTES: 35
READ_BYTES: 0
APPROXIMATE_SIZE(MB): 1
APPROXIMATE_KEYS: 0
***** 4. row *****
REGION_ID: 3
START_KEY: t_197_
END_KEY:
LEADER_ID: 475
LEADER_STORE_ID: 9
PEERS: 458, 471, 475
SCATTERING: 0
WRITTEN_BYTES: 217
READ_BYTES: 0
APPROXIMATE_SIZE(MB): 1
APPROXIMATE_KEYS: 0
4 rows in set (0.05 sec)

```

除了可以通过改造为 hash 表之外，TiDB 3.1 版本提供的 Follower Read 功能增加了集群的吞吐能力，也能在一定程度上缓解读热点。未来在 TiDB 4.0，PD 会提供 Load Base Splitting 策略，除了根据 Region 的大小进行 Region 分裂之外，还会根据访问 QPS 负载自动分裂频繁访问的小表的 Region，具体参考本书“弹性调度”章节。

## 7.2.4 写热点解决方案

TiDB 写入热点的业务场景通常有：

1. 业务从 MySQL 迁移到 TiDB 时保留了自增主键
2. 高并发写入无主键表/主键非 int 的表/联合主键的表
3. 高并发写入的表上存在递增索引，比如时间索引等
4. 高并发更新小表
5. 秒杀或者类似的场景下的单行热点问题

### 7.2.4.1 自增主键

MySQL 为了提高顺序写入的性能，通常都建议业务使用自增 ID 作为主键。而 TiDB 中数据按照主键的 Key 切分成很多 Region，每个 Region 的数据只会保存在一个节点上面。业务带着自增主键迁移到 TiDB 后，最新写入的数据大概率都在同一个 Region 上，也就是同一个 TiKV 节点上，从而引起热点。

从 MySQL 迁移到 TiDB 的时候，建议去掉自增主键，同时通过设置 SHARD\_ROW\_ID\_BITS，把 rowid 打散写入多个不同的 Region，缓解写入热点。但是设置的过大会造成 RPC 请求数放大，增加 CPU 和网络开销。

- SHARD\_ROW\_ID\_BITS = 4 表示 16 个分片
- SHARD\_ROW\_ID\_BITS = 6 表示 64 个分片
- SHARD\_ROW\_ID\_BITS = 0 表示默认值 1 个分片

语句示例：

```
CREATE TABLE : CREATE TABLE t (c int) SHARD_ROW_ID_BITS = 4;
ALTER TABLE : ALTER TABLE t SHARD_ROW_ID_BITS = 4;
```

SHARD\_ROW\_ID\_BITS 的值可以动态修改，每次修改之后，只对新写入的数据生效。未来 TiDB 4.0 会引入 auto\_random 特性，彻底解决数据打散问题。

### 7.2.4.2 隐式主键

如果业务的表没有主键或主键不是 int 类型或联合主键时，TiDB 内部会自动生成隐式的 \_tidb\_rowid 列作为行 ID。在不使用 SHARD\_ROW\_ID\_BITS 的情况下，\_tidb\_rowid 列的值基本也为单调递增，大量 INSERT 时数据会集中写入单个 Region，造成热点。

要避免由 \_tidb\_rowid 带来的写入热点问题，可以在建表时，使用 SHARD\_ROW\_ID\_BITS 和 PRE\_SPLIT\_REGIONS 两个建表选项。SHARD\_ROW\_ID\_BITS 用于将 \_tidb\_rowid 列生成的行 ID 随机打散。pre\_split\_regions 用于在建完表后预先进行 Split region。

**注意：** pre\_split\_regions 必须小于或等于 shard\_row\_id\_bits。

示例：

```
create table t2 (a int, b int) shard_row_id_bits = 4 pre_split_regions=2;
```

- SHARD\_ROW\_ID\_BITS = 4 表示 \_tidb\_rowid 的值会随机分布成 16 ( $16=2^4$ ) 个范围区间。
- pre\_split\_regions=2 表示建完表后提前切分出 4 ( $2^2$ ) 个 Region。

### 7.2.4.2 高并发更新小表

类似小表的并发读取过高场景，小表并发更新过高导致的写入热点，4.0 之前，需要通过手工切分热点 Region 来临时解决。定位到热点 Region，命令示例如下：

```
$ pd-ctl -u http://[PDIP]:2379 -i
// 将 Region 1 对半拆分成两个 Region, 基于粗略估计值
> operator add split-region {hotRegionId} --policy=approximate
// 将 Region 1 对半拆分成两个 Region, 基于精确扫描值
> operator add split-region {hotRegionId}1 --policy=scan
```

注意：手工切分的 Region 在经过 split-merge-interval 之后，可能会被合并，split-merge-interval 控制对同一个 Region 做 split 和 merge 操作的间隔，即对于新 split 的 Region 一段时间内不会被 merge，这个参数可以通过 pd-ctl 来更改，默认值是 1h。

中长期的解决方式是通过改造小表为 hash 分区表解决，或者是业务改造后通过队列缓存更新后批量提交等方式解决。未来 TiDB 4.0 版本的解决方式在 Region 并发读取过高时已经说明，此处略。

### 7.2.4.3 单调递增索引

当表上存在时间字段的索引、或者订单 ID 等单调递增的索引导致的写入热点，目前可以通过手工切分热点 Region 来临时解决。切分的方式在高并发更新小表时有过说明。

### 7.2.4.4 秒杀等单行热点

秒杀减库存等单行热点更新的场景下，大量请求并发更新同一行记录，推荐调整为 TiDB 的悲观锁。目前对于这类“极端”场景，建议通过异步队列或者缓存在来解决，TiDB 上云之后，可以考虑通过弹性调度隔离到高性能机器来解决。

## 7.2.5 调整 PD 的热点调度策略

上面介绍了针对不同热点场景的解决方案，下面说一下通用的通过 PD 调度策略来缓解热点的思路，可以作为热点问题解决方案的补充。PD 对于写热点，热点调度会同时尝试打散热点 Region 的 Peer 和 Leader；对于读热点，由于只有 Leader 承载读压力，热点调度会尝试将热点 Region 的 Leader 打散。

### 7.2.5.1 热点调度规则

热点调度对应的调度器是 hot-region-scheduler-limit，根据 Store 上报的信息，统计出持续一段时间读或写流量超过一定阈值的 Region，并用与负载均衡类似的方式把这些 Region 分散开来，对于写热点，热点调度会同时尝试打散热点 Region 的 Peer 和 Leader；对于读热点，由于只有 Leader 承载读压力，热点调度会尝试将热点 Region 的 Leader 打散。

### 7.2.5.2 提高 PD 的热点调度速度

在 TiDB 3.0 版本开始，将热点调度并发度从 region-schedule-limit 拆出到 hot-region-schedule-limit，可以通过加大 hot-region-schedule-limit、并减少其他调度器的 limit 配额，加速热点调度。还可调小 hot-region-cache-hits-threshold 使 PD 更快响应流量的变化。

```
// 调大 hot-region-schedule-limit
config set hot-region-schedule-limit 8
// 调小其他调度器 limit 配额
config set merge-schedule-limit 2
config set region-schedule-limit 2
```

### 7.2.5.3 整体打散表 Region 分布

在之前所述的读写热点解决方案都无法打散热点时，可以尝试在业务低峰时间添加 scatter-range-scheduler 调度器使这个 table 的所有 Region 均匀分布。添加 scatter-range-scheduler 调度器示例：

```
curl -X POST http://{TiDBIP}:10080/tables/{db}/{table}/scatter
```

清理 scatter-range-scheduler 调度器示例：

```
curl -X POST http://{TiDBIP}:10080/tables/{db}/{table}/stop-scatter
```

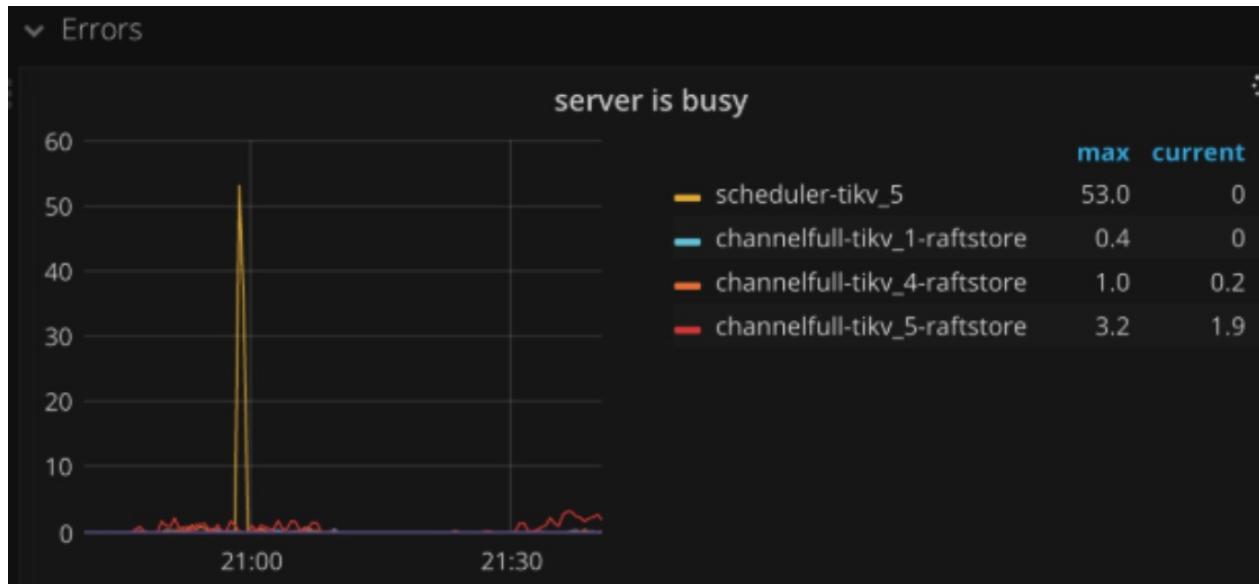
## 7.2.6 热点案例：

核心业务有 2 个表，一个是业务数据表 A，一个是相对应流水表 B。开始做设计时业务 A 和 B 都是一个字符型主键加一个时间索引。A 的数据大小 1.5K。B 的数据比较小。按照这种设计模式，表 A、B 的数据本身和相对应的时间索引都是热点。另外 2 个自定义的主键也可能存在热点。

考虑到 TiDB 2.1 版本单 TiKV 只有一个线程处理 raft 消息。并且 A 的数据比较大。最终只对 A 表的主键进行改造，把原主键的一部分拿出来做成 bigint 类型，并且在最前面 1 位做成 0-9 的随机数。人为的把 A 表的数据分成 10 片。因为 A 表数据比较大。此时，其他几部分的热点相对于整个业务来说，虽然是热点，但是成为不了瓶颈。

## 7.3 TiKV is busy 处理思路

日常 TiDB 运维中，当你在 TiKV 监控 Trouble - Shooting - Server Is Busy 看到以下这样的监控时，



可能此时的 TiDB 集群在该时间段内响应延时会大幅度增加，甚至会出现大量请求超时并且伴随大量告警出现。

### 7.3.1 Server is Busy 的影响

Server is Busy 本质上就是 tikv-server 繁忙，暂时无法对该请求做出响应，所以此时从 TiDB 集群到业务都会受到影响。以下从两个角度观察这个问题。

#### 1. 运维角度

1. 集群性能迅速下降，现象可以从 TiDB 监控 - Query Summary - Duration 明显看到。
2. TiKV 服务器负载增加，现象可以从 TiKV 监控 - Server / Thread CPU / Error 中看到。
3. TiDB 日志中大量的 server is busy 日志，可以查看 tidb.log 过滤 Server is Busy 关键字。
4. 慢查询大量增加，常规基于主键查询的请求，也会很慢。

#### 2. 业务角度

1. 业务访问数据库响应耗时大幅度增加，例如：5ms -> 5s。
2. 业务告警，数据库访问超时，例如：TimeoutException。
3. 部分请求访问 DB 不响应。

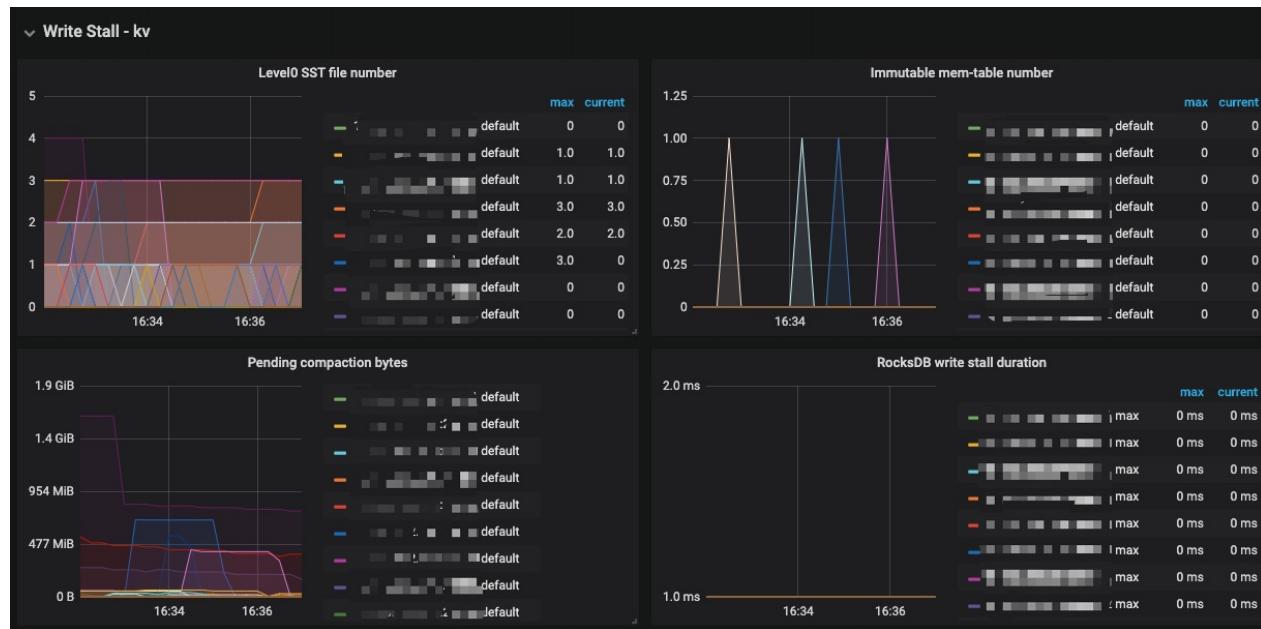
每次出现 Server is Busy 对于运维以及业务都是比较紧张，那么接下来分析一下 Server is busy 的原因。

### 7.3.2 Server is Busy 的原因

#### 1. 写保护 ( write stall )

TiKV 底层有 2 个 RocksDB 作为存储，RocksDB 使用的 LSM Tree，LSM Tree 牺牲了一部分读的性能和增加了合并的开销，换取了高效的写性能，但如果写入过快，超过了 RocksDB 处理的极限，RocksDB 就会考虑对写入进行降速处理。

- 密集写入，导致 level0 sst 太多导致 stall。
- pending compaction bytes 太多导致 stall，服务器磁盘IO能力在写入高峰跟不上写入。
- memtable 太多导致 stall。



处理建议：

1. 是否存在热点写入/写入倾斜，是否可以打散写入
2. 调整 rocksdb 参数 max-sub-compactions 至 2~3，将 level0 到 level1 的 compaction 拆分为多个子任务，加快并行 compaction 的速度
3. 适当调大 level0-slowdown-writes-trigger = 40, level0-stop-writes-trigger = 56，这不一定能根本解决问题，只是加大了限制进行缓解

## 2. scheduler too busy

- 写入冲突严重，两阶段提交时都需要申请 latch，如果冲突严重，latch 申请就会排队，导致 latch wait duration 比较高，现象 TiKV 监控 - scheduler prewrite|commit 的 latch wait duration

scheduler prewrite - latch wait duration | scheduler commit latch wait duration



- 写入慢导致写入堆积，该 TiKV 正在写入的数据超过了 [storage] scheduler-pending-write-threshold = "100MB" 设置的阈值。

处理建议：

- 是否可以将针对单行数据的并行写入，改为串行写入
- 可以考虑分布式锁
- 开启悲观事务

### 3. 线程池排队

常规的线程池设计中，请求处理的越快，线程池压力越小，整体处理能力就越强。当单个请求变慢时，整个线程池也不受影响，当变慢的请求逐渐堆积时，整个线程池也会逐渐变得处理能力下降甚至不响应。超出线程池上限后会返回 Server Is Busy。

#### (1) 关键配置 tikv.yml

在 3.0 的版本中，不同的查询会在 2 套线程池中执行，分别是 `readpool.storage` 和 `readpool.coprocessor`，每个线程池分为三个优先级，分别用于处理高优先级，普通优先级和低优先级请求。TiDB 点查选择是高优先级，范围扫描是普通优先级，而诸如表分析之类的后台作业是低优先级。

既然是使用线程池处理请求，接下来看下线程池的限制，线程数，以及单个线程允许积压的最大任务数量。

```
// 高优先级线程池， 默认值 cpu core 数 * 80%， 最小值 1
high-concurrency
// 普通优先级线程池， 默认值 cpu core 数 * 80%， 最小值 1
normal-concurrency
// 低优先级线程池， 默认值 cpu core 数 * 80%， 最小值 1
low-concurrency
// 指定低优先级线程池中每个线程的最大运行操作数，处理高优先级读取请求
// 默认值 2000， 最小值 2000
max-tasks-per-worker-high
// 指定低优先级线程池中每个线程的最大运行操作数，处理低普通先级读取请求
max-tasks-per-worker-normal
// 指定低优先级线程池中每个线程的最大运行操作数，处理低优先级读取请求
max-tasks-per-worker-low
```

其中以高优先线程池为例，因为调整线程资源的是线程池级别而不是单线程级别，所以高优先级线程池默认最大运行操作数的限制为

```
max-tasks-per-worker-high * high-concurrency = 2000 * 4 = 8000
```

### (2) 推荐配置（针对 TiDB 集群）

- 单机单实例( TiKV )，不应超过服务器 CPU 核数
  - 例如：一台 48 Core 服务器运行 1 个 tikv-server，则每个实例的高并发值应小于 48
  - 最大性能推荐值：**48**
  - 均衡性能推荐值：**36**
- 单机多实例(TIKV) ,
  - 例如：一台 48 Core 服务器运行 3 个 tikv-server，则每个实例的高并发值应小于 16
  - 最大性能推荐值：**16**
  - 均衡性能推荐值：**12**

### (3) 4.0 的线程池整合

从 4.0 版本开始，将 readpool.storage 和 readpool.coprocessor 整合为一个 unified read pool 线程池，并且不再需要配置 3 个优先级，解决资源分配不均的问题，并且大大提高了使用体验，相关配置：

```
[readpool]
# unify-read-pool = true

[readpool.unified]
# min-thread-count = 1
# max-thread-count = 8

## Size of the stack for each thread in the thread pool.
# stack-size = "10MB"

## Max running tasks of each worker, reject if exceeded.
# max-tasks-per-worker = 2000
```

处理建议：

1. 考虑是否出现大量扫描现象。
2. 考虑是否是可用线程较少，可以通过增加 TiKV 节点提高集群整体处理能力

## 4. raftstore is busy

- append log 遇到了 stall，监控在 2 个地方可以看到
  - [tikv-detail]->[RocksDB - raft]->[Write stall duration]
  - [tikv-detail]->[RocksDB - kv]->[Write stall duration]
  - 以上也可以直接看 [tikv-detail]->[Errors]->[Server is busy]
- append log duration 比较高，导致处理消息不及时，监控在 [tikv-detail]->[Raft IO]->[append log duration]
  - 考虑 append log 慢，查看磁盘 IO 情况，通常是写盘慢了，查看 [tikv-detail]->[Raft IO]->[Write duration]

- 考虑 [raftstore] store-pool-size 配置是否过小，该值建议在[1,5]之间，不建议太大。可以通过 [tikv-detail]->[Thread CPU]->[Raft store CPU] 看确定是否过小，如果[Raft store CPU]超过了 [store-pool-size 数量] \* 70% 说明需要加大 store-pool-size。
- 通常单个 tikv-server 实例的 region 数量超过 5 万之后，region 之间的心跳也会占用很多 raftstore cpu，建议开启 hibernate region 来解决这个问题

```
[raftstore]
hibernate-regions = true
```

处理建议：  
考虑是否磁盘写入存在瓶颈  
是否\*\*\*\*\* store-pool-size 配置是否过小，适当调整参数

以上为 TiKV Server is busy 的主要的几个原因，在使用 TiDB 过程中需要尽力避免过程中出现 Server is Busy 的情况，可以通过优化 SQL 优化、参数调整、增加节点等手段避免该问题。

### 7.3.3 触发 Server is Busy 的常见场景

#### 1. SQL 开销较大

1. 常规慢查询，有很多的场景，例如：全表扫描。
2. 大表索引未被命中正确的索引的情况。
3. 高并发导入数据，导致 tikv 写入繁忙。

处理建议：针对开销较大的 SQL，如果是读 SQL 可以做相应的 SQL 优化，来避免大量扫表。（4.0 的 unified thread pool 针对这种情况有优化）。高并发导入的问题可以降低导入并发。

#### 2. 事务冲突

乐观锁事务模式下事务冲突严重，会导致大量的线程进行重试，从而导致 tikv is busy，例如计数器功能。

处理建议：针对\*\*乐观锁事务模式下的事务\*\*冲突的场景，可以通过添加分布式锁，或者使用悲观事务模型来解决。TiDB v3.0.8 默认使用悲观锁事务模式，如果集群是从 v3.0.8 版本以下升级上来的集群，默认还是乐观锁事务模式。

#### 3. 集群 region 数量太大

在 TiDB 2.1 等低版本中，因为 TiKV 的 raft 是单线程，当管理的 region 数达到一定量级时，性能会下降，多大一定程度，单核只够管理 region。并没有空闲的能力处理业务。业务就会出现 server is busy。

处理建议：  
1. 升级到 3.0 版本以上，设置 store-pool-size 启用多线程 raftstore  
2. 设置 hibernate-regions，开启静默 region

### 7.3.4 热点问题

1. 如果业务写入集中在某一个 region 范围内，比如自增 id 的写入，这个 region 所在的 tikv-server 的压力会增大，导致线程处理变慢，线程会出现排队，最终导致 server is busy。
2. 客户端在短期内发起密集的写入，其中主要是热点写入，可能出现写入倾斜，并导致单个 tikv 节点出现 server is busy。

处理建议：  
对于热点更新的场景，可以通过 region 拆分、shard\_row\_id\_bits、pre\_split\_regions 等方式优化。

### 7.3.5 常规查询变慢的原因

当 Server is Busy 出现后，查询为什么突然变慢，平时 99% 6ms 返回, 为啥突然 3s 都没有返回?

如果确认是在 Server is Busy 的情况下，Query Duration 明显增加，此时可以通过观察 tidb.log 日志，可以看到，正常查询主要耗时在 wait 阶段，并不是消耗在 exec 时间。

例如：如果 coprocessor 的每个线程排队超过 2000 个任务，本次查询是第 2001 个任务，那么需要队列中任务任一个任务执行完成，后第 2001 才会开始执行，所以看似简单的查询会变慢，主要时间消耗在队列等待上面。

### 7.3.6 总结处理思路

通过前面的原因和场景，总结一下可能的处理思路及手段如下：

1. 针对开销较大的 SQL，可以做出相应的 SQL 优化，来避免大量扫表。（4.0 的 unified thread pool 针对这种情况有优化）
2. 针对事务冲突的场景，可以通过添加分布式锁，或者使用悲观事务模型来解决。
3. 对于热点更新的场景，可以通过 region 拆分、shard\_row\_id\_bits、pre\_split\_regions 等方式优化。
4. 如果是可用的线程较少，导致了线程池排队，可以增加 tikv 节点，来提高集群的处理能力。

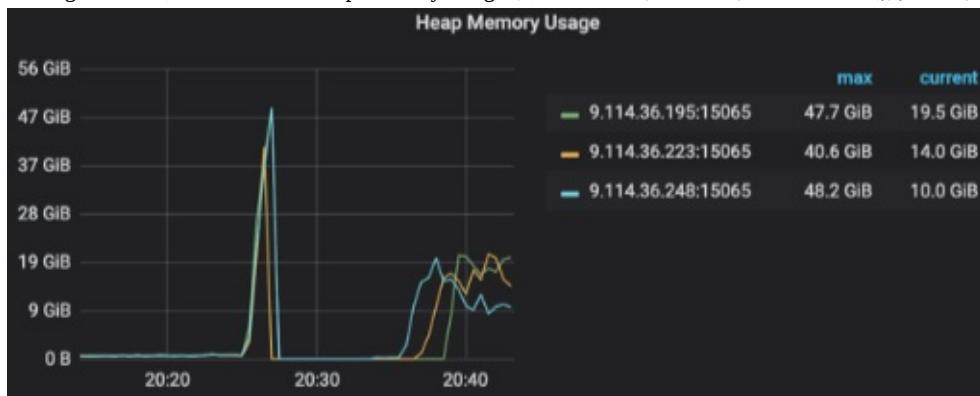
## 7.4 TiDB OOM 的常见原因

日常 TiDB 运维中，会遇到组件 OOM 的问题，通常出现在 TiDB-Server 和 TiKV-Server，分别来看一下出现的常见原因以及处理建议。

### 7.4.1 TiDB-Server

#### 1. 如何快速确认 TiDB-Server 出现了 OOM

- 客户端收到 tidb-server 报错 "ERROR 2013 (HY000): Lost connection to MySQL server during query"
- TiDB grafana 面板中 Server 的 Heap Memory Usage 项，出现一次或者多次内存上涨并突然下跌到底的情况



- 查看 TiDB service 的启动时间

```
$ systemctl status tidb-4000.service
● tidb-4000.service - tidb-4000 service
  Loaded: loaded (/etc/systemd/system/tidb-4000.service; disabled; vendor preset: disabled)
  Active: active (running) since Fri 2020-02-21 12:30:04 CST; 5 min ago
```

- 查看日志

- /var/log/message&kern，是否出现“out of memory”字样，和“TiDB service was killed”
- tidb.log 中可以 grep 到事故发生后附近时间的“Welcome to TiDB”的日志（即 tidb-server 发生重启）
- tidb\_stderr.log 中能 grep 到“fatal error: runtime: out of memory”或“cannot allocate memory”

#### 2. TiDB-Server 出现 OOM 的常见原因

导致 TiDB oom 的本质原因是 TiDB 现有的内存小于 TiDB 要使用的内存。

- 复杂查询

SQL 执行计划中的 root 任务占比过高，TiDB-Server 需要缓存大量数据。比如执行计划中使用了 hash join、过多的子查询、Update&Delete 涉及数据量过大等等。

- 机器内存不足，机器上部署的其他应用占用了较多内存，如 Prometheus
- 活跃线程数过多，导致内存用尽
- mydumper导数据时，并发量太大

#### 3. 如何缓解 TiDB-Server OOM 问题

- 参照本书“快速定位慢 SQL”的章节，通过添加索引或者优化 SQL 语句来解决
- 配合 tidb\_mem\_quota\_query 限制单个 Query 的内存消耗，默认值为 32GB

```
global:
mem-quota-query: 209715200 //举个例子：将阈值设置为 200MB
oom-action: "cancel" //超过阈值的 SQL 自动杀掉
```

- max\_execution\_time 限制单个 SQL 执行时间

```
set @@GLOBAL.max_execution_time=10000; //限制 SQL 执行最长时间为 10s
```

- 开启 SWAP，可以缓解由于大查询使用内存过多而造成的 OOM 问题。

但该方法会在内存空间不足时，由于存在 IO 开销，因此大查询性能造成一定影响。性能回退程度受剩余内存量、读写盘速度影响。

- 目前 4.0 版本已经支持 oom-use-tmp-storage 功能

设置为 `true` 可以使单条 SQL 内存使用在超出 `mem-quota-query` 时为某些算子启用临时磁盘

- 增大 TiDB 现有内存

TiDB 实例默认使用服务器全部内存的，提升物理服务器内存真实场景中并不现实。但当前市场上服务器多 NUMA 架构，TiDB 使用建议中存在着 NUMA 绑核的机制。

如果在做绑核动作时，如果使用了 membind 参数 (Only allocate memory from nodes. Allocation will fail when there is not enough memory available on these nodes. nodes may be specified as noted above) 绑定到 NUMA NODE 上，则默认只能分配该 NUMA NODE 上的内存，而非全部；

推荐使用 preferred 参数 (Only allocate memory from nodes. Allocation will fail when there is not enough memory available on these nodes. nodes may be specified as noted above) 进行绑定，其含义为首先使用分配的 NUMA NODE 的内存资源，不足时会使用其他节点的内存。

这样可以提高 TiDB 使用内存的水位。

```
sudo -s numactl --preferred=node
```

- 减少 TiDB 使用内存。

#### 1) 优化单一大 SQL

当出现 TiDB OOM 时，去查看对应的 `tidb_slow_query.log` 通过 `Query_time` 和 `Mem_max` 值定位使用内存过多的慢 SQL。

如果是查询 SQL 可通过其执行计划定位，是否存在以下动作：

全表扫描 table scan 优化思路：考虑新建索引。

hash join 优化思路：注意表之间的排列顺序，让筛选性好的表优先 join；考虑使用 index loop join 替代。

如果是 delete 语句，可以考虑使用 `tidb_batch_delete` 方式，或业务上分片方式，达到少量多次的效果。

#### 2) 横向扩容 TiDB

面对没有单一大 SQL，而是并发较高的场景，可以选择横向扩展 TiDB 节点来缓解单一 TiDB 实例的压力，从而起到缓解 oom 的作用。

## 7.4.2 TiKV-Server

TiKV-Server OOM 的常见场景有如下几种，不同场景的处理建议也有不同。

### 1. block-cache 配置过大

在 TiKV grafana 选中对应的 instance 之后查看 RocksDB 的 block cache size 监控来确认是否是该问题。同时检查 `capacity` 参数是否设置合理，默认情况下 TiKV 的 block-cache 设置为机器总内存的 45%，在 container 部署的时候需要显式指定该参数，因为 TiKV 获取的是物理机的内存，可能会超出 container 的内存限制。

```
[storage.block-cache]
capacity = "30GB"
```

处理建议：

- 适当增大机器内存
- 适当减小 block-cache

### 2. coprocessor 收到大量大查询，需要返回大量的数据

gRPC 发送速度跟不上 coprocessor 往外吐数据的速度导致 OOM。可以通过检查 [tikv-detail]->[Coprocessor Overview]->[Total Response Size] 是否超过 network outbound 流量来确认是否属于这种情况。

处理建议：

1. 使用万兆网卡，提高数据传输速度
2. 参照本书 "快速定位慢 SQL" 的章节，检查是否存在全表扫描的大查询
3. 检查 gRPC poll CPU 是否不足

### 3. Raft apply 线程短时间需要处理大量 raft 日志，apply log 过程速度慢

Apply 日志不及时可能导致 apply channel 中内存堆积，堆积严重导致系统内存不足则会出现 oom。通常是 apply wait duration 和 apply log duration 过高。相关监控位于：

- [tikv-detail]->[Raft IO]->[apply log duration]
- [tikv-detail]->[Raft propose]->[apply wait duration]
- [tikv-detail]->[Thread CPU]->[async apply CPU]

处理建议：

1. 检查 apply cpu 线程是否存在瓶颈，如果 [Async apply CPU] 超过了 [apply-pool-size 数量] \* 70% 说明需要加大 apply-pool-size。
2. 检查 io 负载情况如磁盘吞吐量是否打满、写延迟是否过高。

## 7.5 TiKV 磁盘空间占用与回收常见问题

TiKV 作为 TiDB 的存储节点，用户通过 SQL 导入或更改的所有数据都存储在 TiKV。这里整理了一些关于 TiKV 空间占用的常见问题

### TiKV 的空间放大

- 监控上显示的 Number files at each levels 是什么含义？如果用户向 TiDB 中写入了 10G 数据，那么实际占用的物理空间是多大？

TiKV 采用 LSM-Tree 架构的 RocksDB 作为底层存储引擎，最新写入的数据会在最上层，最老的数据在最底层。如果用户只执行过 INSERT 而没有 UPDATE 和 DELETE 的话，那么按照默认配置 `max-bytes-for-level-multiplier`，每一层的大小是上一层的十倍。RocksDB 相同层不会有重复的数据，再乘以三个副本，因此 10GB 数据最多占据  $(512\text{MB} + 1\text{GB} + 10\text{GB}) * 3$  的物理空间，由于 RocksDB 还采取了针对 key 的前缀压缩，以及针对 block 的 LZ4 或 ZSTD 压缩，因此最终占用的磁盘空间肯定小于 33.5GB. (512MB 为 L0 的 SST 文件大小。这里没有考虑索引的大小)

- TiDB 文档和配置中提到的 GC 是什么意思？

TiDB 采用 MVCC 事务模型，并且支持了 Snapshot Isolation 级别的事务隔离，因此为了保证正在进行中的事务能够读取到一致的数据，所有的 DELETE 以及 UPDATE 操作在 TiDB 中都不会立刻将原来的数据在物理上删除或者更改，而是为其新增一个版本，这样就保证了旧的版本仍然能被尚未结束的事务读取到。每隔一段时间 TiDB 会确认某个时间点之前的事务已经全部结束了，那么所有的数据在该时间点之前的版本都可以只保留最新的那一个，于是 TiDB 会将这个时间点通知给 TiKV，TiKV 则会发起清理旧版本数据以回收物理空间的操作，这个操作被称作 `GC`。

- 为什么我执行了 UPDATE SQL 之后，集群占用的空间在不停地增长？UPDATE 的数据会占用额外的空间吗？

参见上一条，对于 UPDATE 的数据不会立刻覆盖其原有的数据，而是为其新增一个版本，因此会占用额外的物理空间。TiDB 默认的 `tikv_gc_life_time` 为 10 分钟，因此 UPDATE 所覆盖的旧版本数据会在至少 10 分钟后才被删除。由于 TiKV 上的 GC 线程为单线程，因此目前的版本还存在 UPDATE 过快而导致旧版本来不及回收，数据大小膨胀的问题，未来 TiDB 会解决这个问题。倘若 GC 及时的话，那么用户 UPDATE 后 TiKV 占用的实际空间为 "用户 10 分钟内更新的数据量 + 数据库有效数据量 \* 1.12". (这里的 1.12 参考 上上条推断的空间放大系数)

- TiKV 的写入性能周期性下降（10~20分钟一轮）这是怎么回事？

建议检测 TiKV 监控中的 GC 一栏中，GC speed 的指标是否与 TiKV 写入性能下降的周期波动重合。TiKV 的 GC 由 raft leader 发起，然后将需要删除的旧版本通过一致性协议发送给 follower 删除，因此会抢占正常的业务写入的资源。可以通过 TiKV 的配置 `gc.max-write-bytes-per-sec` 来限制 GC 的速度，根据机器配置建议该值设置为 `128KB ~ 512KB`，默认值为 `0KB`，即不进行任何限速。

### 如何高效地回收磁盘空间

- 为什么我执行了 `DELETE FROM table_xx;` 后磁盘空间迟迟没有回收？(监控上显示的磁盘剩余空间并没有增大)

参考上文中对 GC 的解释，TiDB 删除数据也是为其增加一个特殊的新版本，旧版本要等待至少 10 分钟后才会真正从 RocksDB 中删除，而 RocksDB 回收物理空间还需要更多的额外时间。因此我们建议用户如果要删除某个表的数据尽量使用 `DROP TABLE table_xxx`，而不是 `DELETE FROM table_xx`。前者会在超过 GC 时间后，直接删除 RocksDB 在磁盘上的物理文件。

- TiDB 旧版本数据过期时间是可配置的吗？应该如何调整这个配置的大小？

可以通过 MySQL 客户端连接 TiDB，查看 TiDB 的系统表 `SELECT * from mysql.tidb`，`tikv_gc_life_time` 即为旧版本的过期时间，用户可以动态调整该配置，但是 TiDB 不允许该配置的值低于 10 分钟，更低的值将被忽略。建议用户不要把这个值设置得过大，以免浪费更多的磁盘空间，同时还可能因积累旧版本数据过多，导致 GC 流量过大影响了其他业务。

- GC 删除数据所占据的物理空间能在 RocksDB 中被立刻回收吗？

GC 删除的数据会很快被 compact 到下一层。在 TiKV 的 CPU 资源充足，RocksDB compact 足够及时的情况下，由于相同层内不会有重复数据，因此最多存在 12% 应该被删除的重复无效数据，这是由于 rocksdb 的写放大带来的数据。

## Dynamic Level 相关问题

- 为什么 TiKV 的监控上显示 level-1 和 level-2 都没有数据，但是 level-3 和 level-4 却有数据？

因为 TiKV 使用 RocksDB 开启了 **Dynamic Level Bytes**，所以数据文件会优先放更底层。计算规则：如果当前数据总大小低于 `max-bytes-for-level-base`（默认为 512MB），则所有数据都会在 level-6，此时 level-6 实际上相当于 level-1。如果数据总大小超过 `max-bytes-for-level-base`，但低于 `max-bytes-for-level-base * max-bytes-for-level-multiplier`，则 level-6 视作 level-2，level-5 视作 level-1。但是无论如何，除了 level-0 以外的各层数据比例都按照上层比下层 1 : 10 进行分布。

- 磁盘空间不够，如何提高 TiKV 的压缩效果？

TiKV 提供 snappy，zlib，bzip2，lz4，lz4hc，zstd 等六种压缩算法。默认为 `["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]` 注意我们采取了 dynamic level，所以只有当数据量超过 500G 时 RocksDB 的层数才会超过 4，超过 500G 部分的数据才会启动 ZSTD 压缩算法。如果希望能够进一步提高压缩效果，可以将 defaultcf 以及 writecf 的配置 `compression-per-level` 设置为 `["no", "no", "lz4", "lz4", "zstd", "zstd", "zstd"]`，这样的话，50G ~ 500G 之间的数据的也能按照 zstd 压缩。

## 第8章 TiDB 调优指南

“性能”一词，耳熟能详。伴随业务发展，数据库性能挑战更剧。为此，本章节主要从 TiDB 分布式数据库配置、索引加速两方面入手，详解 TiDB、TiKV、PD 三大核心组件的常见配置优化项，保障数据库性能。

目录：

- 8.1 TiDB 常见配置优化
- 8.2 TiKV 常见配置优化
- 8.3 添加索引调优加速

## 8.1 TiDB 常见配置优化

在前一章节，介绍了 TiDB 常见问题处理思路，本章节主要介绍一些 TiDB 常见的配置优化。

### 8.1.1 限制 SQL 内存使用和执行时间

对于关系型数据库来说，SQL 效率毋庸置疑至关重要。可以想象一下，假设在数据库内没有任何控制机制，某条或某几条 SQL 执行时间长且耗用内存高，那么只能依赖告警系统且人工去快速定位 SQL 然后 Kill，期间效率可想而知，而在 TiDB 存在参数能让我们能够很好的限制 SQL 内存使用和执行时间。

#### (1) 执行时间限制

max\_execution\_time

- 作用域：GLOBAL | SESSION | SQL HINT
- 默认值：0
- 含义：该变量会限制语句的执行时间不能超过 N 毫秒，否则服务器会终止这条语句的执行，SQL Hint 方式具体示例如下：

SQL Hint:

```
mysql> SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM t1 INNER JOIN t2 WHERE ...;
```

#### (2) 内存使用限制

tidb\_mem\_quota\_query

- 作用域：SESSION
- 默认值：32 GB
- 含义：该变量是系统变量，用来设置一条查询语句的内存使用阈值。如果一条查询语句执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 OOMAction 项所指定的行为，配置文件 oom-action 默认值 log，表示打印超过内存阈值 SQL，可通过配置 cancel 实现 Kill SQL 语句，SQL 设置具体示例如下：

```
--设置内存使用限制为10G
mysql> set @@session.tidb_mem_quota_query=10737418240;
Query OK, 0 rows affected (0.00 sec)
mysql> show variables like '%tidb_mem_quota_query%';
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| tidb_mem_quota_query | 10737418240 |
+-----+-----+
1 row in set (0.00 sec)
```

mem-quota-query

- 作用域：GLOBAL
- 默认值：32GB
- 含义：该变量是 TiDB 全局 Global 配置，需在 TiDB 配置文件设置（可在 tidb-ansible conf/tidb.yaml 配置 mem-quota-query 滚更生效），如果一条查询语句执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 OOMAction 项所指定的行为，配置文件 oom-action 默认值 log，表示打印超过内存阈值 SQL，可通过配置 cancel 实现 Kill SQL 语句，具体示例如下：

```
conf/tidb.yaml
---
# default configuration file for TiDB in yaml format

global:
  ...
  # Only print a log when out of memory quota.
  # Valid options: ["log", "cancel"]
  # oom-action: "log"

  # Set the memory quota for a query in bytes. Default: 32GB
  # mem-quota-query: 34359738368
```

## 8.1.2 事务重试设置

TiDB 数据库锁机制有别于传统数据库悲观锁，采用乐观锁，2PC 提交，此处不作具体展开，详情可查看 TiDB 事务模型章节。针对事务冲突处理，可根据业务场景按需决定是否事务重试。

### tidb\_retry\_limit

- 作用域：SESSION | GLOBAL
- 默认值：10
- 含义：该变量用来设置最大重试次数。一个事务执行中遇到可重试的错误（例如事务冲突、事务提交过慢或表结构变更）时，会根据该变量的设置进行重试。注意当 tidb\_retry\_limit = 0 时，也会禁用自动重试，SQL 设置具体示例如下：

```
--设置SESSION作用域
mysql> set @@session.tidb_retry_limit=0;
Query OK, 0 rows affected (0.01 sec)

--设置GLOBAL作用域
mysql> set @@global.tidb_retry_limit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like '%tidb_retry_limit%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| tidb_retry_limit | 0     |
+-----+-----+
```

### tidb\_disable\_txn\_auto\_retry

- 作用域：SESSION | GLOBAL
- 默认值：on
- 含义：该变量用来设置是否禁用显式事务自动重试，若将变量值设置为 on 时，表示不会自动重试，遇到事务冲突需要在应用层重试。若将变量值设为 off，遇到事务冲突 TiDB 将会自动重试事务，这样在事务提交时遇到的错误更少。需要注意的是，这样可能会导致数据更新丢失。该变量不会影响自动提交的隐式事务和 TiDB 内部执行的事务，它们依旧会根据 tidb\_retry\_limit 的值来决定最大重试次数，SQL 设置具体示例如下：

```
--设置SESSION作用域
mysql> set @@session.tidb_disable_txn_auto_retry=OFF;
Query OK, 0 rows affected (0.00 sec)

--设置GLOBAL作用域
mysql> set @@global.tidb_disable_txn_auto_retry=OFF;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like '%tidb_disable_txn_auto_retry%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| tidb_disable_txn_auto_retry | OFF   |
+-----+-----+
1 row in set (0.01 sec)
```

### 8.1.3 Join 算子优化

TiDB 数据库 SQL 执行，Join 算子天然并发，当系统资源富余时，可根据数据库 TP | AP 应用可适当调整 Join 算子并发提高 SQL 执行效率，提升数据库系统性能。

`tidb_distsql_scan_concurrency`

- 作用域：SESSION | GLOBAL
- 默认值：15
- 含义：该变量用来设置 scan 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值。对于 AP 类应用，最大值建议不要超过所有 TiKV 节点的 CPU 核数，SQL 设置具体示例如下：

```
--设置SESSION作用域
mysql> set @@session.tidb_distsql_scan_concurrency=30;
Query OK, 0 rows affected (0.00 sec)

--设置GLOBAL作用域
mysql> set @@global.tidb_distsql_scan_concurrency=30;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like '%tidb_distsql_scan_concurrency%';
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| tidb_distsql_scan_concurrency | 30   |
+-----+-----+
1 row in set (0.01 sec)
```

`tidb_index_lookup_size`

- 作用域：SESSION | GLOBAL
- 默认值：20000
- 含义：该变量用来设置 index lookup 操作的 batch 大小，AP 类应用适合较大的值，TP 类应用适合较小的值，SQL 设置具体示例如下：

```
--设置SESSION作用域
mysql> set @@session.tidb_index_lookup_size=40000;
Query OK, 0 rows affected (0.00 sec)

--设置GLOBAL作用域
mysql> set @@global.tidb_index_lookup_size=40000;
Query OK, 0 rows affected (0.01 sec)

mysql> show variables like '%tidb_index_lookup_size%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| tidb_index_lookup_size | 40000 |
+-----+-----+
1 row in set (0.00 sec)
```

#### tidb\_index\_lookup\_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：4
- 含义：该变量用来设置 index lookup 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值，SQL设置具体示例如下：

```
--设置SESSION作用域
mysql> set @@session.tidb_index_lookup_concurrency=8;
Query OK, 0 rows affected (0.00 sec)

--设置GLOBAL作用域
mysql> set @@global.tidb_index_lookup_concurrency=8;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like '%tidb_index_lookup_concurrency%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| tidb_index_lookup_concurrency | 8 |
+-----+-----+
1 row in set (0.00 sec)
```

#### tidb\_index\_lookup\_join\_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：4
- 含义：该变量用来设置 index lookup join 算法的并发度，SQL设置具体示例如下：

```
--设置SESSION作用域
mysql> set @@session.tidb_index_lookup_join_concurrency=8;
Query OK, 0 rows affected (0.00 sec)
--设置GLOBAL作用域
mysql> set @@global.tidb_index_lookup_join_concurrency=8;
Query OK, 0 rows affected (0.01 sec)
mysql> show variables like '%tidb_index_lookup_join_concurrency%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| tidb_index_lookup_join_concurrency | 8 |
+-----+-----+
1 row in set (0.00 sec)
```

#### tidb\_hash\_join\_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：5
- 含义：该变量用来设置 hash join 算法的并发度，SQL设置具体示例如下：

```
--设置SESSION作用域
mysql> set @@session.tidb_hash_join_concurrency=10;
Query OK, 0 rows affected (0.01 sec)

--设置GLOBAL作用域
mysql> set @@global.tidb_hash_join_concurrency=10;
Query OK, 0 rows affected (0.00 sec)

mysql> show variables like '%tidb_hash_join_concurrency%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| tidb_hash_join_concurrency | 10    |
+-----+-----+
1 row in set (0.01 sec)
```

#### tidb\_index\_serial\_scan\_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：1
- 含义：该变量用来设置顺序 scan 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值，SQL 设置具体示例如下：

```
--设置SESSION作用域
mysql> set @@session.tidb_index_serial_scan_concurrency=4;
Query OK, 0 rows affected (0.00 sec)

--设置GLOBAL作用域
mysql> set @@global.tidb_index_serial_scan_concurrency=4;
Query OK, 0 rows affected (0.01 sec)

mysql> show variables like '%tidb_index_serial_scan_concurrency%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| tidb_index_serial_scan_concurrency | 4    |
+-----+-----+
1 row in set (0.00 sec)
```

### 8.1.4 常见 Mysql 兼容问题

#### compatible-kill-query

- 默认值：false
- 含义：设置 Kill 语句的兼容性，TiDB 中 Kill sessionID 的行为和 MySQL 中的行为不相同。杀死一条查询，在 TiDB 里需要加上 TiDB 关键词，即 Kill TiDB sessionID。但如果把 compatible-kill-query 设置为 true，则不需要加上 TiDB 关键词。这种区别很重要，因为当用户按下 Ctrl+C 时，MySQL 命令行客户端的默认行为是：创建与后台的新连接，并在该新连接中执行 Kill 语句。如果负载均衡器或代理已将该新连接发送到与原始会话不同的 TiDB 服务器实例，则该错误会话可能被终止，从而导致使用 TiDB 集群的业务中断。只有当确定在 Kill 语句中引用的连接正好位于 Kill 语句发送到的服务器上时，才可以启用 compatible-kill-query，具体示例如下(修改 tidb-ansible conf/tidb.yaml 配置再滚更)：

```
conf/tidb.yaml
---
# default configuration file for TiDB in yaml format

global:
  ...
  # Make "kill query" behavior compatible with MySQL. It's not recommend to
  # turn on this option when TiDB server is behind a proxy.
  compatible-kill-query: false
```

```
tidb_constraint_check_in_place
* 作用域: SESSION | GLOBAL
* 默认值: 0
* 含义: TiDB 支持乐观事务模型, 即在执行写入时, 假设不存在冲突。冲突检查是在最后 commit 提交时才去检查。这里的检查指 unique key 检查。
该变量用来控制是否每次写入一行时就执行一次唯一性检查。注意, 开启该变量后, 在大批量写入场景下, 对性能会有影响。示例:
```

```
--默认关闭 tidb_constraint_check_in_place 参数行为
create table t (i int key);
insert into t values (1);
begin;
insert into t values (1);
Query OK, 1 row affected
commit;    --commit 时才去做检查, 并报错重复主键
ERROR 1062 : Duplicate entry '1' for key 'PRIMARY'
```

```
--打开 tidb_constraint_check_in_place 参数行为
set @@tidb_constraint_check_in_place=1;
begin;
insert into t values (1);
ERROR 1062 : Duplicate entry '1' for key 'PRIMARY'
```

## 8.1.5 其他优化项

prepared-plan-cache 以及 txn\_local\_latches 两个参数主要是 TiDB 配置参数, 需要在 TiDB 配置文件中设置, 可在 tidb-ansible conf/tidb.yaml 设置, 再滚更 tidb-server 节点。

```
conf/tidb.yaml
-- 执行计划缓存
prepared_plan_cache:
  enabled: true          -- 是否开启 prepare 语句的 plan cache, 默认值 false
  capacity: 100           -- 缓存语句的数量
  memory-guard-ratio: 0.1 -- 用于防止超过 performance.max-memory, 超过 max-proc * (1 - prepared-plan-cache.memory-guar-
                           rd-ratio) 会剔除 LRU 中的元素, 最小值为 0 ; 最大值为 1, 默认值 0.1

-- 事务内存锁相关配置, 当本地事务冲突比较多时建议开启
txn_local_latches:
  enable: true            -- 是否开启事务内存锁相关配置, 默认值 false
  capacity: 2048000        -- Hash 对应的 slot 数, 会自动向上调整为 2 的指数倍。每个 slot 占 32 Bytes 内存。当写入数据的范围
                           比较广时 (如导数据), 设置过小会导致变慢, 性能下降。
```

## 8.2 TiKV 常见配置优化

本章节用于描述如何根据机器配置情况来调整 TiKV 的参数，使 TiKV 的性能达到最优。

TiKV 最底层使用的是 RocksDB 做为持久化存储，所以 TiKV 的很多性能相关的参数都是与 RocksDB 相关的。TiKV 使用了两个 RocksDB 实例，默认 RocksDB 实例存储 KV 数据，Raft RocksDB 实例（简称 RaftDB）存储 Raft 数据。

TiKV 使用了 RocksDB 的 `column Families (CF)` 特性。

- 默认 RocksDB 实例将 KV 数据存储在内部的 `default`、`write` 和 `lock` 3 个 CF 内。
  - `default` CF 存储的是真正的数据，与其对应的参数位于 `[rocksdb.defaultcf]` 项中；
  - `write` CF 存储的是数据的版本信息 (MVCC) 以及索引相关的信息，相关的参数位于 `[rocksdb.writecf]` 项中；
  - `lock` CF 存储的是锁信息，系统使用默认参数。
- Raft RocksDB 实例存储 Raft log。
  - `default` CF 主要存储的是 Raft log，与其对应的参数位于 `[raftdb.defaultcf]` 项中。

所有的 CF 默认共同使用一个 block cache 实例。通过在 `[storage.block-cache]` 下设置 `capacity` 参数，可以配置该 block cache 的大小。block cache 越大，能够缓存的热点数据越多，读取数据越容易，同时占用的系统内存也越多。如果要为每个 CF 使用单独的 block cache 实例，需要在 `[storage.block-cache]` 下设置 `shared=false`，并为每个 CF 配置单独的 block cache 大小。例如，可以在 `[rocksdb.writecf]` 下设置 `block-cache-size` 参数来配置 `write` CF 的大小。

每个 CF 有各自的 `write-buffer`，大小通过 `write-buffer-size` 控制。

## TiKV 内存使用情况

除了以上列出的 `block-cache` 以及 `write-buffer` 会占用系统内存外：

1. 需预留一些内存作为系统的 page cache
2. TiKV 在处理大的查询的时候（例如 `select * from ...`）会读取数据然后在内存中生成对应的数据结构返回给 TiDB，这个过程中 TiKV 会占用一部分内存

## TiKV 机器配置推荐

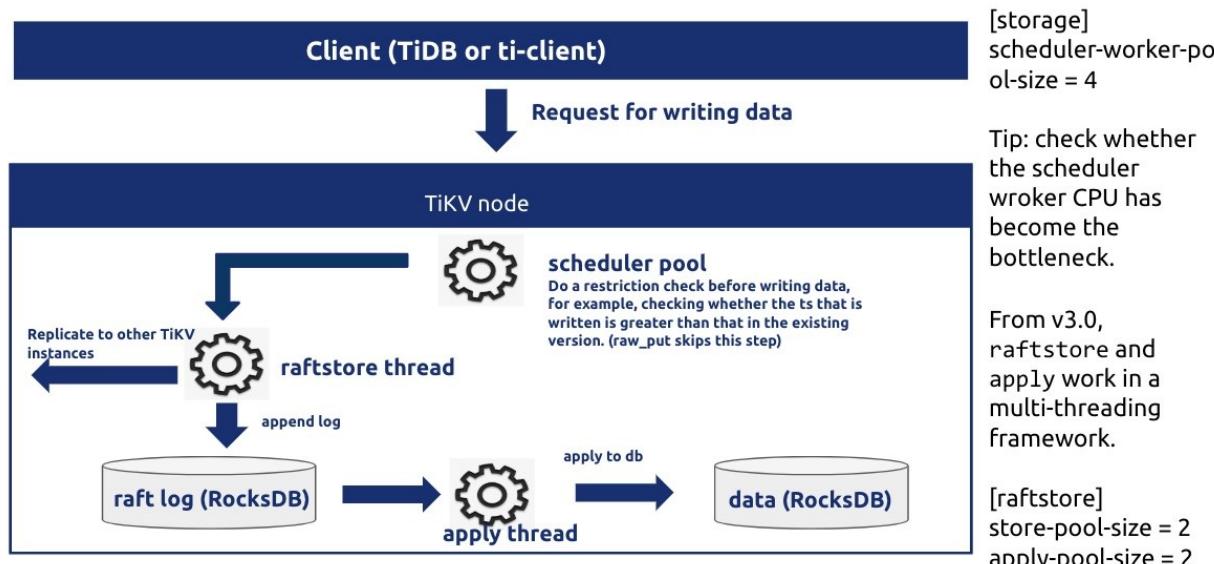
1. 生产环境中，不建议将 TiKV 部署在 CPU 核数小于 8 或内存低于 32GB 的机器上
2. 如果对写入吞吐要求比较高，建议使用吞吐能力比较好的磁盘
3. 如果对读写的延迟要求非常高，建议使用 IOPS 比较高的 SSD 盘

具备上述基础知识后，本章节将详细介绍 TiKV 线程池优化，海量 Region 集群调优，以及其他常见优化设置，希望可以帮助读者了解如何根据业务场景需要配置 TiKV。

## 8.2.1 TiKV 线程池优化

在 TiKV 4.0 中，线程池主要由 gRPC、Scheduler、UnifyReadPool、Store、Apply、RocksDB 以及其它一些占用 CPU 不多的定时任务与检测组件组成。

- gRPC 线程池是 TiKV 所有读写请求的总入口，它会把不同任务类型的请求转发给不同的线程池。
- Scheduler 线程池负责检测写事务冲突，把事务的两阶段提交、悲观锁上锁、事务回滚等请求转化为 key-value 对数组，然后交给 Store 线程进行 Raft 日志复制。
- Raftstore 线程池负责处理所有的 Raft 消息以及添加新日志的提议（Propose）、将日志写入到磁盘，当日志在 Raft Group 中达成多数一致（即 Raft 论文中描述的 Commit Index）后，它就会把该日志发送给 Apply 线程。
- Apply 线程收到从 Store 线程池发来的已提交日志后将其解析为 key-value 请求，然后写入 RocksDB 并且调用回调函数通知 gRPC 线程池中的写请求完成，返回结果给客户端。
- RocksDB 线程池是 RocksDB 进行 Compact 和 Flush 任务的线程池，关于 RocksDB 的架构与 Compact 操作请参考 [RocksDB: A Persistent Key-Value Store for Flash and RAM Storage](#)
- UnifyReadPool 是 TiKV 4.0 推出的新特性，它由之前的 Coprocessor 线程池与 Storage Read Pool 合并而来，所有的读取请求包括 kv get、kv batch get、raw kv get、coprocessor 等都会在这个线程池中执行。



## 1. GRPC

gRPC 线程池的大小默认配置 (`server.grpc-concurrency`) 是 4。由于 gRPC 线程池几乎不会有计算开销，它主要负责网络 IO、反序列化请求，因此该配置通常不需要调整，如果部署的机器 CPU 特别少（小于等于 8），可以考虑将该配置 (`server.grpc-concurrency`) 设置为 2，如果机器配置很高，并且 TiKV 承担了非常大量的读写请求，观察到 Grafana 上的监控 Thread CPU 的 gRPC poll CPU 的数值超过了 `server.grpc-concurrency` 大小的 80%，那么可以考虑适当调大 `server.grpc-concurrency` 以控制该线程池使用率在 80% 以下。

## 2. Scheduler

Scheduler 线程池的大小配置 (`storage.scheduler-worker-pool-size`) 在 TiKV 检测到机器 CPU 数大于等于 16 时默认为 8，小于 16 时默认为 4。它主要用于将复杂的事务请求转化为简单的 key-value 读写。但是 **scheduler** 线程池本身不进行任何写操作，如果检测到有事务冲突，那么它会提前返回冲突结果给客户端，否则的话它会把需要写入的 key-value 合并成一条 Raft 日志交

给 raftstore 线程进行 raft 日志复制。通常来说为了避免过多的线程切换，最好确保 scheduler 线程池的利用率保持在 50% ~ 75% 之间。（如果线程池大小为 8 的话，那么 Grafana 上的 Thread CPU 的 scheduler worker CPU 应当在 400% ~ 600% 之间较为合理）

### 3. Raftstore

Raftstore 线程池是 TiKV 最为复杂的一个线程池，默认大小（raftstore.store-pool-size）为 2，所有的写请求都会先在 store 线程 fsync 的方式写入 RocksDB（除非手动将 raftstore.sync-log 设置为 true；而 raftstore.sync-log 设置为 false，可以提升一部分写性能，但也会造成数据丢失的可能）。由于存在 IO，store 线程理论上不可能达到 100% 的 CPU。为了尽可能地减少写磁盘次数，将多个写请求攒在一起写入 RocksDB，最好控制其 CPU 使用在 40% ~ 60%。千万不要为了提升写性能盲目增大 store 线程池大小，这样可能反而会适得其反，增加了磁盘负担让性能变差。

- 监控 Raft IO 中的 append log duration 表示 Store 线程处理一批 Raft Message 并且将需要持久化的日志写入 RocksDB（这里存储的格式是 Raft 日志格式）中的时间。如果该指标较高（P99 大于 500ms），说明盘比较慢（查看监控 Node Exporter 中的 disk latency 判断盘的性能是否有问题），或者是写入 RocksDB 时触发了 [Write Stall](#)。此时可以查看 RocksDB-raft 中的 write stall duration，该指标正常情况下应该为 0，若不为 0 则可能发送了 write stall。

### 4. UnifyReadPool

UnifyReadPool 负责处理所有的读取请求。默认配置（readpool.unified.max-thread-count）大小为机器 CPU 数的 80%。通常建议根据业务负载特性调整其 CPU 使用率在 60% ~ 90% 之间。

### 5. RocksDB

RocksDB 线程池是 RocksDB 进行 Compact 和 Flush 任务的线程池，通常不需要配置，如果机器 CPU 数较少，可将 rocksdb.max-background-jobs 与 raftdb.max-background-jobs 同时改为 4。如果遇到了 [Write Stall](#)，查看 Grafana 监控上 RocksDB-kv 中的 Write Stall Reason 有哪些指标不为 0。如果是由 pending compaction bytes 相关原因引起的，可将 rocksdb.max-sub-compactions 设置为 2（该配置表示单次 compaction job 允许使用的子线程数量）。如果原因是 memtable count 相关，建议调大所有列的 max-write-buffer-number（默认为 5）。如果原因是 level0 file limit 相关，建议调大如下参数：

```
rocksdb.defaultcf.level0-slowdown-writes-trigger
rocksdb.writecf.level0-slowdown-writes-trigger
rocksdb.lockcf.level0-slowdown-writes-trigger
```

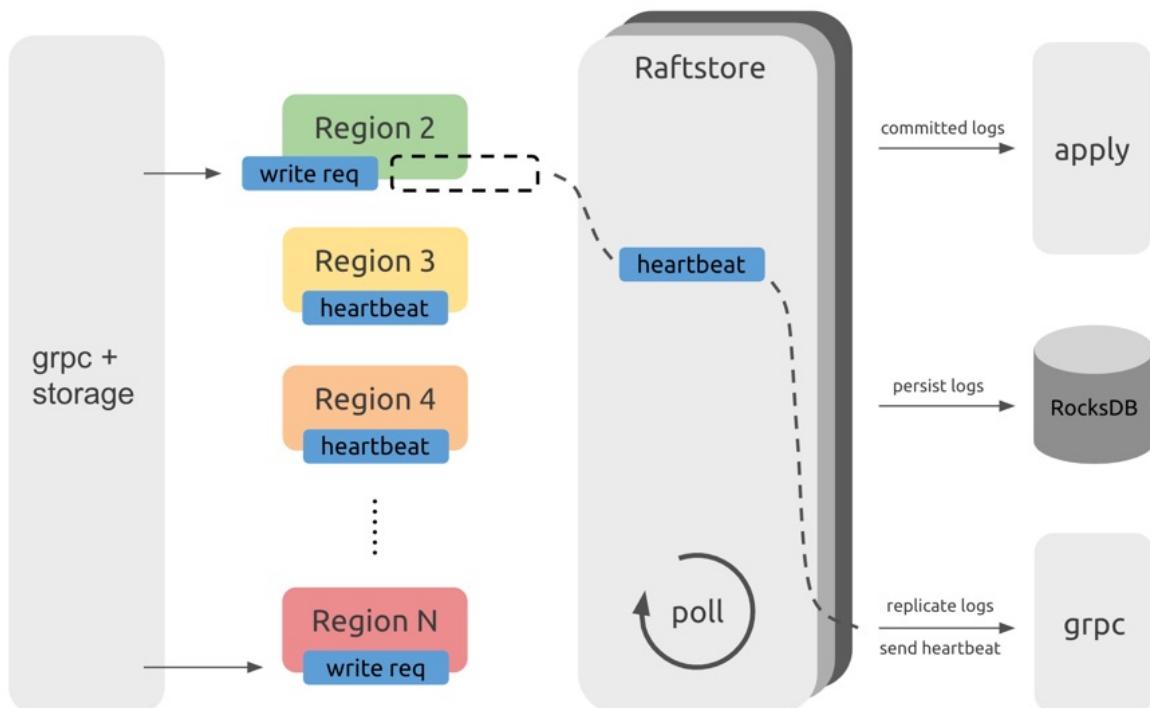
## 8.2.2 海量 Region 集群调优

在 TiDB 的架构中，所有数据以一定 range 将数据 key 切分成若干 Region 分布在多个 TiKV 实例上。随着数据的写入，一个集群中会产生上百万个甚至千万个 Region。单个 TiKV 实例上产生过多的 Region 会给集群带来较大的负担，影响整个集群的性能表现。

本将介绍 TiKV 核心模块 Raftstore 的工作流程，海量 Region 导致性能问题的原因，以及优化性能的方法。

### 1. Raftstore 的工作流程

一个 TiKV 实例上有多个 Region。Region 消息是通过 Raftstore 模块驱动 Raft 状态机来处理的。这些消息包括 Region 上读写请求的处理、Raft log 的持久化和复制、Raft 的心跳处理等。但是，Region 数量增多会影响整个集群的性能。为了解释这一点，需要先了解 TiKV 的核心模块 Raftstore 的工作流程。



注意：该图仅为示意，不代表代码层面的实际结构。

上图是 Raftstore 处理流程的示意图。如图所示，从 TiDB 发来的请求会通过 gRPC 和 storage 模块变成最终的 KV 读写消息，并被发往相应的 Region，而这些消息并不会被立即处理而是被暂存下来。Raftstore 会轮询检查每个 Region 是否有需要处理的消息。如果 Region 有需要处理的消息，那么 Raftstore 会驱动 Raft 状态机去处理这些消息，并根据这些消息所产生的状态变更去进行后续操作。例如，在有写请求时，Raft 状态机需要将日志落盘并且将日志发送给其他 Region 副本；在达到心跳间隔时，Raft 状态机需要将心跳信息发送给其他 Region 副本。

### 2. 性能问题

从 Raftstore 处理流程示意图可以看出，需要依次处理各个 Region 的消息。那么在 Region 数量较多的情况下，Raftstore 需要花费一些时间去处理大量 Region 的心跳，从而带来一些延迟，导致某些读写请求得不到及时处理。如果读写压力较大，Raftstore 线程的 CPU 使用率容易达到瓶颈，导致延迟进一步增加，进而影响性能表现。

通常在有负载的情况下，如果 Raftstore 的 CPU 使用率达到了 85% 以上，即可视为达到繁忙状态且成为了瓶颈，同时 Grafana TiKV 监控 Raft Propose 面板 propose wait duration 可能会高达百毫秒级别。

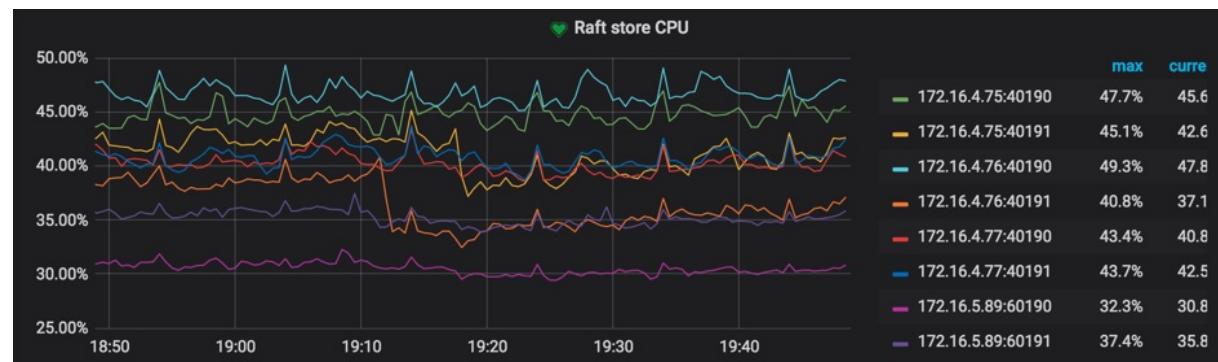
注意：Raftstore 的 CPU 使用率是指单线程的情况。如果是多线程 Raftstore，可等比例放大使用率。由于 Raftstore 线程中有 I/O 操作，所以 CPU 使用率不可能达到 100%。

## 性能监控

可在 Grafana 的 TiKV 面板下查看相关的监控 metrics：

Thread-CPU 下的 Raft store CPU

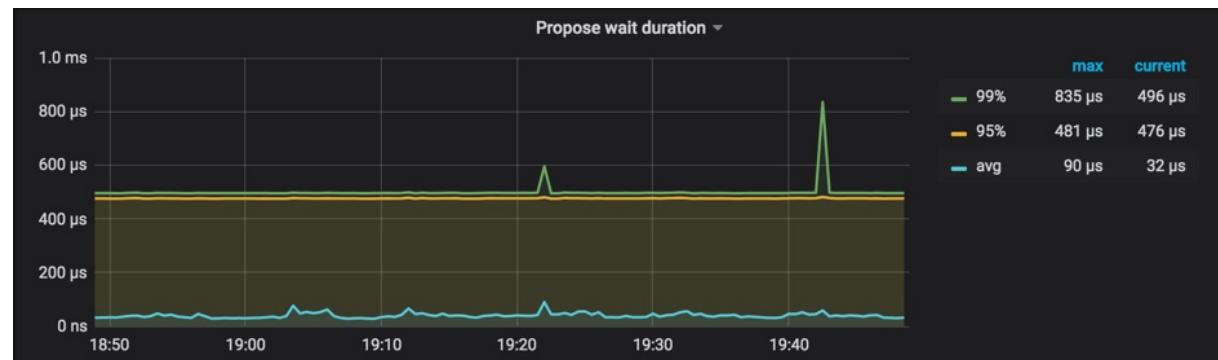
参考值：低于 raftstore.store-pool-size \* 85%。



Raft Propose 下的 Propose wait duration

Propose wait duration 是从发送请求给 Raftstore，到 Raftstore 真正开始处理请求之间的延迟时间。如果该延迟时间较长，说明 Raftstore 比较繁忙或者处理 append log 比较耗时导致 Raftstore 不能及时处理请求。

参考值：低于 50-100ms。



## 3. 性能优化方法

找到性能问题的根源后，可从以下两个方向来解决性能问题：

- 减少单个 TiKV 实例的 Region 数
- 减少单个 Region 的消息数

### 方法一：增加 TiKV 实例

如果 I/O 资源和 CPU 资源都比较充足，可在单台机器上部署多个 TiKV 实例，以减少单个 TiKV 实例上的 Region 个数；或者增加 TiKV 集群的机器数。

### 方法二：调整 raft-base-tick-interval

除了减少 Region 个数外，还可以通过减少 Region 单位时间内的消息数量来减小 Raftstore 的压力。例如，在 TiKV 配置中适当调大 raft-base-tick-interval：

```
[raftstore]
raft-base-tick-interval = "2s"
```

raft-base-tick-interval 是 Raftstore 驱动每个 Region 的 Raft 状态机的时间间隔，也就是每隔该时长就需要向 Raft 状态机发送一个 tick 消息。增加该时间间隔，可以有效减少 Raftstore 的消息数量。需要注意的是，该 tick 消息的间隔也决定了 election timeout 和 heartbeat 的间隔。示例如下：

```
raft-election-timeout = raft-base-tick-interval * raft-election-timeout-ticks
raft-heartbeat-interval = raft-base-tick-interval * raft-heartbeat-ticks
```

如果 Region Follower 在 raft-election-timeout 间隔内未收到来自 Leader 的心跳，就会判断 Leader 出现故障而发起新的选举。raft-heartbeat-interval 是 Leader 向 Follower 发送心跳的间隔，因此调大 raft-base-tick-interval 可以减少单位时间内 Raft 发送的网络消息，但也会让 Raft 检测到 Leader 故障的时间更长。

### 方法三：提高 Raftstore 并发数

TiKV 默认将 raftstore.store-pool-size 配置为 2。如果 Raftstore 出现瓶颈，可以根据实际情况适当调高该参数值，但不建议设置过高以免引入不必要的线程切换开销。

### 方法四：开启 Hibernate Region 功能

在实际情况中，读写请求并不会均匀分布到每个 Region 上，而是集中在少数的 Region 上。那么可以尽量减少暂时空闲的 Region 的消息数量，这也就是 Hibernate Region 的功能。无必要时可不进行 raft-base-tick，即不驱动空闲 Region 的 Raft 状态机，那么就不会触发这些 Region 的 Raft 产生心跳信息，极大地减小了 Raftstore 的工作负担。

截至 TiDB v4.0，Hibernate Region 仍是一个实验功能，在 [TiKV master](#) 分支上已经默认开启。可根据实际情况和需求来开启该功能。Hibernate Region 的配置说明请参考[配置 Hibernate Region](#)。

### 方法五：开启 Region Merge

注意：Region Merge 在 TiDB v4.0 中默认开启。

开启 Region Merge 也能减少 Region 的个数。与 Region Split 相反，Region Merge 是通过调度把相邻的小 Region 合并的过程。在集群中删除数据或者执行 Drop Table/Truncate Table 语句后，可以将小 Region 甚至空 Region 进行合并以减少资源的消耗。

通过 pd-ctl 设置以下参数即可开启 Region Merge 以及调整 Region Merge 速度：

```
>> pd-ctl config set max-merge-region-size 20
>> pd-ctl config set max-merge-region-keys 200000
>> pd-ctl config set merge-schedule-limit 8
```

详情请参考[如何配置 Region Merge 和 PD 配置文件描述](#)。同时，默认配置的 Region Merge 的参数设置较为保守，可根据需求参考[PD 调度策略最佳实践](#)中提供的方法加快 Region Merge 过程的速度。

## 8.2.3 其他常见优化设置

1. Block-cache TiKV 使用了 RocksDB 的 Column Family (CF) 特性，KV 数据最终存储在默认 RocksDB 内部的 default、write、lock 3 个 CF 内。
2. default CF 存储的是真正的数据，与其对应的参数位于 [rocksdb.defaultcf] 项中。
3. write CF 存储的是数据的版本信息 (MVCC) 、索引、小表相关的数据，相关的参数位于 [rocksdb.writecf] 项中。
4. lock CF 存储的是锁信息，系统使用默认参数。
5. Raft RocksDB 实例存储 Raft log。default CF 主要存储的是 Raft log，与其对应的参数位于 [raftdb.defaultcf] 项中。
6. TiDB 3.0 版本及以上所有 CF 共享一个 Block-cache，用于缓存数据块，加速 RocksDB 的读取速度，TiDB 2.1 版本及以下通过参数 block-cache-size 控制每个 CF Block-cache 大小，Block-cache 越大，能够缓存的热点数据越多，对读取操作越有利，同时占用的系统内存也会越多。
7. 每个 CF 有各自的 Write-buffer，大小通过 write-buffer-size 控制。

TiDB 3.0 版本及以上部署 TiKV 多实例情况下，需要修改 conf/tikv.yml 中 block-cache-size 下面的 capacity 参数：

```
storage:
  block-cache:
    capacity: "1GB"
```

注意：TiKV 实例数量指每个服务器上 TiKV 的进程数量。推荐设置：capacity = MEM\_TOTAL \* 0.5 / TiKV 实例数量

## Sync-log

通过使用 [Raft 一致性算法](#)，数据在各 TiKV 节点间复制为多副本，以确保某个节点挂掉时数据的安全性。只有当数据已写入超过 50% 的副本时，应用才返回 ACK（三副本中的两副本）。但理论上两个节点也可能同时发生故障，所以除非是对性能要求高于数据安全的场景，一般都强烈推荐开启 sync-log。一般来说，开启 sync-log 会让性能损耗 30% 左右。

可以修改 conf/tikv.yml 中 raftstore 下面的 sync-log 参数：

```
[raftstore]
sync-log = true
```

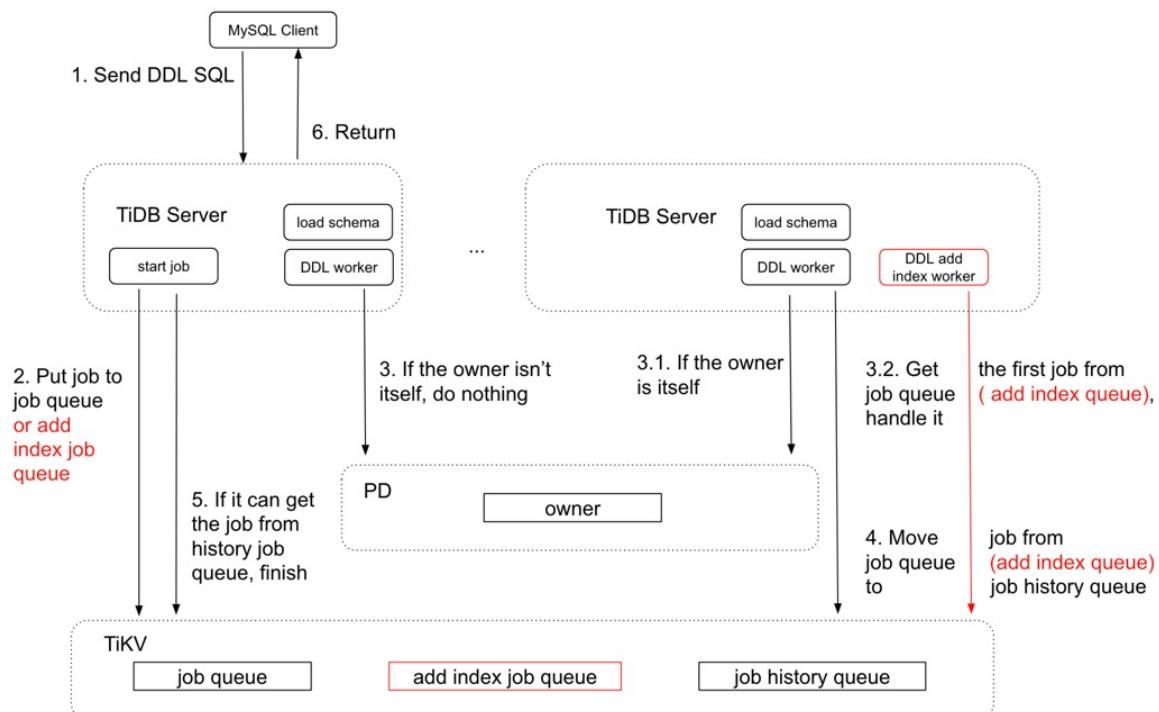
## 8.3 添加索引调优加速

索引是应用设计和开发过程中一个重要方面，我们常常发现，在应用开发中总是在出现性能问题的时候才会想起创建索引，而可能此时表上的数据已经非常巨大，创建索引的代价往往非常巨大。得益于 TiDB 的 Online DDL 特性，在 TiDB 中创建索引无需考虑 DDL 与业务 DML 之间互相阻塞的问题，但如何平衡索引创建的速度和集群负载的影响是一个值得思考的问题。

## 8.3.1 TiDB 增加索引原理

在普通关系型数据库中增加索引通常会有时间过长、锁表等风险，特别是在一张频繁更新数据的表上增加索引的时候，风险变得很大且不可控。TiDB 的 DDL 通过实现 Google F1 的在线异步 schema 变更算法，来完成在分布式场景下的无锁，在线 schema 变更。从 TiDB 2.1 开始实现了并行 DDL，新增了增加索引队列 (add index job queue) 以及增加索引线程 (add index worker) 用以加速增加索引的执行速度。

增加索引流程：



增加索引核心操作:

1. 将索引的元信息添加到系统表中。
2. 批量从读原始表中读数据并构建索引。

增加索引主要流程:

1. 客户端发送添加索引请求到 TiDB，TiDB 会检查表、索引等是否符合规范。
2. 将添加索引请求转化成 Job 发送到添加索引的队列中 (add index job queue)。
3. TiDB 会启动 Worker，将 Job 从添加索引的队列中取出，并且写入到对应表信息中。
4. 这时候 Worker 从 PD 中获取需要添加表的所有 region 范围，并且默认分成 256 个子 Job，并发的去扫 region 中的所有数据，生成索引信息。
5. 当所有子 Job 都完成之后，会将该 Job 放入历史队列中 (history ddl job)。

## 8.3.2 动态调整索引创建速度

由于创建索引在扫表回填索引的时候会消耗大量资源，甚至与一些频繁更新的字段会发生冲突导致正常业务受到影响。大表创建索引的过程往往会长时间，所以要尽可能地平衡执行时间和集群性能之间的关系。

创建索引相关的几个参数：

参数	默认值	说明
tidb_ddl_reorg_worker_cnt	4	控制创建索引并发度
tidb_ddl_reorg_batch_size	256	控制每次创建索引数据的数量
tidb_ddl_reorg_priority	PRIORITY_LOW	调整创建索引优先级。参数有 PRIORITY_LOW/PRIORITY_NORMAL/PRIORITY_HIGH
tidb_ddl_error_count_limit	512	失败重试次数，如果超过该次数创建索引会失败

参数调整：

目前主要使用 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 这两个参数来动态调整索引创建速度，通常来说它们的值越小对系统影响越小，但是执行时间越长。

一般情况下，先将值保持为默认的 4 和 256，观察集群资源使用情况和响应速度，再逐渐调大 `tidb_ddl_reorg_worker_cnt` 参数来增加并发，观察监控如果系统没有发生明显的抖动，再逐渐调大 `tidb_ddl_reorg_batch_size` 参数，但如果索引涉及的列更新很频繁的话就会造成大量冲突造成失败重试。

另外还可以通过调整 `tidb_ddl_reorg_priority` 为 `PRIORITY_HIGH` 来让创建索引的任务保持高优先级来提升速度，但在通用 OLTP 系统上，一般建议保持默认。

如何评估创建索引的速度：

1. 使用 `admin show ddl` 命令来查询 `RowCount` 和 `START_TIME` 字段，记录当前 DDL 已经更新了的行数 `r1`，利用开始时间计算出已执行时间 `t1`。
2. 再使用 `show stats_meta` 命令来查看 `RowCount` 字段，查看表数据的总行数 `r0`。
3. 此时就可以用： $t1/(r1/r2) - t1$  来估算剩余执行时间，再根据系统集群使用情况及响应速度来评估是否动态调整参数。

总结：

1. 更新不频繁的字段创建索引时，可以根据实际负载合理动态调整参数，可参考：[添加索引和负载测试](#)。
2. 如果创建索引的字段上更新十分频繁，可以保持默认参数值，来优先系统及业务的稳定。

# 第 1 章 TiDB 开源社区历史及现状

## 1.1 TiDB 开源社区现状及发展简史

### 1.1.1 TiDB 开源社区现状

TiDB 分布式数据库从 2015 年 4 月份开源以来，马上就要 5 岁了，在这 5 年中，TiDB 分布式数据库收到了无数的关注和鼓励，在大家的帮助下，一步步地从嗷嗷待哺的婴儿，慢慢成长成为一个充满活力的少年。在这 1800 多个日夜努力的背后，有来自社区的开发者，用户，布道师，设计师等不同角色的贡献者们的细心呵护，也有来自 PingCAP 持续不断的资源投入，截止到 2020 年 3 月，TiDB 社区项目已经聚集了来自全球的 760 多位 Contributor，100 多位核心用户，30000+ GitHub Stars 支持，1000+ 的 AskTUG 社区问答注册会员，700+ PingCAP University 的社区培养人才，TiDB 现已被近 1000 家不同行业的领先企业应用在实际生产环境。

2019 年，TiDB 开源社区共输出 94 篇技术文章，123 场技术布道，包括 38 场 Infra Meetup 和 23 次在线的 Paper Reading，累计覆盖 60000+ 受众人群。

2020 年，TiDB 4.0 GA 版本会在 5 月底和大家见面，在新的一年，TiDB 会承担更大的责任，与社区一起，连接世界各地的用户，与用户共同成功。

### 1.1.2 TiDB 开源社区发展简史

时间	事件
2015 年 04 月	写下第一行 TiDB 代码
2015 年 09 月	在 GitHub 上开源，一个月 Star 数超过 2700
2015 年 12 月	TiDB 1.0 Alpha 发布，成为全球第一个开源的 Google F1 实现
2016 年 04 月	独立研发的基于 Google Spanner 的下一代分布式存储引擎 TiKV 开源
2017 年 10 月	TiDB 1.0 GA
2018 年 04 月	TiDB 2.0 GA, TiDB + TiKV 16000+ Stars, 240+ Contributors
2018 年 08 月	TiDB Operator 开源
2018 年 08 月	CNCF 接纳 TiKV 作为 CNCF Sandbox 的云原生项目
2018 年 12 月	面向社区人才培养的 Talent Plan 正式启动
2019 年 01 月	TiDB Lightning Toolset & TiDB Data Migration 正式开源
2019 年 03 月	TiDB 社区在线学习视频网站 PingCAP University 正式上线
2019 年 05 月	TiDB Binlog 正式开源
2019 年 05 月	CNCF 升级 TiKV 作为 CNCF Incubation 的云原生项目
2019 年 06 月	TiDB User Group 正式成立
2019 年 06 月	TiDB 3.0 GA , TiDB + TiKV 22000+ Stars, 390+ Contributors
2019 年 06 月	TiDB 社区问答网站 AskTUG 正式上线
2019 年 06 月	TiDB 社区在线学习视频网站 2.0 Courses 正式上线
2019 年 07 月	TiDB Operator 1.0 GA，并提供 AWS/GCP/阿里云一键部署方案
2019 年 12 月	混沌测试平台 Chaos Mesh 正式开源
2020 年 01 月	实时查询分析引擎 TiFlash Beta 版本发布
2020 年 03 月	TiDB 开源项目 Contributor 突破 400
2020 年 03 月	TiDB 4.0 RC
2020 年 03 月	教大家从零到一写分布式数据库的 Talent Plan Courses 正式发布
2020 年 05 月	TiDB 4.0 GA (计划)

## 1.2 TiDB 分布式数据库开源生态项目介绍

### 1.2.1 TiDB 相关项目

#### 1. TiDB

TiDB 为整个 TiDB 分布式数据库的 SQL 处理层。这一层最重要的工作是处理用户请求，执行 SQL 运算逻辑，用户的 SQL 请求会直接或者通过 Load Balancer 发送到 TiDB。TiDB 会解析 MySQL Protocol Packet，获取请求内容，然后做语法解析、查询计划制定和优化、执行查询计划获取和处理数据。TiDB 是无状态节点，本身并不存储数据，数据全部存储在 TiKV 集群中，所以在这个过程中 TiDB 需要和 TiKV 交互，获取数据。最后 TiDB 需要将查询结果返回给用户。

#### 2. Parser

Parser 是由 Yacc 生成的解析器，并且与 MySQL 语法高度兼容。Parser 的功能是把 SQL 语句按照 SQL 语法规则进行解析，将文本转换成抽象语法树（AST）。

#### 3. TiSpark

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时融合 TiKV 分布式集群的优势，和 TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。TiSpark 依赖于 TiKV 集群和 Placement Driver (PD)，也需要你搭建一个 Spark 集群。

### 1.2.2 TiKV 相关项目

#### 1. TiKV

TiKV 是一个分布式、支持事物的 K-V 数据库。它通过 RocksDB 进行本地储存，使用 Raft 协议来维护一致性，依照 Percolator 事务模型。在 Raft 和 PD 的帮助下，它能够支持横向扩展和异地副本。它既能够作为普通的分布式 K-V 数据库使用，也提供了能够满足 ACID 的事物接口。TiDB 使用它完成底层储存、分布式下推计算。与此同时，TiKV 也提供 java、c 等客户端库可供使用。

#### 2. PD

PD (Placement Driver) 是 TiDB 里面全局中心总控节点，它负责整个集群的调度，负责全局 ID 的生成，以及全局时间戳 TSO 的生成等。PD 还保存着整个集群 TiKV 的元信息，负责给 client 提供路由功能。

#### 3. grpc-rs

grpc-rs 是为 gRPC Core 提供的 rust 包装层。它已经支持了朴素的异步调用、流式调用、SSL 等常用功能。TiKV 使用它完成与 TiDB 中其他部分的通信。

#### 4. raft-rs

raft-rs 是 Raft 协议的 rust 实现。它借鉴了 etcd 的 Raft 实现的设计。

#### 5. rust-rocksdb

rust-rocksdb 是 Rocksdb 的 rust 包装层。为 Rust 应用程序提供了方便易用的使用 Rocksdb 的方式。TiKV 使用它完成硬盘存储。

#### 6. rust-prometheus

rust-prometheus 是为 rust 应用设计的 [Prometheus](#) instrumentation 库。赋予 Rust 程序接入 [Prometheus](#) 的能力。

## 7. pprof-rs

pprof-rs 是 rust 程序在线 profiling 工具。TiKV 使用它提供了在线 profiling、采样生成火焰图的能力。

### 1.2.3 Tools 相关项目

#### 1. TiDB Lightning

TiDB Lightning 是一个将全量数据高速导入到 TiDB 集群的工具，其特点是将 DDL 和其他数据记录分离开，DDL 依旧通过 TiDB 进行执行，而其他数据则直接通过 KV 的形式导入到 TiKV 内部。支持 Mydumper 或 CSV 输出格式的数据源，尤其适用于对于需要从 MySQL 将大量数据导入到 TiDB 上的场景。

#### 2. Dumpling

Dumpling 是 Mydumper 的替代工具，能够从任何兼容 MySQL 协议的数据库中导出数据，Dumpling 的导出速度和 Mydumper 不相上下，由于能够生成二进制的输出文件，在使用 Lightning 将数据导入到 TiDB 时会加快速度。此外，Dumpling 还支持云存储功能。

#### 4. ticdc

ticdc 是一个 TiDB 的事件日志复制工具，用于数据备份，下游支持 MySQL, TiDB, Kafka 等。ticdc 会捕获上游 TiKV 的 KV 变化日志，将其还原为 SQL 之后由下游的 MySQL 或 TiDB 进行消费；写入 Kafka 的 KV 日志会在 Kafka 的下游进行 SQL 还原。

#### 5. DM

DM 是一体化的数据同步任务管理平台，能够支持从 MySQL 到 TiDB 分布式数据库的全量数据同步，主要包括 DM-master, DM-worker 和 dmctl 三个组件，其中 DM-master 负责管理和调度数据同步任务的各项操作，DM-worker 负责执行具体的数据同步任务，dmctl 用于控制 DM 集群。上游 MySQL 产生的 binlog 由 DM-worker 进行消费后插入到下游的 TiDB 分布式数据库里。

#### 6. BR

BR 是 TiDB 分布式数据库专用的备份恢复工具，BR 的备份和恢复速度都远超过一般通过 SQL 进行的数据备份恢复，适合在大数据量场景下使用。在备份时，BR 会从 PD 服务器获取一个时间戳作为备份时间点，TiKV 会将自己节点上所有 region leader 中符合要求的 KV 写入给定的路径生成 sst 文件，恢复时，TiKV 会读取其生成的 sst 文件，并且通过 Raft 协议保证数据的一致性。

#### 7. tidb-binlog

tidb-binlog 是 TiDB 的 binlog 搜集工具，TiDB 中执行成功的 SQL 会被 Pump 实时记录，Drainer 会从 Pump 中收集 binlog 并进行归并后同步给下游，Tidb Binlog 组件能够对接 TiDB, MySQL, Kafka，是基于 SQL 的数据备份和同步方案。

### 1.2.4 Cloud 相关项目

#### TiDB Operator

TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供了 TiDB 部署、升级、扩缩容、备份恢复和配置变更的能力。只需要简单定义 TiDB 集群的配置和集群信息，TiDB Operator 就可以把 TiDB 运行在 Kubernetes 上。同时，TiDB Operator 支持私有云和常用的公有云，能够降幅降低 Kubernetes 上 TiDB 的管理成本。

### 1.2.5 测试、部署、文档相关项目

## 1. Chaos Mesh

Chaos Mesh 是 Kubernetes 上原生的混沌测试工具，通过在 Kubernetes 环境中将进程退出，生成网络错误、文件系统错误、内核错误等生产环境中常见或罕见的错误，增强测试的覆盖率，发现传统测试难以出现的问题。Chaos Mesh 是通用的 Chaos 测试方案，可以用于测试所有运行在 Kubernetes 的应用。

## 2. tipocket

tipocket 是使用 Chaos Mesh 对 TiDB 进行混沌测试的实践，同时使用了 TiDB Operator 和 Chaos Mesh，在 Kubernetes 上对 TiDB 和 Chaos 进行编排管理，并运行各种数据库测试用例。

## 3. TiDB Ansible

TiDB Ansible 是 TiDB 在物理机集群生产环境中使用的部署运维工具，能够对集群进行部署、升级、扩缩容、变更配置等操作。

## 4. docs

docs 是 PingCAP 所有开源项目的文档，可以在 PingCAP 官方进行查阅，地址为：<https://pingcap.com/docs/>。

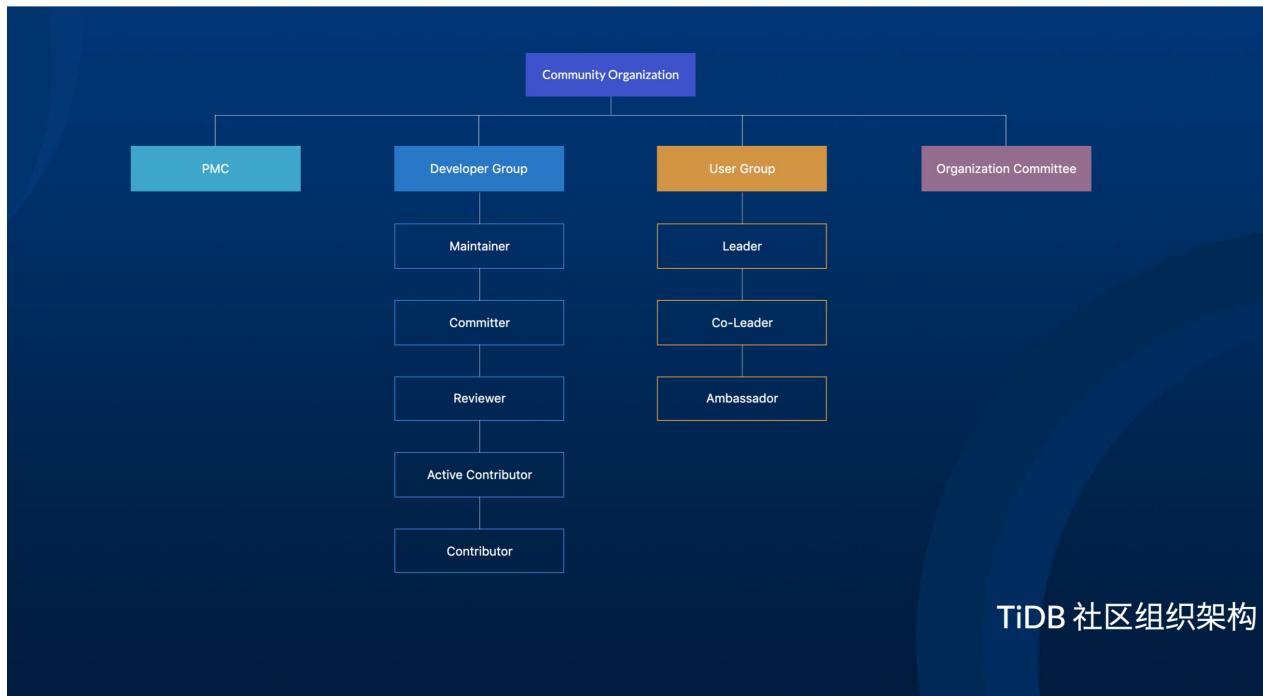
## 5. docs-cn

docs-cn 是 docs 文档的中文版本，可以在 PingCAP 官方进行查阅，地址为：<https://pingcap.com/docs-cn/>。

## 1.3 TiDB 开源社区治理

经过几年的发展，TiDB 社区已经逐渐成熟，但是随着社区的发展壮大，为了更好地组织对 TiDB 社区感兴趣的人一起高效地参与社区的活动，以便更好的激发社区活力，维护积极健康的社区环境，TiDB 开源社区一直持续打造透明公正的治理规范，参见 [community](#)。

在社区组织架构下面，下设四个不同的组织，PMC，Developer Group，User Group 以及 Organization Committee。



### 1.3.1 PMC

TiDB 社区的顶层治理架构我们借鉴了 Apache 基金会的 PMC(Project Management Committee) 概念，即项目管理委员会，参与 TiDB 社区项目 Roadmap 制定以及社区相关的重大决议。

### 1.3.2 Developer Group

Developer Group，就是大家常说的开发者社区。在开发者社区中，在大家耳熟能详的 Contributor，Committer 和 Maintainer 中，考虑到 Contributor 到 Committer 中间的升级链路太长，TiDB 社区新增了 Active Contributor 和 Reviewer 的角色，贡献者可以一步步进阶，更好地参与到项目协作和推进。



目前 TiDB 社区项目越来越多，本身 TiDB 和 TiKV 项目也越来越复杂，所以 TiDB 社区引入了两个新的治理概念：

- 特别兴趣小组（Special Interest Group，简称 SIG），特别兴趣小组是一个非常古老的词语，它最早出现在上世纪 70 年代，因为 ARPAnet 网络所形成的各种 mailing list 邮件组，这个也是开源社区的雏形。TiDB 社区也沿用了这么一个古老且广泛应用于开源社区的组织形式（词语）。SIG 主要负责例如 TiDB/TiKV 某个模块的开发和维护工作，对该模块代码的质量负责。满足条件的 Active Contributor 将被邀请加入专项兴趣小组，开发者们将在专项兴趣小组中获得来自 Tech Lead 们以及小组活跃 Committer、Reviewer 的持续指导，一边锻炼技术能力，一边优化和完善该模块。社区开发者们可通过专项兴趣小组逐渐从初始的 Active Contributor 成长为受到社区认可的 Reviewer、Committer 和 Maintainer。一般而言每个专项兴趣小组都会周期性的组织会议，讨论最近进展和遇到的问题，所有的会议讨论都公开在社区上，方便感兴趣的同学一起参与和讨论。
- 工作小组（Working Group，简称 WG）是由为了完成某个特定目标而聚集在一起的临时工作组。为了完成目标，有些工作小组可能跨越多个 SIG，有些小组也可能只会专注在某个具体的 SIG 中做某个具体的事情。工作小组具有生命周期，一旦目标完成，工作小组即可解散。工作小组运营和管理的唯一目标是确保该小组成立时设置的目标在适当的时间内完成。一般而言，工作小组也会有周期性的会议，用于总结目前项目进展，确定下一步实施方案等。

### 1.3.3 User Group

User Group，即 TiDB User Group（TUG）是由 TiDB 用户发起的独立、非盈利的第三方组织，用户实行自我管理，旨在加强 TiDB 用户之间的交流和学习。TUG 的形式包括但不限于线上问答和技术文章分享、线下技术沙龙、走进名企、官方互动活动等等。TUG 成员可以通过线上、线下的活动，学习前沿技术知识，发表技术见解，共同建设 TiDB 项目。

随着 TiDB 产品的成熟，TiDB 用户群体愈发壮大，用户在使用过程中遇到的问题反馈及实践经验，对于 TiDB 产品的完善及应用推广有着不可忽视的重要作用，希望用户与开发者有更好的交流互动，一起推动 TiDB 社区的健康发展。

### 1.3.4 Organization Committee

Organization Committee，项目组织委员会，负责执行、输出，推广技术内容的组织者们，成员包括各地区用户组组长以及社区活动负责人。目前暂时囊括了很多具体方向的 Committee，比如 Legal / Quality / Marketing 等等，都会在这个委员会里面孵化和展开。

## 1.4 TiDB 开源社区重要合作开发

在 TiDB 开源社区中，我们与多家公司以及高校建立起了深入的合作关系，就 TiDB 分布式数据库不同方向的开发工作与多家高校实验室开展了开发实践，包括但不限于：

- 与中国科学技术大学某实验室对存储引擎进行性能优化，共同在 ICDE 发表 Paper - UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing
- 与华中科技大学某实验室为 TiDB 在最新的 Intel Optane DC persistent memory 盘上面定制存储引擎。
- 与华中师范大学某实验室一起进行分布式数据库测试的研究。

也跟国内多家公司就 TiDB 分布式数据库开发建立合作关系，包括但不限于：

- 与美团在生态工具、查询优化器等多个方向合作，加速 TiDB 增量备份的速度，开发从外部异构数据库同步数据到 TiDB 的组件。
- 与知乎在存储引擎、Cloud 等方向合作，提速 TiDB 自研存储引擎 scan 性能，提升 TiDB 大规模集群（250+ 节点，500+ TB 数据）下面的稳定性。
- 与一点资讯在 TiKV 上合作，加速 TiKV 性能，为 TiKV 提供类似新的 KV 模式，适配更多的业务场景。

感谢一直以来社区的支持和鼓励，后续 TiDB 社区会将更多社区合作的成果和细节跟大家分享，敬请期待。

## 第 2 章 TiDB 开源社区生态

### 2.1 社区重要活动介绍

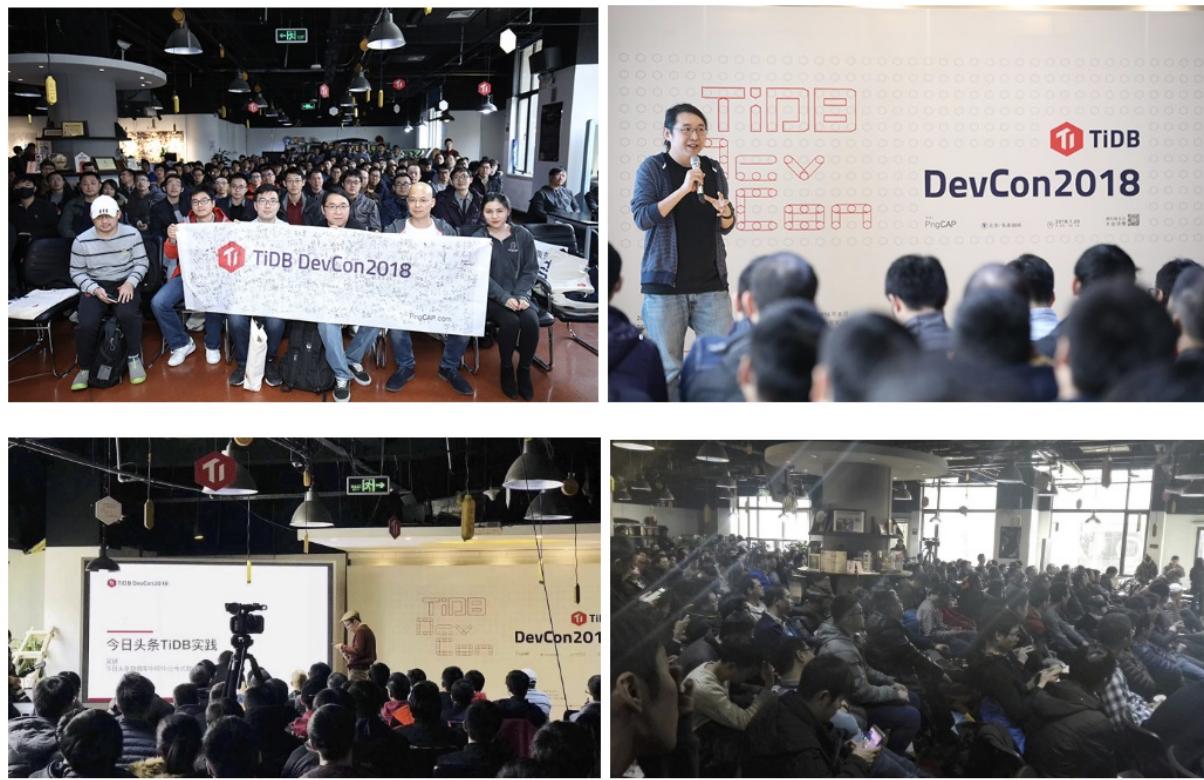
#### 2.1.1 TiDB DevCon

TiDB DevCon 是 PingCAP 团队面向 TiDB 社区推出的年度最高规格的技术盛会，一般在每年年初举办，迄今已成功举办 2018 及 2019 两届。TiDB DevCon 聚焦 TiDB 项目核心技术的最新进展和未来规划，以及来自社区一线用户的最佳实践经验，展示 TiDB 分布式数据库在海内外的最新动态。TiDB DevCon 旨在帮助社区更好的理解 TiDB 分布式数据库的技术理念，汇总用户从技术选型到应用落地各阶段的实操经验，挖掘 TiDB 分布式数据库场景适配的更多可能性。

[TiDB DevCon 2019](#) 于 2019 年 01 月 19 日在北京举办，出席人数 750+，社区讲师来自：美团、小米、转转、新浪微博、VIPKID、贝壳找房、Bilibili、新东方、北京银行、微众银行、中证券 等。



[TiDB DevCon 2018](#) 于 2018 年 01 月 20 日在北京举办，出席人数 300+，社区讲师来自：今日头条、Mobike、饿了么、去哪儿等，参见。



## 2.1.2 TiDB TechDay

为了让更多的社区伙伴能够近距离与我们展开交流，并快速 Get TiDB 分布式数据库的技术细节和正确使用姿势，从 2017 年开始，PingCAP 在每年年中面向社区小伙伴举行 TiDB TechDay 活动，用一天的时间为大家深入拆解当年 TiDB 分布式数据库技术层面的各种大招。

首届 TechDay 在 2017 年的上海，这次活动为社区小伙伴带来了 TiDB Internal, TiSpark, TiDB on Kubernetes 等技术分享。

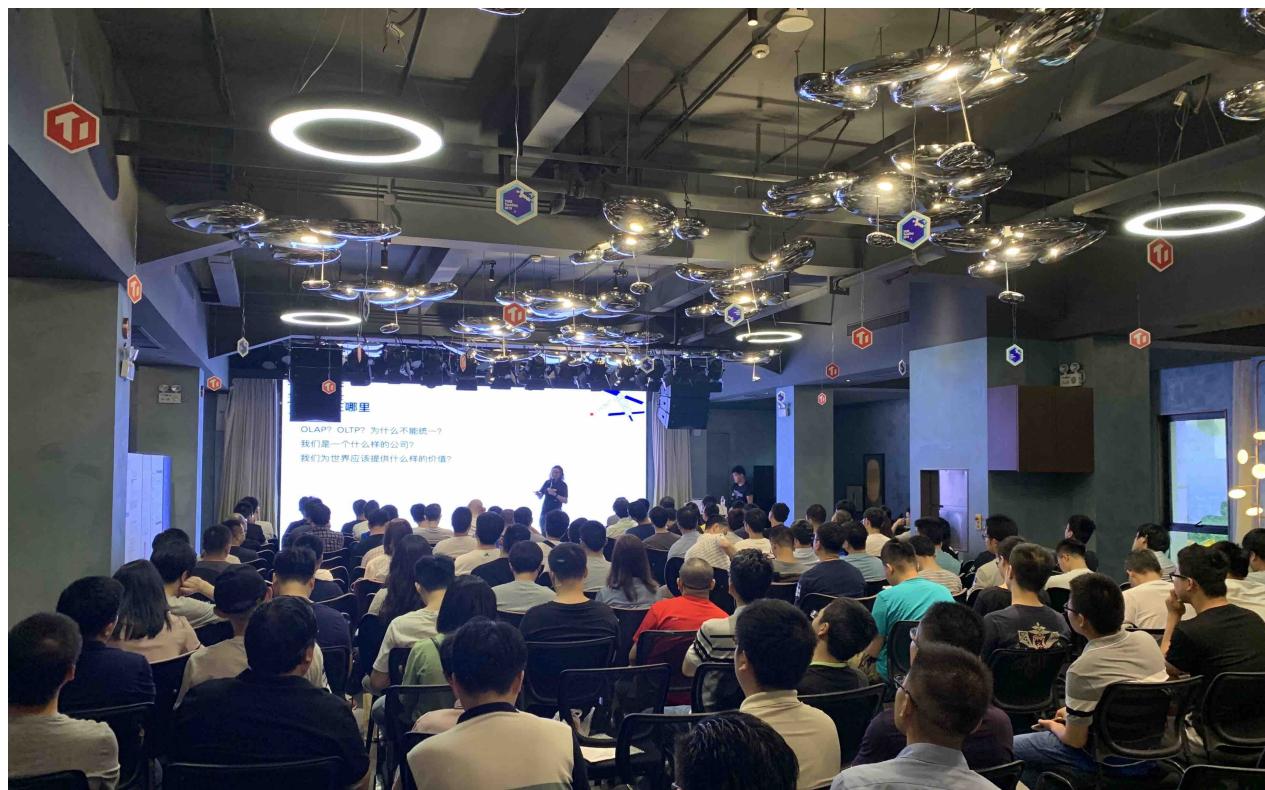


第二届 TechDay 在 2018 年的深圳，这次活动除了 TiDB 2.1 Beta 版本、Chaos Practice in TiDB 等技术分享之外，还带来了精彩的用户实践和社区小伙伴的贡献心得。



第三届 TechDay 启动「TiDB TechDay 2019 全国巡讲」，打破「一年一城」的传统，巡回北京、上海、成都、深圳、武汉、杭州 6 座城市，为社区小伙伴带来了 TiDB 3.0 的技术细节，一起交流全球社区运营的新想法，各地用户伙伴也一起交流分享了 TiDB 实践经验。





## 2.1.3 Infra Meetup

### 1. 活动简介

作为一个基础架构领域的前沿技术公司，PingCAP 希望能为国内真正关注技术本身的 Hackers 打造一个自由分享的平台。自 2016 年 3 月 5 日开始，我们定期在周末举办 Infra Meetup，与大家深度探讨基础架构领域的前瞻性技术思考与经验，目前已在北京、上海、广州、成都、杭州、深圳、西安等地举办。在这里，我们希望提供一个高水准的前沿技术讨论空间，让大家真正感受到自由的开源精神魅力。

### 2. 过去现在与未来

2015 年一群崇尚开源精神的程序员在民居里写下了 TiDB 分布式数据库最初的代码，周末大家不想写代码的时候，就聚一起聊技术聊方案。慢慢地有更多外部公司的技术小伙伴加入了进来，于是就有了现在的 Infra Meetup。

截止 2020 年 3 月，PingCAP Infra Meetup 已经举行了 125 期，近四千人次参与了现场活动，有更多人观看了视频及线上直播。Infra Meetup 100 期，刘奇和大家分享了 TiDB 作为明星级开源分布式数据库的架构演进哲学，对 TiDB 产品升级迭代做了一次最系统的回溯与展望。而最新一期 Meetup，由于新冠病毒疫情影响，活动从线下移到线上，上万人同时在线观看了此次直播，这也是截止目前单次 Meetup 参与人数最多的一次。

No.125 只是 Infra Meetup 新的起点，希望未来有更多对基础架构前沿技术讨论感兴趣的小伙伴，通过无论线下还是线上方式，一起加入我们。





## 2.1.4 TiDB Hackathon

### 1. 活动简介

对于熟悉技术圈的伙伴来说，Hackathon 想必大家已经很熟知了，在这个流传于程序员和技术爱好者的活动当中，大家相聚在一起，以合作的方式去编程，在短时间内激发灵感，进而探索出更多的可能性。基于 Hackathon 的模式和理念，从 18 年开始，PingCAP 已经成功举办了两届 TiDB Hackathon，在这场属于 TiDB 开发者的狂欢中，伴随着啤酒 Pizza 不眠夜，大家围绕整个 TiDB 生态在 48 小时内头脑风暴、全力燃烧做出一个完整的作品，并由评审最终评选出获奖者。

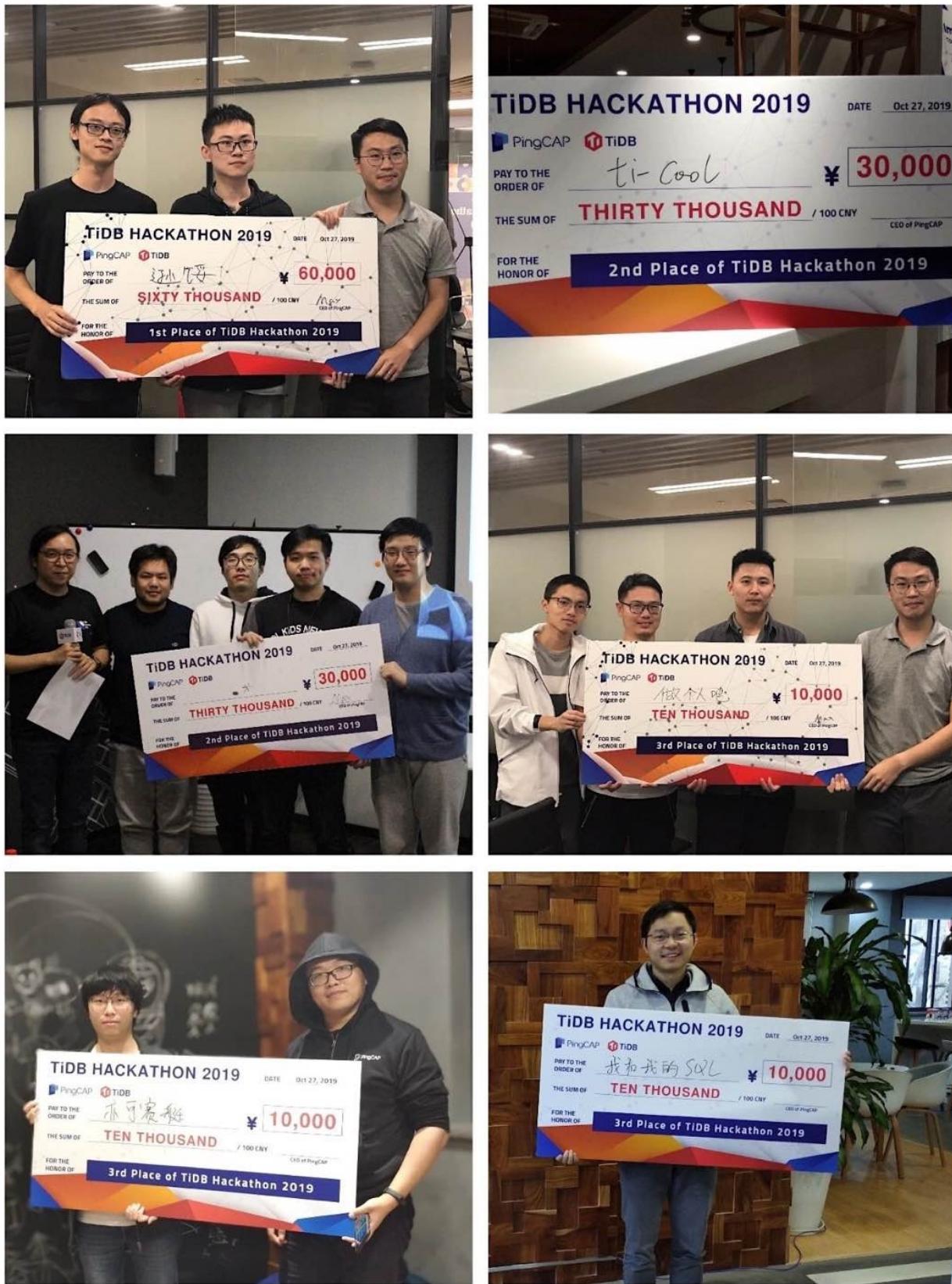
### 2. 回顾

PingCAP 作为国内一家开源的分布式数据库公司，我们一直坚信着开源的力量。基于 TiDB 本身作为开源项目的独特优势，TiDB Hackathon 既能让大家感受即刻编程的快感，也能带给社区更多优秀的创意和灵感。

- 第一届 TiDB Hackathon 于 2018 年 12 月 1 日在北京总部开启，以“TiDB Ecosystem”作为大方向，参赛者可围绕着 TiDB 及其周边生态实现自由选题，22 支参赛队伍经过两天一夜的疯狂竞技，最终 6 支队伍杀出重围，赢得大奖。短短 48 小时内，我们见证了将外部数据源以流式表的方式硬核扩展的 TiDB Batch and Streaming SQL，能让 TiDB 访问多种数据源的 CC，为开发者提供了方便查询 Region 历史的可视化工具 TiEye，All diagnosis in SQL 的 TiQuery。在这些项目中，我们惊喜于技术所带来的魅力，也看到了通过 Hackathon 而衍生出对于社区项目更多的可能性。



- 第二届 TiDB Hackathon 于 2019 年 10 月 26 日重磅回归，相比去年选题更加广泛，弱化了地域限制，北上广三地联动，各路大神云集，再一次为大家线上献上了一场技术盛宴。“Improve·跃界”，跨越、爆发，所见「可提升」之处皆为竞技场。两天一夜的 Hacking Time，三地共 39 只队伍参赛，长达 7 小时精彩的 Demo Show 直播，大家都为 TiDB 性能、易用性、稳定性、功能及周边工具献出无数的好点子，用社区的力量为 TiDB 装上更多的“大杀器”：技术小白的你，也能在浏览器里直接运行数据库；程序员的你，即使呼呼大睡，机器人也能帮你找 bug；资深用户的你，也可以像使用单机数据库一样使用 TiDB，虽然叫大杀器，但这些特性似乎还有些可爱和温柔~



### 3. 展望

TiDB 社区有一个信念：我们从来不给自己设限，突破&提升才是一直追求的常态。对于社区、对于产品，我们想看到的从来不是一道简简单单的填空题，而希望把它变成选择题、简答题甚至一道作文题。在众多面向社区的重要活动中，TiDB Hackathon 是一个两岁的孩子，从第一次见面看到他初成长的潜力，到第二次重逢看到他富有多样性、包容性的样子带给我们的惊喜。而这一切，跟来自每一位社区的小伙伴是分不开的。

当然，我们还有待完善和提高的空间，通过 Hackathon 也给 PingCAPer 带来了许多思考：我们如何鼓励并激发大家更有创造力的想法？如果建立更加高效的组织和机制带动社区百花齐放？如何践行我们的开源信念？我们希望能将开源社区平台变成一片更加肥沃的土壤，为所有充满理想、富有好奇心，想用技术去改变世界的小伙伴提供可耕耘的天地。可喜的是，即使前路漫长我们依旧在路上，也见证着 TiDB Hackathon 衍生出的项目融入社区重要合作的开发中，在大家的精心呵护下开始慢慢生根，发芽。

今年的 Hackathon 又会给大家带来哪些惊喜？希望这道题由你来决定，青春不散场，我们赛场见~

## 2.1.5 TUG 企业行

在 TiDB 分布式数据库的演进过程中，TiDB 社区意识到一个成熟的开源产品和活跃的社区，不仅依靠全球开发者的积极贡献，也离不开活跃用户的使用反馈和建议。用户是 TiDB Community 的重要组成部分，因此需要为用户搭建一个开放、活跃、独立学习和交流的平台，于是 [TiDB User Group](#) 应运而生。

与此同时，上线 TiDB 产品的企业用户越来越多，尤其是在以互联网为主的新经济里，TiDB 产品的普及度非常高，典型如微信的十二宫格里，其中的 11 家服务提供商将 TiDB 产品应用在生产环境。TiDB 产品就像一个魔盒，因各企业使用姿势的不同而变化万千，满足了多种多样的业务需求；它也是业务创新的利器，将很多以前不可能的业务变成了可能。比如餐饮 SaaS 服务行业，同时十几个维度的实时数据分析在早期的架构下很难实现，现在却变成行业标配。因此，我们常常听到用户的这些心声：

"我对 TiDB 感兴趣，但想深入了解一下别的用户是怎么使用 TiDB 的"

"对于这些具体案例，我还有一些问题想跟深度使用用户一起交流"

"我想结交更多同行，了解更多使用 TiDB 的企业"

为了顺应大家的呼声，TUG 成立之后的第一个系列活动，就是“TUG 企业行”。TUG 企业行是 TUG 联合 TiDB 产品众多企业用户共同承办的系列线下活动，一次活动时长约半天，活动的宗旨是“倾听用户的声音”，注重参与者的实际收获与参与感，它和之前官方主导的 Infra Meetup（见 2.1.3 节）有很多不同。

1. 主题与嘉宾不同：每次 TUG 企业行活动都会根据用户的呼声，确定一个大的主题、比如“数据库选型”、“异构数据库迁移”、“Cloud + Database”、“数据高可用”、“金融行业数据库实践”等等，并根据活动主题邀请在这个方向有成功经验的企业用户来做主讲嘉宾。
2. 活动形式更灵活：TUG 企业行活动很注重参加者之间的互动，除了常见的主题演讲外，还有小组讨论、圆桌会议、大咖说等丰富多样的形式，可以近距离和各个大佬直面交流一线问题。
3. 场地、氛围不一样：TUG 企业行会轮流在各知名企业举办，接待企业带领到访客人参观，准备精心的茶点，既开拓大家视野，还能结识很多同行、朋友，参与感十足。
4. 举办方不同：对于 TUG 企业行活动来说，TUG 是组织者，企业用户是承办者，PingCAP 官方则负责提供技术支持与赞助。

TUG 企业行自 2019 年 8 月正式开启，一经推出便立即得到了广大用户企业的支持和响应，先后走入了转转、Shopee、VIPKID、爱奇艺、沪江、喜马拉雅、UCloud、随手科技、知乎、网易游戏、执御等 10+ 企业，来自用户的各种最佳实践、数据库选型等几十个演讲受到了 TUG 小伙伴们和广大企业的普遍欢迎。



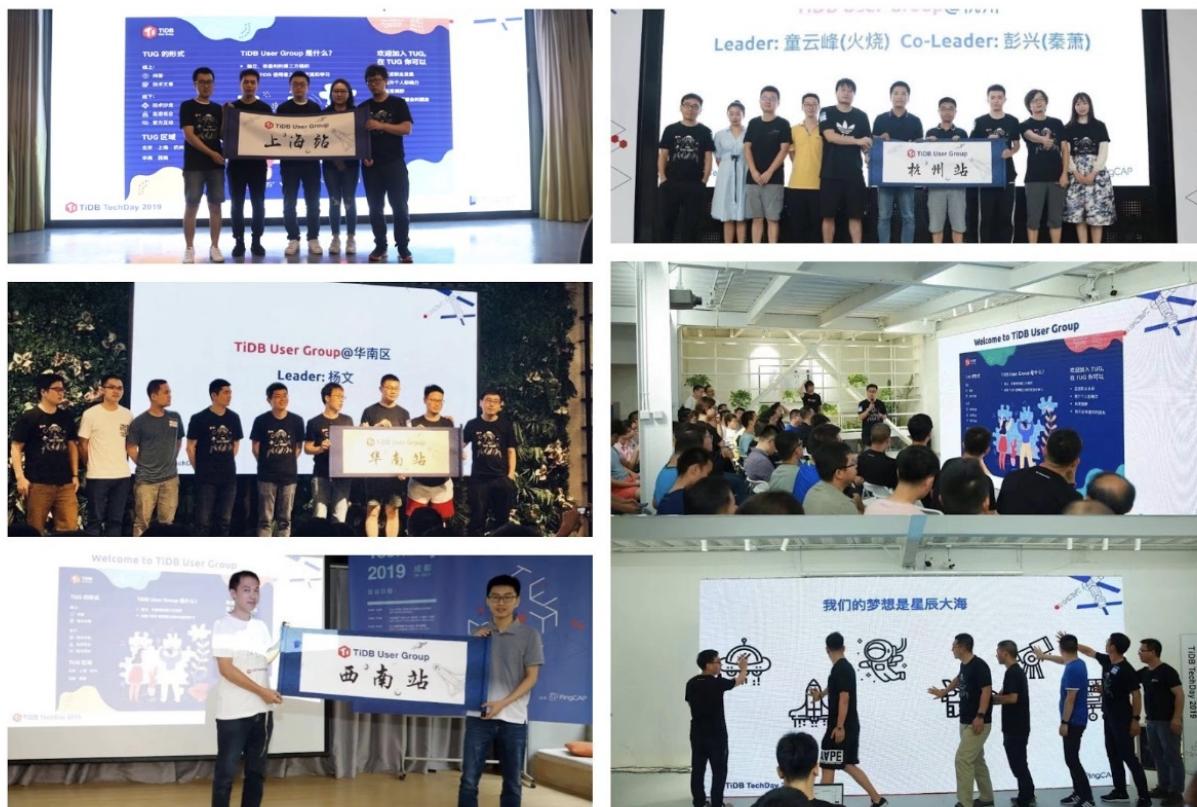
2020年伊始，整个国家都在经受疫情的考验，TUG 企业行活动也会做一些调整。做为 TUG 最重要的系列活动，线下企业行会在疫情结束后的第一时间和大家见面，而线上直播的企业行即将开启，敬请期待，欢迎广大企业和 TUG 组委会取得联系进行合办。在 TUG 里，用户的声音得以放大，用户企业的技术影响力也得以扩大。

## 2.2 TUG (TiDB User Group)

### 2.2.1 TUG 成立背景

截至 2019 年底，TiDB 分布式数据库已被近 1000 家不同行业的领先企业应用在实际生产环境，涉及互联网、游戏、银行、保险、证券、航空、制造业、新零售等多个行业，用户遍及中国、美国、欧洲、日本、东南亚等全球各地。随着 TiDB 分布式数据库的用户不断快速增长，用户层面迫切需要一个可以相互交流学习、了解行业动态、更好地与 TiDB 产品链接的平台，这些需求促使了 TUG 的形成。

TUG 成立于 2019 年 6 月，是由 TiDB 用户自发组织、管理的独立社区，PingCAP 官方提供赞助与支持，目前共有北京、上海、杭州、华南（广州、深圳为中心）、西南（成都为中心）5 个活跃的 Local TUG 小组。TUG 成立的初衷是“连接用户，共建社区”，在成立初期就吸引了众多对数据库技术有兴趣的架构师、DBA、开发等技术同学，早期的成员来自美团、58 集团、转转、爱奇艺、知乎、京东云、腾讯、Shopee、网易游戏、微众银行、平安科技、小红书、bilibili 等 100+ 深度使用 TiDB 的企业。



TUG 社区以优质的技术内容为核心资产，倡导开放透明、相互尊重、争相积极的社区氛围，所有成员都可以在 TUG 学习、交流 TiDB 相关技术内容，并自发在社区分享内容，从而扩大 TiDB 技术及 TUG 社区的影响力。到 2019 年底，TUG 在成立后的短短半年时间内，已累计沉淀了 78 篇优质技术文章和 78 个技术 Talk，直接和间接影响了 5000 余人，TUG 成员的活跃度和 TUG 的号召力可见一斑。

### 2.2.2 TUG 组织架构

作为一个用户自主管理的社区，TUG 的组织架构分为管理 (Management) 和专业 (Professional) 两条线。

- TUG Management 主要负责统筹规划 TUG 的发展，并积极推广 TUG 组织，其成员每年进行一次换届选举。管理线成员主要有以下角色：
  - TUG Governance Committee：TUG 管理委员会成员，TUG 最高决策者，确定 TUG 组织架构、治理结构、管理流程、全年大事，利用自身资源推广和发展 TUG 组织；
  - TUG Infra Team Leader/Co-Leader：主要负责 TUG 内容审核、社区活动的音视频字幕整理、社群网站答疑、社群管

理等；

- TUG Local Leader/Co-Leader：主要负责本地区的 TUG 活动和本地 TUG 组织的发展壮大。其中 Leader 负责制定全年活动计划、邀请嘉宾、拓展人脉等，Co-leader 负责组织和执行。
- TUG Professional 目前主要包括 TiDB 最有价值布道师（TiDB MVA），他们是为 TUG 贡献了优质的专业技术内容的人员，后续我们会有更具体和更进阶的角色露出，敬请期待。
  - TiDB MVA (Most Valuable Advocate)：TUG 社区荣誉称号。通过演讲、撰文等多种方式分享 TiDB 相关知识和经验，当年度为 TUG 贡献的优质内容数量  $\geq 3$  的成员可获得 TiDB MVA 的称号。到 2019 年底，TUG 已经涌现了 29 位 TiDB MVA；

TUG 社区中愿意积极参与社区建设，并利用社交媒体等多渠道推广 TiDB 技术及 TUG 的成员为 TUG Ambassador。TUG Ambassador 是 TUG 管理线成员和专业线成员的候选人，和 TUG Management、Professional 成员一起组成了 TUG 核心成员。

### 2.2.3 TUG 成员的贡献、积分和权益

TUG 鼓励所有成员以各种方式积极为社区做贡献，为 TiDB 使用者提供舞台发挥自己的价值。不管是在 TUG 中组织活动，通过技术 Talk 或文章做内容贡献，还是审核文章，又或者是为社区其他成员答疑解惑，进行社群管理等，无论事务大小，都是为 TUG 的发展贡献了自己的力量。为了感谢 TUG 成员的努力和付出，TUG 的各种贡献都可以获得相应的积分，并根据积分换取包括周边礼品、技术支持和技术培训等在内的 TUG 权益，也可获得对等的社区荣誉。

### 2.2.4 TUG 展望

2020 年起，TUG 进一步完善自身组织架构和管理规则，开设面向行业、场景的技术小组，使 TUG 的运行更加流畅自如，成员都能有所成长，用户生态更加积极活跃。与此同时，TUG 也开始探索 Go Global 之路，让全球的 TiDB 分布式数据库使用者都能找到归属地。欢迎更多 TiDB 分布式数据库使用者加入 TUG，和 TUG 一起成长！

## 2.3 Talent Plan: Made for Community, by Community

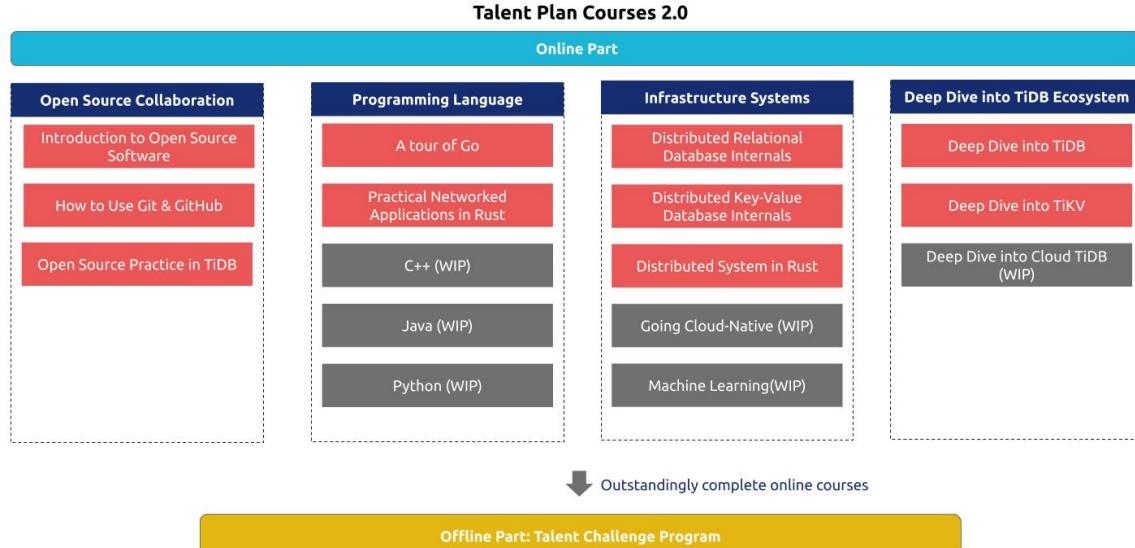
Talent Plan 是由 PingCAP 发起的、面向社区人才培养的开源项目，该项目在 GitHub 上已拥有 3800+ Star。Talent Plan 旨在开发并整合一系列开源协作、编程语言、基础设施等课程资料并与 TiDB 生态系统实践经验相结合，帮助开源爱好者以及基础设施爱好者逐步获得基础知识沉淀和技能提升。

自 2018 年 12 月份 Talent Plan 项目启动至今（北京时间 2020 年 3 月 7 日），线上课程共吸引国内外实名认证课程爱好者 400 余人，40 余人顺利通过线上课程的考核；线下实训项目成功举办 4 期，38 名同学顺利结业，15 名同学在通过 Talent Plan 课程考核后陆续加入 PingCAP，其他同学也持续在 TiDB Community 中发挥自己的光和热。



### 2.3.1 课程内容

Talent Plan 是为社区提供的学习资料，也在社区的建议和反馈中不断优化成长。在社区小伙伴的共同努力下，Talent Plan 正在进行新一轮的升级，这是一个涵盖了开源协作、Rust 语言、分布式数据库、分布式系统、TiDB/TiKV 原理精讲等线上学习课程以及 TiDB 开源社区项目线下实训课程的更大的学习版图。课程框架见下图：



## 1. 线上课程

线上课程包括 4 个课程系列，分别是：[Open Source Collaboration 课程系列](#)、[Programming Language 课程系列](#)、[Infrastructure Systems 课程系列](#)、[Deep Dive into TiDB Ecosystem 课程系列](#)。

每个课程系列中会有相关的课程供大家选择，课程之间是相互解耦的，大家可以结合自己需求自由规划学习路径。

### Series 1: Open Source Collaboration(WIP)

这是专门为零基础开源爱好者准备的全新课程系列，我们希望通过这个系列课程的学习，即使是技术小白也能快速了解开源是什么、不同开源软件许可协议的差异、知名开源基金会（Linux、Apache、CNCF 等）的运作方式以及 TiDB 在开源方面的实践，快速掌握参与开源项目的小技巧。

这个课程系列目前仍在小范围测试阶段，如果你对于这个课程感兴趣，欢迎通过[参与通道](#)与我们取得联系。

### Series 2: Programming Language

这个课程系列中将逐步对当下常用的编程语言学习课程进行整合，包括但不限于 Go、Rust、C++、Python 等。

需要特别介绍的是由 Rust 核心作者 Brian Anderson 精心设计的 Rust 学习课程——[Practical Networked Applications in Rust](#)，通过这部分课程的学习，你将能够独立创建一个基于 Rust 语言的 Key-Value 数据库。

### Series 3: Infrastructure Systems

这个课程系列专为基础设施爱好者设计，其中包括：

- 用 Go 语言全新设计的分布式关系型数据库 ([TinySQL](#)) 课程 (WIP)
- 用 Go 语言全新设计的分布式 Key-Value 数据库 ([TinyKV](#)) 课程 (WIP)
- 用 Rust 语言打造的分布式系统 ([Distributed Systems in Rust](#)) 课程

TinySQL 几乎涵盖了分布式数据库 SQL 层最重要的部分，课程介绍按照由简单到复杂，由静态到动态的顺序展开：

- 首先对 SQL 和关系代数做简要介绍，为后面的课程做准备。
- 然后关注于一个只读 SQL 的执行，从 Parser 开始解析语义，到执行器如何执行语义，再去了解优化器如何选出最优的执行计划；
- 最后关注于那些改变数据状态的 SQL（包括 DML 以及 DDL），以及如何处理它们和只读语句之间的相互影响。

TinyKV 类似已有的 [Distributed Systems in Rust](#) 课程，它同样受著名的 MIT 6.824 所启发，但这次将更加接近 TiKV 的实现，引入调度相关逻辑，学员可以从 0 到 1 实现一个完整可用的分布式 KV 服务。课程主要分为四个部分：

- LAB1: 实现单机 KV server
- LAB2: 基于 Raft 实现多副本高可用 KV server
- LAB3: 实现 multi-Raft 以及数据均衡调度
- LAB4: 基于 percolator 模型实现分布式事务

通过 TinyKV 课程的学习，你将会从实践中对 Raft 协议，Percolator 分布式事务模型有一个更深刻的理解。同时，在实现 TinyKV 的过程中也有助于了解 TiDB + TiKV + PD 的实际框架，之后深入研究 TiDB/TiKV/PD 的源码会更加游刃有余。

目前，全新设计的 TinySQL 和 TinyKV 课程已经基本实现了一个可用的课程框架和相关测试，接下来会进行进一步的优化调整，同时课程材料也在紧锣密鼓地进行编写中。如果你对于这个课程感兴趣，欢迎通过 [参与通道](#) 与我们取得联系。

#### Series 4: Deep Dive into TiDB Ecosystem

这个课程系列将深入解读 TiDB 生态项目内部设计原理，TiDB、TiKV、Cloud TiDB 深度原理解析会逐步呈现在大家面前。

## 2. 线下实训——Talent Challenge Program

线上课程成绩优秀的小伙伴将会被邀请参与线下实训项目，实训项目以小组方式进行，每个小组选择一个与 TiDB 生态系统相关的实训项目，在 1 个月左右的时间里通力协作完成项目并进行最终答辩，答辩通过的同学将获得专属 PingCAP Talent Plan 结业证书，线下实训期间表现优秀的还将有机会拿到 PingCAP 校招/实习免面试绿色通道/Special Offer、PingCAP/TiDB 全球 Meetup 的邀请函等。

截至目前，线下实训已成功举办 4 期，累计线下学员数 41 人，累计覆盖 10 所高校，38 名同学顺利结业。



### 2.3.2 学习路径

#### 路径 1: Distributed Storage Engineer

如果你想要成为一名分布式存储工程师，可以选择以下课程组合：

- Programming Language: "[Practical Networked Applications in Rust](#)"
- Infrastructure Systems: "[Distributed Key-Value Database Internals\(WIP\)](#)" & "[Distributed Systems in Rust](#)"
- "[Deep Dive into TiKV](#)"

#### 路径 2: Distributed Relational Database Engineer

如果你想要成为一名分布式关系型数据库工程师，可以选择以下课程组合：

- Programming Language: "[A Tour of Go](#)"
- Infrastructure Systems: "[Distributed Relational Database Internals\(WIP\)](#)"
- "[Deep Dive into TiDB](#)"

### 路径 3: Cloud TiDB Engineer

如果你想要成为一名云数据库工程师，可以选择以下课程组合：

- Programming Language: "[A Tour of Go](#)"
- Container & Container Orchestration (Docker、K8s ...)
- "[Deep Dive into Cloud TiDB\(WIP\)](#)"

### 路径 4: 开源社区运营

如果你对开源社区运营感兴趣，可以选择：

- Open Source Collaboration(WIP): "Introduction to Open Source Software" & "Build a Welcoming Community"
- 其他社区运营相关书籍，如：[The Art of Community: Building The New Age Of Participation](#), [The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary](#), [People Powered: How communities can supercharge your business, brand, and teams](#)

除了以上学习路径，你也可以结合自己的需求，将上述课程自由组合，挖掘新的学习路径。

### 2.3.3 学员们说

- “我超级喜欢 PingCAP 的氛围还有培训的方式。这次培训从语言和数据库理论学习到跟进最新论文，再到动手实操小 Demo，了解 TiDB 各个部分实现原理以及最后阅读分析 TiDB 各个模块的代码，丰富的课程让我对数据库的理解又加深了一层。希望自己以后的研究东西能够贴近到具体的场景和系统去发现问题，并把自己的研究成果落地。我也非常期待 PingCAP Global Meetup 之旅和各路大牛交流。其实，现在内心还没平复下来，这次来北京收获太多了，现在话都组织不好了。”——兰海（第一期优秀学员）
- “参加 Talent Plan 是一次非常珍贵的体验，一方面是学到了许多的没有接触过的分布式领域的知识，另一方面也结识了来自全国各个高校的优秀的小伙伴以及 PingCAP 的各位厉害的导师，也为我之后来 PingCAP 实习埋下了伏笔。”（节选自[这门分布式 KV 存储系统课程教会了我什么？| 我与 Talent Plan](#)）——张艺文（第二期优秀学员）
- “参加这次 Talent Plan，我不仅学习到了丰富的知识，还深入地参与到具有挑战性的工程项目。更重要的是交到了一群非常优秀、靠谱的朋友。非常感谢 PingCAP 举办这个活动，希望 Talent Plan 越办越好。”——黄文俊（第三期优秀学员）
- “经过 Talent Plan 的学习让我明白了，在实验室里 fancy 的想法在工业界可能并不 work，实际应用环境要比实验环境严格苛刻很多很多。经过这次线下课程的学习，我以后在设计方案的时候会着重考虑从现实的角度出发。”——邹欢（第三期最具潜力奖获得者）
- “这次参加 Talent Plan 收获十分巨大，首先是认识了一群很棒的小伙伴，正如崔秋老师说的，一个月时间很短，但是友谊却是一辈子的。然后，我对整个 TiDB 生态以及其中的各个模块有了更高层次的认识和理解，也切身体验到了 PingCAP 很 Cool 的 Geek 氛围，总之度过了很有意义的一个月。”——林宁（第三期最具潜力奖获得者）
- “工作后参加 Talent Plan 是非常神奇的体验。在这不仅能学习到 TiDB 各个模块的基础理论，还能听到一线的开发直接分享在生产环境中实践的一些细节和经验。最后还能和大家一起齐心协力去落地一个项目，并见证它成为这个优秀开源数据库的一部分。感觉自己回到了久违的学生时光，再一次体验快速成长，并重新找到当初选择这个行业的初心”——郑向升（第四期学员代表）

### 2.3.4 如何参与 Talent Plan

Talent Plan 规划了一个巨大的学习版图，我们期待着能与社区小伙伴一起逐步实现、不断优化，真正做到“Made for Community, by Community”。

- 如果你已经迫不及待想要开始 Talent Plan 课程的学习，[Talent Plan 官方网站](#) 中有已经规划好的 TiDB 及 TiKV 两条路径供你学习。

- 如果你想要为 Talent Plan 升级版课程贡献自己的一份力量，我们在 [TiDB Community Slack Workspace](#) 中开通了channel: **#wg-talent-plan-courses**，欢迎感兴趣的小伙伴们加入进来，一起打造更加“酷炫”的 2.0 版本！

## 2.4 Challenge Program

### 2.4.1 起源

TiDB 产品的每一次微小进步都离不开社区小伙伴的支持和帮助，在产品不断迭代的过程中，越来越多的小伙伴不断的参与到 TiDB 开源社区的建设当中，越来越多的小伙伴在 TiDB 开源社区用自己的方式表达着对于开源的热情和对于技术的追求，TiDB 也在社区小伙伴的推动下，不断地刷新着过去的成绩。

为了让 TiDB 产品在稳定性、性能、易用性等方面更上一层楼，PingCAP 推出一个中长期的挑战计划——TiDB Challenge Program。每一期的挑战赛都会在 TiDB 产品线的各个代码仓库中开放一些 Issue 来供社区小伙伴探讨和解决。每一个 Issue 都会对应一定的积分，参赛选手可在每个赛季结束后用获得的积分兑换社区礼品。

目前为止，挑战赛已经进行了 2 期：

season	period
Challenge Program season 2	2020.03~2020.05
Challenge Program season 1	2019.11~2020.02

### 2.4.2 第一季：性能



性能挑战赛的官网地址为：[TiDB Performance Challenge](#)。本次比赛奖项设置为：一等奖 1 名，二等奖 2 名，三等奖 3 名，其余分数高于 600 分的团队或个人为优秀奖，各团队和个人的获奖情况如下：

- 一等奖：.\* Team (15050 积分)。
- 二等奖：niedhui (4300 积分) 和 cator (3500 积分)。
- 三等奖：pingyu (2600 积分)、Renkai (2550 积分) 和 js00070 (1800 积分)。
- 优秀奖：ekalinin (1450 积分)、mmyj (1050 积分)、AerysNan (750 积分)、MaiCw4J (650 积分)、Rustin-Liu (650 积分) 和 koushiro (650 积分)。

在这次比赛中，选手们提升了 TiDB SQL 引擎的计算速度（优化了 in/like 表达式的执行性能），提升了 TiTan 的 GC 性能，极大的降低了 GC 对写入的影响，优化了 PD API 的性能，减少资源使用，降低对 PD 在线服务的影响。

更多精彩内容可查看 [性能挑战赛回顾](#)。

### 2.4.3 第二季：易用性



作为 TiDB Challenge Program 系列的第二赛季，这一季将聚焦 TiDB 易用性。在书写本书的同事，第二季正在如火如荼的进行。第二季官网地址：[TiDB Usability Challenge](#)

在进行第一季的过程中，PingCAP 在 AskTUG 网站上发起了“我的 TiDB 听我的”的需求征集活动。需求收集从 2019.12.17 开始，2020.01.12 结束。经过历经 1 个月的需求收集，整理后对 20 个用户需求进行了投票活动。需求投票从 2020 年 2 月 11 日开始，2020 年 2 月 20 日结束。每人只能投票一次，投票可多选，最少可投一个选项，最多可投 5 个选项。

经过前期 2 轮和需求有关的用户活动，挑战赛第二季从 2020 年 3 月 2 日正式开始，2020 年 05 月 30 日结束，持续 3 个月。本着「用户的需求就是我们的方向」，这一季的大部分需求都将由用户投票产生，这些需求将以任务的形式呈现在第二季挑战赛中，参赛选手可以通过认领任务的方式获得积分，在赛季结束后进行奖品兑换。

另外，比赛过程中，排名前三的需求，整体上各自分别加 10000，8000，6000 分。等需求完整的实现或者挑战赛结束，需求的加分将由需求的子任务完成者们一起分享。考虑到需求不一定能在挑战赛期间完整做完，这些需求额外积分的分享规则为：(该挑战者完成的该需求的子任务积分和/所有挑战者完成的该需求的子任务积分和)\*这个需求的总加分。

相比于上一季，第二季首次面向了海外，首次在 TiDB、TiKV、PD 以外的代码仓库开启了挑战赛活动，几乎覆盖了 TiDB 产品线上所有的开源代码仓库。

#### 2.4.4 比赛规则

TiDB Challenge Program 全流程包括：查看任务->领取任务->实现任务->提交任务->评估任务->获得积分->积分兑换，其中“获得积分”之前的步骤都将在 GitHub 上实现。详细的比赛规则可查看 [这里](#)。

#### 2.4.5 学习资料

PingCAP 提供了 [TiDB 精选技术讲解文章](#)，帮助大家轻松掌握 TiDB 各核心组件的原理及功能。此外还有 [数据库小课堂](#)，帮助选手快速熟悉数据库知识，点击以上链接即可轻松获取。

## 2.5 PingCAP Incubator

PingCAP Incubator 的灵感来自于 CNCF 的运作方式，TiDB 社区一直有一个想法：在用户与开发者之间建立沟通的桥梁，让来自真正生产环节的需求由真正有兴趣且有能力的开发者来实现。其创立初衷是为了让来自社区的项目可以快速的获得 TiDB 社区的资源支持，比如社区治理经验，贡献者对接，导师指导，TiDB 产品用户企业场景磨砺等等。

PingCAP Incubator 旨在梳理一套相对完整的 TiDB 生态开源项目孵化体系，将关于 TiDB 开源生态的想法与实际生产环境中的需求相关联，通过开源项目协作方式，共同将想法落地。力求想法项目化，“提出->孵化->毕业”以及毕业后的项目走向都有章可循，同时结合项目不同特征及孵化目的，将项目划分为 Feature 类和 Project 类，针对性地给出孵化流程建议。

PingCAP Incubator 设立了导师制度，导师来源于 TiDB 核心研发工程师 TUG 核心成员等，TiDB 核心研发工程师将从技术实现的角度为项目提供支持，无论是最初的立项还是孵化过程中的技术难点探讨，都可以向研发导师寻求帮助；TUG 项目导师将从实际生产环境中的需求出发，明确项目的推进方向，在项目孵化过程中，也会定期进行指导，帮助项目完成孵化。此外，在项目推进的过程中，TiDB Community 项目管理委员会（PMC）也会全程助力，为项目孵化提供必要的资源支持。

截止到 2020 年 3 月，有好几个潜在的优秀项目进入 [PingCAP Incubator](#) 进行孵化，非常值得期待，包括：

- TiDB 4.0 最期待的功能之一的 TiDB Dashboard(包含 TiDB 集群热点可视化 Key Visualization, 全局日志检索，火焰图，自动巡检和生成报告)
- TiDB 第一本开源书籍 - TiDB In Action , 48 小时 Book Rush
- TiDB 轻量级的集群管理工具 - TiUP
- 从零到一教大家做分布式数据库 - Talent Plan Courses(TinySQL/TinyKV)
- AskTUG 背后的开源论坛 Discourse 后端数据库从 PG 迁移到 TiDB
- 跑在浏览器里面的 TiDB - TiDB Wasm

## 2.6 PingCAP University —— Embrace Unlimited Possibilities

PingCAP University 是 PingCAP 官方面向企业和个人，致力于培养开发、管理和应用分布式关系型数据库系统的一流人才而设立的培训和认证机构。

PingCAP University 目前提供两大人才培养项目：

- 培养具备独立运维和管理分布式关系型数据库的 [TiDB DBA 认证项目](#)，即 PCTA (PingCAP Certificated TiDB Associate) — PingCAP 公司认证 TiDB 数据库专员和 PCTP (PingCAP Certified TiDB Professional) — PingCAP 公司认证 TiDB 数据库专家；
- 培养独立开发 TiDB 生态细分模块能力的 [Talent Plan 项目](#)。

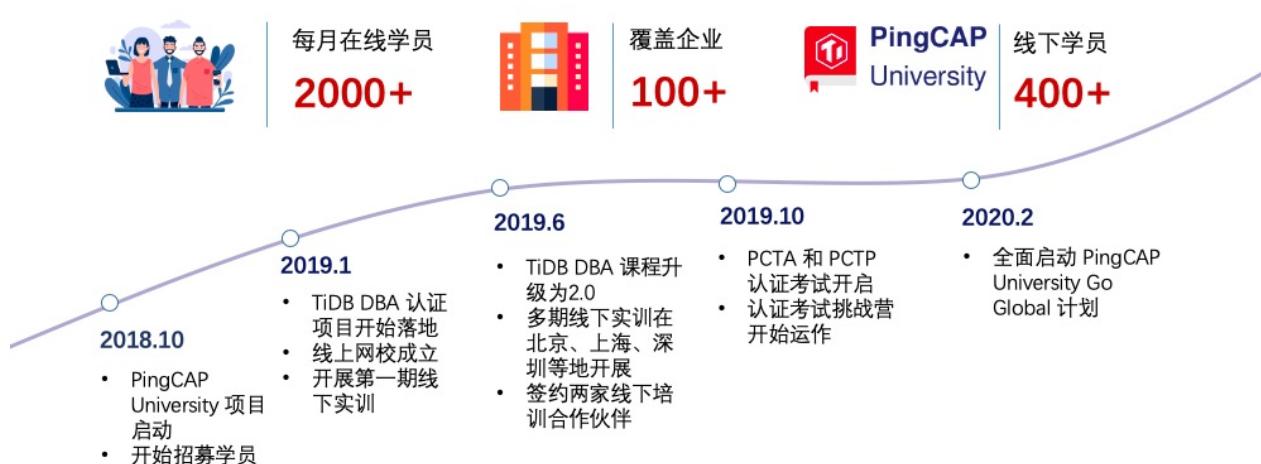
PingCAP 讲师团队均来自 PingCAP 官方的资深解决方案架构师、核心技术研发工程师、高级 TiDB DBA 及 TiDB 官方认证讲师，具备专业的 TiDB 理论素养，拥有丰富的实战经验。

### 2.6.1 TiDB DBA 认证项目简介

#### 1. 项目简介

TiDB DBA 认证项目自 2018 年 10 月启动、2019 年 1 月正式落地，得到了社区伙伴的广泛响应。截止 2020 年 3 月 7 日，线上课程每月有来自国内外的 2000 余人观看，线下实训项目成功举办 11 期，400 余同学顺利毕业，数十名同学通过 PCTA、PCTP 认证考试，这批应用 TiDB 的核心人才正服务于各行各业。

### PingCAP University : Embrace Unlimited Possibilities



#### 2. 课程亮点

- (1) 核心工程师亲授：所有授课讲师均为 TiDB 核心研发工程师或客户支持工程师，原厂技术原厂传授，更有明星讲师倾囊相授。
- (2) 课程体系完善：原厂技术核心、理论实操并重、线上线下结合，助力深度理解 TiDB 架构、原理及最佳实践，带来更多同行认可。
- (3) 前沿技术解读：不止于 TiDB，更有 NewSQL 数据库领域前沿技术、多种行业大厂应用场景等丰富呈现，持续更新，与未来同行。

(4) 原厂权威认证：PCTA 基础认证注重实践，PCTP 进阶认证注重深度原理，严格的考试体系保障原厂背书价值，打开高起点职场大门。

## 2.6.2 TiDB DBA 认证项目课程内容

随着 TiDB 产品的持续迭代、用户场景的不断丰富，PingCAP University 在实践中保持与学员的沟通，持续打磨课程，TiDB DBA 认证项目的课程内容不断升级。TiDB DBA 认证项目 2.0 于 2019 年 6 月推出，在保留高密度干货、理论和实操相结合的一贯特点之外，本次升级有以下方面的优化：

- 课程内容扩展：2.0 课程增加了分布式事务原理、存储引擎内核原理、计算引擎内核原理、优化器深度解析等内容，整个课程深入浅出、更加完整和体系化；
- 优化学习曲线：2.0 课程分为基础篇、高级进阶篇和扩展篇三个层次，层层递进，能满足不同层级的学员，从入门到进阶一次搞定；
- 知其然，更知其所以然：2.0 课程不但教学员如何操作，还会讲解 TiDB 计算、存储、调度等底层架构原理，以及时下火热的云原生技术、混合数据库（HTAP），让学员们能从全局更好地了解和使用 TiDB，深刻理解数据库发展的新趋势；
- 理论实践两手抓：2.0 课程延续了 1.0 课程的设计，理论知识和实操课程并重。在实操课程，我们给每个学员配备了硬件环境，从安装部署升级、数据迁移、到跨机房多活高可用部署，老师都会全程通过 Demo show 的方式，让学员们真正可以快速学以致用。

TiDB DBA 认证项目 2.0 课程框架见下图：

课程类别	培训课程	培训目标	考试认证
基础知识	一、概述 【2 课时】	初级 TiDB DBA	PCTA: PingCAP Certified TiDB Associate
	二、系统安装部署与管理 【3 课时】	需要学习并熟练掌握如下内容： TiDB 概论、架构简介 TiDB 集群管理概述及使用实操 TiDB SQL 与调优 TiDB 对象管理与维护 TiDB 同构和异构平台数据迁移解决方案及实操 TiDB 备份与恢复原理及实操 TiDB 集群监控报警及日志管理 TiDB 常见问题处理 TiDB 业务开发最佳实践及应用案例介绍 TiDB 性能 Benchmark 介绍	
	三、使用管理手册 【2 课时】		
	四、生态工具 【6.5 课时】		
	五、备份恢复 【1 课时】		
	六、TiDB 监控与告警 【2 课时】		
	七、业务开发优化 【2 课时】		
	总结综述		
高级进阶	八、TiDB 计算内核体系 【4 课时】	高级 TiDB DBA	PCTP: PingCAP Certified TiDB Professional
	九、TiKV 存储内核体系 【4 课时】	需要学习并熟练掌握如下内容： TiDB Server 原理 PD 原理 TiKV 原理及架构 深入 Coprocessor 原理 TiDB Server 性能调优 TiKV 性能调优 TiDB 分布式事务原理 TiDB SQL 实现原理 TiDB SQL 性能调优 TiDB 高可用原理、方案及实战	
	十、事务 【4 课时】		
	十一、系统性能优化 【5 课时】		
	十二、SQL 优化 【8.5 课时】		
	十三、高可用及容灾 【3 课时】		

## 1. 线上课程

PingCAP University TiDB DBA 认证项目 [官方网校](#) 已正式上线，可免费进行线上基础课程 PCTA 的学习，有助于学员快速了解 TiDB 产品全貌。学员除了可以在线自主学习外，还可以参加 [在线学习打卡营](#)，在讲师集中答疑中进行深入交流，优秀学员还可享受线下培训的奖励计划。

## 2. 线下实训

线上课程成绩优秀的小伙伴，对社区有突出贡献的布道师以及购买商业培训认证服务的企业将会被邀请参与线下实训项目。实训项目包含 PCTA 和 PCTP 两部分内容， $6\text{天} \times 8\text{学时/天} = 48\text{学时}$ ，一般在周末两天进行，持续3周。

截至目前，线下实训已成功举办 11 期，累计线下学员数 400 余人，累计覆盖 100 多家企业。

学员们说

- 内容安排得很充实，课程设计很好。——孙同学，某证券公司
- 之前没怎么接触过 TiDB，所以觉得目前上课方式和内容很好，干货比较多。——柳同学，某银行
- 整体课程都很不错，前面的理论有一些没有懂，通过后面的实践这些原理都得到了很好的理解。老师们都非常厉害，时间允许的话，希望在优先保证正常内容讲解的基础上能扩展更多知识点。——王同学，某 IT 服务商
- 课程安排得很饱满、合理，整体感觉不错。——彭同学，某网约车平台
- 课程讲到了 Binlog 和 DM 以及高可用，不过感觉只有两地三中心的实战是不够的，周边生态工具其实在平时运维场景中可能是使用比较多的，可以通过几个重要场景实践和理论一起讲感觉会更好些。——Mike 同学，某电商平台
- 希望可以有更多生产环境中的问题和建议可以分享，尤其是不同类型的公司使用的重度或者轻度 TiDB 的场景介绍，甚至是其中部分组件的使用特点以及适配程度。大家业务不同，需求不同，可能会产生一些其他的思维碰撞。——陈同学，某移动支付解决方案提供商
- 课程内容丰富全面，对于掌握 TiDB 运维知识及其实现原理非常有用。——李同学，某大型网络银行
- 课程形式方面，线上学习灵活，线下学习高效，可以根据自己的时间安排选择。官方在 B 站也有定期的分享视频。课程内容方面，分类详细，学习安排紧凑，可以全方位深入了解 TiDB 的设计和使用理念。——张同学，某知名弹幕网站
- PingCAP University 的课程优点：课程视频丰富，文档丰富详细。个人学习过程中也感到稍有不足，比如部分视频过长，不精练，需要很长时间才能看完。——刘同学，某知名本地信息网站



### 2.6.3 TiDB DBA 认证考试

PCTA 和 PCTP 是对于即将或已经从事 DBA、架构师等职位的人员掌握和使用 TiDB 的能力进行测试和评定，并由 PingCAP 公司作为唯一权威机构向考核通过的人才颁发相应证书。

PCTA (PingCAP Certified TiDB Associate) 是 PingCAP 公司认证 TiDB 数据库专员的缩写。PCTA 要求具备安装部署及日常运维分布式关系型数据库的能力。PCTA 需要学习并熟练掌握 TiDB 架构原理、安装部署、周边工具等基础知识。

PCTP (PingCAP Certified TiDB Professional) 是 PingCAP 公司认证 TiDB 数据库专家的缩写。PCTP 要求具备管理大型分布式关系型数据库集群的能力。PCTP 需要学习并熟练掌握 TiDB 的深度原理及高级 Feature、性能调优、SQL 优化、Trouble Shooting 等进阶内容，要成为 PCTP 必须先获得 PCTA 认证。

目前 PCTA 和 PCTP 认证主要面向 TiDB 3.0 版本。

#### 认证考试学习资料

- 依据 [考试大纲](#)，自学 [官方网校](#) 及 [官方文档](#)，可同时参考 [AskTUG](#) 相关内容。
- 学习 PingCAP University 免费开放的 PCTA（基础）视频课程（18 门课程共计近 20 课时），自学 [官方文档](#)，报名参加 [认证考试挑战营](#)。
- 参加 PingCAP University 原厂线下商业培训和参加 [认证考试挑战营](#)，包含 PCTA（基础）和 PCTP（进阶）全方面内容，注重理论和实操，助力快速熟悉、掌握 TiDB 的进阶技能。

- 通过培训 [授权合作伙伴](#) 参加线下商业培训课程，包含 PCTA（基础）和 PCTP（进阶）两部分内容。

## 2.7 AskTUG

### 2.7.1 AskTUG 建立背景

“通过一个平台，一定能找到 TiDB 分布式数据库所有问题的满意答案”，因为这样的愿望，TUG 汇集 TiDB 产品用户的集体智慧建立了 AskTUG 技术问答网站，并于 2019 年 8 月正式公开上线。作为 TUG 成员学习、分享的“聚集地”，TiDB 产品用户可以在这里提出、解答问题，相互交流探讨，并与 TiDB 产品产生更多链接。

自上线以来，AskTUG 逐渐吸引了越来越多用户的关注，截止 2019 年 2 月底，AskTUG 已有 1500+ 注册用户，沉淀了 1300+ 问题和 100+ 技术文章，已是一个日回复数 100+ 的活跃网站。

The screenshot shows the AskTUG website interface. At the top, there's a navigation bar with links for '问答' (Questions), '博客' (Blog), 'TUG', '活动' (Events), and 'PingCAP University'. Below the navigation is a search bar and a 'New Topic' button. The main content area has two sections: '分类' (Categories) on the left and '主题' (Topics) on the right. The '分类' section lists various categories like '数据迁移与同步', 'SQL 优化', 'TiSpark & TiFlash', etc., each with a count of posts (e.g., 59 / 月). The '主题' section shows recent posts, each with a thumbnail, title, and timestamp. For example, one post about Oracle, MySQL, Greenplum, Apache Spark, MariaDB ColumnStore and Oracle..... was posted 25 days ago.

### 2.7.2 AskTUG 内容简介

AskTUG 上超过 80% 都是 UGC 内容，全站的核心内容为以下板块：

- **问答**：AskTUG 建立的初衷是帮助 TiDB 产品的使用者准确高效地解决问题。截至目前，AskTUG 的问题回复率达到 100%，包括 TiDB 产品各组件从安装部署到性能调优的问题，都能得到专业及时的解答。配备网站的搜索功能，用户可以自助搜索获取信息，省时高效。
- **技术博客**：用户可以在这里自发分享 TiDB 产品、数据库和云原生相关的技术原理、心得和应用实践，干货满满。优质的技术文章不仅可以供其他用户参考学习，内容贡献者还能通过总结输出巩固技术能力，提升个人影响力。
- **TiDB 活动**：TUG 线上线下技术活动信息会第一时间在这里更新。除了技术干货分享活动，AskTUG 还经常发起一些泛技术类话题进行探讨，比如行业发展趋势、突发事件应对与影响等，对于这类话题大家可以各抒己见，AskTUG 也成为大家吐露心声和吐槽的社交平台。
- **AskTUG Weekly**：网站每周热门内容的总结，让用户能快速获取网站精华内容以及用户关注热点。

作为 TUG 优质内容的载体，AskTUG 可以说是一个“宝库”，一方面可以为用户“排忧解难”，提供参考学习的宝贵资源，另一方面也可以让 TiDB 产品设计和开发人员倾听用户的声音，了解用户的关注点，做到了产品与用户的链接，让 TiDB 能更好地服务于用户。

### 2.7.3 社区小伙伴们说

- AskTUG 对于解决 TiDB 产品使用问题很有帮助，尤其是比较复杂的问题，例如内存相关的问题。个人认为咱们 AskTUG 解决效率很高。希望 AskTUG 上再多一些性能优化方面的文章，最好是从实践中得到的经验。
- AskTUG 能很好地解决平时遇到的问题，(问题解答的) 效率也很高，专业程度、技术支持都很好，有时候很多大佬也过来解答问题。总体来说 AskTUG 的体验很好。

#### 2.7.4 AskTUG 展望

当前，AskTUG 还处在最基本的“有问必答”阶段，距离“真正解决实际问题，得到满意答案”这个目标，还有很长的路要走。在 2020 年，AskTUG 将从四个方面进行优化：

- 首先，提升问答准确率和效率。这既需要广大用户参与回答和讨论，也需要 PingCAP 专职技术支持小组的专业解答；同时，对于平台展现出来的产品优化问题，PingCAP 也成立了“产品易用性改进小组”进行定向跟进；
- 其次，优质内容的整理与沉淀。根据用户的使用阶段，AskTUG 将陆续推出 POC 最佳实践、SOP 运维、典型 Case Study、FAQ 等系列文章，这四大系列都会在上半年发布，供大家参考；
- 再次，收敛问题入口，通过统一的标签、主题、搜索权重等，进行更精准的问题定位，归类相似问题，避免重复提问；同时，提升用户体验，AskTUG 平台将优化自身的 UI 设计、功能、搜索准确度等方面，更精准地进行资料、内容推送，让优质内容浏览起来更加方便、快捷；
- 最后，AskTUG 引入了“未解决问题”的自动工单，升级了支持和跟进流程，不让任何一个有价值的问题被遗漏。

相信在 TUG 小伙伴们的共同努力下，AskTUG 问答平台将不断发展壮大，成为一个活跃、饱含优质内容，高效支持的平台。

## 2.8 Contribution Map

### 2.8.1 Contribution Map 的由来

TiDB 是一个开源的分布式数据库，目前（2020.3.7）在 GitHub 上有 22.7K 个 star，是分布式数据库领域关注度非常高的一个项目，目前有 400 多个 contributors 参与了该项目的贡献，可以说 TiDB 是一个在大家的共同努力下成长起来的项目。为了能让更多的人方便地参与 TiDB 及其周边的项目，我们制作了 [Contribution Map](#)。Contribution Map 致力于解决下面几个潜在 contributor 经常会碰到的问题：

- 如何快速找到自己感兴趣的模块？Contribution Map 对各个项目的各个模块进行了描述，并且提供了对应模块的源码路径，这样 contributor 就能很方便地找到自己感兴趣的模块。
- 不知道自己能做什么贡献？Contribution Map 将各个模块的任务会不定期地更新到 What I Can Contribute 那一列，contributor 能快速找到对应模块的任务。
- 暂时不具备贡献所需要的技能，该学习哪些资料？Contribution Map 的 Learning Materials 列列出了如果 contributor 不具备相关的技能可以通过学习哪些资料来提升自己的技能，以达到能够贡献的水平。
- 如何在特定的领域进行进阶提升？可以参考 Contribution Map 的 SIG (Special Interest Group) 一节，加入特定的兴趣小组，和兴趣相投的一批人来一起在某个领域进行提升。每个 SIG 会有对应的 tech-lead 和 mentors 来维护和更新每个 SIG 的任务，并帮助大家进行提升。大家可以在 TiDB community 和 TiKV community 来查看各个 SIG 的详细信息。

### 2.8.2 Contribution Map 解读

下面我们来看下 Contribution Map 的大概结构。开头部分是各个项目的索引，包含了各个项目的简单说明，比如 TiDB 是一个开源的兼容 MySQL 协议的分布式 HTAP 数据库，TiKV 是分布式的事务 KV 数据库等等。

## A map that guides what and how contributors can contribute

### Table of Contents

- [TiDB is an open-source distributed HTAP database compatible with the MySQL protocol](#)
- [TiKV, distributed transactional key-value database](#)
- [PD Placement driver for TiKV](#)
- [TiKV Clients](#)
- [Libraries depended by TiKV](#)
- [Ecosystem Tools: DM Data Migration Platform](#)
- [Ecosystem Tools - Binlog : A tool used to collect and merge tidb's binlog for real-time data backup and synchronization](#)
- [Ecosystem Tools - Lightning: A high-speed data import tool for TiDB](#)
- [Ecosystem Tools - BR: A command-line tool for distributed backup and restoration of the TiDB cluster data](#)
- [TiDB on K8S/Docker : Creates and manages TiDB clusters running in Kubernetes](#)
- [Deployment Tools - tidb-ansible: A tool to capture data change of TiDB cluster](#)
- [Chaos-Mesh: A Chaos Engineering Platform for Kubernetes](#)
- [Documentation](#)
- [AskTUG\(CN\)](#)
- [PingCAP University\(CN\)](#)
- [SIG - Special Interest Group](#)

接下来就是每个项目各个模块更详细的介绍，主要分为 6 部分：模块描述；代码位置；如果想要在该模块进行贡献需要具备哪些技能；相关的学习资料；该模块的任务列表；贡献指南。下图展示的是 TiKV 项目各个模块的描述代码位置等：

## TiKV, distributed transactional key-value database

Module	Description	Code Directory	Required Skills	Learning Materials	What I can Contribute	Contributing Tutorials
Util	Utilities like thread-pool, logger, encoding and decoding, etc.	Utilities, Pipeline batch system	Rust	Rust book, Practical networked applications in Rust, Protocol buffers, TiKV source code reading: <a href="#">service layer(CN)</a> , gRPC concepts	Issues want help	Land your first Rust PR in TiKV, Became TiKV Contributor in 30 minutes(CN)
Network	Network layer	Server	Rust, Protobuf, gRPC	Ditto	Ditto	Ditto
Raw KV API		API entrance, Storage struct	Rust	Ditto	Ditto	Ditto
Transaction KV API		API entrance, Implementation of <a href="#">Transaction</a> , GC worker, Pessimistic transaction	Rust, 2PC, Percolator transaction model	Two Phase Commit, Percolator <a href="#">paper</a> , TiKV source code reading : <a href="#">storage(CN)</a> , TiKV source code reading : <a href="#">distributed transaction(CN)</a> , TiKV source code reading <a href="#">MVCC read(CN)</a>		
Multi-raft		Raftstore	Rust, Raft	Rust book, Practical networked applications in Rust, Raft <a href="#">paper</a> , Raft implementation in <a href="#">etcd</a>		
Engine		Engine traits, Engine rocks	Rust, RocksDB		Engine abstraction	
Coprocessor		TiDB query, TiDB query codegen, TiDB query datatype	Rust	TiKV source code reading : <a href="#">Coprocessor Overview(CN)</a> , TiKV source code reading : <a href="#">Coprocessor Executor</a>	Coprocessor Issues	
Backup		Backup source code				

Contribution Map 仍然在持续完善中，会将越来越多的相关内容包含进来，希望能帮助到更多的 contributors。



