# **新一代数据库TiDB从入门到实践**

# 学习目标

1.了解数据库发展史

2.了解TiDB数据库的特点和应用场景

3.掌握TiDB数据库的安装部署

4.掌握TiDB数据库的SQL操作

5.掌握TiDB数据库与Spark整合

6.掌握TiDB数据库数据迁移

7.了解TiDB数据库核心原理

8.了解TiDB数据库源码

9.了解TiDB数据库实践应用

# TiDB引入

## 数据库技术发展简史

数据库技术产生于20世纪60年代末70年代初，其主要主要研究如何存储，使用和管理数据。随着计算机硬件和软件的发展，数据库技术也不断地发展。数据库技术在理论研究和系统开发上都取得了辉煌的成就。

* 从数据管理的角度看，数据库技术到目前共经历了如下三个阶段：

1. 人工管理阶段-数据量小独立,用户直接管理
2. 文件系统阶段-使用文件存取数据,冗余度高,管理维护难
3. 数据库系统阶段-专门的数据库软件系统管理数据,高效方便,易于共享维护

* 按照数据模型发展的主线，数据库技术的形成过程和发展可分为如下三个阶段：

1. 层次和网状数据库管理系统-可以理解为使用指针来表示数据之间的联系
2. 关系数据库管理系统(RDBMS)-可以理解为理解为使用二维表来表示维护数据间的关系
3. 新一代数据库技术的研究和发展-针对关系型数据库存在数据模型，性能，扩展性，伸缩性等方面的缺点，出现了：

ORDBMS:面向对象数据库技术。如：PostGreSQL

NoSQL:非结构化数据库技术。如

1):键值存储数据库：Redis

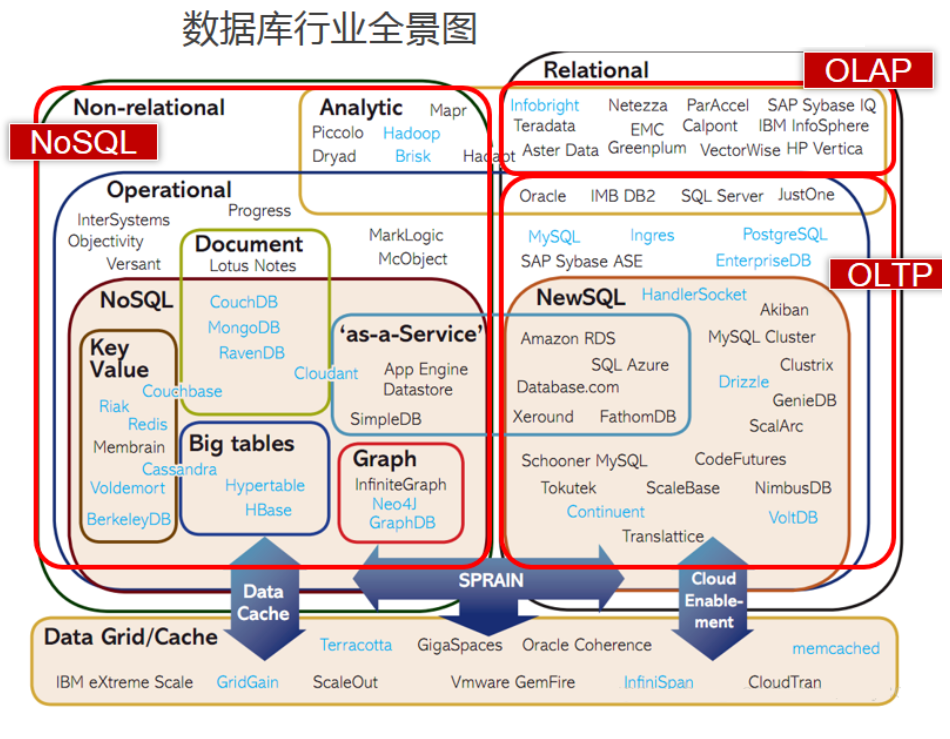
2):列式储数数据库：HBase

3):文档型数据库：MongoDB

4):图形数据库：Neo4J

NewSQL:这类数据库不仅具有NoSQL对海量数据的存储管理能力，还保持了传统数据库支持ACID和SQL等特性。如：TiDB

## 从MySQL到TiDB



如今的数据库种类繁多，RDBMS（关系型数据库）、NoSQL（Not Only SQL）、NewSQL，在数据库领域均有一席之地，可谓百家争鸣之势。那么我们为什么要学习使用TiDB呢？接下来就从我们最熟悉的MySQL的使用说起！

### 场景引入

假设现在有一个高速发展的互联网公司,核心业务库MySQL的数据量已经近亿行,且还在不断增长中,公司对于数据资产较为重视,所有数据要求多副本保存至少5年,且除了有对历史数据进行统计分析的离线报表业务外,还有一些针对用户数据实时查询的需求,如用户历史订单实时查询

### 问题分析

**1.MySQL能否满足上述场景需求?**

根据以往的MySQL使用经验，MySQL单表在 5000 万行以内时，性能较好，单表超过5000万行后，数据库性能、可维护性都会极剧下降。当然这时候可以做MySQL分库分表,如使用Mycat或Sharding-jdbc

**2.分库分表的能否解决问题?**

**分库分表的优点非常明显，如：**

将大表拆分成小表，单表数据量控制在 5000 万行以内，使 MySQL 性能稳定可控。

将单张大表拆分成小表后，能水平扩展，通过部署到多台服务器，提升整个集群的 QPS、TPS、Latency 等数据库服务指标。

**但是，此方案的缺点也非常明显：**

分表跨实例后，产生分布式事务管理难题，一旦数据库服务器宕机，有事务不一致风险。

分表后，对 SQL 语句有一定限制，对业务方功能需求大打折扣。尤其对于实时报表统计类需求，限制非常之大。事实上，报表大多都是提供给高层领导使用的，其重要性不言而喻。

分表后，需要维护的对象呈指数增长（MySQL实例数、需要执行的 SQL 变更数量等）。

### 问题解决

基于以上核心痛点，我们需要探索新的数据库技术方案来应对业务爆发式增长所带来的挑战，为业务提供更好的数据库服务支撑。

调研市场上的各大数据库,我们可以考虑选用NewSQL技术来解决,因为NewSQL技术有如下显著特点:

- 无限水平扩展能力

- 分布式强一致性，确保数据 100% 安全

- 完整的分布式事务处理能力与 ACID 特性

而TiDB数据库 GitHub的活跃度及社区贡献者方面都可以算得上是国际化的开源项目,是NewSQL技术中的代表性产品,所以我们可以选择使用TiDB数据库!

### 总结

传统关系型数据库历史比较久，目前RDBMS的代表为Oracle、MySQL、PostgreSQL，在数据库领域也是“辈份”比较高的，其广泛应用在各行各业，RDBMS大多为本地存储或共享存储。

但是此类数据库存在着一些问题，如自身容量的限制。随着业务量不断增加，容量渐渐成为瓶颈，此时DBA会通过多次的库表sharding，以此来缓解容量问题。大量的分库分表，不仅耗费了大量人力，还使得业务访问数据库的路由逻辑变得复杂。除此之外，RDBMS伸缩性比较差，通常集群扩容缩容成本较高，且不满足分布式的事务。

NoSQL类数据库的代表为Hbase、Redis、MongoDB、Cassandra等，这类数据库解决了 RDBMS伸缩性差的问题，集群容量扩容变得方便很多，但是由于存储方式为多个KV存储，所以对SQL的兼容性就大打折扣。对于NoSQL类数据库来说，只能满足部分分布式事务的特点。

NewSQL领域的代表是Google的spanner和F1，其号称可以实现全球数据中心容灾，且完全满足分布式事务的ACID，但是只能在Google云上使用。

TiDB诞生在大背景下，也弥补了国内在NewSQL领域中的空缺。TiDB自2015年5月写下第一行代码以来，至今已发布大小版本几十次，版本迭代十分迅速

## TiDB概述

### 官网

<https://pingcap.com/index.html>

TiDB可以理解为是MySQL的加强版/分布式MySQL/MySQLPlus



### 简介

TiDB 是 PingCAP 公司设计的开源分布式 HTAP (Hybrid Transactional and Analytical Processing) 数据库，结合了传统的 RDBMS 和 NoSQL 的最佳特性。TiDB 兼容 MySQL，支持无限的水平扩展，具备强一致性和高可用性。TiDB 的目标是为 OLTP (Online Transactional Processing) 和 OLAP (Online Analytical Processing) 场景提供一站式的解决方案。

TiDB数据库具备「分布式强一致性事务、在线弹性水平扩展、故障自恢复的高可用、跨数据中心多活」等核心特性，是大数据时代理想的数据库集群和云数据库解决方案。目前，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、电信、新零售、政府等多个行业，包括美国、欧洲、日本、东南亚等海外用户。

TiDB 的设计目标是 100% 的 OLTP 场景和 80% 的 OLAP 场景，更复杂的 OLAP 分析可以通过 TiSpark 项目来完成。

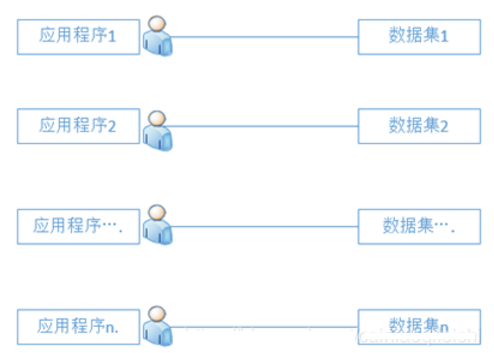
TiDB 对业务没有任何侵入性，能优雅的替换传统的数据库中间件、数据库分库分表等 Sharding 方案。同时它也让开发运维人员不用关注数据库 Scale 的细节问题，专注于业务开发，极大的提升研发的生产力。

## 扩展阅读

### 数据管理技术发展阶段

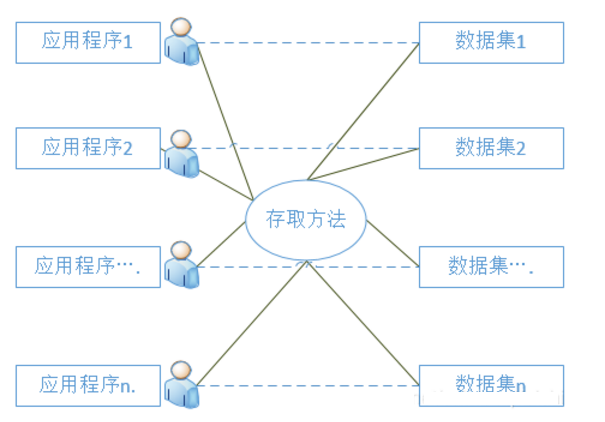
1. 人工管理阶段

20世纪50年代以前，计算机主要用于数值计算.从当时的硬件看，外存只有纸带，卡片，磁带，没有直接存取设备；从软件看（实际上，当时还未形成软件的整体概念），没有操作系统以及管理数据的软件；从数据看，数据量小，数据无结构，由用户直接管理，且数据间缺乏逻辑组织，数据依赖于特定的应用程序，缺乏独立性.



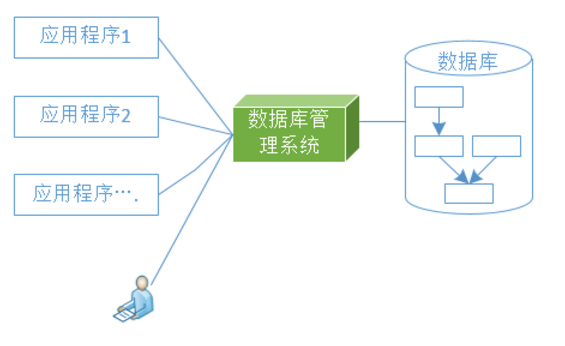
1. 文件系统阶段

50年代后期到60年代中期，出现了磁鼓，磁盘等数据存储设备.新的数据处理系统迅速发展起来.这种数据处理系统是把计算机中的数据组织成相互独立的数据文件，系统可以按照文件的名称对其进行访问，对文件中的记录进行存取，并可以实现对文件的修改，插入和删除，这就是文件系统.文件系统实现了记录内的结构化,即给出了记录内各种数据间的关系.但是，文件从整体来看却是无结构的.其数据面向特定的应用程序，因此数据共享性差，且冗余度大，管理和维护的代价也很大.



1. 数据库系统阶段

60年代后期，出现了数据库这样的数据管理技术.数据库的特点是数据不再只针对某一特定应用，而是面向全组织，具有整体的结构性，共享性高，冗余度小，具有一定的程序与数据间的独立性，并且实现了对数据进行统一的控制.



### 数据库模型发展阶段

1. 第一代数据库系统  层次和网状数据库管理系统

    层次和网状数据库的代表产品是IBM公司在1969年研制出的层次模型数据库管理系统。层次数据库是数据库系统的先驱，而网状数据库则是数据库概念、方法、技术的奠基。

1. 第二代数据库系统  关系数据库管理系统(RDBMS)

    1970年，IBM公司的研究员E.F.Codd在题为《大型共享数据库数据的关系模型》的论文中提出了数据库的关系模型，为关系数据库技术奠定了理论基础。到了80年代，几乎所有新开发的数据库系统都是关系型的。真正使得关系数据库技术实用化的关键人物是James Gray。Gray在解决如何保障数据的完整性、安全性、并发性以及数据库的故障恢复能力等重大技术问题方面发挥了关键作用。关系数据库系统的出现，促进了数据库的小型化和普及化，使得在微型机上配置数据库系统成为可能。

1. 新一代数据库技术的研究和发展

    目前已从多方面发展了现行的数据库系统技术。我们可以从数据模型、新技术内容、应用领域三个方面概括新一代数据库系统的发展。

    (1) 面向对象的方法和技术对数据库发展的影响最为深远

    80年代，面向对象的方法和技术的出现，对计算机各个领域，包括程序设计语言、软件工程、信息系统设计以及计算机硬件设备等都产生了深远的影响，也给面临新挑战的数据库技术带来了新的机遇和希望。数据库研究人员借鉴和吸收了面向对象的方法和技术，提出了面向对象的数据库模型(简称对象模型)。当前有许多研究是建立在数据库已有的成果和技术上的，针对不同的应用，对传统的DBMS，主要是RDBMS进行不同层次上的扩充，例如建立对象关系(OR)模型和建立对象关系数据库(ORDB)。

    (2) 数据库技术与多学科技术的有机结合

    数据库技术与多学科技术的有机结合是当前数据库发展的重要特征。计算机领域中其他新兴技术的发展对数据库技术产生了重大影响。传统的数据库技术和其他计算机技术的结合、互相渗透，使数据库中新的技术内容层出不穷。数据库的许多概念、技术内容、应用领域，甚至某些原理都有了重大的发展和变化。建立和实现了一系列新型的数据库，如分布式数据库、并行数据库、演绎数据库、知识库、多媒体库、移动数据库等，它们共同构成了数据库大家族。

    (3) 面向专门应用领域的数据库技术的研究

为了适应数据库应用多元化的要求，在传统数据库基础上，结合各个专门应用领域的特点，研究适合该应用领域的数据库技术，如工程数据库、统计数据库、科学数据库、空间数据库、地理数据库、Web数据库等，这是当前数据库技术发展的又一重要特征。同时，数据库系统结构也由主机/终端的集中式结构发展到网络环境的分布式结构，随后又发展成两层、三层或多层客户/服务器结构以及Internet环境下的浏览器/服务器和移动环境下的动态结构。多种数据库结构满足了不同应用的需求，适应了不同的应用环境。

### SQL，NoSQL，NewSQL

1. 关系型数据库(RDBMS,即SQL数据库)

商业软件： Oracle，DB2

开源软件：MySQL，PostgreSQL

单机版本已经很难满足海量数据的需求

1. NoSQL

NoSQL = Not Only SQL，意即“不仅仅是SQL，提倡运用非关系型的数据存储

普遍选择牺牲掉复杂 SQL 的支持及 ACID 事务换取弹性扩展能力

通常不保证强一致性的(支持最终一致)

* **主要分类**
* 键值（Key-Value）数据库：如 MemcacheDB，Redis
* 文档存储：如 MongoDB
* 列存储：方便存储结构化和半结构化数据，并做数据压缩，对某几列的查询有非常大的IO优势： 如 HBase，Cassandra
* 图数据库：存储图关系(注意：不是图片)。如 Neo4J

1. NewSQL

针对OLTP的读写，提供与NOSQL相同的可扩展性和性能，同时能支持满足ACID特性的事务，即保持NoSQL的高可扩展和高性能，并且保持关系模型

* **为什么需要NewSQL**

NoSQL 不能完全取代 RDBMS

单机RDBMS 无法满足性能需求

使用“单机RDBMS + 中间件”方式，在中间件层很难解决分布式事务、高可用问题

* **NewSQL设计架构**

可以基于全新的数据库平台，也可以基于现有的SQL引擎优化。

无共享存储(MPP架构)是比较常见的架构

基于多副本实现高可用和容灾

分布式查询

数据Sharding机制

通过2PC，Paxos/Raft等协议实现数据一致

* **代表产品**

Google Spanner

OceanBase

TiDB

### OLTP和OLAP

* OLTP

强调支持短时间内大量并发的事务操作（增删改查）能力，每个操作涉及的数据量都很小（比如几十到几百字节）

强调事务的强一致性（想想银行转账交易，容不得差错）

举例：“双十一”期间，可能有几十万用户在同一秒内下订单。后台数据库要能够并发的、以近乎实时的速度处理这些订单请求（如果下了订单，十几分钟还没有反应，用户肯定要骂人了）

* OLAP

偏向于复杂的只读查询，读取海量数据进行分析计算，查询时间往往很长

举例：“双十一”结束，淘宝的运营人员对订单进行分析挖掘，找出一些市场规律等等。 这种分析可能需要读取所有的历史订单进行计算，耗时几十秒甚至几十分钟都有可能。

代表产品：

Greenplum

TeraData

阿里 AnalyticDB

### TiDB怎么诞生的?

著名的开源分布式缓存服务 Codis 的作者，PingCAP联合创始人& CTO ，资深 infrastructure 工程师的黄东旭，擅长分布式存储系统的设计与实现，开源狂热分子的技术大神级别人物。即使在互联网如此繁荣的今天，在数据库这片边界模糊且不确定地带，他还在努力寻找确定性的实践方向。

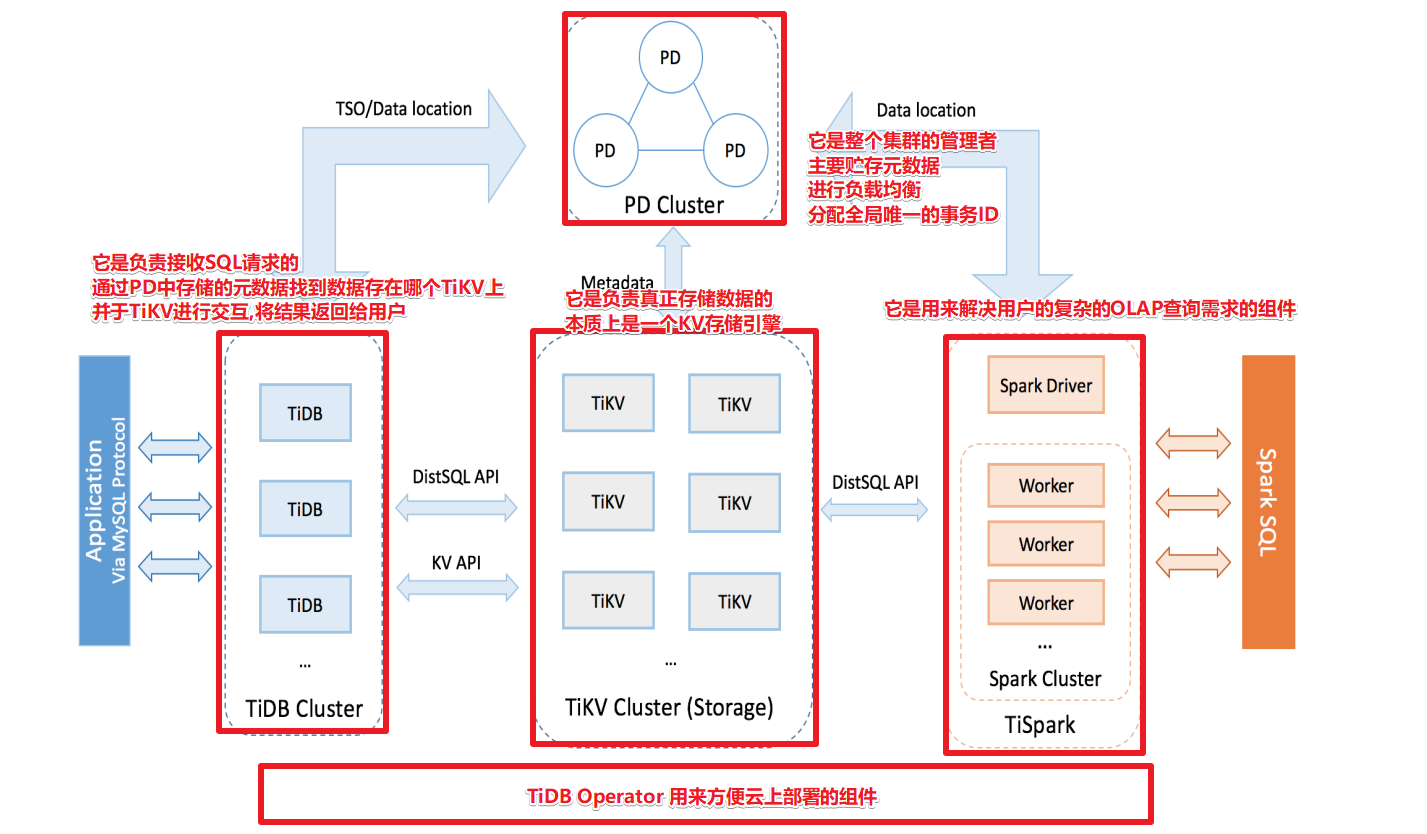
2012 年底，他看到 Google 发布的两篇论文，如同棱镜般，折射出他自己内心微烁的光彩。这两篇论文描述了 Google 内部使用的一个海量关系型数据F1/Spanner，解决了关系型数据库、弹性扩展以及全球分布的问题，并在生产中大规模使用。“如果这个能实现，对数据存储领域来说将是颠覆性的”，黄东旭为完美方案的出现而兴奋， PingCAP 的 TiDB 在此基础上诞生了。

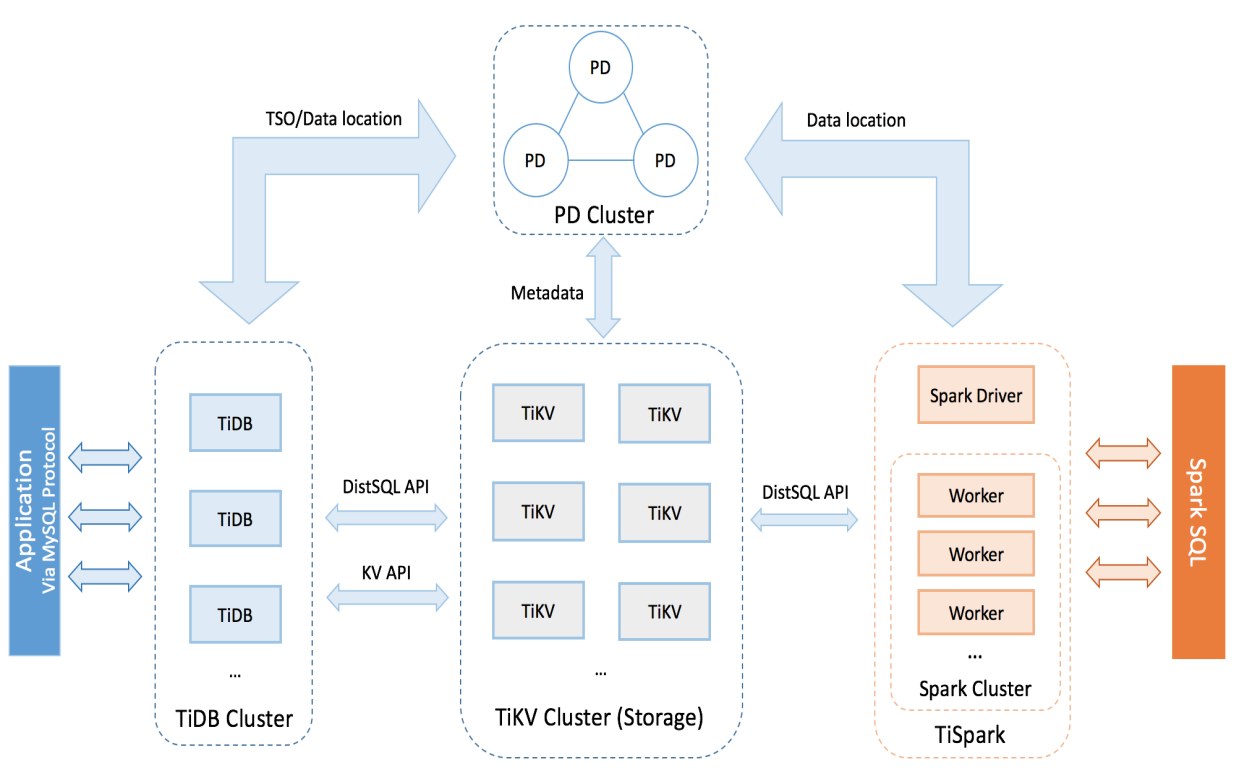
# TiDB架构特性

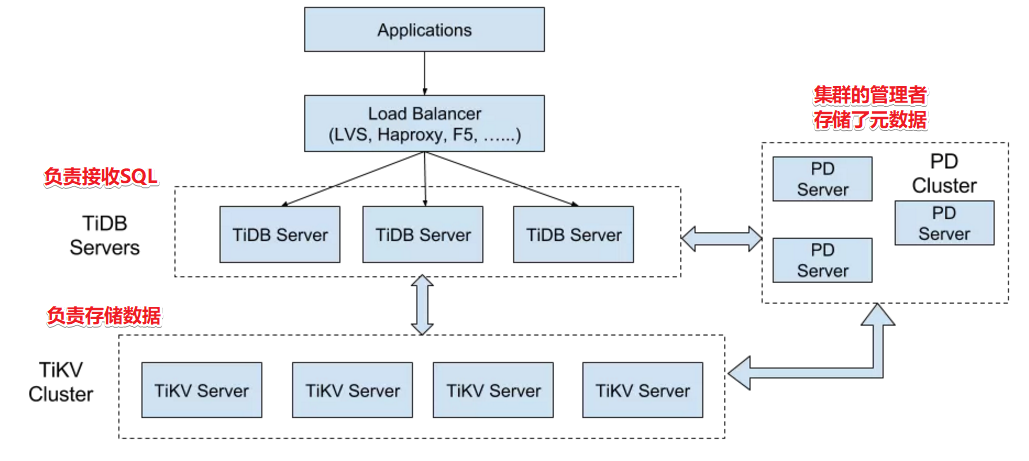
## TiDB 整体架构

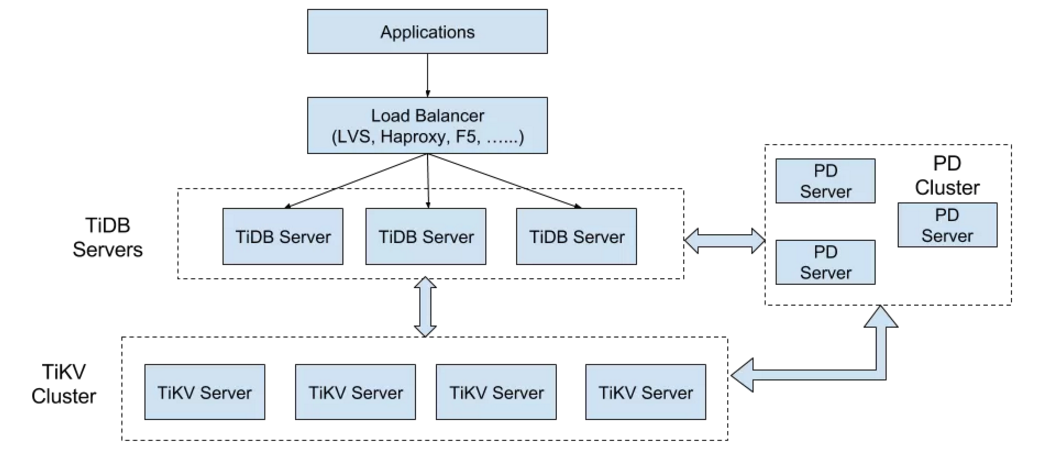
TiDB 集群主要包括三个核心组件：TiDB Server，PD Server 和 TiKV Server。此外，还有用于解决用户复杂 OLAP 需求的 TiSpark 组件和简化云上部署管理的 TiDB Operator 组件。

* 架构图解









### TiDB Server

TiDB Server 负责接收 SQL 请求，处理 SQL 相关的逻辑，并通过 PD 找到存储计算所需数据的 TiKV 地址，与 TiKV 交互获取数据，最终返回结果。TiDB Server 是无状态的，其本身并不存储数据，只负责计算，可以无限水平扩展，可以通过负载均衡组件（如LVS、HAProxy 或 F5）对外提供统一的接入地址。

### PD Server

Placement Driver (简称 PD) 是整个集群的管理模块，其主要工作有三个：一是存储集群的元信息（某个 Key 存储在哪个 TiKV 节点）；二是对 TiKV 集群进行调度和负载均衡（如数据的迁移、Raft group leader 的迁移等）；三是分配全局唯一且递增的事务 ID。

PD 通过 Raft 协议保证数据的安全性。Raft 的 leader server 负责处理所有操作，其余的 PD server 仅用于保证高可用。建议部署奇数个 PD 节点。

### TiKV Server

TiKV Server 负责存储数据，从外部看 TiKV 是一个分布式的提供事务的 Key-Value 存储引擎。存储数据的基本单位是 Region，每个 Region 负责存储一个 Key Range（从 StartKey 到 EndKey 的左闭右开区间）的数据，每个 TiKV 节点会负责多个 Region。TiKV 使用 Raft 协议做复制，保持数据的一致性和容灾。副本以 Region 为单位进行管理，不同节点上的多个 Region 构成一个 Raft Group，互为副本。数据在多个 TiKV 之间的负载均衡由 PD 调度，这里也是以 Region 为单位进行调度。

### TiSpark

TiSpark 作为 TiDB 中解决用户复杂 OLAP 需求的主要组件，将 Spark SQL 直接运行在 TiDB 存储层上，同时融合 TiKV 分布式集群的优势，并融入大数据社区生态。至此，TiDB 可以通过一套系统，同时支持 OLTP 与 OLAP，免除用户数据同步的烦恼。

### TiDB Operator

TiDB Operator 提供在主流云基础设施（Kubernetes）上部署管理 TiDB 集群的能力。它结合云原生社区的容器编排最佳实践与 TiDB 的专业运维知识，集成一键部署、多集群混部、自动运维、故障自愈等能力，极大地降低了用户使用和管理 TiDB 的门槛与成本。

## TiDB 核心特性

TiDB 具备如下众多特性,其中两大核心特性为：水平扩展与高可用

1. 高度兼容 MySQL

大多数情况下，无需修改代码即可从 MySQL 轻松迁移至 TiDB，分库分表后的 MySQL 集群亦可通过 TiDB 工具进行实时迁移。

对于用户使用的时候，可以透明地从MySQL切换到TiDB 中，只是“新MySQL”的后端是存储“无限的”，不再受制于Local的磁盘容量。在运维使用时也可以将TiDB当做一个从库挂到MySQL主从架构中。

1. 分布式事务

TiDB 100% 支持标准的 ACID 事务。

1. 一站式 HTAP 解决方案

HTAP: Hybrid Transactional/Analytical Processing

TiDB 作为典型的 OLTP 行存数据库，同时兼具强大的 OLAP 性能，配合 TiSpark，可提供一站式 HTAP 解决方案，一份存储同时处理 OLTP & OLAP，无需传统繁琐的 ETL 过程。

1. 云原生 SQL 数据库

TiDB 是为云而设计的数据库，支持公有云、私有云和混合云，配合 TiDB Operator 项目 可实现自动化运维，使部署、配置和维护变得十分简单。

1. 水平弹性扩展

通过简单地增加新节点即可实现 TiDB 的水平扩展，按需扩展吞吐或存储，轻松应对高并发、海量数据场景。

1. 真正金融级高可用

相比于传统主从 (M-S) 复制方案，基于 Raft 的多数派选举协议可以提供金融级的 100% 数据强一致性保证，且在不丢失大多数副本的前提下，可以实现故障的自动恢复 (auto-failover)，无需人工介入。

### 水平扩展

无限水平扩展是 TiDB 的一大特点，这里说的水平扩展包括两方面：计算能力(TiDB)和存储能力(TiKV)。

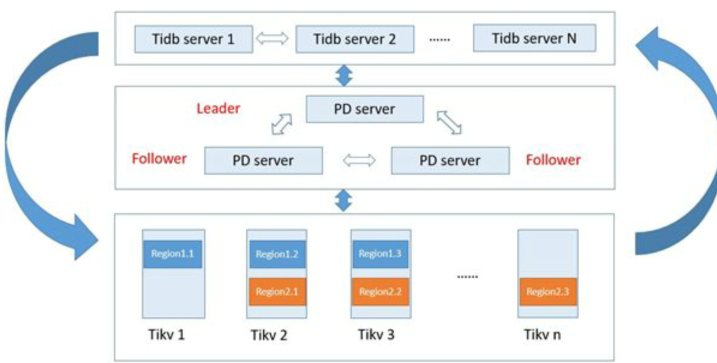
TiDB Server 负责处理 SQL 请求，随着业务的增长，可以简单的添加 TiDB Server 节点，提高整体的处理能力，提供更高的吞吐。

TiKV 负责存储数据，随着数据量的增长，可以部署更多的 TiKV Server 节点解决数据 Scale 的问题。

PD 会在 TiKV 节点之间以 Region 为单位做调度，将部分数据迁移到新加的节点上。

所以在业务的早期，可以只部署少量的服务实例（推荐至少部署 3 个 TiKV， 3 个 PD，2 个 TiDB），随着业务量的增长，按照需求添加 TiKV 或者 TiDB 实例。

### 高可用



高可用是 TiDB 的另一大特点，TiDB/TiKV/PD 这三个组件都能容忍部分实例失效，不影响整个集群的可用性。下面分别说明这三个组件的可用性、单个实例失效后的后果以及如何恢复。

1. TiDB

TiDB 是无状态的，推荐至少部署两个实例，前端通过负载均衡组件对外提供服务。当单个实例失效时，会影响正在这个实例上进行的 Session，从应用的角度看，会出现单次请求失败的情况，重新连接后即可继续获得服务。单个实例失效后，可以重启这个实例或者部署一个新的实例。

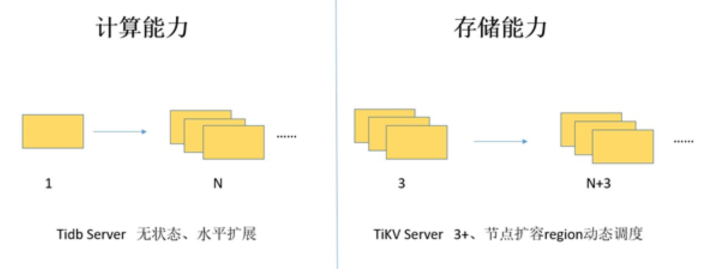
1. PD

PD 是一个集群，通过 Raft 协议保持数据的一致性，单个实例失效时，如果这个实例不是 Raft 的 leader，那么服务完全不受影响；如果这个实例是 Raft 的 leader，会重新选出新的 Raft leader，自动恢复服务。PD 在选举的过程中无法对外提供服务，这个时间大约是3秒钟。推荐至少部署三个 PD 实例，单个实例失效后，重启这个实例或者添加新的实例。

1. TiKV

TiKV 是一个集群，通过 Raft 协议保持数据的一致性（副本数量可配置，默认保存三副本），并通过 PD 做负载均衡调度。单个节点失效时，会影响这个节点上存储的所有 Region。对于 Region 中的 Leader 节点，会中断服务，等待重新选举；对于 Region 中的 Follower 节点，不会影响服务。当某个 TiKV 节点失效，并且在一段时间内（默认 30 分钟）无法恢复，PD 会将其上的数据迁移到其他的 TiKV 节点上。

## TiDB 存储和计算能力



### 存储能力-TiKV-LSM

TiKV Server通常是3+的，TiDB每份数据缺省为3副本，这一点与HDFS有些相似，但是通过Raft协议进行数据复制，TiKV Server上的数据的是以Region为单位进行，由PD Server集群进行统一调度，类似HBASE的Region调度。

TiKV集群存储的数据格式是KV的，在TiDB中，并不是将数据直接存储在 HDD/SSD中，而是通过RocksDB实现了TB级别的本地化存储方案，着重提的一点是：RocksDB和HBASE一样，都是通过 LSM树作为存储方案，避免了B+树叶子节点膨胀带来的大量随机读写。从何提升了整体的吞吐量。

### 计算能力-TiDB Server

TiDB Server本身是无状态的，意味着当计算能力成为瓶颈的时候，可以直接扩容机器，对用户是透明的。理论上TiDB Server的数量并没有上限限制。

## 总结

TiDB作为新一代的NewSQL数据库，在数据库领域已经逐渐站稳脚跟，结合了Etcd/MySQL/HDFS/HBase/Spark等技术的突出特点，随着TiDB的大面积推广，会逐渐弱化 OLTP/OLAP的界限，并简化目前冗杂的ETL流程，引起新一轮的技术浪潮。

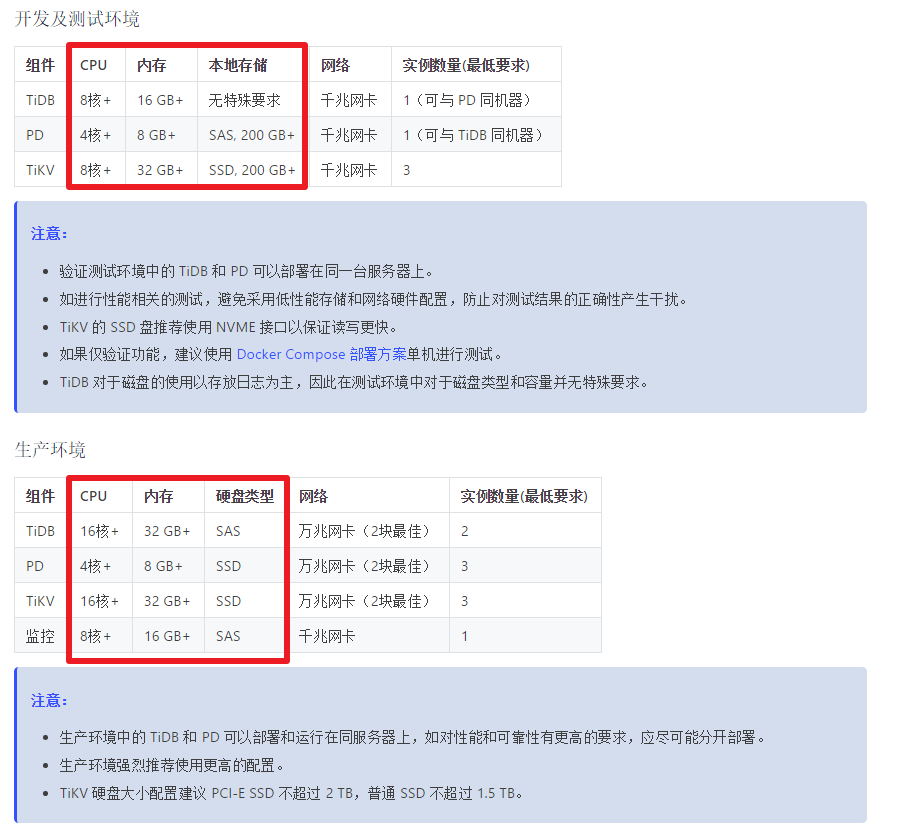
一言以蔽之，TiDB，前景可待，未来可期。

# TiDB安装部署

<https://pingcap.com/docs-cn/stable/overview/#部署方式>

## 部署方式选择





<https://gitee.com/pingcap/tidb?_from=gitee_search>



通过查阅官方文档和网友踩坑经验,我们学习测试时就就直接使用官网推荐的Docker Compose方式快速构建TiDB集群即可(或者使用单机版)

## TiDB-Local单机版

部署TiDB就是摆在我们面前的第一节课，其实有很多的部署方式，限于环境，我先开始琢磨单机版如何部署。

在Centos 6的版本中如果要部署，这个难度还是比较大的，而且会有很多未知的坑，根据官方的建议，是需要在Centos 7以上的版本中，否则glibc的版本问题会很快碰到。

我们安装一套Centos7，采用快速的单机部署的方式来尝鲜。

1. 下载安装包

wget http://download.pingcap.org/tidb-latest-linux-amd64.tar.gz

1. 解压文件

tar -zxvf tidb-latest-linux-amd64.tar.gz

cd tidb-latest-linux-amd64

1. 启动

启动PD

./bin/pd-server --data-dir=pd --log-file=pd.log &

启动tikv

./bin/tikv-server --pd="127.0.0.1:2379" --data-dir=tikv --log-file=tikv.log &

启动tidb-server

./bin/tidb-server --store=tikv --path="127.0.0.1:2379" --log-file=tidb.log &

1. 登录：

mysql -h 127.0.0.1 -P 4000 -u root

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 3

Server version: 5.7.25-TiDB-v4.0.0-beta-185-g065deca8f TiDB Server (Apache License 2.0), MySQL 5.7 compatible

Copyright (c) 2000, 2020, 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.

1. 体验

mysql> \s

--------------

mysql Ver 14.14 Distrib 5.7.29, for Linux (x86\_64) using EditLine wrapper

Connection id: 3

Current database:

Current user: root@127.0.0.1

SSL: Not in use

Current pager: stdout

Using outfile: ''

Using delimiter: ;

Server version: 5.7.25-TiDB-v4.0.0-beta-185-g065deca8f TiDB Server (Apache License 2.0), MySQL 5.7 compatible

Protocol version: 10

Connection: 127.0.0.1 via TCP/IP

Server characterset: utf8mb4

Db characterset: utf8mb4

Client characterset: utf8

Conn. characterset: utf8

TCP port: 4000

--------------

mysql> show databases;

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

| Database |

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

| INFORMATION\_SCHEMA |

| INSPECTION\_SCHEMA |

| METRIC\_SCHEMA |

| PERFORMANCE\_SCHEMA |

| mysql |

| test |

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

6 rows in set (0.00 sec)

mysql> select tidb\_version();

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

| tidb\_version() |

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

| Release Version: v4.0.0-beta-185-g065deca8f

Git Commit Hash: 065deca8f00cd61b357ba0f1a61119d55111f758

Git Branch: master

UTC Build Time: 2020-02-20 12:18:56

GoVersion: go1.13

Race Enabled: false

TiKV Min Version: v3.0.0-60965b006877ca7234adaced7890d7b029ed1306

Check Table Before Drop: false |

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

1 row in set (0.00 sec)

mysql> select version();

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

| version() |

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

| 5.7.25-TiDB-v4.0.0-beta-185-g065deca8f |

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

1 row in set (0.00 sec)

mysql> select user,host from mysql.user;

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

| user | host |

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

| root | % |

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

1 row in set (0.00 sec)

mysql> exit

## TiDB-Docker集群版

1.准备环境

确保你的机器上已安装：

Docker（17.06.0 及以上版本）

Docker Compose

Git

2.快速部署

下载 tidb-docker-compose

git clone <https://github.com/pingcap/tidb-docker-compose.git>

3.创建并启动集群

获取最新 Docker 镜像：

cd tidb-docker-compose && docker-compose pull && docker-compose up -d

注意:

得先启动Docker

sudo systemctl start docker

再执行上面的docker-compose命令

4.访问集群

mysql -h 127.0.0.1 -P 4000 -u root

[root@node01 tidb-docker-compose]# mysql -h 127.0.0.1 -P 4000 -u root

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 1

Server version: 5.7.25-TiDB-v4.0.0-beta-177-g5a00c80e4 TiDB Server (Apache License 2.0), MySQL 5.7 compatible

Copyright (c) 2000, 2020, 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> show databases;

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

| Database |

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

| INFORMATION\_SCHEMA |

| INSPECTION\_SCHEMA |

| METRIC\_SCHEMA |

| PERFORMANCE\_SCHEMA |

| mysql |

| test |

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

6 rows in set (0.00 sec)

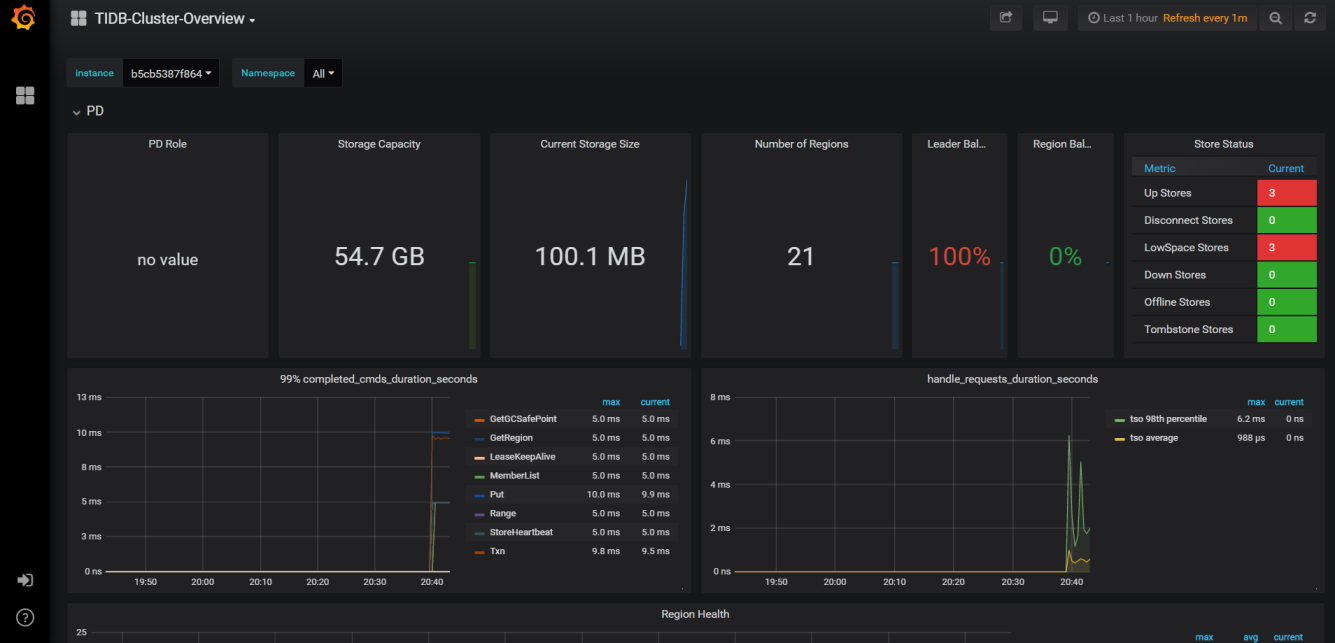
mysql> exit

Bye

[root@node01 tidb-docker-compose]#

5.访问集群 Grafana 监控页面：

<http://node01:3000/>



默认用户名和密码均为 admin。

6.集群数据可视化：

<http://node01:8010/>

## 补充：

### CentOS安装MySQL

1.下载并安装MySQL官方的 Yum Repository

wget -i -c http://dev.mysql.com/get/mysql57-community-release-el7-10.noarch.rpm

2.yum安装

yum -y install mysql57-community-release-el7-10.noarch.rpm

3.安装MySQL服务器。

yum -y install mysql-community-server

4.启动MySQL

systemctl start mysqld.service

5.查看MySQL运行状态

systemctl status mysqld.service

6.设置开机启动

systemctl enable mysqld

7.找出输入初始密码：

grep "password" /var/log/mysqld.log

8.进入数据库：

mysql -uroot -p

输入初始密码

9.修改密码

ALTER USER 'root'@'localhost' IDENTIFIED BY 'Root@1234';

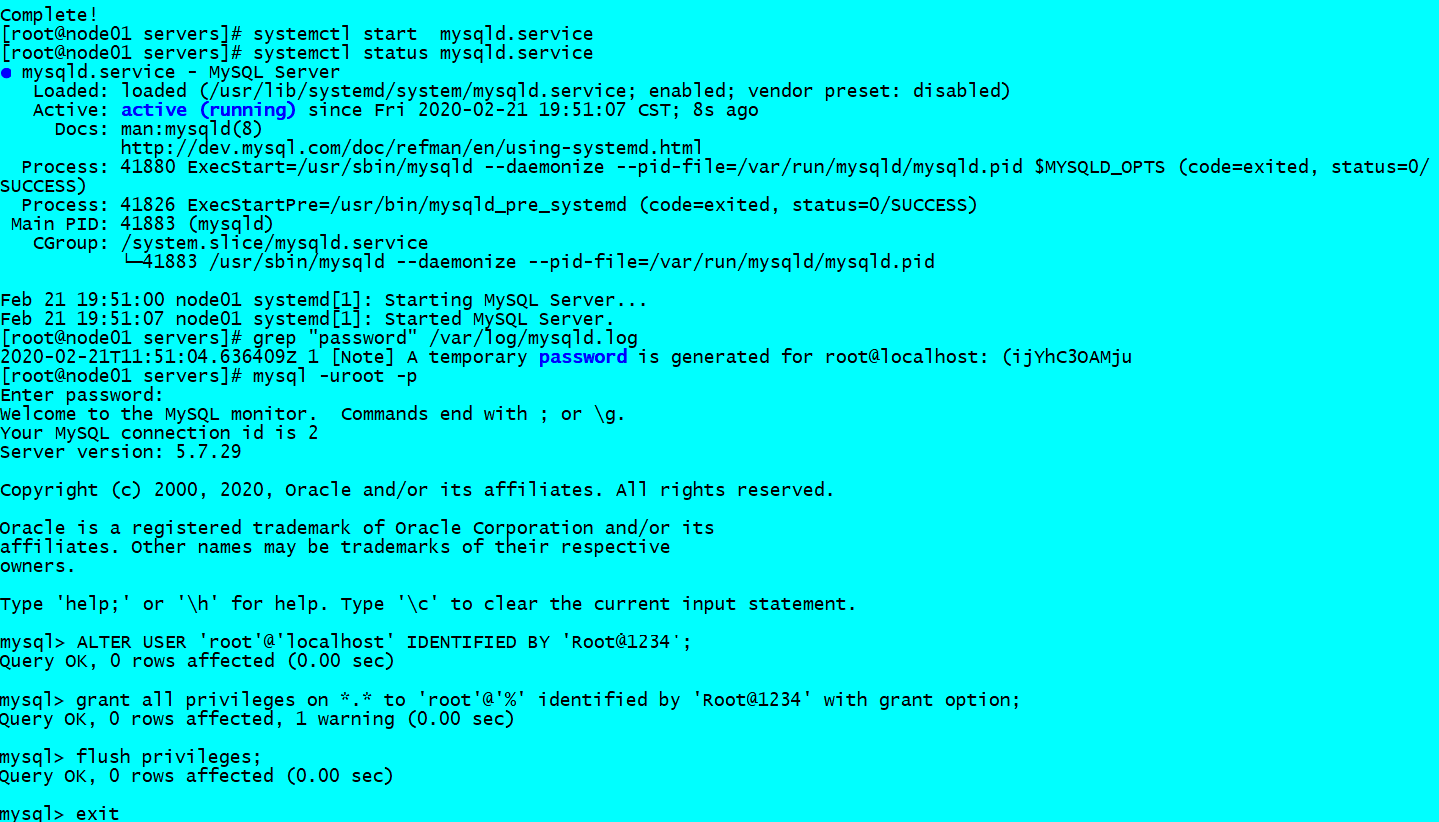
注意:密码设置必须要大小写字母数字和特殊符号（,/';:等）,不然不能配置成功

10.开启mysql的远程访问

grant all privileges on \*.\* to 'root'@'%' identified by 'Root@1234' with grant option;

flush privileges;

exit



### CentOS安装Git

1.查看是否安装过git,若出现版本号，则代表已经安装了git

git --version

2.使用yum安装git

yum -y install git

### CentOS安装Docker

1.卸载旧版本

较旧的 Docker 版本称为 docker 或 docker-engine 。如果已安装这些程序，请卸载它们以及相关的依赖项。

$ sudo yum remove docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-engine

2.设置仓库

在新主机上首次安装 Docker Engine-Community 之前，需要设置 Docker 仓库。之后，您可以从仓库安装和更新 Docker。

安装所需的软件包。yum-utils 提供了 yum-config-manager ，并且 device mapper 存储驱动程序需要 device-mapper-persistent-data 和 lvm2。

$ sudo yum install -y yum-utils \

device-mapper-persistent-data \

lvm2

使用以下命令来设置稳定的仓库。

$ sudo yum-config-manager \

--add-repo \

https://download.docker.com/linux/centos/docker-ce.repo

3.安装 Docker Engine-Community

安装最新版本的 Docker Engine-Community 和 containerd，或者转到下一步安装特定版本：

$ sudo yum install docker-ce docker-ce-cli containerd.io

如果提示您接受 GPG 密钥，请选是。

有多个 Docker 仓库吗？

如果启用了多个 Docker 仓库，则在未在 yum install 或 yum update 命令中指定版本的情况下，进行的安装或更新将始终安装最高版本，这可能不适合您的稳定性需求。

Docker 安装完默认未启动。并且已经创建好 docker 用户组，但该用户组下没有用户。

要安装特定版本的 Docker Engine-Community，请在存储库中列出可用版本，然后选择并安装：

1、列出并排序您存储库中可用的版本。此示例按版本号（从高到低）对结果进行排序。

$ yum list docker-ce --showduplicates | sort -r

docker-ce.x86\_64 3:18.09.1-3.el7 docker-ce-stable

docker-ce.x86\_64 3:18.09.0-3.el7 docker-ce-stable

docker-ce.x86\_64 18.06.1.ce-3.el7 docker-ce-stable

docker-ce.x86\_64 18.06.0.ce-3.el7 docker-ce-stable

2、通过其完整的软件包名称安装特定版本，该软件包名称是软件包名称（docker-ce）加上版本字符串（第二列），从第一个冒号（:）一直到第一个连字符，并用连字符（-）分隔。例如：docker-ce-18.09.1。

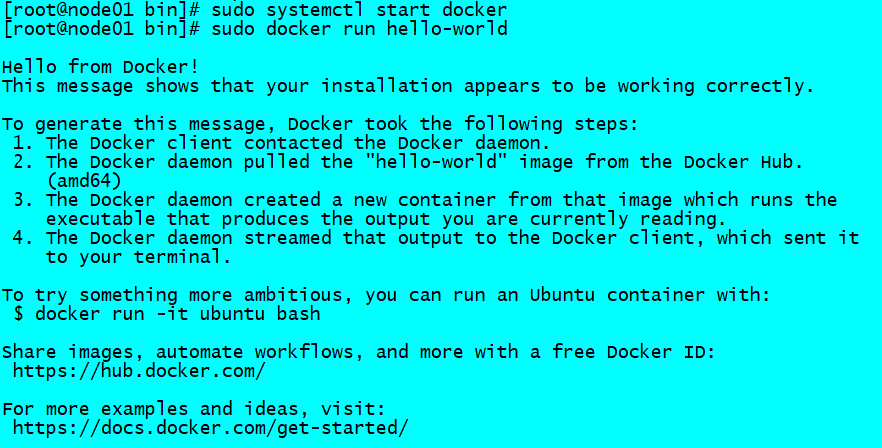
$ sudo yum install docker-ce-<VERSION\_STRING> docker-ce-cli-<VERSION\_STRING> containerd.io

3.启动 Docker

$ sudo systemctl start docker

4.通过运行 hello-world

$ sudo docker run hello-world



### CentOS安装Docker Compose

Compose 简介

Compose 是用于定义和运行多容器 Docker 应用程序的工具。通过 Compose，您可以使用 YML 文件来配置应用程序需要的所有服务。然后，使用一个命令，就可以从 YML 文件配置中创建并启动所有服务。

如果你还不了解 YML 文件配置，可以先阅读 YAML 入门教程。

Compose 使用的三个步骤：

使用 Dockerfile 定义应用程序的环境。

使用 docker-compose.yml 定义构成应用程序的服务，这样它们可以在隔离环境中一起运行。

最后，执行 docker-compose up 命令来启动并运行整个应用程序。

docker-compose.yml 的配置案例如下（配置参数参考下文）：

实例

# yaml 配置实例

version: '3'

services:

web:

build: .

ports:

- "5000:5000"

volumes:

- .:/code

- logvolume01:/var/log

links:

- redis

redis:

image: redis

volumes:

logvolume01: {}

1.Compose 安装

Linux

Linux 上我们可以从 Github 上下载它的二进制包来使用，最新发行的版本地址：https://github.com/docker/compose/releases。

运行以下命令以下载 Docker Compose 的当前稳定版本：

$ sudo curl -L "https://github.com/docker/compose/releases/download/1.24.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

要安装其他版本的 Compose，请替换 1.24.1。

2.将可执行权限应用于二进制文件：

$ sudo chmod +x /usr/local/bin/docker-compose

3.创建软链：

$ sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose

4.测试是否安装成功：

$ docker-compose --version

cker-compose version 1.24.1, build 4667896b

### Docker Compose命令

1、Docker-Compose命令格式

docker-compose [-f <arg>...] [options] [COMMAND] [ARGS...]

命令选项如下：

-f，–file FILE指定Compose模板文件，默认为docker-compose.yml，可以多次指定。

-p，–project-name NAME指定项目名称，默认将使用所在目录名称作为项目名。

-x-network-driver 使用Docker的可拔插网络后端特性（需要Docker 1.9+版本）

-x-network-driver DRIVER指定网络后端的驱动，默认为bridge（需要Docker 1.9+版本）

-verbose输出更多调试信息

-v，–version打印版本并退出

2、docker-compose up

docker-compose up [options] [--scale SERVICE=NUM...] [SERVICE...]

选项包括：

-d 在后台运行服务容器

–no-color 不使用颜色来区分不同的服务的控制输出

–no-deps 不启动服务所链接的容器

–force-recreate 强制重新创建容器，不能与–no-recreate同时使用

–no-recreate 如果容器已经存在，则不重新创建，不能与–force-recreate同时使用

–no-build 不自动构建缺失的服务镜像

–build 在启动容器前构建服务镜像

–abort-on-container-exit 停止所有容器，如果任何一个容器被停止，不能与-d同时使用

-t, –timeout TIMEOUT 停止容器时候的超时（默认为10秒）

–remove-orphans 删除服务中没有在compose文件中定义的容器

–scale SERVICE=NUM 设置服务运行容器的个数，将覆盖在compose中通过scale指定的参数

docker-compose up

启动所有服务

docker-compose up -d

在后台启动所有服务

-f 指定使用的Compose模板文件，默认为docker-compose.yml，可以多次指定。

docker-compose -f docker-compose.yml up -d

3、docker-compose ps

docker-compose ps [options] [SERVICE...]

docker-compose ps

列出项目中目前的所有容器

4、docker-compose stop

docker-compose stop [options] [SERVICE...]

选项包括：

-t, –timeout TIMEOUT 停止容器时候的超时（默认为10秒）

docker-compose stop

停止正在运行的容器，可以通过docker-compose start 再次启动

5、docker-compose -h

docker-compose -h

查看帮助

6、docker-compose down

docker-compose down [options]

停止和删除容器、网络、卷、镜像。

选项包括：

–rmi type，删除镜像，类型必须是：all，删除compose文件中定义的所有镜像；local，删除镜像名为空的镜像

-v, –volumes，删除已经在compose文件中定义的和匿名的附在容器上的数据卷

–remove-orphans，删除服务中没有在compose中定义的容器

docker-compose down

停用移除所有容器以及网络相关

7、docker-compose logs

docker-compose logs [options] [SERVICE...]

查看服务容器的输出。默认情况下，docker-compose将对不同的服务输出使用不同的颜色来区分。可以通过–no-color来关闭颜色。

docker-compose logs

查看服务容器的输出

8、docker-compose build

docker-compose build [options] [--build-arg key=val...] [SERVICE...]

构建（重新构建）项目中的服务容器。

选项包括：

–compress 通过gzip压缩构建上下环境

–force-rm 删除构建过程中的临时容器

–no-cache 构建镜像过程中不使用缓存

–pull 始终尝试通过拉取操作来获取更新版本的镜像

-m, –memory MEM为构建的容器设置内存大小

–build-arg key=val为服务设置build-time变量

服务容器一旦构建后，将会带上一个标记名。可以随时在项目目录下运行docker-compose build来重新构建服务

9、docker-compose pull

docker-compose pull [options] [SERVICE...]

拉取服务依赖的镜像。

选项包括：

–ignore-pull-failures，忽略拉取镜像过程中的错误

–parallel，多个镜像同时拉取

–quiet，拉取镜像过程中不打印进度信息

docker-compose pull

拉取服务依赖的镜像

10、docker-compose restart

docker-compose restart [options] [SERVICE...]

重启项目中的服务。

选项包括：

-t, –timeout TIMEOUT，指定重启前停止容器的超时（默认为10秒）

docker-compose restart

重启项目中的服务

11、docker-compose rm

docker-compose rm [options] [SERVICE...]

删除所有（停止状态的）服务容器。

选项包括：

–f, –force，强制直接删除，包括非停止状态的容器

-v，删除容器所挂载的数据卷

docker-compose rm

删除所有（停止状态的）服务容器。推荐先执行docker-compose stop命令来停止容器。

12、docker-compose start

docker-compose start [SERVICE...]

docker-compose start

启动已经存在的服务容器。

13、docker-compose run

docker-compose run [options] [-v VOLUME...] [-p PORT...] [-e KEY=VAL...] SERVICE [COMMAND] [ARGS...]

在指定服务上执行一个命令。

docker-compose run ubuntu ping www.baidu.com

在指定容器上执行一个ping命令。

14、docker-compose scale

docker-compose scale web=3 db=2

设置指定服务运行的容器个数。通过service=num的参数来设置数量

15、docker-compose pause

docker-compose pause [SERVICE...]

暂停一个服务容器

16、docker-compose kill

docker-compose kill [options] [SERVICE...]

通过发送SIGKILL信号来强制停止服务容器。

支持通过-s参数来指定发送的信号，例如通过如下指令发送SIGINT信号：

docker-compose kill -s SIGINT

17、dokcer-compose config

docker-compose config [options]

验证并查看compose文件配置。

选项包括：

–resolve-image-digests 将镜像标签标记为摘要

-q, –quiet 只验证配置，不输出。 当配置正确时，不输出任何内容，当文件配置错误，输出错误信息

–services 打印服务名，一行一个

–volumes 打印数据卷名，一行一个

18、docker-compose create

docker-compose create [options] [SERVICE...]

为服务创建容器。

选项包括：

–force-recreate：重新创建容器，即使配置和镜像没有改变，不兼容–no-recreate参数

–no-recreate：如果容器已经存在，不需要重新创建，不兼容–force-recreate参数

–no-build：不创建镜像，即使缺失

–build：创建容器前，生成镜像

19、docker-compose exec

docker-compose exec [options] SERVICE COMMAND [ARGS...]

选项包括：

-d 分离模式，后台运行命令。

–privileged 获取特权。

–user USER 指定运行的用户。

-T 禁用分配TTY，默认docker-compose exec分配TTY。

–index=index，当一个服务拥有多个容器时，可通过该参数登陆到该服务下的任何服务，例如：docker-compose exec –index=1 web /bin/bash ，web服务中包含多个容器

20、docker-compose port

docker-compose port [options] SERVICE PRIVATE\_PORT

显示某个容器端口所映射的公共端口。

选项包括：

–protocol=proto，指定端口协议，TCP（默认值）或者UDP

–index=index，如果同意服务存在多个容器，指定命令对象容器的序号（默认为1）

21、docker-compose push

docker-compose push [options] [SERVICE...]

推送服务依的镜像。

选项包括：

–ignore-push-failures 忽略推送镜像过程中的错误

22、docker-compose ps

docker-compose ps[options] [SERVICE...]

显示各个容器运行的进程情况。

23、docker-compose unpause

docker-compose unpause [SERVICE...]

恢复处于暂停状态中的服务。

24、docker-compose version

docker-compose version

打印版本信息。

# TiDB实践案例

## TiDB-SQL操作

<https://pingcap.com/docs-cn/stable/reference/mysql-compatibility/>

sudo systemctl stop docker

sudo systemctl start docker

cd /export/servers/tidb-docker-compose && docker-compose up -d

docker-compose ps

成功部署 TiDB 集群之后，便可以在 TiDB 中执行 SQL 语句了。因为 TiDB 兼容 MySQL，可以使用 MySQL 客户端连接 TiDB，并且大多数情况下可以直接执行 MySQL 语句。

mysql -h 127.0.0.1 -P 4000 -u root

show databases;

### 创建、查看和删除数据库

1.要创建一个名为 samp\_db 的数据库，可使用以下语句：

CREATE DATABASE IF NOT EXISTS samp\_db;

2.使用 SHOW DATABASES 语句查看数据库：

SHOW DATABASES;

3.使用 DROP DATABASE 语句删除数据库，例如：

DROP DATABASE samp\_db;

4.再次查看数据库：

SHOW DATABASES;

### 创建、查看和删除表

1.先创建一个库

CREATE DATABASE IF NOT EXISTS samp\_db;

USE samp\_db;

2.使用 SHOW TABLES 语句查看数据库中的所有表。例如：

SHOW TABLES FROM samp\_db;

3.使用 CREATE TABLE 语句创建表。

如果表已存在，添加 IF NOT EXISTS 可防止发生错误：

CREATE TABLE IF NOT EXISTS person (

number INT(11),

name VARCHAR(255),

birthday DATE

);

4.使用 SHOW CREATE 语句查看建表语句。例如：

SHOW CREATE table person;

5.使用 SHOW FULL COLUMNS 语句查看表的列。 例如：

SHOW FULL COLUMNS FROM person;

6.使用 DROP TABLE 语句删除表。例如：

DROP TABLE person;

或者

DROP TABLE IF EXISTS person;

### 创建、查看和删除索引

1.先创建一张表

CREATE TABLE IF NOT EXISTS person (

number INT(11),

name VARCHAR(255),

birthday DATE

);

2.对于值不唯一的列，可使用 CREATE INDEX 或 ALTER TABLE 语句。例如：

CREATE INDEX person\_num ON person (number);

或者

ALTER TABLE person ADD INDEX person\_num (number);

3.使用 SHOW INDEX 语句查看表内所有索引：

SHOW INDEX from person;

4.使用 ALTER TABLE 或 DROP INDEX 语句来删除索引。与 CREATE INDEX 语句类似，DROP INDEX 也可以嵌入 ALTER TABLE 语句。例如：

DROP INDEX person\_num ON person;

ALTER TABLE person DROP INDEX person\_num;

5.对于值唯一的列，可以创建唯一索引。例如：

CREATE UNIQUE INDEX person\_num ON person (number);

或者

ALTER TABLE person ADD UNIQUE person\_num (number);

### 增删改查数据

1.使用 INSERT 语句向表内插入数据。例如：

INSERT INTO person VALUES("1","tom","20170912");

2.使用 SELECT 语句检索表内数据。例如：

SELECT \* FROM person;

3.使用 UPDATE 语句修改表内数据。例如：

UPDATE person SET birthday='20200202' WHERE name='tom';

SELECT \* FROM person;

4.使用 DELETE 语句删除表内数据：

DELETE FROM person WHERE number=1;

SELECT \* FROM person;

### 创建、授权和删除用户

1.使用 CREATE USER 语句创建一个用户 tiuser，密码为 123456：

CREATE USER 'tiuser'@'localhost' IDENTIFIED BY '123456';

2.授权用户 tiuser 可检索数据库 samp\_db 内的表：

GRANT SELECT ON samp\_db.\* TO 'tiuser'@'localhost';

3.查询用户 tiuser 的权限：

SHOW GRANTS for tiuser@localhost;

4.删除用户 tiuser：

DROP USER 'tiuser'@'localhost';

5.查看所有权限

SHOW GRANTS;

## TiDB-读取历史数据

接下来介绍 TiDB 如何读取历史版本数据，包括具体的操作流程以及历史数据的保存策略。

### 功能说明

TiDB 实现了通过标准 SQL 接口读取历史数据功能，无需特殊的 client 或者 driver。当数据被更新、删除后，依然可以通过 SQL 接口将更新/删除前的数据读取出来。

另外即使在更新数据之后，表结构发生了变化，TiDB 依旧能用旧的表结构将数据读取出来。

### 操作流程

为支持读取历史版本数据， 引入了一个新的 system variable: tidb\_snapshot ，这个变量是 Session 范围有效，可以通过标准的 Set 语句修改其值。其值为文本，能够存储 TSO 和日期时间。TSO 即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： “2020-10-08 16:45:26.999”，一般来说可以只写到秒，比如”2020-10-08 16:45:26”。 当这个变量被设置时，TiDB 会用这个时间戳建立 Snapshot（没有开销，只是创建数据结构），随后所有的 Select 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 的时间为准。

当读取历史版本操作结束后，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。

### 历史数据保留策略

TiDB 使用 MVCC 管理版本，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，所以可以保留历史数据。历史数据不会全部保留，超过一定时间的历史数据会被彻底删除，以减小空间占用以及避免历史版本过多引入的性能开销。

TiDB 使用周期性运行的 GC（Garbage Collection，垃圾回收）来进行清理，关于 GC 的详细介绍参见 [TiDB 垃圾回收 (GC)](https://pingcap.com/docs-cn/dev/reference/garbage-collection/overview)。

这里需要重点关注的是 tikv\_gc\_life\_time 和 tikv\_gc\_safe\_point 这条。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 safePoint，用户可以安全地使用大于 safePoint 的时间戳创建 snapshot 读取历史版本。safePoint 在每次 GC 开始运行时自动更新。

### 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);

insert into t values (1), (2), (3);

2.查看表中的数据：

select \* from t;

+------+

| c |

+------+

| 1 |

| 2 |

| 3 |

+------+

3.查看当前时间：

select now();

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

| now() |

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

| 2020-02-02 16:45:26 |

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

4.更新某一行数据：

update t set c=22 where c=2;

5.确认数据已经被更新：

select \* from t;

+------+

| c |

+------+

| 1 |

| 22 |

| 3 |

+------+

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="2020-02-02 16:45:26";

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。

7.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 |

+------+

8.清空这个变量后，即可读取最新版本数据：

set @@tidb\_snapshot="";

select \* from t;

+------+

| c |

+------+

| 1 |

| 22 |

| 3 |

+------+

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。

## 整合Spark-TiSpark

### 准备数据

向 TiDB 集群中插入一些样本数据：

docker-compose exec tispark-master bash

cd /opt/spark/data/tispark-sample-data

mysql -h tidb -P 4000 -u root < dss.ddl

### 启动Spark shell

当样本数据加载到 TiDB 集群之后，访问 Spark shell。

docker-compose exec tispark-master /opt/spark/bin/spark-shell

...

Spark context available as 'sc' (master = local[\*], app id = local-1527045927617).

Spark session available as 'spark'.

Welcome to

\_\_\_\_ \_\_

/ \_\_/\_\_ \_\_\_ \_\_\_\_\_/ /\_\_

\_\ \/ \_ \/ \_ `/ \_\_/ '\_/

/\_\_\_/ .\_\_/\\_,\_/\_/ /\_/\\_\ version 2.1.1

/\_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_172)

Type in expressions to have them evaluated.

Type :help for more information.

### 执行Spark代码

scala> import org.apache.spark.sql.TiContext

...

scala> val ti = new TiContext(spark)

...

scala> ti.tidbMapDatabase("TPCH\_001")

...

scala> spark.sql("select count(\*) from lineitem").show

+--------+

|count(1)|

+--------+

| 60175|

+--------+

你也可以通过 Python 或 R 来访问 Spark：

docker-compose exec tispark-master /opt/spark/bin/pyspark &&

docker-compose exec tispark-master /opt/spark/bin/sparkR

下面执行另一个复杂一点的 Spark SQL:

scala> spark.sql(

"""select

| l\_returnflag,

| l\_linestatus,

| 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\_linestatus

|order by

| l\_returnflag,

| l\_linestatus

""".stripMargin).show

结果为：

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

|l\_returnflag|l\_linestatus| 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|

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

(续)

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

sum\_charge| avg\_qty| avg\_price|avg\_disc|count\_order|

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

526165934.000839|25.575155|35785.709307|0.050081| 14876|

12282485.056933|25.778736|35588.509684|0.047759| 348|

1029418531.523350|25.454988|35691.129209|0.049931| 29181|

528524219.358903|25.597168|35874.006533|0.049828| 14902|

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

## 数据迁移-TiDB Lightning

MySQL

-->

TiDB

### TiDB Lightning介绍

TiDB Lightning 是一个将全量数据高速导入到 TiDB 集群的工具，目前支持 Mydumper 或 CSV 输出格式的数据源。你可以在以下两种场景下使用 Lightning：

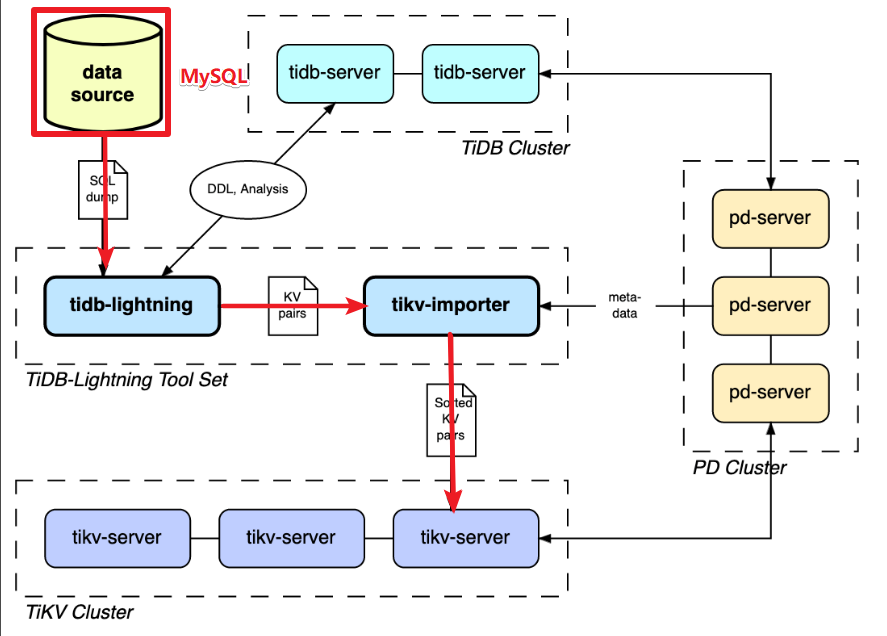
迅速导入大量新数据。

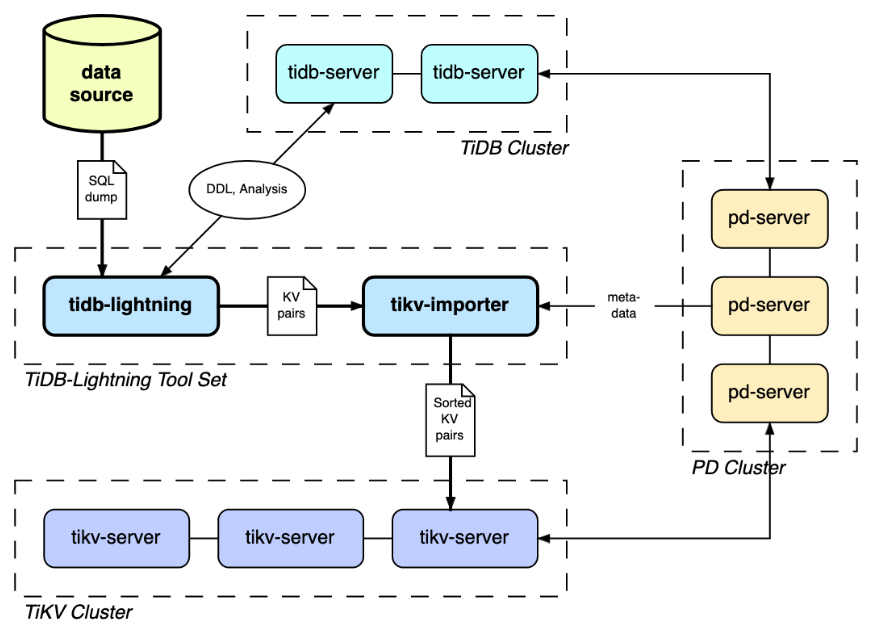
备份恢复所有数据。

TiDB Lightning 主要包含两个部分:

1.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。

2.tikv-importer（“后端”）：主要完成将数据导入 TiKV 集群的工作，把 tidb-lightning 写入的 KV 对缓存、排序、切分并导入到 TiKV 集群。





### 准备迁移工具

wget https://download.pingcap.org/tidb-enterprise-tools-latest-linux-amd64.tar.gz

wget https://download.pingcap.org/tidb-toolkit-latest-linux-amd64.tar.gz

### 准备MySQL数据

|  |
| --- |
| CREATE DATABASE mytest;  USE mytest;  CREATE TABLE mytest.t1 (  id BIGINT UNSIGNED NOT NULL AUTO\_INCREMENT PRIMARY KEY,  c CHAR (32),  PORT INT  );  insert into t1 values (),(),(),(),(),(),(),();  insert into t1 (id) select null from t1;  insert into t1 (id) select null from t1;  insert into t1 (id) select null from t1;  insert into t1 (id) select null from t1;  insert into t1 (id) select null from t1;  update t1 set c=md5(id), port=@@port;  CREATE TABLE mytest.t2 (  id BIGINT UNSIGNED NOT NULL AUTO\_INCREMENT PRIMARY KEY,  c CHAR (32),  PORT INT  );  insert into t2 values (),(),(),(),(),(),(),();  insert into t2 (id) select null from t2;  insert into t2 (id) select null from t2;  insert into t2 (id) select null from t2;  insert into t2 (id) select null from t2;  insert into t2 (id) select null from t2;  update t2 set c=md5(id), port=@@port; |

### 导出数据

mkdir -p /data/my\_database/

cd /export/servers/tidb-enterprise-tools-latest-linux-amd64/bin

./mydumper -h 127.0.0.1 -P 3306 -u root -p Root@1234 -t 16 -F 256 -B mytest -T t1,t2 --skip-tz-utc -o /data/my\_database/

cd /data/my\_database

其中：

-B mytest ：从 mytest 数据库导出。

-T t1,t2：只导出 t1 和 t2 这两个表。

-t 16：使用 16 个线程导出数据。

-F 256：将每张表切分成多个文件，每个文件大小约为 256 MB。

--skip-tz-utc：添加这个参数则会忽略掉 TiDB 与导数据的机器之间时区设置不一致的情况，禁止自动转换。

这样全量备份数据就导出到了/data/my\_database目录中。

### 启动tikv-importer

cd /export/servers/tidb-toolkit-latest-linux-amd64/bin

vim tikv-importer.toml

|  |
| --- |
| # 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、off。  log-level = "info"  [server]  # tikv-importer 监听的地址，tidb-lightning 需要连到这个地址进行数据写入。  addr = "192.168.1.101:8287"  [import]  # 存储引擎文档 (engine file) 的文件夹路径。  import-dir = "/mnt/ssd/data.import/" |

nohup ./tikv-importer -C tikv-importer.toml > nohup.out &

### 启动tidb-lightning

vim run.sh

|  |
| --- |
| #!/bin/bash  nohup ./tidb-lightning \  --importer 192.168.1.101:8287 \  -d /data/my\_database/ \  --pd-urls 0.0.0.0:2379 \  --tidb-host 192.168.1.101 \  --tidb-user root \  --log-file tidb-lightning.log \  > nohup.out & |

chmod 755 run.sh

./run.sh

### 注意

有性能要求,使用单机版演示即可

sudo systemctl stop docker

cd /export/servers/tidb-latest-linux-amd64

./bin/pd-server --data-dir=pd --log-file=pd.log &

./bin/tikv-server --pd="127.0.0.1:2379" --data-dir=tikv --log-file=tikv.log &

./bin/tidb-server --store=tikv --path="127.0.0.1:2379" --log-file=tidb.log &

# TiDB技术内幕

## TiDB 技术内幕 - 存储

### 引言

数据库、操作系统和编译器并称为三大系统，可以说是整个计算机软件的基石。其中数据库更靠近应用层，是很多业务的支撑。这一领域经过了几十年的发展，不断的有新的进展。

很多人用过数据库，但是很少有人实现过一个数据库，特别是实现一个分布式数据库。了解数据库的实现原理和细节，一方面可以提高个人技术，对构建其他系统有帮助，另一方面也有利于用好数据库。

研究一门技术最好的方法是研究其中一个开源项目，数据库也不例外。单机数据库领域有很多很好的开源项目，其中 MySQL 和 PostgreSQL 是其中知名度最高的两个，不少同学都看过这两个项目的代码。但是分布式数据库方面，好的开源项目并不多。 TiDB 目前获得了广泛的关注，特别是一些技术爱好者，希望能够参与这个项目。由于分布式数据库自身的复杂性，很多人并不能很好的理解整个项目，所以我希望能写一些文章，自顶向下，由浅入深，讲述 TiDB 的一些技术原理，包括用户可见的技术以及大量隐藏在 SQL 界面后用户不可见的技术点。

### 保存数据



数据库最根本的功能是能把数据存下来，所以我们从这里开始。

保存数据的方法很多，最简单的方法是直接在内存中建一个数据结构，保存用户发来的数据。比如用一个数组，每当收到一条数据就向数组中追加一条记录。这个方案十分简单，能满足最基本，并且性能肯定会很好，但是除此之外却是漏洞百出，其中最大的问题是数据完全在内存中，一旦停机或者是服务重启，数据就会永久丢失。

为了解决数据丢失问题，我们可以把数据放在非易失存储介质（比如硬盘）中。改进的方案是在磁盘上创建一个文件，收到一条数据，就在文件中 Append 一行。OK，我们现在有了一个能持久化存储数据的方案。但是还不够好，假设这块磁盘出现了坏道呢？我们可以做 RAID （Redundant Array of Independent Disks），提供单机冗余存储。如果整台机器都挂了呢？比如出现了火灾，RAID 也保不住这些数据。我们还可以将存储改用网络存储，或者是通过硬件或者软件进行存储复制。到这里似乎我们已经解决了数据安全问题，可以松一口气了。But，做复制过程中是否能保证副本之间的一致性？也就是在保证数据不丢的前提下，还要保证数据不错。保证数据不丢不错只是一项最基本的要求，还有更多令人头疼的问题等待解决：

* 能否支持跨数据中心的容灾？
* 写入速度是否够快？
* 数据保存下来后，是否方便读取？
* 保存的数据如何修改？如何支持并发的修改？
* 如何原子地修改多条记录？

这些问题每一项都非常难，但是要做一个优秀的数据存储系统，必须要解决上述的每一个难题。 为了解决数据存储问题，我们开发了 TiKV 这个项目。接下来我向大家介绍一下 TiKV 的一些设计思想和基本概念。

### Key-Value

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。简单来讲，可以将 TiKV 看做一个巨大的 Map，其中 Key 和 Value 都是原始的 Byte 数组，在这个 Map 中，Key 按照 Byte 数组总的原始二进制比特位比较顺序排列。 大家这里需要对 TiKV 记住两点：

1. 这是一个巨大的 Map，也就是存储的是 Key-Value pair
2. 这个 Map 中的 Key-Value pair 按照 Key 的二进制顺序有序，也就是我们可以 Seek 到某一个 Key 的位置，然后不断的调用 Next 方法以递增的顺序获取比这个 Key 大的 Key-Value

讲了这么多，有人可能会问了，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要说四遍：

这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 SQL 中的 Table 无关！ 这里的存储模型和 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的巨大的（分布式的） Map。

### 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 是一个非常优秀的开源的单机存储引擎，可以满足我们对单机引擎的各种要求，而且还有 Facebook 的团队在做持续的优化，这样我们只投入很少的精力，就能享受到一个十分强大且在不断进步的单机引擎。当然，我们也为 RocksDB 贡献了一些代码，希望这个项目能越做越好。这里可以简单的认为 RocksDB 是一个单机的 Key-Value Map。

底层LSM树将对数据的修改增量保存在内存中，达到指定大小限制之后批量把数据flush到磁盘中，磁盘中树定期可以做merge操作，合并成一棵大树，以优化性能。

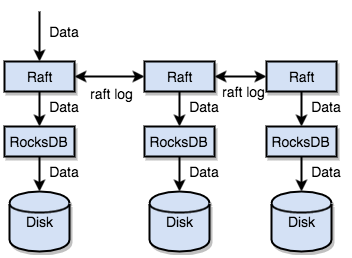
### Raft

好了，万里长征第一步已经迈出去了，我们已经为数据找到一个高效可靠的本地存储方案。俗话说，万事开头难，然后中间难，最后结尾难。接下来我们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，我们需要想办法把数据复制到多台机器上，这样一台机器挂了，我们还有其他的机器上的副本；复杂来说，我们还需要这个复制方案是可靠、高效并且能处理副本失效的情况。听上去比较难，但是好在我们有 Raft 协议。Raft 是一个一致性算法，它和 Paxos 等价，但是更加易于理解。[Raft 的论文](https://raft.github.io/raft.pdf)，感兴趣的可以看一下。本文只会对 Raft 做一个简要的介绍，细节问题可以参考论文。另外提一点，Raft 论文只是一个基本方案，严格按照论文实现，性能会很差，我们对 Raft 协议的实现做了大量的优化，具体的优化细节可参考tangliu 同学的[《TiKV 源码解析系列 - Raft 的优化》](https://zhuanlan.zhihu.com/p/25735592)这篇文章。

Raft 是一个一致性协议，提供几个重要的功能：

1. Leader 选举
2. 成员变更
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 Group 的多数节点中。



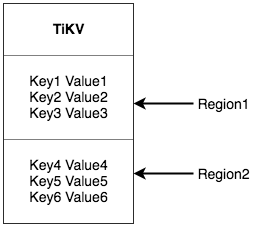
到这里我们总结一下，通过单机的 RocksDB，我们可以将数据快速地存储在磁盘上；通过 Raft，我们可以将数据复制到多台机器上，以防单机失效。数据的写入是通过 Raft 这一层的接口写入，而不是直接写 RocksDB。通过实现 Raft，我们拥有了一个分布式的 KV，现在再也不用担心某台机器挂掉了。

### Region

讲到这里，我们可以提到一个 **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：一种是按照 Key 做 Hash，根据 Hash 值选择对应的存储节点；另一种是分 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，我们将每一段叫做一个 **Region**，并且我们会尽量保持每个 Region 中保存的数据不超过一定的大小(这个大小可以配置，目前默认是 64mb)。每一个 Region 都可以用 StartKey 到 EndKey 这样一个左闭右开区间来描述。



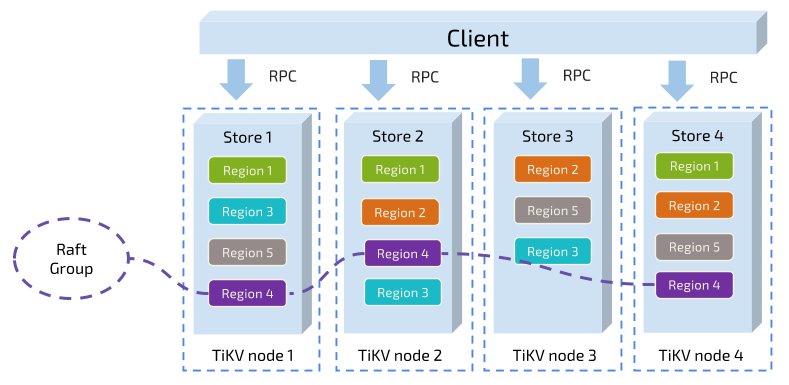
**注意，这里的 Region 还是和 SQL 中的表没什么关系！** 请各位继续忘记 SQL，只谈 KV。 将数据划分成 Region 后，我们将会做 **两件重要的事情**：

* 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。我们的系统会有一个组件来负责将 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（增加新的结点后，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，我们的系统中也会有一个组件记录 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。

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，我们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，再由 Leader 复制给 Follower。



我们以 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。这已经很 Cool，但是还不够完美，我们需要更多的功能。

### MVCC

很多数据库都会实现多版本控制（MVCC），TiKV 也不例外。设想这样的场景，两个 Client 同时去修改一个 Key 的 Value，如果没有 MVCC，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 Version 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，可以将 Key 和 Version 构造出 MVCC 的 Key，也就是 Key-Version。然后可以直接 Seek(Key-Version)，定位到第一个大于等于这个 Key-Version 的位置。

### 事务

TiKV 的事务采用的是 [Percolator](https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Peng.pdf) 模型，并且做了大量的优化。TiKV 的事务采用乐观锁，事务的执行过程中，不会检测写写冲突，只有在提交过程中，才会做冲突检测，冲突的双方中比较早完成提交的会写入成功，另一方会尝试重新执行整个事务。当业务的写入冲突不严重的情况下，这种模型性能会很好，比如随机更新表中某一行的数据，并且表很大。但是如果业务的写入冲突严重，性能就会很差，举一个极端的例子，就是计数器，多个客户端同时修改少量行，导致冲突严重的，造成大量的无效重试。

### 其他

到这里，我们已经了解了 TiKV 的基本概念和一些细节，理解了这个分布式带事务的 KV 引擎的分层结构以及如何实现多副本容错。下一节会介绍如何在 KV 的存储模型之上，构建 SQL 层。

## TiDB 技术内幕 - 计算

### 关系模型到 Key-Value 模型的映射

在这我们将关系模型简单理解为 Table 和 SQL 语句，那么问题变为如何在 KV 结构上保存 Table 以及如何在 KV 结构上运行 SQL 语句。 假设我们有这样一个表的定义：

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

SQL 和 KV 结构之间存在巨大的区别，那么如何能够方便高效地进行映射，就成为一个很重要的问题。一个好的映射方案必须有利于对数据操作的需求。那么我们先看一下对数据的操作有哪些需求，分别有哪些特点。

对于一个 Table 来说，需要存储的数据包括三部分：

1. 表的元信息
2. Table 中的 Row
3. 索引数据

表的元信息我们暂时不讨论，后面介绍。

对于 Row，可以选择行存或者列存，这两种各有优缺点。TiDB 面向的首要目标是 OLTP 业务，这类业务需要支持快速地读取、保存、修改、删除一行数据，所以采用行存是比较合适的。

对于 Index，TiDB 不止需要支持 Primary Index，还需要支持 Secondary Index。Index 的作用的辅助查询，提升查询性能，以及保证某些 Constraint。

查询的时候有两种模式，一种是点查，比如通过 Primary Key 或者 Unique Key 的等值条件进行查询，如 select name from user where id=1; ，这种需要通过索引快速定位到某一行数据；另一种是 Range 查询，如 select name from user where age > 30 and age < 35;，这个时候需要通过idxAge索引查询 age 在 30 和 35 之间的那些数据。Index 还分为 Unique Index 和 非 Unique Index，这两种都需要支持。

分析完需要存储的数据的特点，我们再看看对这些数据的操作需求，主要考虑 Insert/Update/Delete/Select 这四种语句。

对于 Insert 语句，需要将 Row 写入 KV，并且建立好索引数据。

对于 Update 语句，需要将 Row 更新的同时，更新索引数据（如果有必要）。

对于 Delete 语句，需要在删除 Row 的同时，将索引也删除。

上面三个语句处理起来都很简单。对于 Select 语句，情况会复杂一些。首先我们需要能够简单快速地读取一行数据，所以每个 Row 需要有一个 ID （显示或隐式的 ID）。其次可能会读取连续多行数据，比如 Select \* from user;。最后还有通过索引读取数据的需求，对索引的使用可能是点查或者是范围查询。

大致的需求已经分析完了，现在让我们看看手里有什么可以用的：**一个全局有序的分布式 Key-Value 引擎**。全局有序这一点重要，可以帮助我们解决不少问题。比如对于快速获取一行数据，假设我们能够构造出某一个或者某几个 Key，定位到这一行，我们就能利用 TiKV 提供的 Seek 方法快速定位到这一行数据所在位置。再比如对于扫描全表的需求，如果能够映射为一个 Key 的 Range，从 StartKey 扫描到 EndKey，那么就可以简单的通过这种方式获得全表数据。操作 Index 数据也是类似的思路。接下来让我们看看 TiDB 是如何做的。

TiDB 对每个表分配一个 TableID，每一个索引都会分配一个 IndexID，每一行分配一个 RowID（如果表有整数型的 Primary Key，那么会用 Primary Key 的值当做 RowID），其中 TableID 在整个集群内唯一，IndexID/RowID 在表内唯一，这些 ID 都是 int64 类型。

每行数据按照如下规则进行编码成 Key-Value pair：

Key: tablePrefix{tableID}\_recordPrefixSep{rowID}  
Value: [col1, col2, col3, col4]

其中 Key 的 tablePrefix/recordPrefixSep 都是特定的字符串常量，用于在 KV 空间内区分其他数据。

对于 Index 数据，会按照如下规则编码成 Key-Value pair：

Key: tablePrefix{tableID}\_indexPrefixSep{indexID}\_indexedColumnsValue  
Value: rowID

Index 数据还需要考虑 Unique Index 和非 Unique Index 两种情况，对于 Unique Index，可以按照上述编码规则。但是对于非 Unique Index，通过这种编码并不能构造出唯一的 Key，因为同一个 Index 的 tablePrefix{tableID}\_indexPrefixSep{indexID} 都一样，可能有多行数据的 ColumnsValue 是一样的，所以对于非 Unique Index 的编码做了一点调整：

Key: tablePrefix{tableID}\_indexPrefixSep{indexID}\_indexedColumnsValue\_rowID  
Value: null

这样能够对索引中的每行数据构造出唯一的 Key。

注意上述编码规则中的 Key 里面的各种 xxPrefix 都是字符串常量，作用都是区分命名空间，以免不同类型的数据之间相互冲突，定义如下：

var(  
 tablePrefix = []byte{'t'}  
 recordPrefixSep = []byte("\_r")  
 indexPrefixSep = []byte("\_i")  
)

另外请大家注意，上述方案中，无论是 Row 还是 Index 的 Key 编码方案，一个 Table 内部所有的 Row 都有相同的前缀，一个 Index 的数据也都有相同的前缀。这样具体相同的前缀的数据，在 TiKV 的 Key 空间内，是排列在一起。

同时只要我们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，那么就可以将 Row 或者 Index 数据有序地保存在 TiKV 中。这种保证编码前和编码后的比较关系不变 的方案我们称为 Memcomparable，对于任何类型的值，两个对象编码前的原始类型比较结果，和编码成 byte 数组后（注意，TiKV 中的 Key 和 Value 都是原始的 byte 数组）的比较结果保持一致。采用这种编码后，一个表的所有 Row 数据就会按照 RowID 的顺序排列在 TiKV 的 Key 空间中，某一个 Index 的数据也会按照 Index 的 ColumnValue 顺序排列在 Key 空间内。

现在我们结合开始提到的需求以及 TiDB 的映射方案来看一下，这个方案是否能满足需求。

首先我们通过这个映射方案，将 Row 和 Index 数据都转换为 Key-Value 数据，且每一行、每一条索引数据都是有唯一的 Key。

其次，这种映射方案对于点查、范围查询都很友好，我们可以很容易地构造出某行、某条索引所对应的 Key，或者是某一块相邻的行、相邻的索引值所对应的 Key 范围。

最后，在保证表中的一些 Constraint 的时候，可以通过构造并检查某个 Key 是否存在来判断是否能够满足相应的 Constraint。

至此我们已经聊完了如何将 Table 映射到 KV 上面，这里再举个简单的例子，便于大家理解，还是以上面的表结构为例。假设表中有 3 行数据：

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

那么首先每行数据都会映射为一个 Key-Value pair，注意这个表有一个 Int 类型的 Primary Key，所以 RowID 的值即为这个 Primary Key 的值。假设这个表的 Table ID 为 10，其 Row 的数据为：

t10\_r1 --> ["TiDB", "SQL Layer", 10]  
t10\_r2 --> ["TiKV", "KV Engine", 20]  
t10\_r3 --> ["PD", "Manager", 30]

除了 Primary Key 之外，这个表还有一个 Index，假设这个 Index 的 ID 为 1，则其数据为：

t10\_i1\_10\_1 --> null  
t10\_i1\_20\_2 --> null  
t10\_i1\_30\_3 --> null

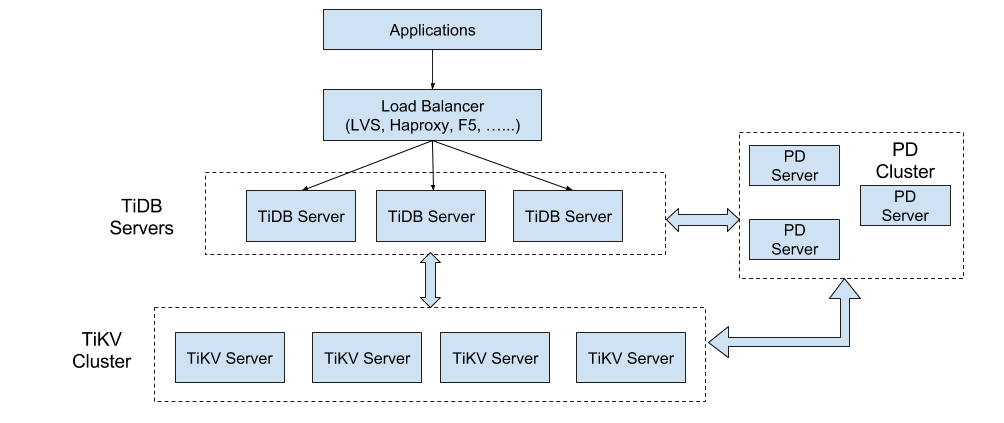
### 元信息管理

上节介绍了表中的数据和索引是如何映射为 KV，本节介绍一下元信息的存储。Database/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，我们也将这些信息存储在 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，还有一个专门的 Key-Value 存储当前 Schema 信息的版本。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 TiKV 上面存储的 Schema 版本是否发生变化，并且保证在一定时间内一定能够获取版本的变化（如果确实发生了变化）。

### SQL on KV 架构

TiDB 的整体架构如下图所示



TiKV Cluster 主要作用是作为 KV 引擎存储数据，前面已经介绍过了细节，这里不再敷述。这里主要介绍 SQL 层，也就是 TiDB Servers 这一层，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。TiDB Server 这一层最重要的工作是处理用户请求，执行 SQL 运算逻辑，接下来我们做一些简单的介绍。

### SQL 运算

理解了 SQL 到 KV 的映射方案之后，我们可以理解关系数据是如何保存的，接下来我们要理解如何使用这些数据来满足用户的查询需求，也就是一个查询语句是如何操作底层存储的数据。

能想到的最简单的方案就是通过上一节所述的映射方案，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。这样一个操作流程转换为 KV 操作流程：

* 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 Row 的 Key 编码规则，就能构造出一个 [StartKey, EndKey) 的左闭右开区间
* 扫描 Key Range：根据上面构造出的 Key Range，读取 TiKV 中的数据
* 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
* 计算 Count：对符合要求的每一行，累计到 Count 值上面 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 操作同 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

### 分布式 SQL 运算

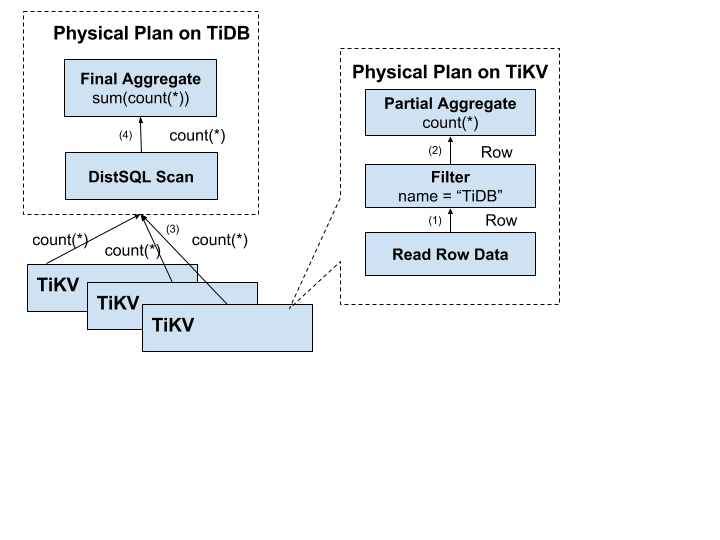
如何避免上述缺陷也是显而易见的，

首先我们需要将计算尽量靠近存储节点，以避免大量的 RPC 调用。

其次，我们需要将 Filter 也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。

最后，我们可以将聚合函数、GroupBy 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 值即可，再由 tidb-server 将 Count 值 Sum 起来。

这里有一个数据逐层返回的示意图：



[MPP and SMP in TiDB](https://mp.weixin.qq.com/s?__biz=MzI3NDIxNTQyOQ==&mid=2247484187&idx=1&sn=90a7ce3e6db7946ef0b7609a64e3b423&chksm=eb162471dc61ad679fc359100e2f3a15d64dd458446241bff2169403642e60a95731c6716841&scene=4) 这篇文章详细描述了 TiDB 是如何让 SQL 语句跑的更快，大家可以参考一下。

### SQL 层架构

上面几节简要介绍了 SQL 层的一些功能，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，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。

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-server 交互，获取数据。

最后 tidb-server 需要将查询结果返回给用户。

### 小结

到这里，我们已经从 SQL 的角度了解了数据是如何存储，如何用于计算。 后面我们将会介绍一些关于 PD 的信息，这部分会比较有意思，里面的很多东西是在使用 TiDB 过程中看不到，但是对整体集群又非常重要。主要会涉及到集群的管理和调度。

## TiDB 技术内幕 - 调度

### 为什么要进行调度

先回忆一下[TiDB 技术内幕 - 存储](https://pingcap.com/blog-cn/tidb-internal-1)提到的一些信息，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用于容灾的时候，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？如果节点只是短暂掉线（重启服务），那么如何处理？如果节点是长时间掉线（磁盘故障，数据全部丢失），需要如何处理？
* 假设集群需要每个 Raft Group 有 N 个副本，那么对于单个 Raft Group 来说，Replica 数量可能会不够多（例如节点掉线，失去副本），也可能会过于多（例如掉线的节点又回复正常，自动加入集群）。那么如何调节 Replica 个数？
* 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足够多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，是需要考虑全局的信息。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。

### 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：

**1.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
* 节点下线后，需要将该节点的数据迁移走

**2.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 Balance 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，以及自动下线失效节点

满足第一类需求后，整个系统将具备多副本容错、动态扩容/缩容、容忍节点掉线以及自动错误恢复的功能。

满足第二类需求后，可以使得整体系统的负载更加均匀、且可以方便的管理。

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。

### 调度的基本操作

我们先来介绍最简单的一点，也就是调度的基本操作，也就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，但是整理下来最终落地的无非是下面三件事：

* 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

刚好 Raft 协议能够满足这三种需求，通过 AddReplica、RemoveReplica、TransferLeader 这三个命令，可以支撑上述三种基本操作。

### 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：

**每个 TiKV 节点会定期向 PD 汇报节点的整体信息**

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](https://github.com/pingcap/kvproto/blob/master/proto/pdpb.proto#L294)，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
* 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](https://github.com/pingcap/kvproto/blob/master/proto/pdpb.proto#L207)，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，就认为是 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。

### 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。

**1.一个 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](https://github.com/pingcap/pd/blob/master/conf/config.toml#L54) 的配置

**2.一个 Raft Group 中的多个 Replica 不在同一个位置**

注意第二点，『一个 Raft Group 中的多个 Replica 不在同一个位置』，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

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 [lables](https://github.com/pingcap/tikv/blob/master/etc/config-template.toml#L16) 并且通过在 PD 上配置 [location-labels](https://github.com/pingcap/pd/blob/master/conf/config.toml#L59) 来指明哪些 lable 是位置标识，需要在 Replica 分配的时候尽量保证不会有一个 Region 的多个 Replica 所在结点有相同的位置标识。

**3.副本在 Store 之间的分布均匀分配**

前面说过，每个副本中存储的数据容量上限是固定的，所以我们维持每个节点上面，副本数量的均衡，会使得总体的负载更均衡。

**4.Leader 数量在 Store 之间均匀分配**

Raft 协议要读取和写入都通过 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。

**5.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。

**6.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。

**7.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(比如已经停服务升级，增加新节点，希望尽快调度)，那么可以通过 pd-ctl 手动加快调度速度。

**8.支持手动下线节点**

当通过 pd-ctl 手动下线节点后，PD 会在一定的速率控制下，将节点上的数据调度走。当调度完成后，就会将这个节点置为下线状态。

### 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 的心跳包收集信息，获得整个集群的详细数据，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 都会检查是否有对这个 Region 待进行的操作，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 自己根据当前自身状态来定。

### 总结

到这里可以发现TiDB的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。TiDB的技术内幕就给大家讲解到这里，希望大家能够对整个 TiDB 的基本概念和实现原理有了解，更多的内容可以查看官网的技术文章

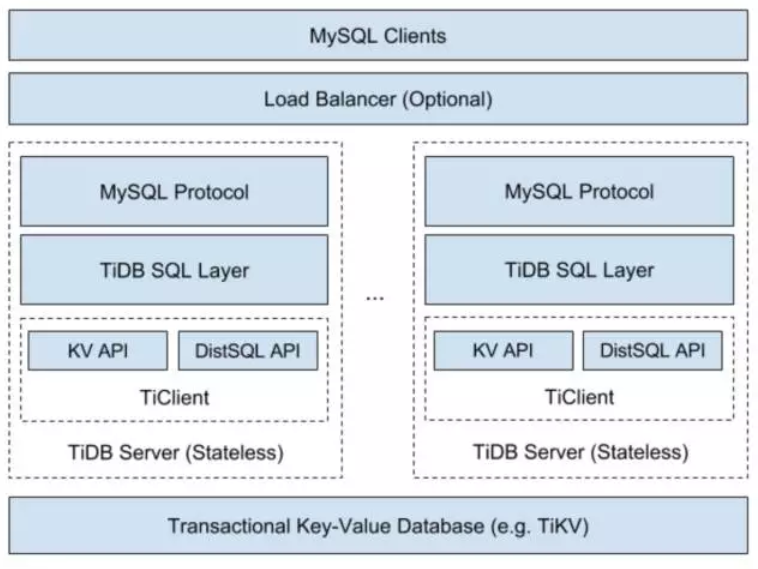
# TiDB源码剖析

<https://pingcap.com/docs-cn/stable/overview/#项目源码>

[https://pingcap.com/blog-cn/#TiDB-%E6%BA%90%E7%A0%81%E9%98%85%E8%AF%BB](https://pingcap.com/blog-cn/" \l "TiDB-%E6%BA%90%E7%A0%81%E9%98%85%E8%AF%BB)

本文面向 TiDB 社区开发者，主要介绍 TiDB 的系统架构、代码结构以及执行流程。 目的是使得开发者阅读文档后，可以对 TiDB 项目有一个整体的了解，更好的参与进来。首先会介绍一下大体的结构以及 Golang 包的结构，然后会介绍内部的执行流程，最后会对优化器、执行器这两个最重要的组件做一些说明。

## **系统架构**



TiDB Server 在整个系统中位于 Load Balancer(或者是 Application) 与底层的存储引擎之间，主要部分分为三层：

* MySQL Protocol 层

接收 MySQL Client 的请求，解析 MySQL Protocol 的包并转换为 TiDB Session 中的各种命令；处理完成后，将结果结果为 MySQL Protocol 格式，返回给 Client。

* SQL 层

解析并执行 SQL 语句，制定查询计划并优化，生成执行器并通过 KV 层读取或写入数据，最后返回结果给 MySQL Protocol层。这一层是重点，后面会详细介绍。

* KV 层

提供带事务的（分布式/单机）存储，在 KV 层和 SQL 层之间，有一层抽象，使得 SQL 层能够忽略下面不同的 KV 存储的差异，看到统一的接口。

## **代码结构**

这里首先会把所有的 package 列出来，然后介绍其主要功能。

* tidb 这个包可以认为是 MySQL Protocol Layer 和 SQL Layer 之间的接口，主要的文件有三个：
  + session.go: 每一个 session 对象对应一个 MySQL Client 的 connection，MySQL Protocol 层负责管理 connection 与 Session 之间的绑定关系，并且各种 MySQL 查询/命令都是调用 Session 的接口来执行。
  + tidb.go:一些函数，供 session.go 调用
  + bootstrap.go: 当 TiDB Server 启动后，如果发现系统未经初始化，会执行初始化流程，详细信息会在下面的章节中介绍。
* docs 一些简略的文档，更详细的文档参见中文文档（https://github.com/pingcap/docs-cn）以及英文文档（https://github.com/pingcap/docs）。
* executor TiDB 执行器，SQL 语句最终会转化为一系列执行器（物理算子）的组合。这个包对外暴露的最主要的接口是 Executor:

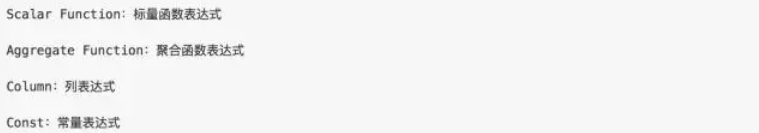


各种执行器都会实现这个接口，TiDB 的执行引擎采用 Volcano 模型，执行器之间通过上述三个接口交互，每一个执行器只需要通过 Next 接口从其他执行器获取数据以及通过 Schema 接口获取数据的元信息。

* plan 这里是整个 SQL 层的核心，SQL 语句解析成 AST 之后，在这个包中制定出查询计划，并对查询计划进行优化，包括逻辑优化和物理优化。 这个包中还包括下面几个功能：
  + validator.go：对 AST 进行合法性验证
  + preprocess.go： 目前只有 name resolve 这一项 resolver.go：名称解析，将 SQL 语句中的标识符（database/table/column/alias）解析并绑定到对应的 column 或者是 Field 上。 typeinferer.go：推导结果类型。对于 SQL 语句，不需要执行即可推导出结果的类型。 logical*plan*builder.go: 制定出优化的逻辑查询计划 physical*plan*builder.go: 根据逻辑查询计划制定出物理查询计划
* privilege 权限控制相关接口，具体实现在 privilege/privileges 包中
* sessionctx 存放 session 中的状态信息，比如 session variable 信息，这些信息可以在 session 中获取，放在单独的包中主要是理清依赖关系，避免循环依赖问题。
* table table接口，对数据库中的表做了一层抽象，提供很多对表的操作（如获取表的 column 信息、读取一行数据等），具体实现在 table/tables 中。另外这里还有对于 Column 以及 Index 的抽象。
* tidb-server tidb-server 程序的 main.go，主要是启动为 server 的代码。
* server MySQL Procotol 层的实现，主要工作是解析协议、传递命令/Query。
* ast sql 文本会被解析为一棵抽象语法树，ast 中定义了树的数据结构，每个节点都需要实现 visitor 接口，然后调用树中节点的的 Accept 方法，对树进行遍历。 如果需要添加新的语法支持，那么除了在 parser 中增加规则之外，还需要在这个包中添加数据结构。
* ddl 异步 Schema 变更相关代码，类似 Google F1 的论文实现。
* domain domain 可以认为是一个存储空间，可以在其中创建数据库、创建表，不同的 domain 之间，可以存在相同名称的数据库，有点像 Name Space。domain 上会绑定 information schema 信息。
* expression 表达式的定义，最重要的接口是 :



目前实现这个接口的表达式包括：



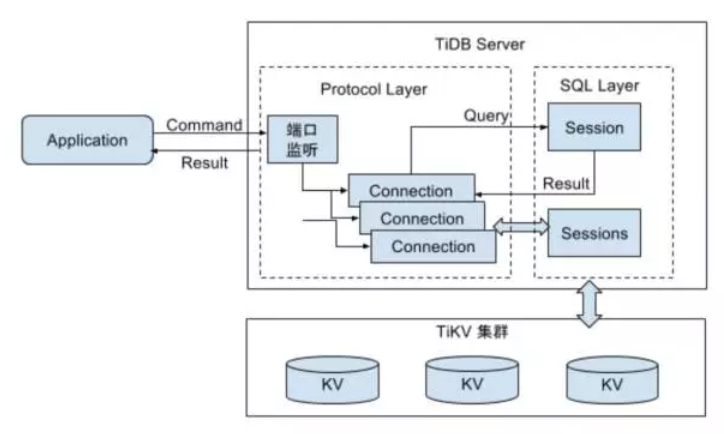
* evaluator 表达式求值相关逻辑，所有的表达式求值方法都在这里。
* infoschema InformationSchema 的实现，提供 db/table/column 相关信息。
* kv KV 相关的接口定义和部分实现，包括 Retriever / Mutator / Transaction / Snapshot / Storage / Iterator 等。对下层的多种KV存储做了一个统一的抽象。
* model TiDB 支持的 ddl / dml 相关的数据结构，包括 DBInfo / TableInfo / ColumnInfo / IndexInfo 等
* parser 语法解析模块，主要包括词法解析 (lexer.go) 和语法解析 (parser.y)，这个包对外的主要接口是 Parse()，用于将 SQL 文本解析成 AST。
* store 底层 KV 存储的实现，如果要接入新的存储引擎，可以将其进行包装，代码放在这个包下面，目前接入两个引擎：分布式引擎（tikv）和单机引擎（localstore/{goleveldb/boltdb}）。这里的新引擎需要实现 kv 包中定义的接口。 关于kv和store，可以参考 TiDB 存储引擎接入指南（https://github.com/ngaut/builddatabase/blob/master/tidb/storage.md）
* terror 定义了 TiDB 的 error 体系。
* context context接口，session 是 context 接口的一个实现，这里抽象出接口来，主要是避免循环依赖。session 各种各种状态信息都通过这个接口存取。
* inspectkv TiDB SQL Data 和 KV 辅助 check 包，以后会用于外部对于 TiDB KV 的访问，将被重新定义和扩展
* meta TiDB 的 meta 数据相关常量定义以及常用函数定义。meta/autoid 定义了一个用于生成全局唯一session内自增 ID 的API，meta 信息依赖于这个工具。
* mysql MySQL 相关的常量定义。
* structure 在 key-value上做了一层封装，支持更丰富的支持 Transaction 的 KV 类型，包括 string / list / hash 等。主要在异步 Schema 变更中使用。
* util 一些工具类，这里有一个包比较重要，就是 types 包，里面有很多和类型的定义以及对各种类型对象的操作。
* distsql 分布式 SQL 执行接口，如果下层的存储引擎支持分布式执行器，可以通过这个接口发送请求，后面会详细介绍这个模块。

## **协议层**

协议层是和应用交互的接口，目前 TiDB 只支持 MySQL 协议，相关的代码都在 server 包中。

这一层的主要功能是管理客户端 connection，解析 MySQL 命令并返回执行结果。这一层是按照 MySQL 协议实现，具体的协议可以参考：https://dev.mysql.com/doc/internals/en/client-server-protocol.html

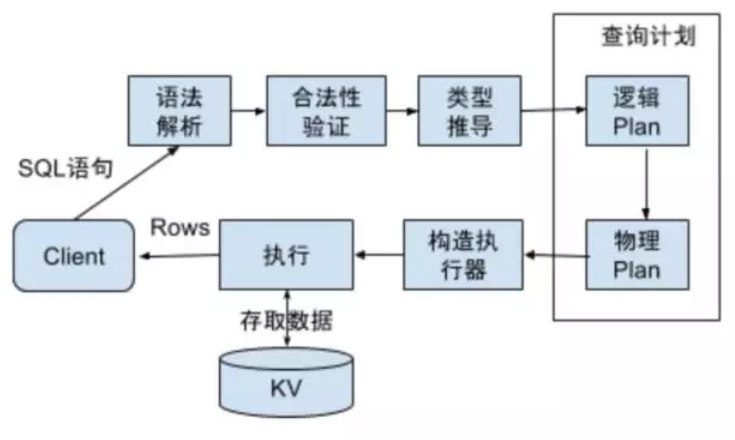
单个 connection 处理命令的入口方法是 clientConn 类的 dispatch 方法，这里会解析协议并调用不同的处理函数。



## **SQL层**

经过上面的章节已经了解了 TiDB 整体框架以及各个包的细节，本章开始讲解核心章节，对 TiDB 的 SQL 层如何 work 做一个简要的介绍。这里忽略具体的 KV 层，只关注 SQL 层。希望通过本章，读者可以了解一条 SQL 语句是如何从一段文本变成执行结果集。整个流程中的详细过程会在接下来的几个章节中说明。

大体上讲，一条 SQL 语句需要经过，语法解析-->合法性验证-->制定查询计划-->优化查询计划-->根据计划生成查询器-->执行并返回结果 等一系列流程。TiDB 的执行流程以这个为基础，流程图如下：



整个流程的入口在 tidb 包的 session.go 中，TiDB-Server 调用 Session.Execute() 接口，输入为文本格式的 SQL 语句，实现在 session.Execute()。

首先会调用 Compile() 对 SQL 语句进行语法解析（tidb.Parse()），解析后会得到一个 stmt 的列表，这里每条语句都是一个 AST（抽象语法树），每一个语法单元都是数的一个 Node，结构定义都在 ast 包中。

得到 AST 之后，调用 executor 包中的 Compiler，输入 AST，得到执行器（Compiler.Compile()）。在这个过程中，会完成合法性验证、制定计划以及优化查询计划。

进入 Compiler.Compile() 函数，首先会调用 plan.Validate() 对语句进行合法性验证（见 plan/validator.go），然后进入 Preprocess 流程，目前这个阶段，Preprocess 只做了名称解析工作，及将 SQL 语句中的提到的 column 名或者 alias name 绑定到对应的 field上。比如”select c from t;”这个语句，会将 c 这个名字绑定到 t 这个表的对应列上（具体的实现见 plan/resolver.go）。之后就进入 optimizer.Optimize()。

Optimize() 方法中，首先对 AST 中各个node的结果进行推导，如 "select 1, ’xx‘, c from t;”，对于 select fields，第一个 field是”1“，其类型为 Longlong，第二个 field 是”’xx‘“, 其类型为 VarString，第三个field是 “c” 其类型为表 t 中 column c 的类型。注意这里除了类别之外还有 charset 等信息，都要进行推导，具体实现见 plan/typeinferer.go。完成类型推导后，进行逻辑优化（planBuilder.build()），主要工作是根据代数运算，对 AST 进行等价变换，化简 AST。比如 ”select c from t where c > 1+1\*2;“ 可以等价变换为”select c from t where c > 3;“。

逻辑优化完成后，进行物理优化，生成查询计划树，并利用索引、根据一些 rule 以及 cost 模型，对树进行变换，减少查询过程的代价，入口在 plan/optimizer.go 中的 doOptimize()方法。

生成查询计划后，会转换为执行器，通过 Exec 接口执行得到 RecordSet对象，对其调用 Next() 方法获取查询结果。

## **优化器**

优化器是数据库的核心，决定了每条语句如何执行。如果说数据库是一支军队，那么优化器就是这支军队的主将、军师，需要运筹帷幄，决胜于千里之外。俗话说一将无能累死三军，同样的一条语句，选择不同的查询计划，最终的运行时间可能会相差很大。对优化器的研究一直是学术界比较活跃的领域，优化是永无止境，可以说在这块投入多大的精力都不为过。 从优化方法上，大致可以分为三类：

* Rule based optimizer：通过启发式规则对 plan 进行优化
* Cost based optimizer：通过计算查询代价对 plan 进行优化
* History based optimizer：通过历史查询信息对 plan 进行优化

基于规则的优化器比较容易实现，只要选取一些常用的规则，就可以对大多数常用的查询有较好的效果。但是其缺陷也比较明显：无法根据数据的真实情况，选择最优的方案。比如对于查询语句 “select \* from t where c1 = 10 and c2 > 100” 在选择索引时，如果只根据规则，那么一定是选择 c1 上面的索引进行查询，但是如果 t 中 c1 所有的值都是 10，那么这个查询计划就很差。这个时候如果有表中数据分布的信息，对选择好的查询计划很有帮助。

基于代价的优化器复杂一些，其核心问题有两个，一个是如何获取数据的真实分布信息，另一个是如何根据这些信息，估算出某一个查询计划所需的代价。

基于历史信息的查询优化器用的比较少，一般 OLTP 数据库中不会涉及。

TiDB 的优化器相关代码在 plan 包中，这个包的主要工作是将 AST 转换为查询计划树，树中的节点是各种逻辑算子或者是物理算子，对查询计划化的各种优化都是通过调用树根节点的各种方法，递归地对所有节点进行优化，并且会不断的对树中的节点进行转换和裁剪。 最重要的几个接口在 plan.go 中，包括：

* Plan： 所有查询计划的接口
* LogicalPlan：逻辑查询计划，所有的逻辑算子都需要实现这个接口
* PhysicalPlan：物理查询计划，所有的物理算子都需要实现这个接口

逻辑优化的入口是 planbuilder.build()，输入是 AST，输出是逻辑查询计划树。然后在这棵树上进行逻辑查询优化：

* 调用 LogicalPlan 的 PredicatePushDown 接口，将谓词尽可能下推
* 调用 LogicalPlan 的 PruneColumns 接口，将不需要的列裁减掉
* 调用 aggPushDownSolver.aggPushDown，将聚合算子下推到 Join 之前

拿到逻辑优化后的查询计划树之后，会进行物理优化，代码的入口是对逻辑查询计划树的根节点调用 convert2PhysicalPlan(&requiredProperty{})，其中 requiredProperty 是对下层返回结果顺序、行数的要求。 逻辑查询计划树从根节点开始，不断的递归调用，将每个节点从逻辑算子转成物理算子，并且根据每个节点的查询代价找到一条比较好的查询路径。

## **执行器**

虽然优化器是最核心的组件，但是缺少优秀的执行器，依旧无法构成一个优秀的数据库。同样以军队为例，执行器就是军队中冲锋陷阵的士兵。再厉害的将军，如果没有一群能征善战的士兵，同样无法打胜仗。

相比 MySQL，TiDB 的执行器有两个优点，第一是整个计算框架是一个 MPP 的框架，计算会在多台 TiKV 以及 TiDB 节点上进行，尽可能提高效率和速度；第二是单个算子会尽可能并行，比如 Join/Union 等算子，会启动多个线程同时计算，整个数据计算流程构成一个 pipeline，尽可能缩短每个算子的等待时间。所以 TiDB 在处理大量数据时，比 MySQL 表现好。

执行器最重要的接口在 executor.go 中:

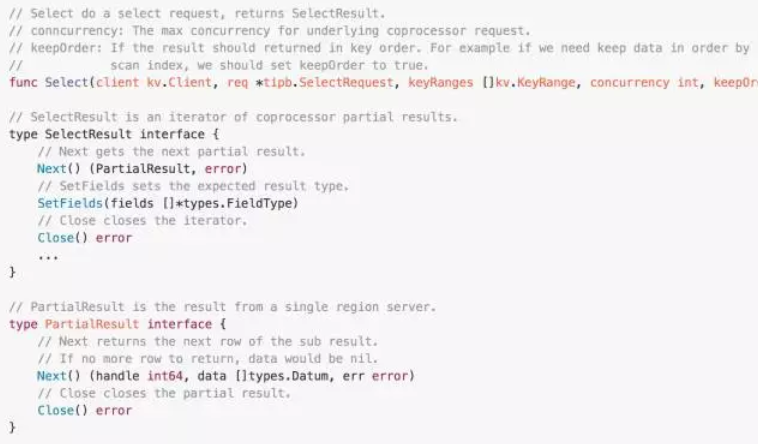


通过优化器得到的物理查询计划树会转换为一个执行器树，树中的每个节点都会实现这个接口，执行器之间通过 Next 接口传递数据。

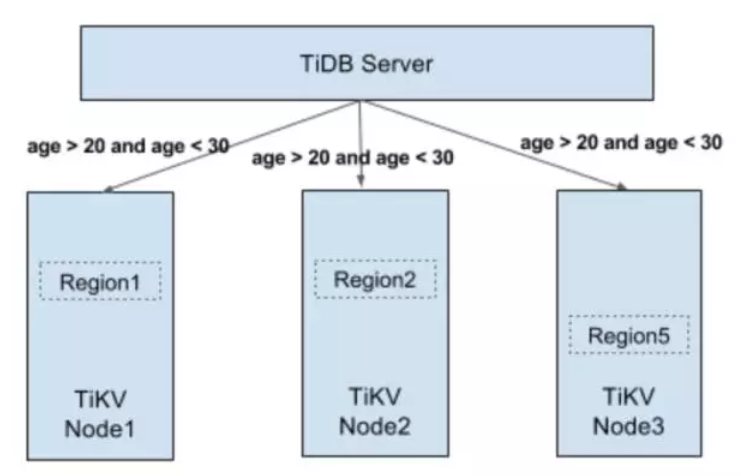
比如 “select c1 from t where c2 > 10; ” 最终生成的执行器是 Projection->Filter->TableScan 这三个执行器，最上层的 Projection 会不断的调用下层执器的 Next 接口，最终调到底层的 TableScan，从表中获取数据。

## **分布式执行器**

TiDB 作为分布式数据库，内置一个分布式的计算框架，Query 在执行的时候，会尽量分布式+并行。这个框架的入口在 distsql 包中，最重要的是下面这个 API 以及两个 Interface：



DistSQL 提供的对外最重要的一个接口是 Select()。第一个参数 kv.Client，只要 KV 引擎满足带事务、满足 KV 接口，并且满足这个 Client 的一些接口，就可以接入 TiDB。目前有一些其他的厂商和 TiDB 合作开发，在其他的 KV 上 run TiDB，并且支持分布式的 SQL。第二个参数是 SelectRequest 。这个东西是由上层执行器构造出来的，它把计算上的逻辑，比如说一些表达式要不要排序、要不要做聚合，所有的信息都放在 req 里边，是一个 Protobuf 结构，然后发给 Select 接口，最终会发送到进行计算的 TiKV region server 上。



分布式执行器在 TiDB 端的主要工作是做任务的分发和结果的收集。Select 接口返回一个数据结构，叫 SelectResult ，这个结构可以认为是一个迭代器，因为下层是有很多 region server，每个节点返回的结果是一个 PartialResult。在这些部分结果之上封装了一个 SelectResult ，就是一个 PartialResult 的迭代器。通过这个的 next 方法可以拿到下一个 PartialResult 。 SelectResult 的内部实现可以认为是个 pipeline。TiDB 会并发地向各个 region server 发请求，并且按照预定的顺序返回结果给上层，这里的顺序是由下层结果返回顺序以及 Select 接口的 KeepOrder 参数共同决定。

这部分相关的代码可以参看 distsql 包以及 store/tikv/coprocessor.go。

# TiDB大厂实践

<https://pingcap.com/cases-cn/user-case-beijing-bank/>

