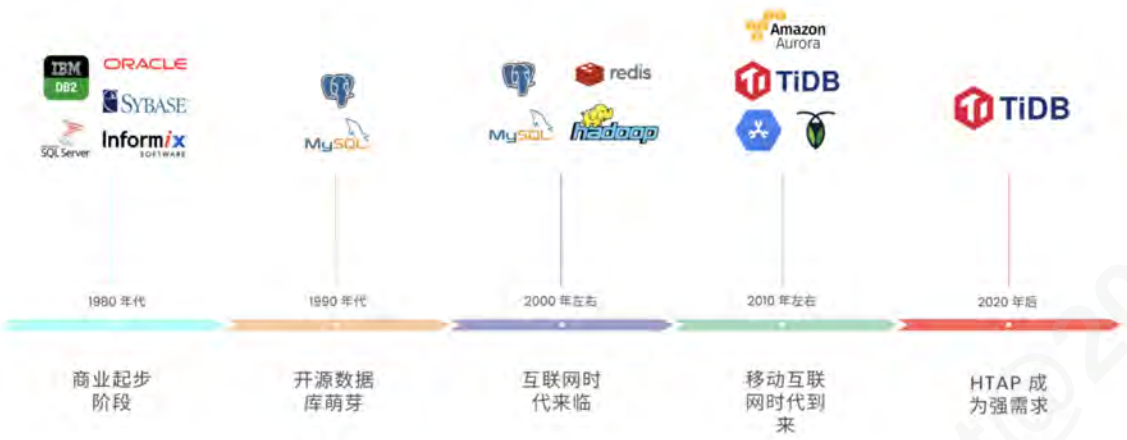
第 1 课：数据库、大数据发展简史与趋势

数据库系统发展历史表

自上世纪 70 年代， EF codd 提出了关系型数据库模型开始，数据库已经发展了将近半个世纪。可以说是一个历史悠久的计算机学科，在这个半个世纪里，这个学科发生了很多变化，但也有一些共性的驱动在里面。我们尝试一起探讨下数据技术发展的内在驱动、以及展望趋势。

我们先按照时间维度，回顾数据库的发展简史：

- 上世纪 80 年代，也就是从提出关系型数据库模型后 10 多年，关系型数据库产品逐步完成了工程与产品实现， Oracle、IBM DB2、Sybase 以及 SQL Server 和 Informix 第一批关系型商业数据库开始出现。
- 到 90 年代中后期，MySQL、PostgreSQL 这类开源数据库开始萌芽。
- 上世纪末到本世纪初， IT 与通信两门技术发生第一次碰撞，开启互联网时代，数据量开始爆发增长，快速发展的各类互联网公司，更加青睐 MySQL、PostgreSQL 这类开源数据库，同时也给这类开源数据库带来丰富的产品生态，比如基于开源数据库的各种分库分表方案产生。同期，2006 年谷歌的三驾马车（GFS、Bigtable、Mapreduce）开启了大数据时代，涌现了 Hadoop（Hbase）、Redis 等 NoSQL 大数据生态。
- 2010 年后，IT 技术与通信技术发生第二次碰撞，4G 网络开启了移动互联网时代，在这个时代，产生了很多业务与场景创新，底层数据技术更是进入前所未有的快速通道，有几个特点：
  - 数据技术栈与方向百花齐放。
  - 集成了分布式技术与关系模型的 NewSQL 数据库开始出现，代表产品有 Spanner、TiDB 等。
  - 云计算与数据库、大数据发生融合；很多云原生数据库出现，云计算不仅给数据技术带来新的变革，也给数据库厂商提供了更高效的服务交付、商业模式，各种 DBaaS（Database-as-a-Service）产品涌现。
- 2020 年后，各行各业都逐步进入数字化时期，虽然从技术创新角度，会有越来越多的数据技术栈与产品将出现；但这么多的技术栈也会大大增加使用成本，从用户需求的角度看，能不能将在线处理业务与分析业务进行整合，也就是我们经常说的 HTAP ，（Hybrid Transactional/Analytical Processing） 作为一种统一的数据服务变成了一个强需求。



数据库技术发展内在驱动

那么，在这个数据库发展的简史背后，驱动数据库技术发展与创新的内因是什么呢？
数据最终服务于上层应用，所以我们认为数据技术发展的驱动力总结起来，主要有三点：

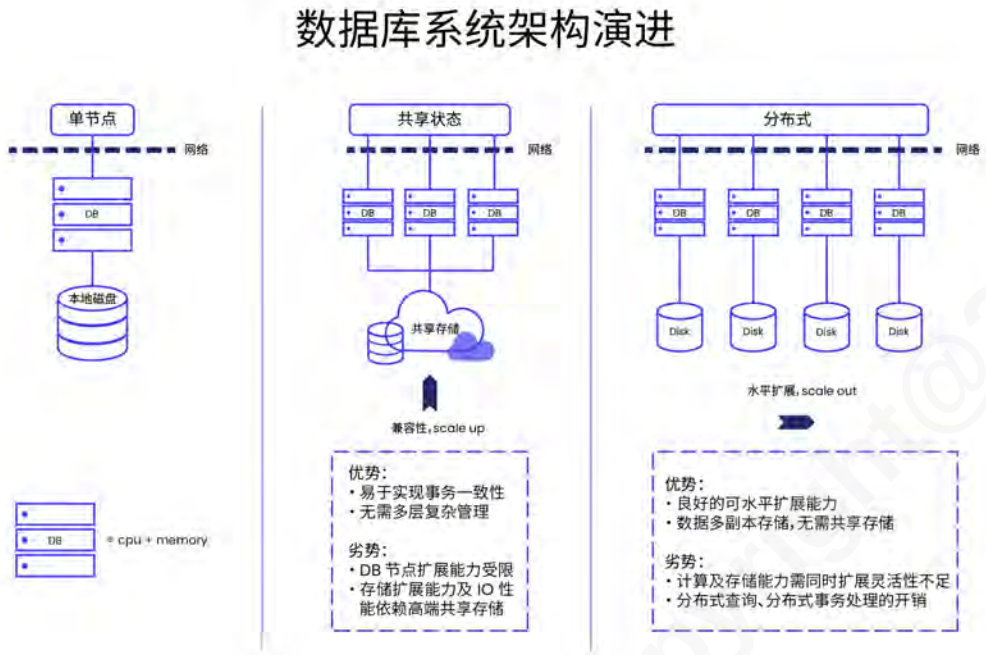
- **业务发展**
- **场景创新**
- **硬件与云计算的发展**

业务发展需求最主要体现在**数据容量持续爆发增长**，数据容量不仅包括数据存储量，更重要的包括数据的吞吐量、读写 QPS 等。
场景创新主要体现**数据的交互效率与数据模型多样性**，比如查询语言、计算模型、数据模型、读写延时等。
硬件与云计算主要体现在**数据架构的变迁**上，比如读写分离、一体机、云原生等。

数据容量催生数据架构演进

从数据容量的角度来看，数据库系统总是被动去适配业务数据量增长。

- 在早期，业务数据量基本在几百 GB 时代，单节点本地磁盘是最高效的数据架构，单机关系型数据库是主流。
- 随着数据量增加，人们可以通过加入更多 CPU、更大内层、更快硬盘等升级硬件的方式来应对，也就是大家说的纵向扩展 Scale up，包括引入网络存储（SAN）Share Disk 成了一种代表性的架构。
- 但硬件的升级总是有上限，当数据量的增加速度远远超过了硬件升级速度，Scale up 失效，只能通过分布式的（Share Nothing） 架构来应对，也就是我们说的横向扩展 Scale out。所以这个时期，分布式逐渐成为海量数据技术的代名词。



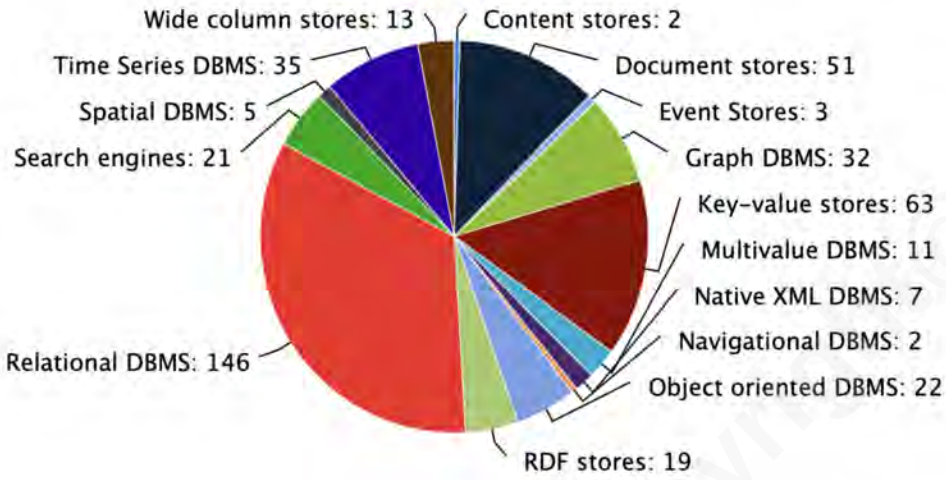
数据模型与交互效率的演进

我们在“**数据模型与交互效率**”这个维度来看：

除了，上面提到的对于水平扩展的需求外，还有关于数据存储结构、事务的需求也是我们要解决的问题。

- 早期的单机数据库查询使用标准 SQL 语言，支持事务并绝大部分采用结构化的方式进行存储。
- 2000 年后，特别是伴随着移动互联网的发展，在很多场景下，结构化的存储方式慢慢显得力不从心，诸如 Key-value、文本、图、时序数据、地理信息等非结构化数据的需求变得突出起来。于是就衍生出了 NoSQL（Not Only SQL）数据库。

伴随着 NoSQL 数据库的快速发展，又一个新的问题摆在我们面前。那就是大部分 NoSQL 数据库对于事务无法支持。而事务对于 OLTP 场景是不可或缺的，也也就是为什么关系型数据库需求依然是当前的主流，如下图所示。

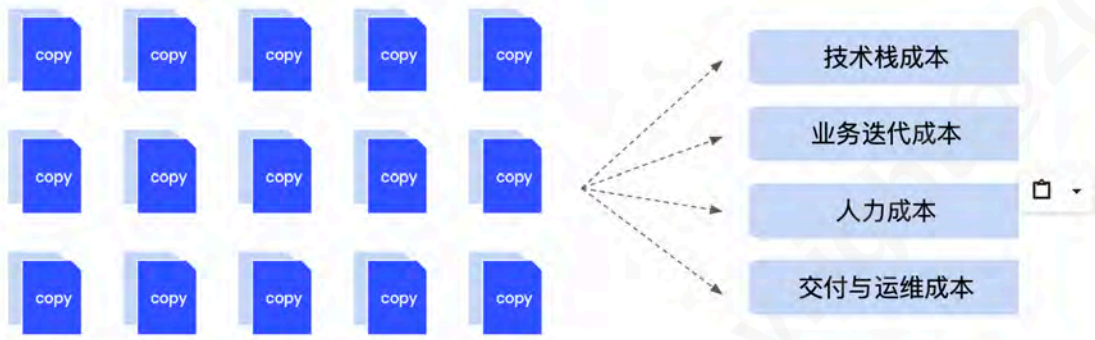


按数据库模型类别细分的 DBMS 流行度排名，
DB-Engines.com，2021.6

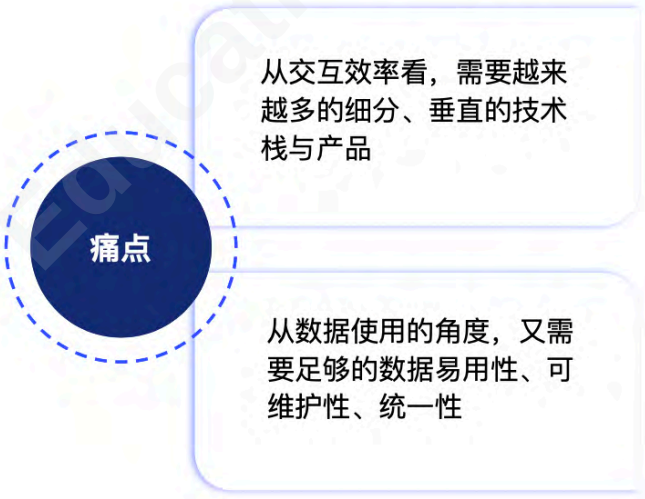
于是，即要求支持事务（OLTP 场景）又同时兼容多种数据模式的可扩展需求就自然提了出来。这就是 NewSQL 的由来。

数据架构，融合还是细分？

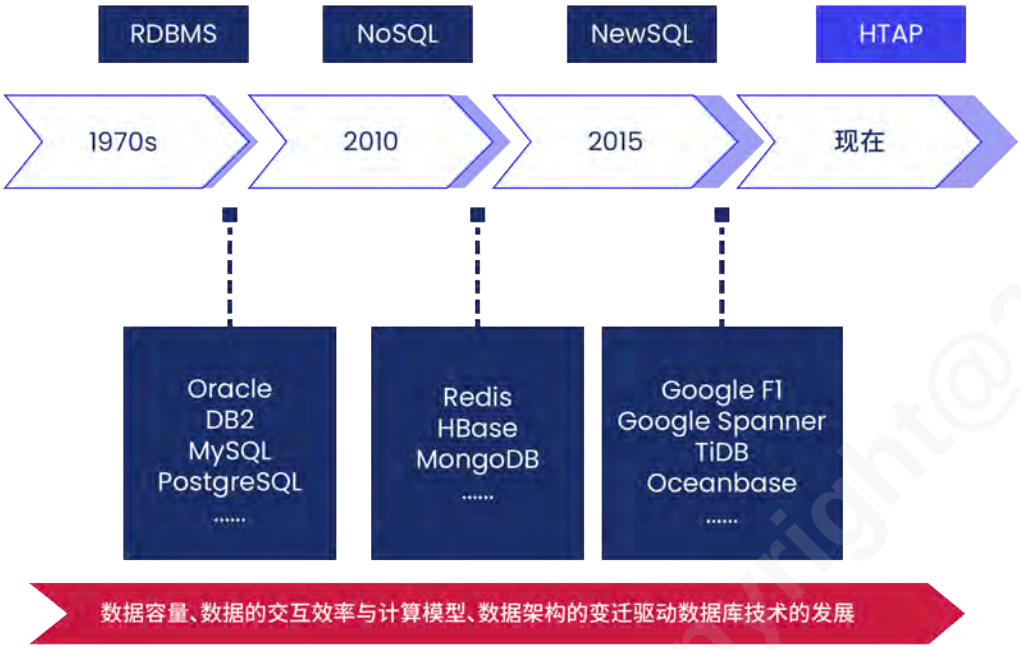
场景创新催生了数据技术栈与产品的多样性。但同时，数据技术栈与产品多样性导致了数据副本无序扩大。以一个真实电商平台为例，一份商品订单数据在电商平台的整个业务链条里，存在 30+ 份副本。而数据副本背后导致了巨大的技术栈成本、业务迭代时间成本、人力成本、交付与运维成本等。如何有效减少业务数据副本将是一个越来越重要的需求。



一方面从交互效率看，需要越来越多的细分、垂直的技术栈与产品；另一方面，从数据使用的角度，又需要足够的数据易用性、可维护性、统一性。这两个真实的需求形成一个巨大的矛盾与痛点。企业数据化加速，业务创新加速，数据将会持续以指数级增长，如何解决这一个痛点，是未来数据技术很重要的一个课题，或许未来趋势将是数据技术的细分与数据服务的融合。



总之，数据模型、数据结构、存储算法、复制协议、算子、计算模型、硬件构建了数据基础技术要素，而数据技术栈方向或者数据产品的本质是：**应对不同业务场景，基于这些相对固定数据基础技术，进行数据架构 Trade Off（选择与平衡）**，我们会在接下来的课程详细解读，在数据架构里无处不在的 Trade Off 。



课后作业

1. 2006 年谷歌的三驾马车开启了大数据时代，请问这三架马车分表是什么？

第 2 课： 分布式关系数据库的发展

分布式系统是数据爆发增长的强需

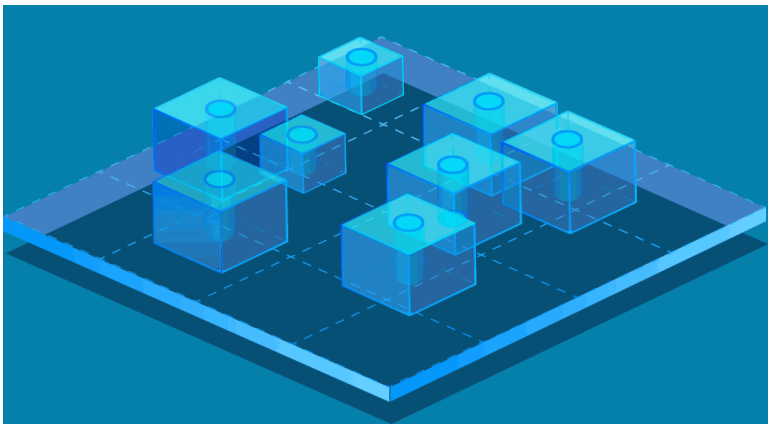
1965 年高登·摩尔（Gordon Moore）提出了摩尔定律。核心观点是：集成电路上可以容纳的晶体管数目在大约每经过 18 个月便会增加一倍。换言之，处理器的性能每隔两年翻一倍，这就意味着，集中式系统的运算能力每隔一段时间能提升一倍。这里面隐含了另外一层意思：“**如果你的系统需承载的计算量的增长速度大于摩尔定律的预测，集中式系统将无法承载你所需的计算量**。”这就是推动分布式系统发展的内因，另外，分布式系统的发展还有一个经济原因，用相对廉价机器的集合组成的分布式系统，除了可以获得超过 CPU 发展速度的性能外，并且具有更好的弹性，可以根据需要弹性增加或者减少机器的数量。
那么，从 2008 年后，数据与其背后的计算需求呈现了爆发增长，并且在可以预测的未来还会快速。基于此，从 2008 年后，分布式系统技术在工程界逐渐涌现了很多技术产品。

分布式系统概述

分布式系统定义

那么，什么是分布式系统呢？我们先看下维基百科的定义：
分布式系统是一种其组件位于不同的联网计算机上的系统，然后通过互相传递消息来进行通信和协调。为了达到共同的目标，这些组件会相互作用。

怎么理解这句话呢？我们可以简单的理解为：把需要进行大量计算的工程数据分割成小块，由多台计算机分别计算，然后将结果统一合并得出数据结论的科学。本质上就是进行数据存储与计算的“分治”。
另外，假设每台机器出问题的概率是固定的，那么分布式系统里因为存在多台计算机，整体的故障率就会更高，为了解决分布式的高可用问题，就需要将数据进行副本冗余，分治和冗余讲究的都是分散化，最终形成一个完整的系统还需要将它们“协作”起来。获得分布式系统价值的同时，这个“再协作”的过程就是我们相比集中式系统要做的额外工作与成本。



分布式系统的历史

那么我们在看看分布式系统的历史，2006 年，谷歌推出了 GFS（Google File System）、Google Bigtable、Google MapReduce 三大组件。Google File System 解决了分布式文件系统问题，Google Bigtable 解决了分布式 KV（Key-Value）存储的问题，Google MapReduce 解决了在分布式文件系统和分布式 KV 存储上面如何做分布式计算和分析的问题。谷歌三大件在业界诞生以后，很快的衍生了一个以分布式系统为基座的领域，并且针对非结构化、半结构化的海量数据处理系统，我们把他模糊称为 NoSQL（Not Only SQL），现在也有很多很好的商业公司基于 NoSQL 发展，比如说文档数据（MongoDB）、缓存（Redis）等大家平常应用开发都会用到的 NoSQL 系统。但这些只是分布式系统发展的上半场。稍后，我们会介绍分布式系统发展的下半场。

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

分布式技术的主要挑战

在此之前，我们先看看分布式技术的主要挑战，分布式系统的本质是分治与协助，这带来极大的系统复杂度，在技术层面挑战主要包括：

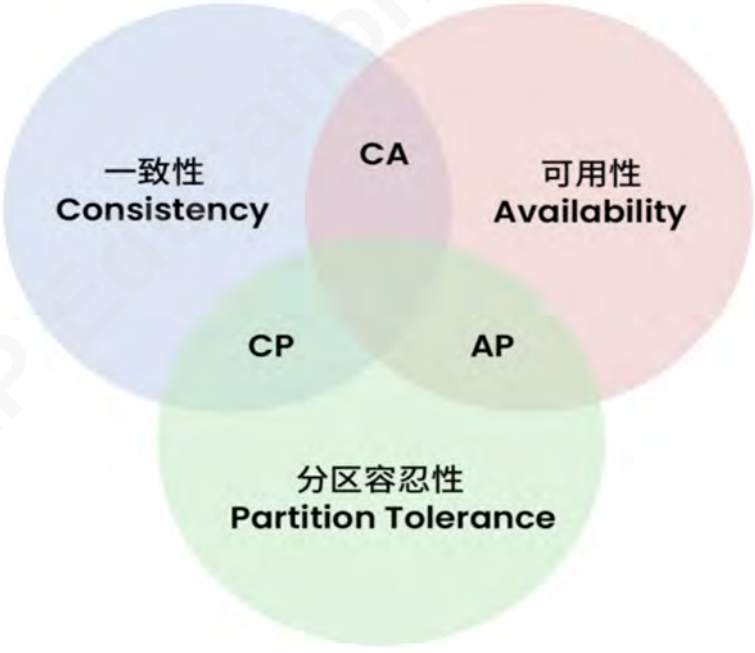
- 如何最大程度的实现分治；
- 如何实现全局的一致性，包括序列化与全局时钟；
- 如何进行故障与部分失效的容错；
- 如何应对不可靠的网络与网络分区；

分布式系统里著名的 CAP 理论

2000 年初，计算机科学家埃里克·布鲁尔提出的一个定理[1]。这个定理讲的是**一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项**。

这就是臭名昭著的帽子理论：

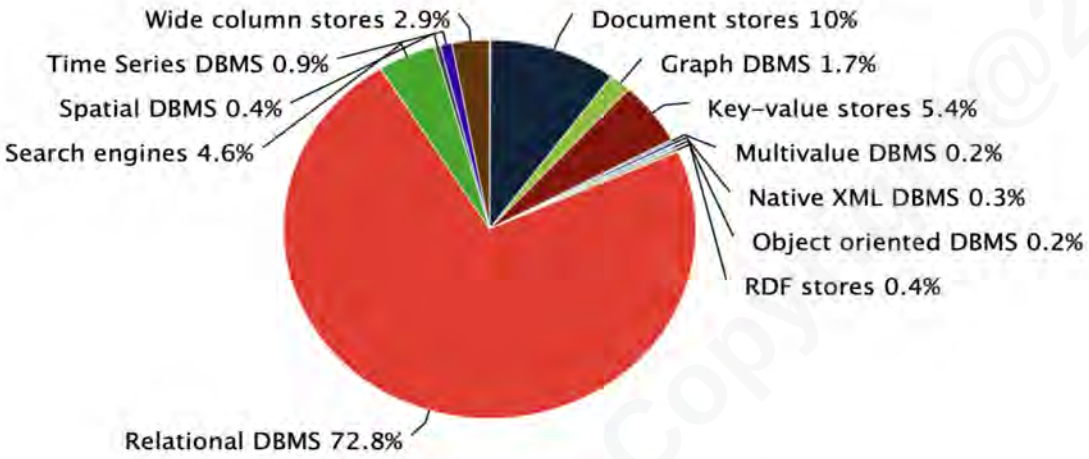
- 一致性指即所有节点在同一时间的数据完全一致。
- 可用性指服务在正常响应时间内一直可用。
- 分区容错性指分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性或可用性的服务。



按照这个定理，所有的分布式系统都会在这三点进行 Trade Off（平衡与选择），那么这里给大家埋一个问题，你能区分出当前主流的各种分布式产品他们如何进行选择的吗？而作为分布式关系型数据库，你觉得它应该取舍，是 CA、CP、AP 呢？

关系型模型与事务

我们再聊一下关系型数据库，虽然现在各种模型的数据库越来越多，但同时，在传统的在线交易场景（OLTP) 中，关系型模式仍然是标准，按照 DB-Engines 的报告，2021 年，关系型数据库在整个数据库模型中权重上仍然占比 72.8%。



各类别数据库模型排名分数（百分比），DB-Engines.com，2021.6

既然是关系型，就一定要具有事务，事务的本质是“并发控制的单元，是用户定义的一个操作序列。这些操作要么都做，要么都不做，是一个不可分割的工作单位”。说白了就是为了保证系统始终处于一个完整且正确的状态。 它要具备大家熟知 ACID 四个特性。

- 原子性（Atomicity）：事务包含的全部操作是一个不可分割的整体，要么全部执行，要么全部都不执行。
- 一致性（ Consistency）：事务前后，所有的数据都保持一致的状态，不能违反数据的一致性检查。
- 隔离性（Isolation）：主要规定了各个事务之间相互影响的程度，主要用于规定多个事务访问同一数据资源，各个事务对该数据资源访问的行为。不同的隔离性是应对不同的现象，比如脏读、可重复读、幻读等
- 持久性（Durability）：事务一旦完成，要将数据所做的变更记录下来(冗余存储或多数据网络备份)。

这里给大家再提一个问题，前面我们提到了 CAP 理论里有一个 Consistency（一致性）、在事务四个特性里，也有一个 Consistency（一致性），那么这两个 Consistency（一致性）描述的是同一个概念吗？

答案是不一样的，简单来说，前者描述的是副本一致性、后者强调是事务一致性，这里面的区别各位可以仔细想想。

NewSQL： 原生分布式关系型数据库

聊到这，我们总结一下，分布式系统通过扩展性应对了数据的容量、关系型模型仍然是市场需求最强数据模型，那有没有哪种数据库来同时解决这两个问题呢？这类数据库就是大家经常提到的 NewSQL，维基百科的定义如下：

> *"NewSQL 是一类关系数据库，它寻求为在线交易处理（OLTP）工作提供 NoSQL 系统的可扩展性，同时维护传统数据库系统的 ACID 保证。"*
>
> —— 维基百科

NewSQL 是个概念，从实现路径上，主要是分布式系统 + SQL + 事务，也就是原生分布式关系型数据库。所以原生分布式关系型数据库，某种意义上讲是 NewSQL 的代名词。
那么这么强大的分布式关系型数据库是如何产生的呢？这个领域的代表产品又是哪个呢？聊到现在，我们这个课程的主角即将登场。

附录

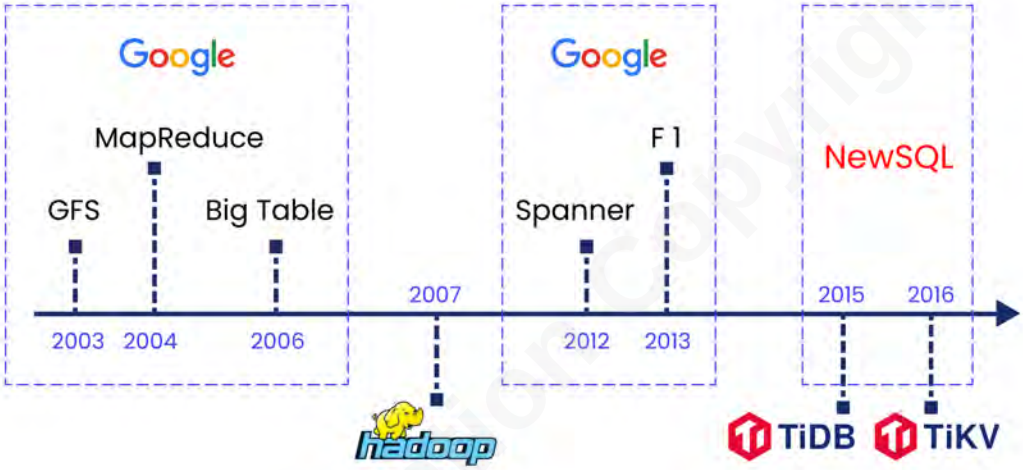
[1] https://zh.wikipedia.org/wiki/CAP%E5%AE%9A%E7%90%86
[2] What's Really New with NewSQL?, Sigmod '16, https://dl.acm.org/doi/abs/10.1145/3003665.3003674

第 3 课：TiDB 产品与开源社区演进

分布式关系型数据库 TiDB 的起源

在分布式关系系统里有二个比较重要的基础：

- 基于 2013 年 Google Spanner [1] / F1 [2] 论文
- 基于 2014 年 Stanford 工业级分布式一致性协议实现 Raft [3] 博士论文



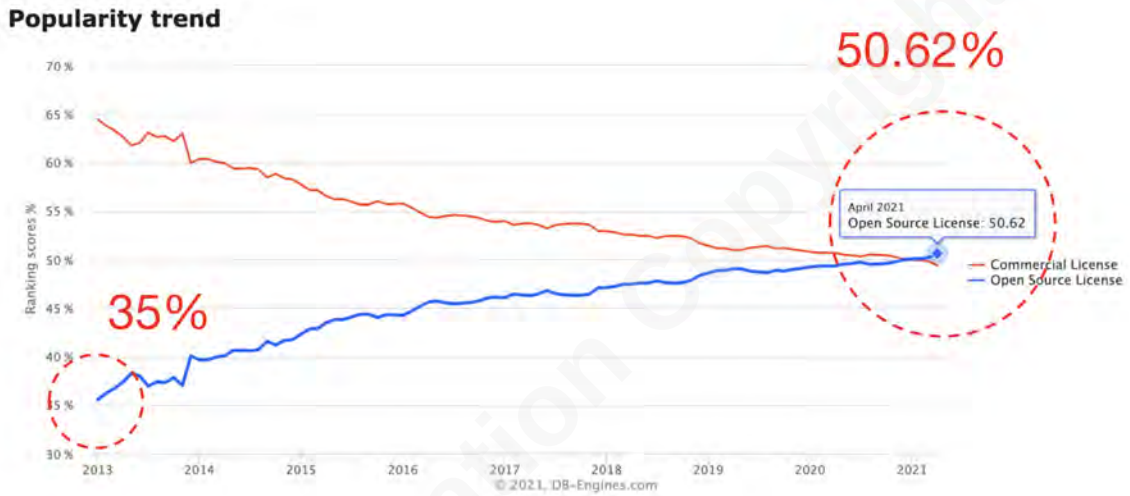
TiDB 就是在这两个理论基础上，通过开源社区的方式，涌现的一款很有代表性的原生分布式关系型数据库。它采取了兼容 MySQL 的策略，最初定位就是一个可以横向扩展的 MySQL，是一款分库分表的替代方案。它有很多标签，开源、原生分布式关系型数据库、HTAP 等等

开源：基础软件成功的一个最佳路径

2015 年 5 月，TiDB 在 GitHub 上开源立项。

开源软件（Open-Source Software）又称开放源代码软件，是一种源代码可以任意获取的计算机软件，这种软件的著作权持有人在软件协议的规定之下保留一部分权利并允许用户学习、修改以及向任何人分发使用该软件。

2010 年后，移动互联网的发展，推动了数据架构升级，也将开源软件带入了一个新的快速发展时期，各种开源数据库软件像雨后春笋涌现，2013-2021 年，8 年时间，开源数据库软件从占比 35% 上升到了 50.62%。



基础软件更倾向通用与标准化，从市场看最终属于寡头竞争，同时软件的生命周期在缩短。所以对每个人细分的基础软件来看，需要在更短的窗口期发展起来，而开源软件模式，具备几个明显的特点：

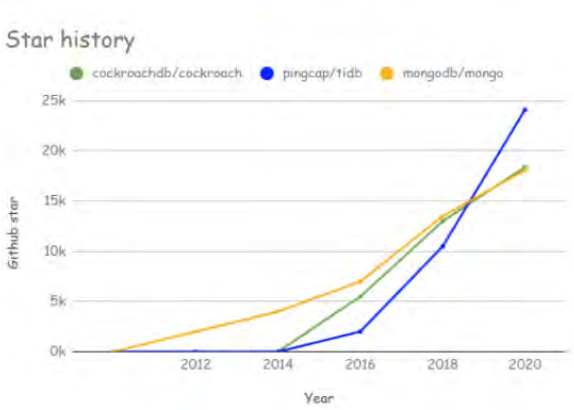
- 开放源码：吸引开发者共同参与，加快产品的迭代速度。
- 开放态度：更容易建立用户信任，在早期获取第一批使用者尤为重要，完成产品市场匹配度验证。
- 开源社区治理：通过社区方式构建人才生态、产品生态。

所以说，开源是基础软件成功的一个最佳路径。

开源助力 TiDB 快速成长

2015 年 TiDB 开始立项，借助于开源与社区，很快吸引了众多开发者的追捧，快速的成长为了全球顶级的开源数据库项目 [4] 。

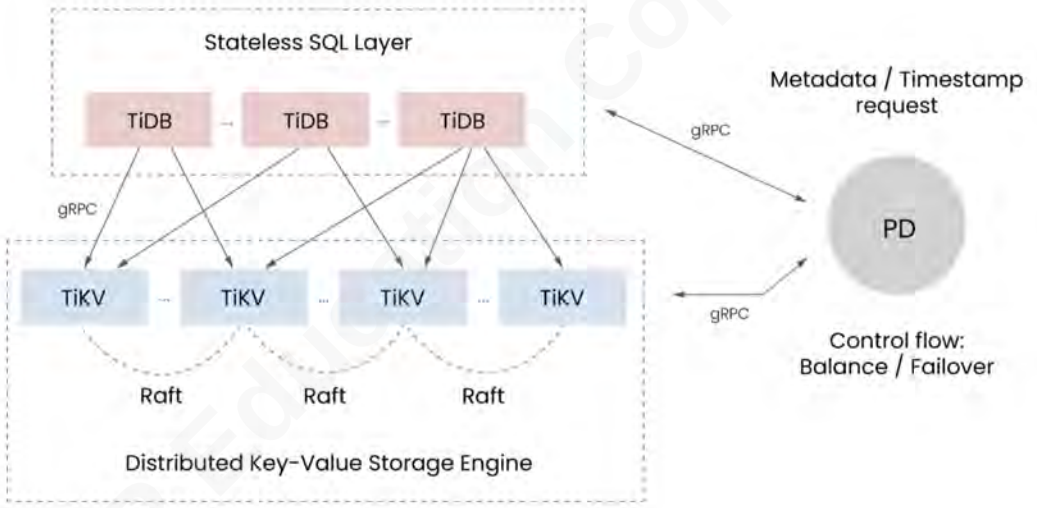
到 2021 年 6 月为止，TiDB 项目在 Github stars 数量已经 24000+，分布式 NewSQL 领域第一。Contributors 数量 1000+，遍布亚、欧、北美、南美、大洋洲。



Github: 24000+ stars, 开源分布式 NewSQL 领域排名第一
https://github.com/pingcap/tidb

Github: TiDB 440+ contributors（TiKV 240+）
https://github.com/pingcap/tidb/graphs/contributors

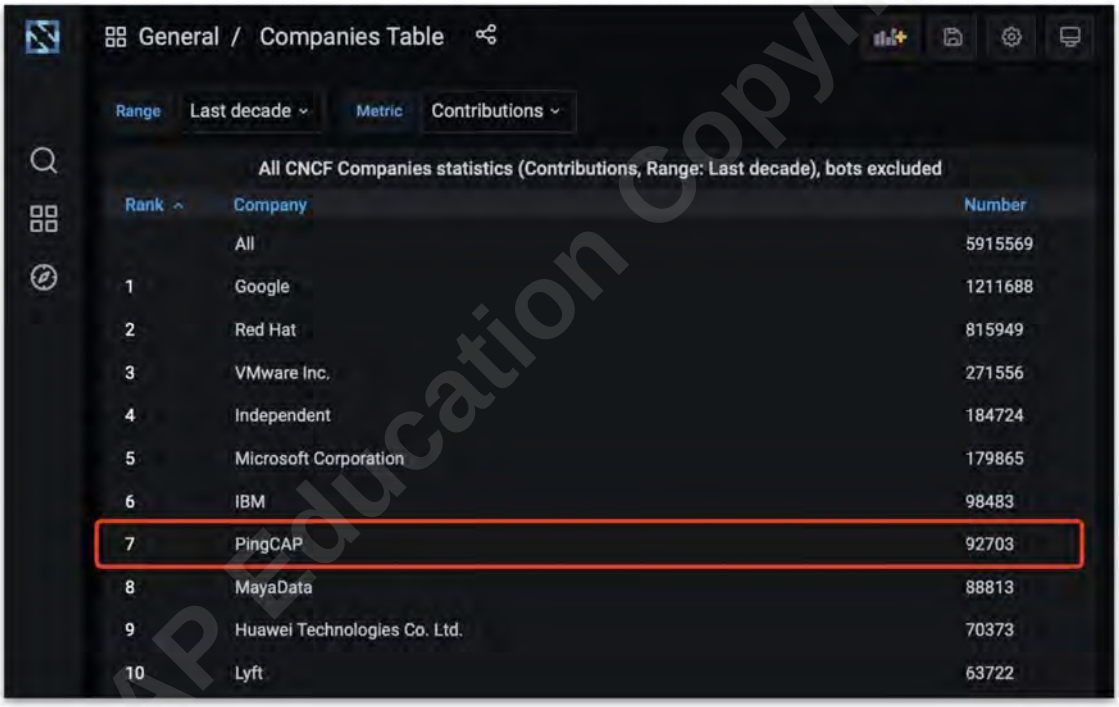
TiDB 是一个计算与存储分离的架构，它的存储引擎叫 TiKV，2018 年 TiKV 捐献给了 CNCF 基金会，并与 2020 年正式毕业。



tikv.org 首页

CNCF 宣布 TiKV 毕业，2020.9.2

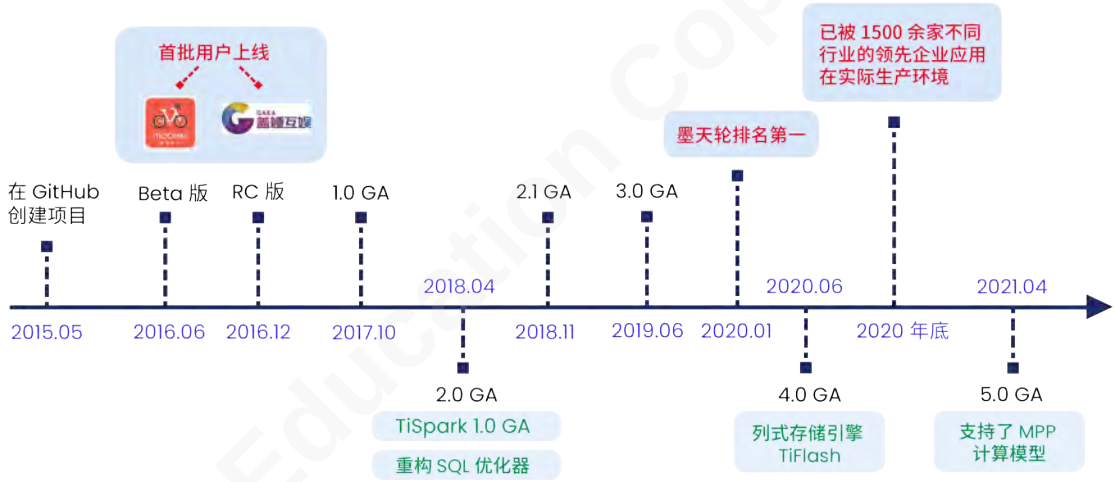
2019- 2020 年，CNCF 统计的社区贡献排名，TiKV 项目母公司 PingCAP 连续两年在全球排名第七。[5]



TiDB 版本迭代

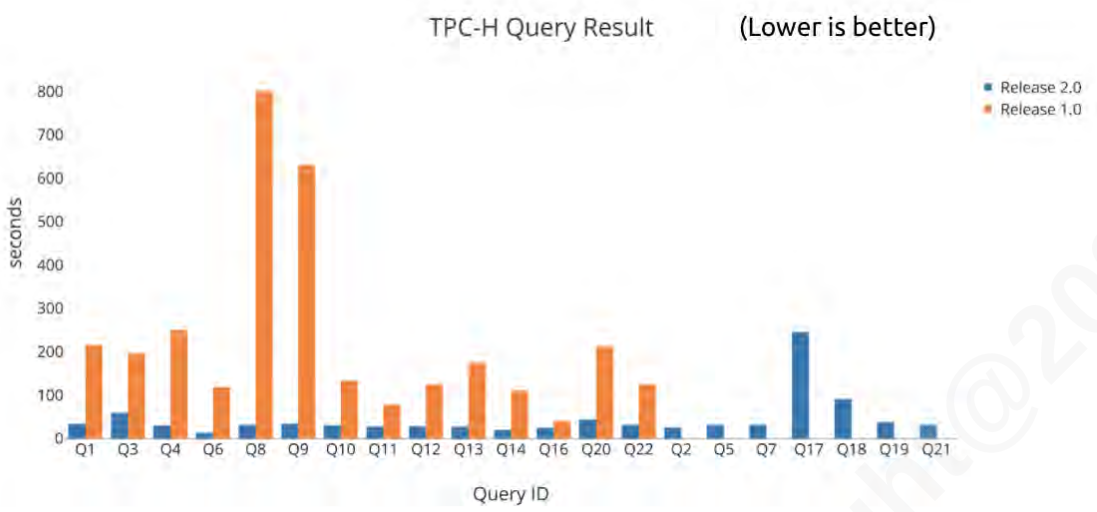
- 2015 年 05 月，在 GitHub 创建项目。
- 2016 年 06 月，发布 Beta 版，2016 年 12 月，发布 RC 版本；同一时期，首批用户开始上线，比如盖娅娱乐、摩拜单车。
- 2017 年 10 月，经过 2 年 6 个月的开发，TiDB 发布 1.0 GA 正式版。

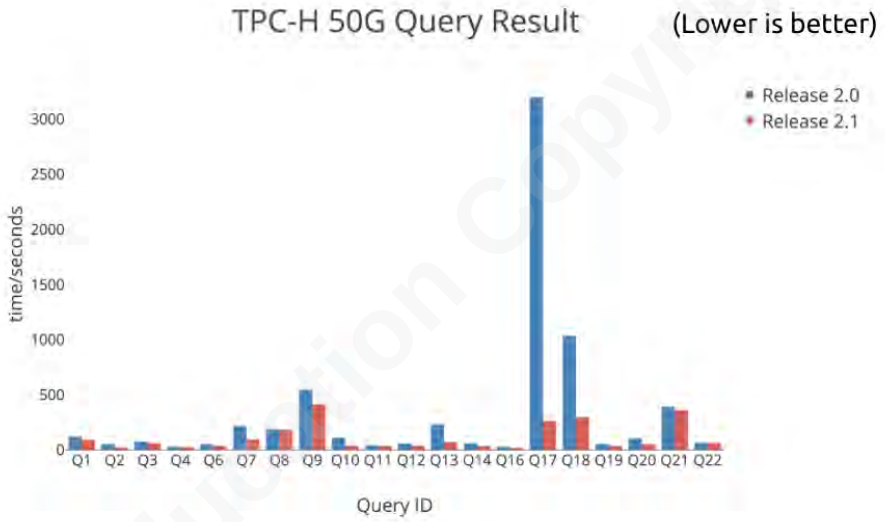
- 2018 年 04 月，发布 2.0 GA 正式版，重构 SQL 优化器，OLAP 性能大幅度提升，同月发布 TiSpark 1.0 GA 正式版。
- 2018 年 11 月，发布 2.1 GA 正式版，性能再次大幅提升。
- 2019 年 06 月，发布 3.0 GA 正式版，显著提升了大规模集群的稳定性和易用性，OLTP 性能大幅提升，增加了窗口函数、视图（实验特性）、分区表、插件系统、悲观锁（实验特性）等新功能。
- 2020 年 1 月，TiDB 在摩天轮数据库排名开启了长期排名第一。
- 2020 年 6 月，发布 4.0 GA 版本，推出了列式存储引擎 Tiflash，向着 HTAP 方向迈出了关键一步。
- 2020 年底，已被 1500 余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。
- 2021 年 4 月，发布了 5.0 GA 版本，TiFlash 支持了 MPP 计算模型，完成了 HTAP 另一块拼图，同步 5.0 在稳定性与性能得到很大提升，一款面向金融核心场景的企业级数据库软件。



TiDB 版本迭代重要功能与提升

以下几张图都显示了随着 TiDB 版本的不断迭代，各项功能不断在提高。





附录：

[1] Spanner: Google's Globally-Distributed Database, https://research.google/pubs/pub39966/
[2] F1: A Distributed SQL Database That Scales, https://research.google/pubs/pub41344/
[3] In Search of an Understandable Consensus Algorithm, https://raft.github.io/raft.pdf
[4] Github:24000+stars，开源分布式 NewSQL 领域排名第一 https://github.com/pingcap/tidb, Github:TiDB 440+ contributors https://github.com/pingcap/tidb/graphs/contributors
[5] CNCF 统计的社区贡献排名 https://all.devstats.cncf.io/d/5/companies-table?orgId=1&var-period_name=Last%20decade&var-metric=contributions

第 4 课：我们到底需要一个什么样的数据库

我们到底需要一个什么样的数据库？

之前的章节我们讨论数据库、大数据的发展历史，也阐述了，数据技术发展内在驱动是：业务发展、场景创新、硬件与云计算发展。那么如果我们在当下重新设计一个数据库，我们主要的考量点是什么？哪些是这个数据库 Must have 功能，哪些又是 Better have 能力呢？

1. 扩展性（Scale-out）：

Scale out 也就是我们经常听到的弹性横向扩展

   a. 里面还包括两个关键点：一是从弹性的角度看，颗粒度越小越好，常见的颗粒度有 Cluster、Database、Schema、表、分表或者分区表。那我们能不能设计一个比分区更小的颗粒度呢？

   b. 数据库写入是昂贵资源，我们一定要面向写入能力的线性扩展机制。

2. 强一致性与高可用性：

   a. 这里的强一致指的是 CAP 理论的一致性，也就是副本一致性，也就是每一份新增数据都会在多个物理节点保存，节点数量和网络延迟是正向关系，理论上保存的节点越多，写入延迟就越高，所以在分布式系统里，普遍采用了多数派强一致。最常见的是，强制写入两个节点。

   b. 这个强一致属性对于我们比较核心的业务，比如金融类场景，能做到更好的数据容灾。这里面还有两个概念：

      i. RPO（Recovery Point Objective）即数据恢复点目标，主要指的是业务系统所能容忍的数据丢失量。

      ii. RTO（Recovery Time Objective）即恢复时间目标，主要指的是所能容忍的业务停止服务的最长时间，也就是从灾难发生到业务系统恢复服务功能所需要的最短时间周期。

   c. 强一致与高可用这两个概念，等价过来，就是实现 RPO=0，RTO 足够小，一般和网络环境相关，目前工程层面普遍在若干秒级。

3. 标准 SQL 与事务（ACID）的支持：

   a. 我们讲过，SQL 因为高度易用性，已经是数据交互语言的事实标准。

   b. 在很多场景下，尤其是线业务场景 OLTP，完整事务仍然是必须项。

4. 云原生数据库：

由于计算与存储的高度分离架构设计，使得云场景中基于资源的弹性设计成为可能。

5. HTAP：

我们说过 HTAP 更多是数据服务的概念，我们认为当前的 HTAP 应该有几个属性：

   a. 必须是基于解决数据容量的前提下，也就是在海量数据的 OLTP 与 OLAP 的数据服务；

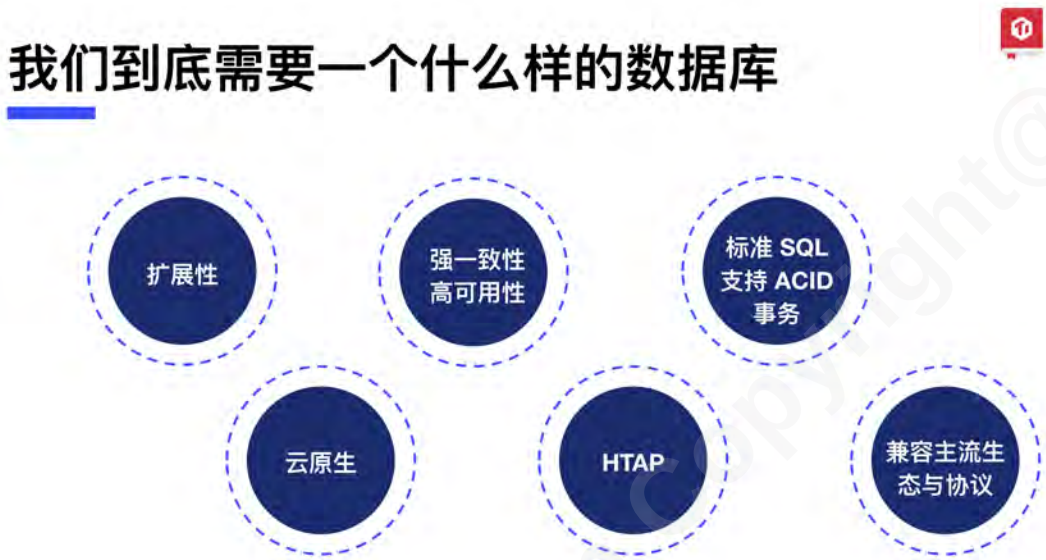
   b. 行列混合，同时也具备更彻底的资源隔离；

c. 承载 HTAP 数据服务的底层的数据架构或者产品，必须需要开放的；兼容数据库与大数据生态。

d. 统一数据查询服务。

6. 兼容主流生态与协议：

站在用户的角度，兼容主流生态与协议，可以大大降低使用门槛与成本。



同时我们在之前也提到，数据技术栈与架构的本质是，在相对固定的基础数据技术元素上进行各种选择、平衡与优化（trade off），结合上面我们讲到的数据架构设计目标，我们可以先笼统的去考虑下都需要哪些基础数据技术元素？

1. 数据模型：包括关系模型、文档模型、Key-Value 模型等。
2. 数据存储与检索结构：包括数据结构、索引结构，常见 B-tree、LSM-tree 等。
3. 数据格式（Data format）：数据在存储引擎的格式，一般分为关系型和非关系型。
4. 存储引擎（Storage engine）：负责数据的存取和持久化存储，比如 Innodb、Rockdb 等。
5. 复制协议（Replication）：为了保证数据可用性，复制协议与算法，常见的 Raft、Paxos 等。
6. 分布式事务模型：是事务处理时的并发控制方法，以保证可串行化。常见模型有 XA、TCC、Percolator 等。
7. 数据架构（Data Architecture）：一般指多个计算引擎共享一份数据，或者每个计算引擎都有一份数据，常见有 Share-Nothing、Share-Disk 等。
8. 优化器算法（Optimizer）：根据数据分布、数据量和统计信息，生成成本最低的执行计划。
9. 执行引擎（Execution Engine）：比如火山模型、向量化、大规模并行计算等。
10. 计算引擎（Computer engine）：负责 SQL 语句接口，解析生成执行计划并发送给存储引擎来执行，同时大部分计算引擎具有缓存数据的功能。

计算与存储分离

提到数据架构，你一定经常听到“计算与存储分离”，那这背后的逻辑是什么呢？

在前面的章节，我们提过数据技术驱动第三个内因是“硬件与云计算的发展”。

让我们回到 2000 年左右，当时的主流网络带宽只有百兆，那个时期主流数据库是单机架构（冯诺依曼架构）计算与存储紧密在一起， Google GFS（分布式文件系统） 尝试用普通的 x86 机器和普通硬盘搭建了大规模存储系统，但受限于网络带宽问题，所以将计算的代码移动到存储层，而不是将数据传输到计算节点。

在后面，网络硬件发生了很大的发展，带宽迅速完成从百兆、千兆、到万兆进化，稳定性也在持续加强。延迟也在大大的降低，同时各种高效的压缩算法进一步减少了磁盘 IO 与网络 IO。在这种背景下，存储和计算强耦合的方式，开始展现一些弊端：

1. 计算与存储强绑定，意味着两种资源总有一个是浪费的。
2. 我们在对服务器进行选型的过程中，开始纠结是计算型、还是存储型，大大增加复杂度和降低通用性。
3. 在云计算场景下，弹性的颗粒度是机器，不能真正做到资源的弹性。

到了云时代，基于高性能的网络的块存储 （EBS）开始替换主机存储成为主流，计算和存储耦合架构更加不合理，存储和计算分离的架构开始进一步普及。

TiDB 高度分层架构

TiDB 在 2015 立项，弹性是整个架构设计的核心考量点，所以选择了更为未来的计算与存储分离的架构，从逻辑上看，主要分为三层：

1. **支持标准 SQL 的计算　引擎 TiDB-Server**

这里面有一个概念，大家可能会混淆，TiDB 是整个数据库的名称，同时 TiDB 有一个兼容 MySQL 的计算引擎 TiDB-Server（https://github.com/pingcap/tidb），兼容以 MySQL 5.7 为主，在逐步兼容 MySQL 8.0。包括在 TiDB Server 支持 MySQL 的协议，语法，方言语法，DDL/DML，系统变量，内置功能，SQL Mode 以及类型推导等等。

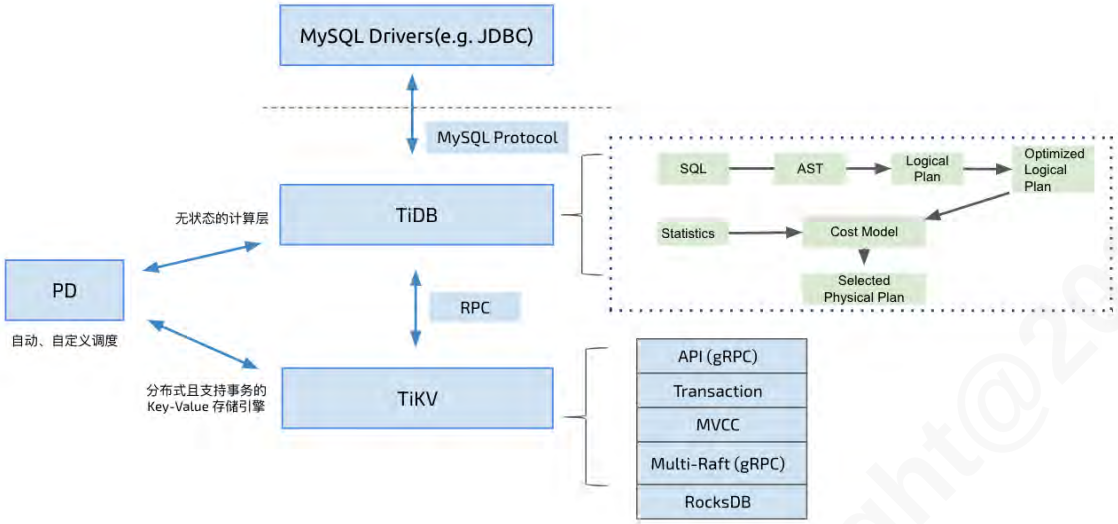
2. **分布式存储引擎 TiKV**

TiDB Server 本身并不存储数据，只是进行计算，实际的数据存储在存储层，也就是 TiKV（https://github.com/pingcap/tikv），目前这个项目已经捐献给 CNCF，属于 CNCF 的毕业项目。

3. **负责元信息管理与调度的，调度引擎 Placement Driver**

Placement Driver (pd-server，简称 PD，https://github.com/pingcap/pd)，PD 主要有几个功能：

- 集群的元信息管理模块，注意这里的元信息，指的是分片（Region）的数据分布、以及集群拓扑结构，不是指的数据库里的 Schema 信息。
- 分布式事务 ID 的分配，可以简单理解为，全局唯一序列。
- 调度中心，默认每个 TiKV 节点会在一个周期内发送节点相关数据给 PD，包括 Region 数量、Leader 数量、最近周期内读写量等，PD 接受后，会进行计算，然后发出再平衡调度，比如将部分 Region 从 Region 数量较多节点调度到 Region 较少的节点上。
- 所以说，PD 是整个集群的大脑，为了保持全局高可用，PD 也至少三个节点，通过 Raft 进行三副本复制。



附录：

1. 最早的分布式事务模型是 X/Open 国际联盟提出的 X/Open Distributed Transaction Processing（DTP）模型，也就是大家常说的 X/Open XA 协议，简称 XA 协议。
2. TCC（Try-Confirm-Cancel）分布式事务模型相对于 XA 等传统模型，其特征在于它不依赖资源管理器（RM）对分布式事务的支持，而是通过对业务逻辑的分解来实现分布式事务。

第 5 课：如何构建一个分布式存储系统

我们需要一个什么样的存储引擎

- 更细粒度的弹性扩容、缩容
- 高并发读写
- 数据不丢不错
- 多副本保障一致性及高可用
- 支持分布式事务

上面几点是我们的设计目标，数据架构的本质是：在相对固定的基础数据技术进行 Trade off，现在我们手里有很多各种形状的积木模块（基础数据技术），我们一起搭建一个很 Cool 的成品。TiDB 的存储引擎是 TiKV，那么我们从下向上，依次按照数据结构、表模型、分片策略、复制冗余、多版本控制、分布式事务等维度，介绍 TiKV 是如何进行 Trade off 的，最终构建一个完整分布式存储系统。

如何构建一个完整的分布式存储系统

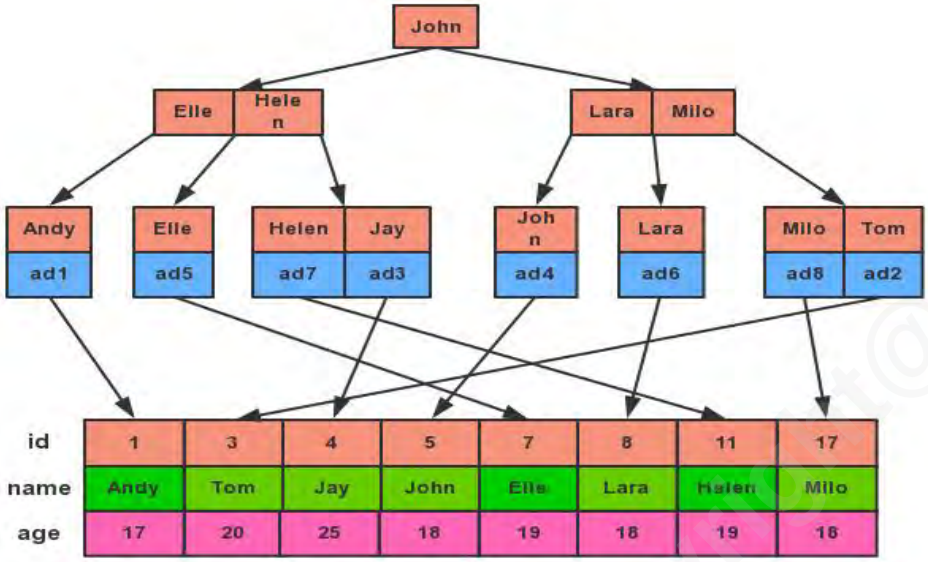
选择数据结构

数据结构是计算机中存储、组织数据的方式，是数据库的核心。

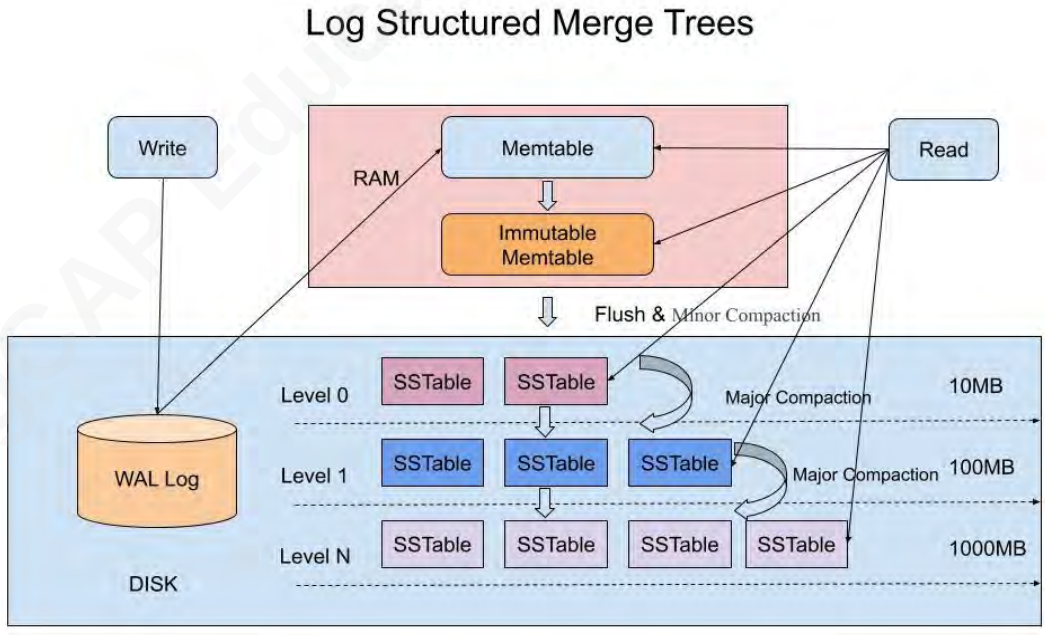
首先，我们以键-值对（key-value）数据的索引开始，key-value 类型并不是唯一可以索引的数据，但它非常简单随处可见，而且是其他更复杂索引的基础构件模块。

第二步，数据结构，在传统的单机关系数据库里，最常见的索引结构是 B-tree 。但最近几年出现了一个很有吸引力的引擎是 LSM-tree（Log Structured Merge Tree），如何选择？还是回到业务场景本身。在传统的 OLTP 系统里，**写入是最昂贵的成本。**

- B-tree 索引必须至少写两次数据，一次是预写日志（WAL），一次是写入树本身；我们需要通过索引加速一次特定查询，就需要创建一个 B-tree 索引。
- B-tree 是一个严格平衡的数据结构，它的设计是对读友好，数据写入触发的 B-tree 的分裂与在平衡成本也非常高。也就是 B-tree 对写入相对不友好。
- 传统的主从（Master-slave）架构里，不管你加多少从节点，集群的写入容量都没法扩展，集群的写入容量完全由主库的机器配置决定；扩展只能通过非常昂贵的集群拆分（分库概念）；



之前我们说过，数据容量更体现在写入吞吐量，分布式系统面对的场景是写入越来越大，基于此，我们选择了 LSM-tree 结构，简单来说，它对写入采取了顺序写的方式，写入到一定阈值后，持久化成一个静态文件，随着写入的越来越大，生产的静态文件会被推到下一层进行再合并（Compaction）。本质上它是一个用空间置换写入延迟，用顺序写入替换随机写入。在后面的课程里，我们会详细解释 LSM-tree 原理。



所以，TiKV 单节点选择了基于 LSM-tree 的 RockDB 引擎

- RockDB 是一款非常成熟的 LSM-tree 存储引擎
- 支持批量写入（Atomic batch write）
- 无锁快照读（Snapshot）
  - 这个功能在数据副本迁移过程会起到作用。

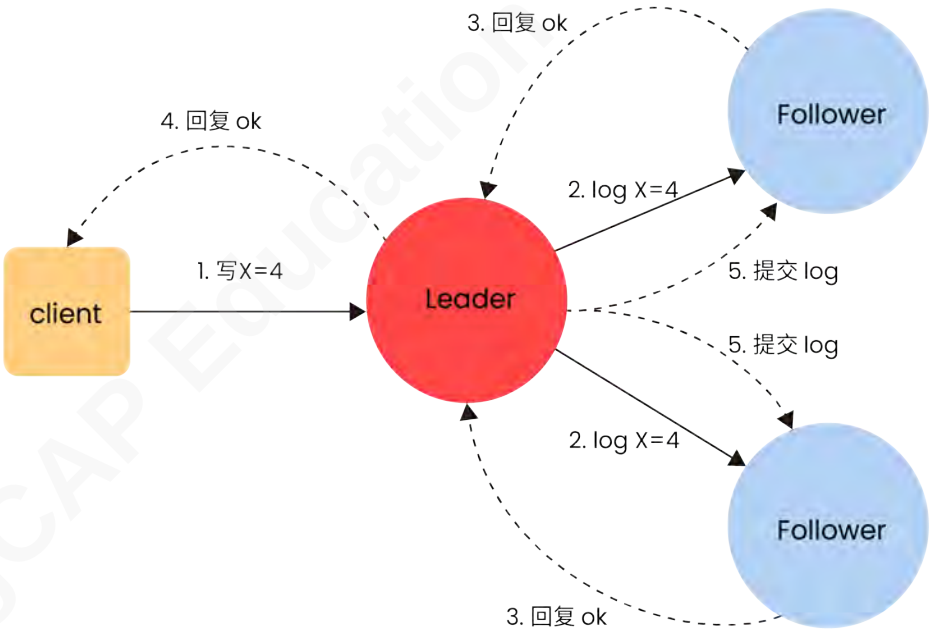
选择数据副本

我们再看下如何做数据副本？数据冗余决定系统的可用性, 所以最重要的是如何选择复制协议，我们要做一个“金融级强一致”的系统，所以选择一个强一致的复制算法，在这个领域，比较流行的是 Raft 与 Paxos，我们先看下维基百科对 Raft 的定义：

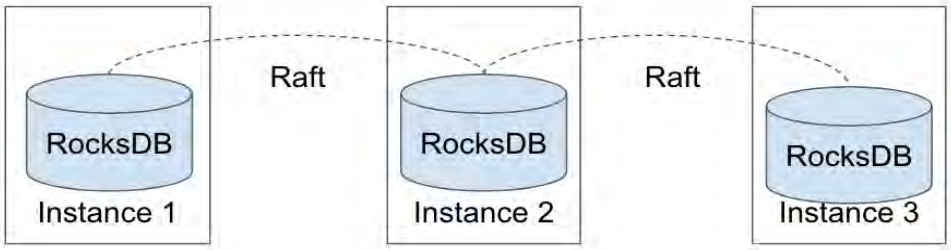
Raft 定义维基百科

Raft 是一种用于替代 Paxos [1] 的共识算法 。相比于 Paxos [1]，Raft 的目标是提供更清晰的逻辑分工使得算法本身能被更好地理解，同时它安全性更高，并能提供一些额外的特性。

Raft 算法通过先选出 leader 节点、有序日志等方式，简化了流程、提高效率，并通过约束减少了不确定性的状态空间。相对 Paxos 逻辑更清晰，容易理解与工程化实现。



有了 Raft，我们就可以基于 Rocksdb 构建一个多副本的集群。



如何实现扩展

数据分片是分布式数据库的关键设计, 如果要是实现扩展，从底层技术看，就是做**分片**，分片最重要的区别，是在于预先分片（静态），还是自动分片（动态），传统分库分表或者分区的方案都是预先分片，比如提前创建 100 个分表，属于静态分片，这种分片只解决表容量的问题，没有解决弹性的问题。所以，第一点，我们要优先使用自动分片算法。第二，分片总是需要一个维度和算法，比较常见的是

- 哈希（hash）
- 范围（range）
- 列举（list）

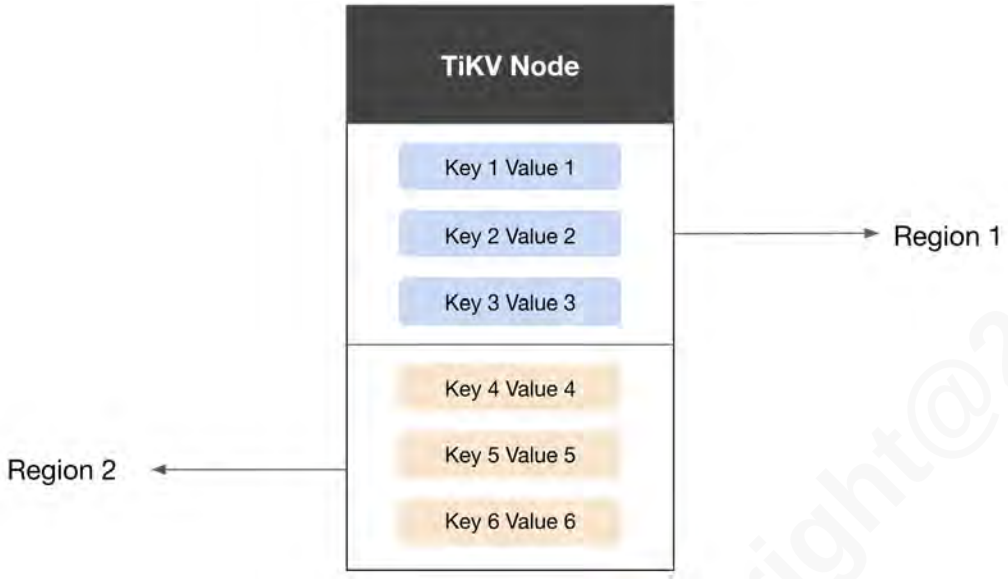
在 TiKV 系统里，采用了 Range 算法，原因如下：

- 类似 “Where age >= 30”，这种范围查询在 OLTP 业务是非常常见场景，在这个场景下，而 Range 分片可以更高效地扫描数据记录，而 Hash 分片由于数据被打散，扫描操作的 I/O 开销更大；
- Range 分片可以简单实现自动完成分裂与合并；
- 弹性优先，分片需要可以自由调度。

所以说 Range 分片在分布式架构下，很多情况下更有优势。但 Range 分片的最大问题就是热点分片问题，也就是最新的分片最热。后面我们会讲怎么样合理解决这类热点问题。

解读完这个问题，我们看看 TiKV 实现方式：

我们首先介绍一个非常重要的概念：Region，Region 是理解后续一系列机制的基础。我们将 TiKV 看做一个巨大的有序的 KV Map，我们通过 range 的方式，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 96MB)。每一个 Region 都可以用 StartKey 到 EndKey 这样一个左闭右开区间来描述。



比如我们刚开始创建一个表，对应存储是一个 KV map，范围是 0~正无穷，随着数据量增加，分离成一个新的分片。

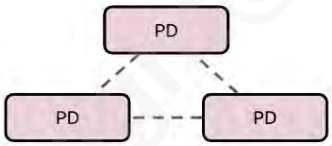
如何进行分离与扩展

当某个 Region 的大小超过一定限制（默认是 144MB）后，TiKV 会将它分裂为两个或者更多个 Region，以保证各个 Region 的大小是大致接近的，这样更有利于 PD 进行调度决策。同样的，当某个 Region 因为大量的删除请求导致 Region 的大小变得更小时，TiKV 会将比较小的两个相邻 Region 合并为一个。

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。以 Region 为单位做数据的分散。

我们有了 Region 的概念，对应每一个分片，通过 Raft 进行三副本复制，那我们看看数据是怎么分裂，已经怎么实现对业务“无”感知，分片从一个物理节点搬运到其他节点。

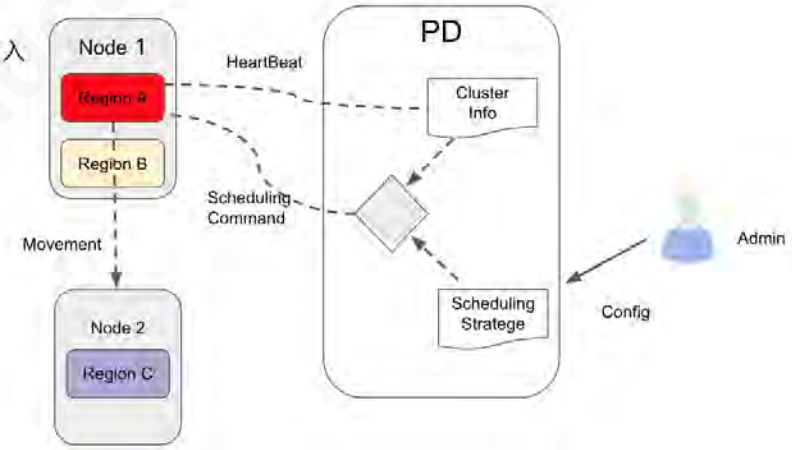
Let's say, the amount of data within Region 1 exceeds the threshold (default: 96MB)



灵活调度机制

将数据划分成 Region 后，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（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）。

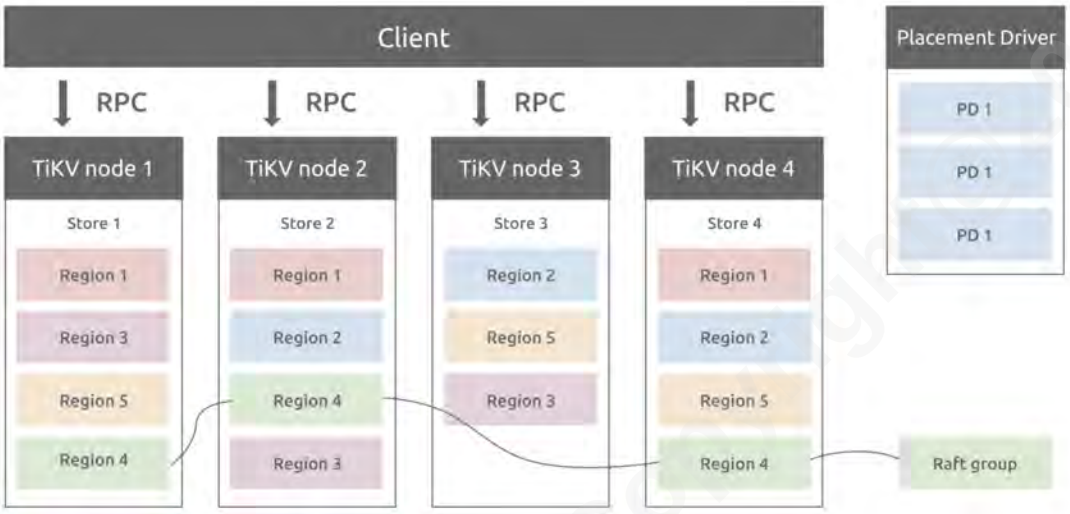
- 分片数量、Leader、吞吐量自动平衡
- 自定义调度接口
  - 支持跨 IDC 表级同时写入



TiKV 整体架构

TiKV 参考 Google Spanner 设计了 multi raft-group 的副本机制，首先将数据按照 key 的范围划分成大致相等的切片（下文统称为 Region），每一个切片会有多个副本（通常是 3 个），其中一个副

本是 leader，提供读写服务。在前面「TiDB 的整体架构中」，我们介绍了 PD 这个调度组件，TiKV 通过 PD 对这些 Region 以及副本进行调度，并决策哪个副本成为 leader，以保证数据和读写负载都均匀地分散在各个 TiKV 上，这样的设计保证了整个集群资源的充分利用并且可以随着机器数量的增加水平扩展。具体详细的调度不在本节细说，会在后续课程中进行详细介绍。



多版本控制（MVCC）

我们还有一个目标是实现完整事务，事务四大属性里有一个叫隔离性，描述的是“各个事务之间相互影响的程度，主要用于规定多个事务访问同一数据资源，各个事务对该数据资源访问的行为”，隔离性最常见的实现技术就是多版本并发控制 (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，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
……
```

有了这样的 MVCC 版本控制后，TiDB 得以实现并发控制、SI 隔离级别、事务支持、恢复历史数据等功能。

分布式事务模型

解决了 Region 的持久化存储之后，TiKV 就需要开始考虑对于分布式事务的支持。TiKV 支持分布式事务，用户可以一次性写入多个 key-value 而不必关心这些 key-value 是否处于同一个 Region 上，TiKV 通过两阶段提交保证了这些读写请求的 ACID 约束。

TiDB 采用的 Google 的 Percolator 事务模型，在 Percolator 的基础上做了大量的优化和改进，简单来说，分布式事务普遍采用的“两阶段”提交的方式，而传统的两阶段提交往往需要一个事务管理器，这个事务管理器往往会成为集群的性能瓶颈。

- TiDB 采用了“去中心化”的两阶段提交算法。
  - 在每个 TiKV 节点上，会单独分批一个存储锁信息的地方，CF Lock（列簇）
  - 通过 PD 进行全局受时（TSO）
- TiDB 默认的隔离级别是 Snapshot Isolation，SI 和 RR（可重复度）隔离级别接近，天然没有幻读的现象；同时也支持 RC（提交读）隔离级别。
- 默认乐观事务模型；3.0+ 版本也支持悲观锁。
- 在 TiDB 5.0+ 版本支持第二阶段的异步提交，可以简单理解为针对特定场景下实现了 1PC，在 OLTP 场景里，比较常见的高并发的小 SQL 写入场景，大大降低了写入延迟。

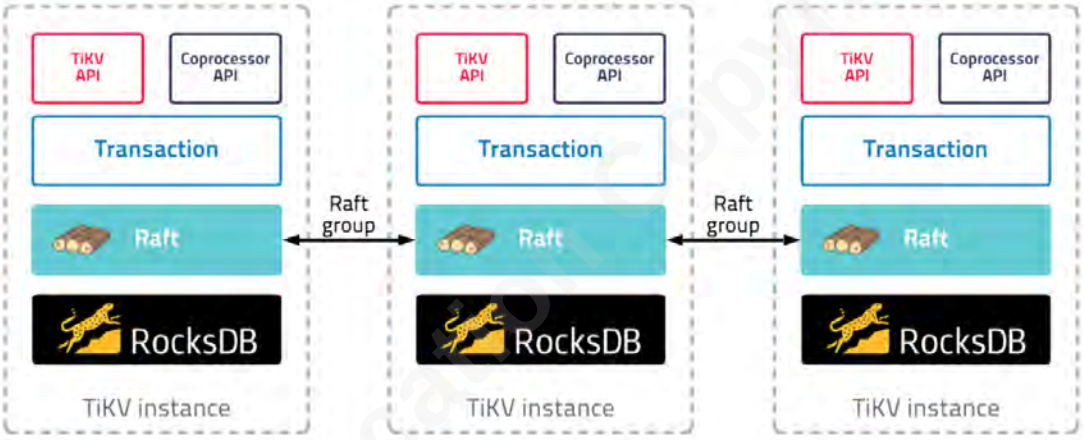
协作处理器（Coprocessor ）

在分布式架构下，一个非常通用的优化思路是，在数据层尽快的进行计算与预处理，比如在本地节点完成数据过滤，部分聚合等，也就是最大程度下推机制。这种机制不仅减少了网络交互成本，还能充分利用分布式存储节点的并行技术能力，所以在每个储存节点需要有一个技术模块。

Coprocessor 就是 TiKV 中读取数据并计算的模块，每个 TiKV 存储节点，我们都有一个协调计算器。

本章小结

到现在为止，我们介绍完了 TiKV 的整体架构，大家可以看到 TiKV 整个系统是高度分层的。在这里我们根据 TiKV 的协议栈实现整体看一下 TiKV 架构的重点：从最底下开始是单机的 RocksDB，然后在上面用 Raft 构建一层可以被复制的 RocksDB，确保所有的写入数据一定复制了足够多的副本足够安全（防止单机故障），之后基于安全的 Key-Value Store 上根据 MVCC 来构建事务的多版本和分布式事务，在分布式事务构建完成之后，就可以轻松的加上支持事务的 TiKV API 和 Coprocessor API 层，我们就得到了一个完整的 TiKV 实例。
在后面章节中，我们将会从下往上逐层来剖析 TiKV 各层组件的原理，便于大家更加深入的理解 TiKV。



附录：

[1] 维基百科-Paxos https://zh.wikipedia.org/wiki/Paxos

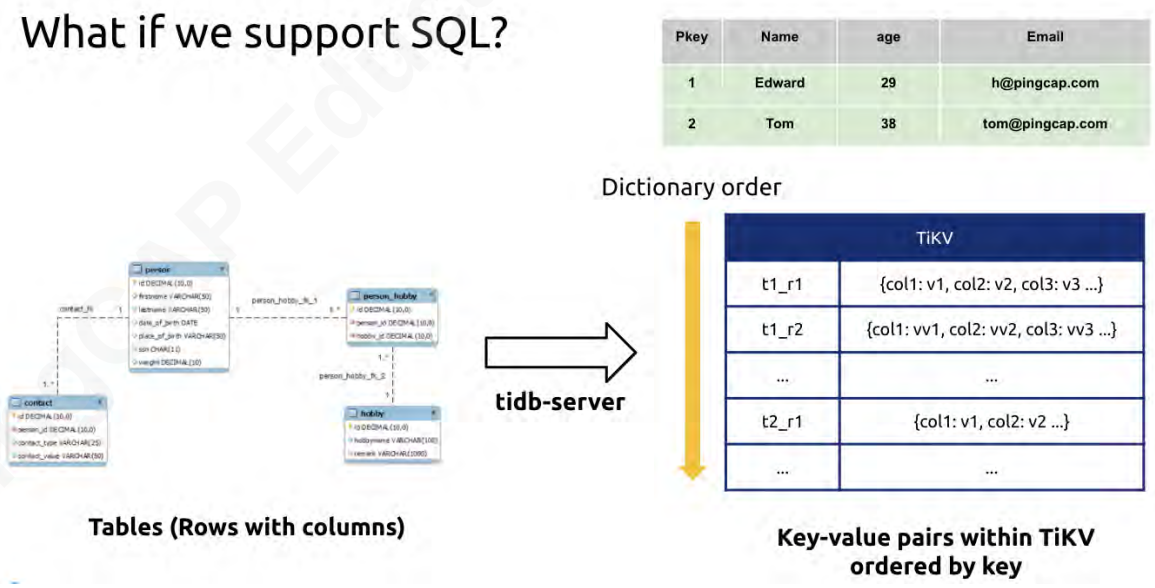
第 6 课：如何构建一个分布式 SQL 引擎

在上一讲，我们从下往上构建了一个支持事务的分布式 KV 系统，实现了颗粒度非常细扩展性（Region），那我们继续回顾下我们的目标，看看如何实现 SQL、关系型模型。

如何在 KV 上实现逻辑表

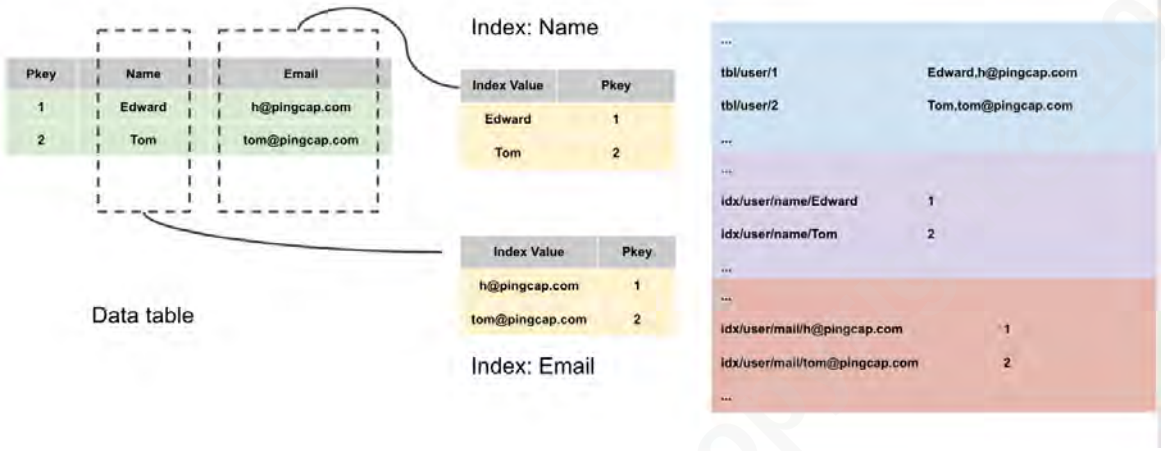
我们现在手里有一个全局有序的分布式 Key-Value 引擎，全局有序这一点重要，可以帮助我们解决不少问题。比如对于快速获取一行数据，我们只需要找到具体 Key，我们就能利用 TiKV 提供的 Seek 方法快速定位到这一行数据所在位置，找到其他列数据。再比如对于扫描全表的需求，如果能够映射为一个 Key 的 Range，从 StartKey 扫描到 EndKey，那么就可以简单的通过这种方式获得全表数据。操作 Index 数据也是类似的思路。接下来让我们看看 TiDB 是如何做的。

- 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）。
- 我们可以简单把 TableID/IndexID + RowID 看成 Key-Value 里的 Key，Value 可以看成所有列按照等位偏移进行连接。
- 在查询过程，我们再通过等位偏移量对 Value 进行反解析，然后对应 Shema 的元信息，匹配列信息。



基于 KV 的二级索引设计

在 TiKV 里，二级索引也是一个全局有序的 KV map，可以简单理解它的 key 就是索引列信息，Values 是原表的 Primary key；然后再通过这个 Primary key 在原表的 KV MAP 进行扫描，找到 Values 的信息。这个过程和 B-tree 回表的逻辑类似，可以理解我们看书过程，先查目录，通过查章节定位页数，再通过具体页数来找到内容。



SQL 引擎过程

SQL 是一个非过程的声明式语言，只描述结果，不解读过程，所以一个 SQL 引擎很重要的一个模块就是优化器，它负责从很多执行计划中找到最优的执行计划，举个例子，我们要的结果是从天安门尽快到达颐和园，我们很多交通方式与路线，优化器就是根据各种交通方式的效率与路况的拥堵情况，来选择一个最快的出行方案。这里的各种交通方式可以理解为数据寻址或者数据计算的各种算子，比如 hash join、Index reader、路况可以理解为表、索引、列的数据分布统计信息。除了优化器外，我们看看，还有哪些是 SQL 引擎必须构建的部分。

- SQL 语法解析、语意解析、协议解析

TiDB 采取了兼容 MySQL 策略，这个模块是按照 MySQL 的语法进行解析。以及语义解析，包括访问表的权限和合理性。

- AST 抽象语法树

AST 主要将 SQL 从文本解析成结构化数据。

- SQL 逻辑优化

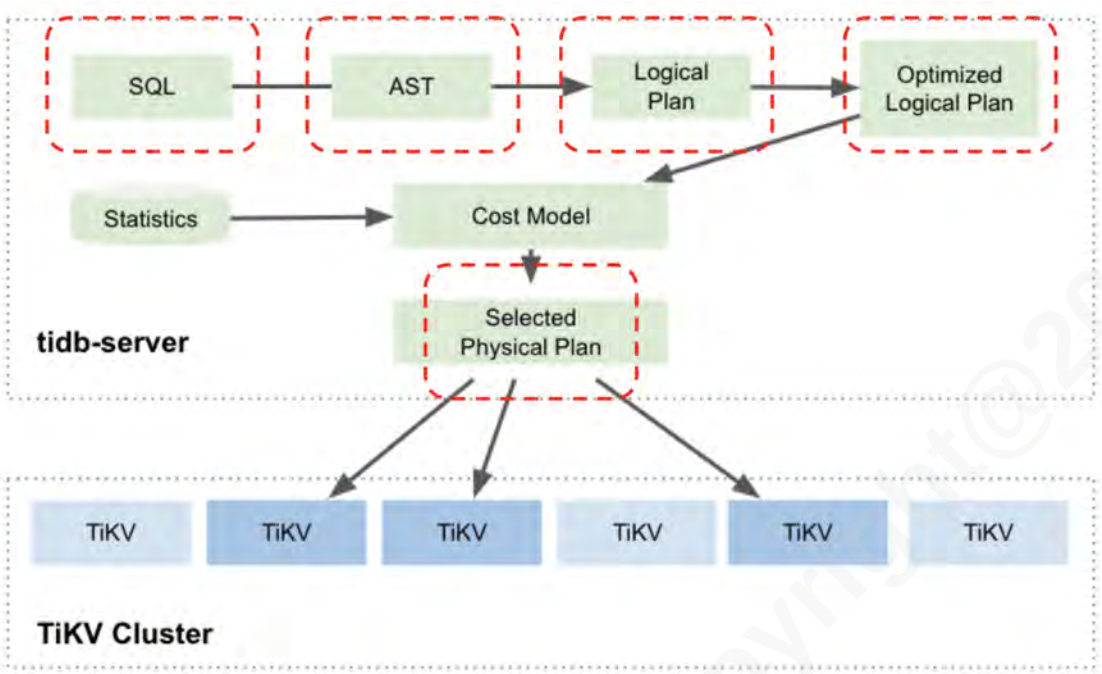
各种 SQL 等价改写与优化，比如子查询改成表关联。各种不必要信息的裁剪，比如列裁剪、分区裁剪等。

- 优化器基于统计信息与成本模型的生产执行计划

这一步最为关键，也就上个例子说的找“最佳出行计划”的过程。这个也是 SQL 优化最关键的部分。

- 执行器执行

执行引擎根据优化器定下的执行计划，进行数据寻址与计算。

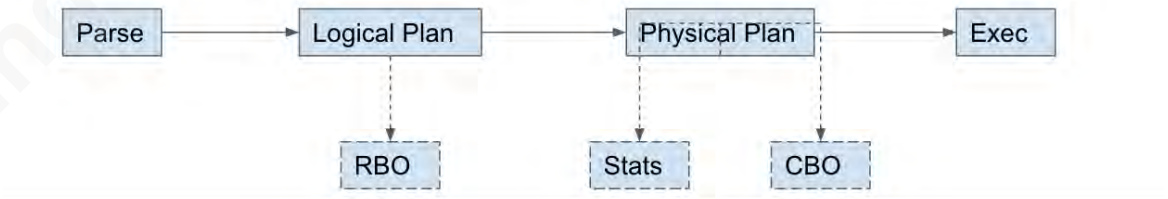


基于成本优化器

优化器怎么才能找到最佳执行计划呢？

- 首先需要知道我们有哪些算子，已经知道每个算子的优势与适应场景，可以简单理解为各种交通工具。
- 基于成本的模型，简单理解，CPU、内存、网络等资源需要一个等价公式。

- Power CBO Optimizer
  - Hash join、Sort merge、Index join、Apply（Nested loop）
  - table_reader、table_scan、index_reader、index_scan、index_lookup
  - Steam aggregation、Hash agg
- Cost
  - Cost(p) = N(p)*FN+M(p)*FM+C(p)*FC, N stands for the network cost, M stands for the memory cost and C stands for the CPU cost.
- task (handle on TiDB or TiKV)
  - corp、root

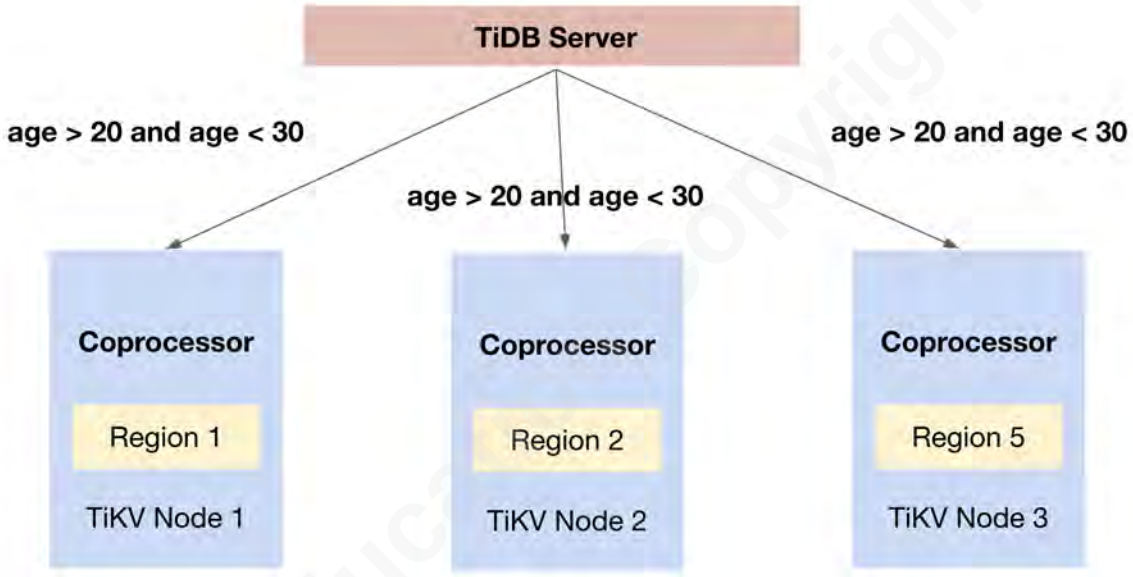


分布式 SQL 引擎主要优化策略

• 执行器构建：最大程度的下推计算

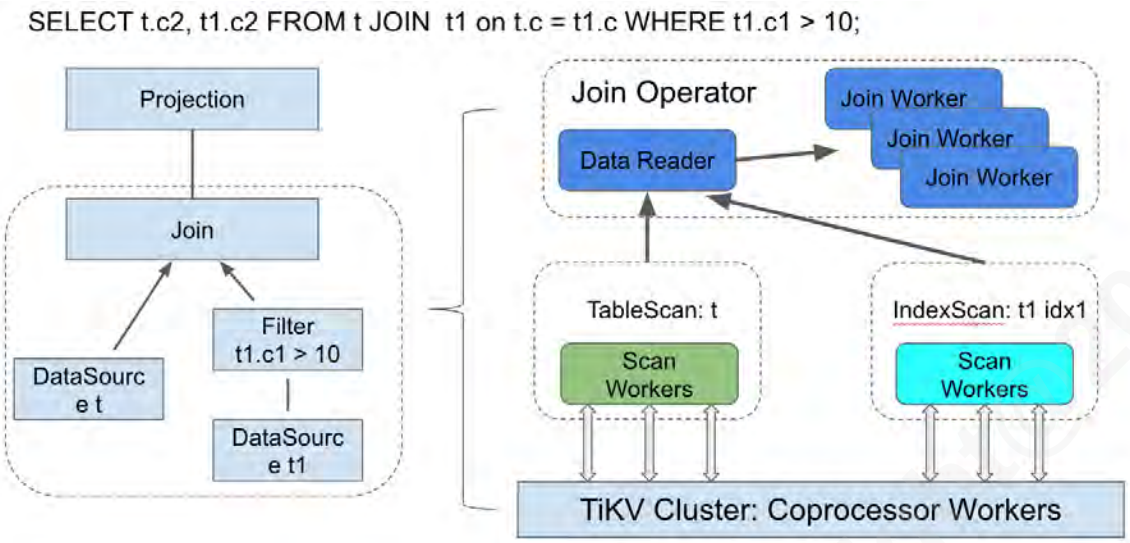
上文我们解读，我们已经构建了一个分布式存储引擎 TiKV，分布式的优势，就是最大程度地利用分布式多节点的存储与算力，所以在分布式 SQL 引擎，一个很重要的优化逻辑就是最大程度的让数据在分布式存储层完成过滤与计算。也就是我们说的最大下推策略（Push Down）。

比如我们要在用户表里统计“ 90 后”有多少人，用户表是按照很多个分片存储在各个存储节点里，那么我们就可以在每个存储节点并行，进行数据扫描与过滤数据，然后统计各自的“ 90 后”数量，最后上报给上的 SQL 引擎，再进行一次简单的 SUM 就可以返回结果。



• 关键算子分布式化

比如 Hash Join，在 TiDB-Server 的 Hash Join 不管是数据寻址，还是在内层进行分批匹配都可以并行与分批处理，这也就是，为什么在 TiDB 这种架构下，大表 join 的场景，比传统的 MySQL 要快很多的原因。

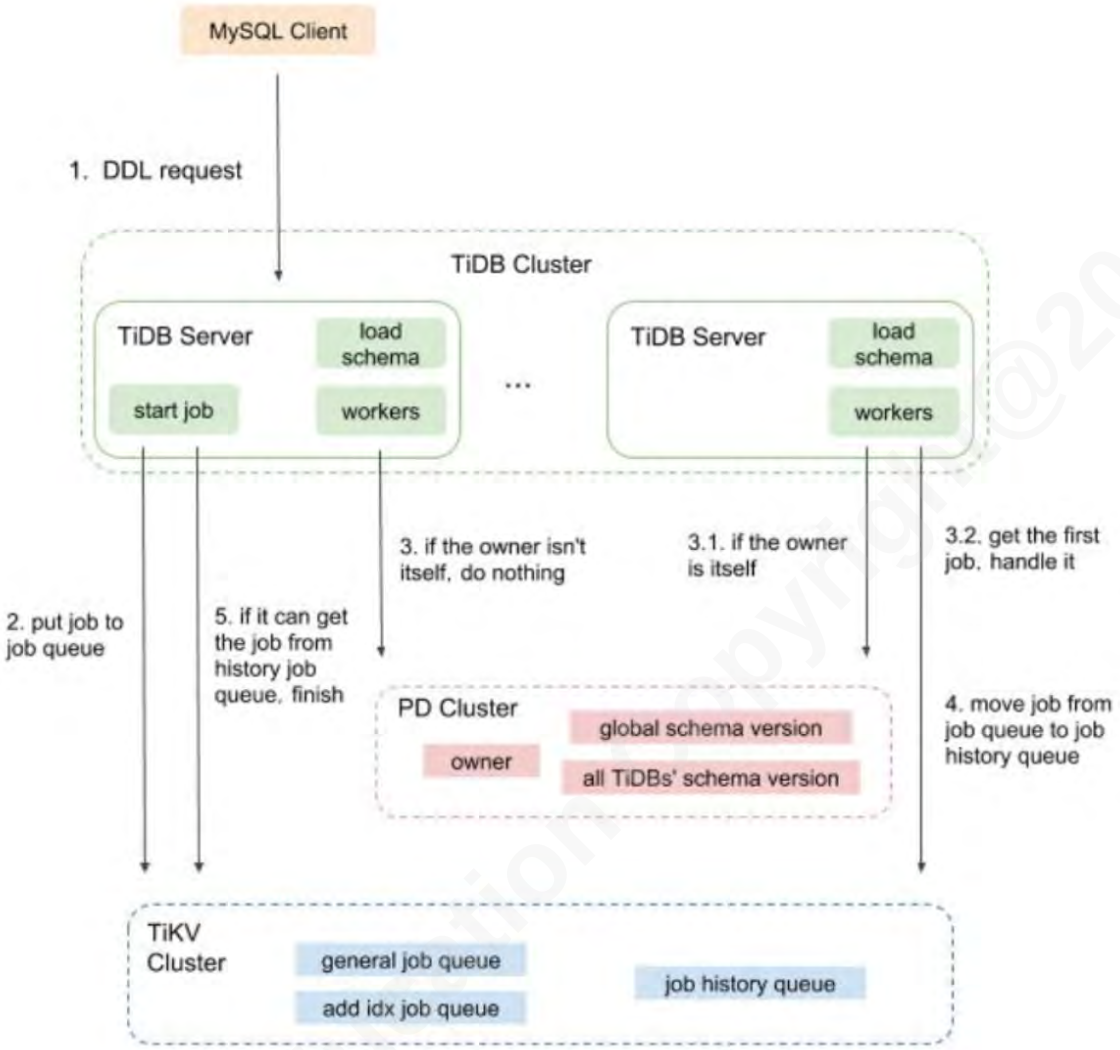


如何构建一个 Online DDL 算法

表变更 DDL 也是 SQL 很常见的操作，传统 MySQL 进行表变更，比如加字段，是需要对表的物理数据进行重组织，这种方式成本很高，效率很低。而分布式数据库面对是数据量更大的表，所以能否实现 Online DDL，对业务快速上线迭代、数据库运维与都非常重要。

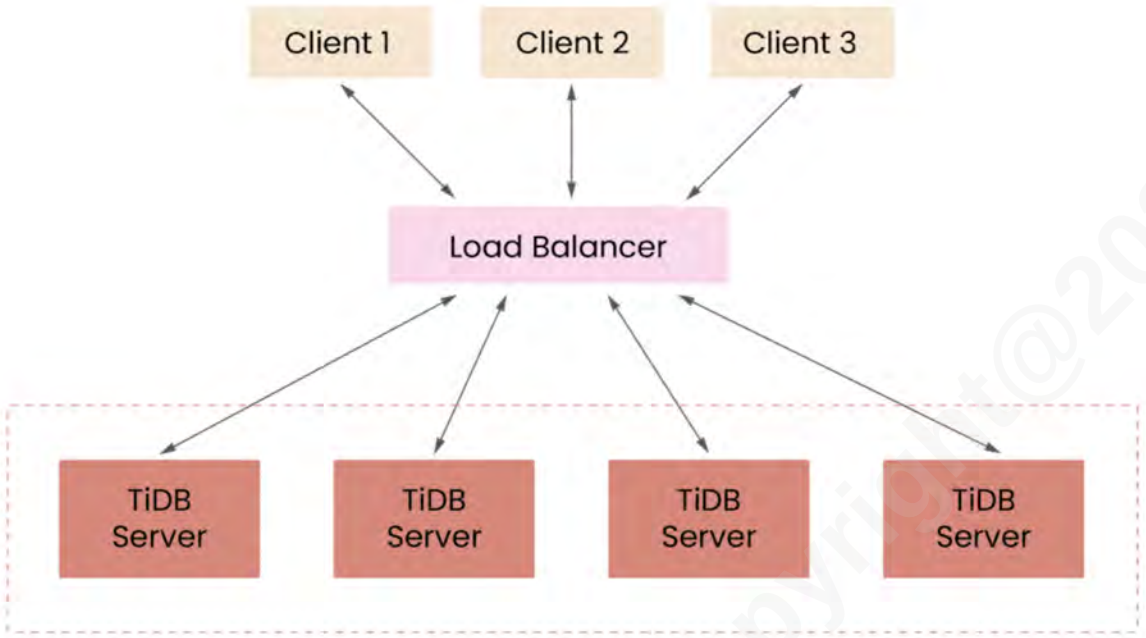
TiDB Online DDL 实现有两个关键点：

- 第一、在 TiDB 里没有分表概念，所以不管多大的表，Schema（表结构）只存储一份，比如我们要在表上加字段，我们只需要在这个 Schema 进行操作，操作后的新数据按照新结构存储，老数据只有在需要变更的时候才重组织，所以整个 DDL 完成过程是非常快速的。
- 第二、多个计算节点怎么保存 Schema 信息一致呢？TiDB-Server 在线 DDL 是根据 Google F1 论文算法实现而来，在此论文中异步在线 DDL 的核心就是 Multi schema versions，它把 DDL 过程分成 *Public*、*Delete-only*、*Write-only* 等几个状态，每状态在多节点之间同步和一致，最终完成完整的 DDL。



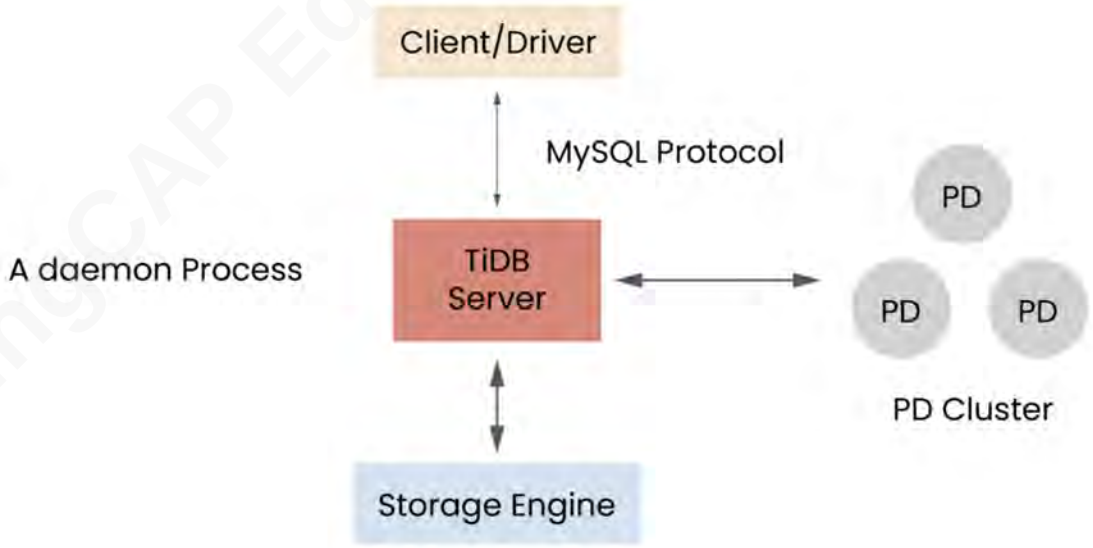
如何连接到 TiDB-Server

回顾前面在架构中对 TiDB-Server 的介绍，它是一个对等、无状态的，可横向扩展的，支持多点写入的，直接承接用户 SQL 的入口，业务层直接使用 MySQL 的客户端或者 SDK 连接到任何一个 TiDB-Server 实例上当作 MySQL 正常使用即可。



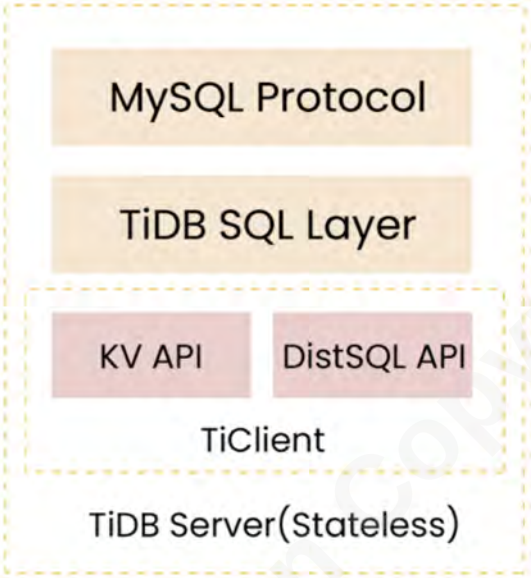
从进程的角度看 TiDB-Server

从一个简单的结构来看，TiDB-Server 就是一个后台进程，启动之后，就会不断地监听它的 4000 端口，与 Client 去通讯，最重要的作用是按照 MySQL 的协议，解析收到的网络请求。SQL 语句解析后，下一步就是要到 TiKV-Server 获取数据，这个过程需要先去 PD 集群获取 TSO（时间戳）、Region（分片）路由，然后根据 Region 的路由信息去相应的 TiKV-Server 节点读写数据。



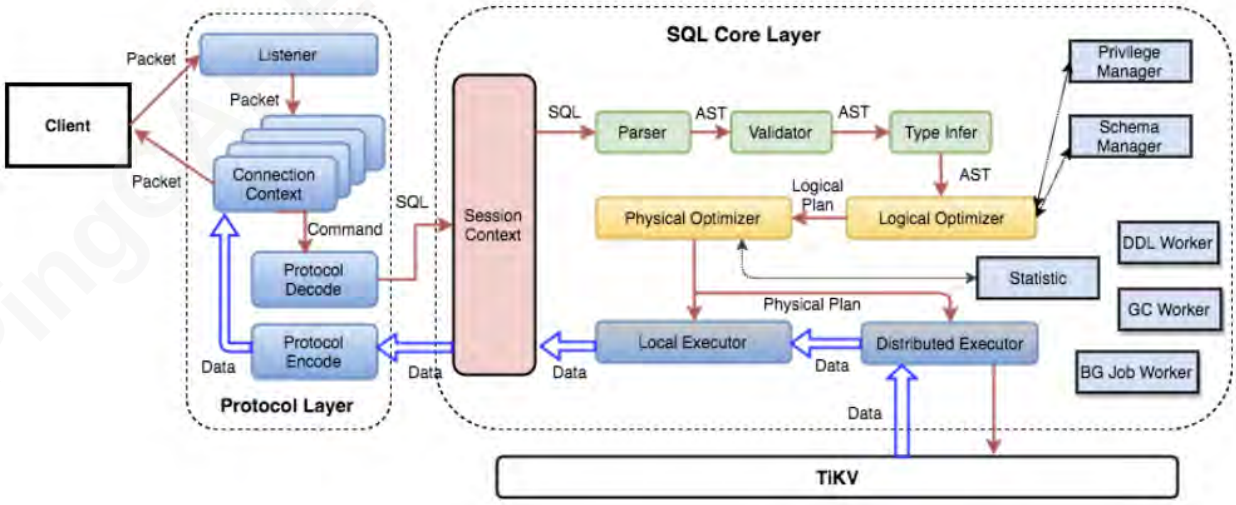
从内部结构看 TiDB-Server

再从一个 TiDB-Server 的内部结构来看，最上面是 MySQL 协议层，按照 MySQL 协议进行编解码，后面是 SQL 核心层，这是 TiDB-Server 最主要的部分，会进行 SQL 的 Parser 分析，进行逻辑优化、物理优化，以及统计信息收集等后台工作的 worker，最下面是数据获取的 API，分为 KV 和事务两种方式。



其他功能

TiDB-Server 作为一个服务，他有一些功能是在前台，就是说大家可以直接看到，还有一些功能是在后台默默工作，下面分别来看一下。



前台功能

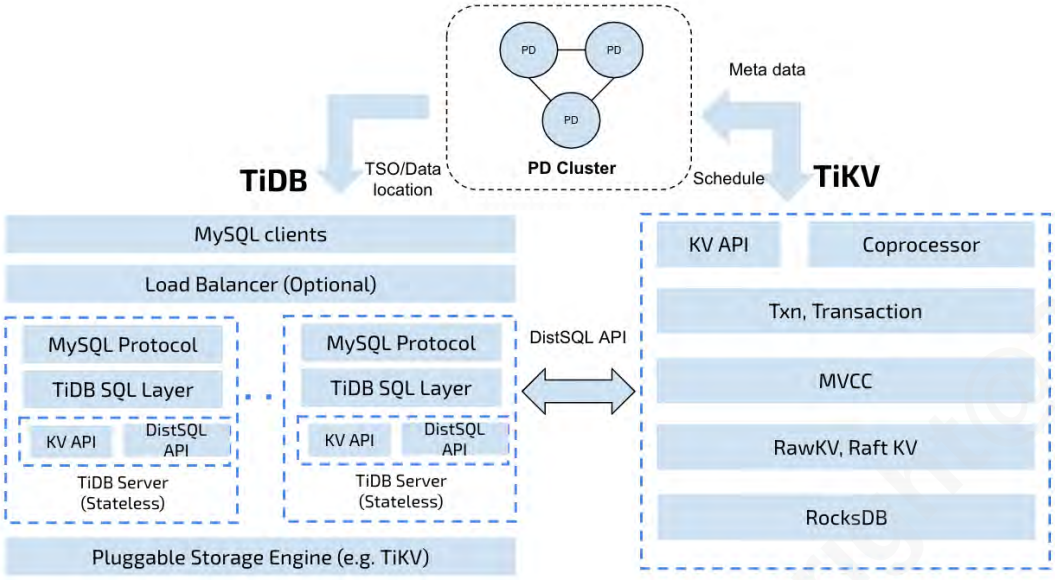
1、管理连接和账号权限管理。 用户通过用户名+密码来进行认证。TiDB 的权限管理系统按照 MySQL 的权限管理进行实现，TiDB 用户目前拥有的权限可以在 INFORMATION_SCHEMA.USER_PRIVILEGES 表中查找到。
2、MySQL 协议编码解码。TiDB 是兼容 MySQL 协议的，对协议的解析是在 TiDB-Server 中实现的。对于 MySQL 的兼容程度，TiDB 4.0 100% 兼容 MySQL 5.7 协议、MySQL 5.7 常用的功能及语法。触发器、存储过程，以及一些细节语法，是不支持的，详细兼容情况可以参考官网 https://docs.pingcap.com/zh/TiDB/stable/MySQL-compatibility
3、独立的 SQL 执行。每一个 TiDB-Server 都是独立且对等的，均能承担读写 SQL，一个 TiDB-Server 不会也无需关心其他 TiDB-Server 上在进行的 SQL。具体来讲，包括逻辑优化、物理优化、语句执行、事务管理、DDL、DML 等。
4、库表元信息，以及系统变量。TiDB 和 MySQL 类似，也有库表信息、参数、变量等系统信息，例如 Information Schema 库，和 System Variables 等。由于是分布式系统，在系统变量的管理上，和单机 MySQL 有所不同，会区分集群级（Global）、实例级（Instance） 两种级别。

后台功能

1、垃圾回收（GC）。GC 是由 TiDB-Server 发起，来驱动 TiKV-Server 回收掉已经过期的老版本数据。
2、执行 DDL。DDL 是从 TiDB-Server 执行的 SQL，TiDB 为表加列是秒回并生效的，但有一些是需要后台慢慢执行的，比如加索引，这里进度管理、操作的数据 Range 范围、版本控制，都是需要 TiDB-Server 协同的。DDL 的详细过程，后面有一节详细介绍。
3、统计信息管理。TiDB 的优化器运行需要依靠数据的统计信息，表的全量、增量的统计信息收集，也是在 TiDB-Server 的后台自动运行的。
4、SQL 优化器与执行器。这些会在后续课程中详细解读，这里先略过。

总结

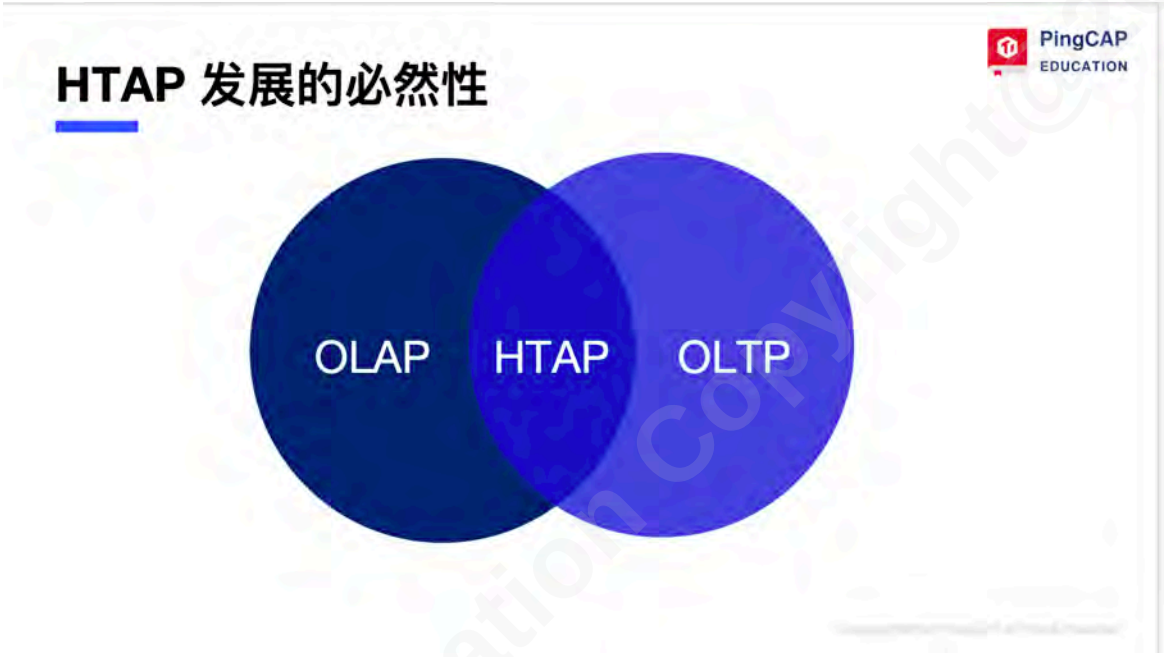
现在我们终于构建一个分布式的 SQL 计算引擎 TiDB-Server，再加上上文构建的分布式存储引擎 TiKV-Server，一个完整的分布式关系型数据库架构就基本搭建完成。



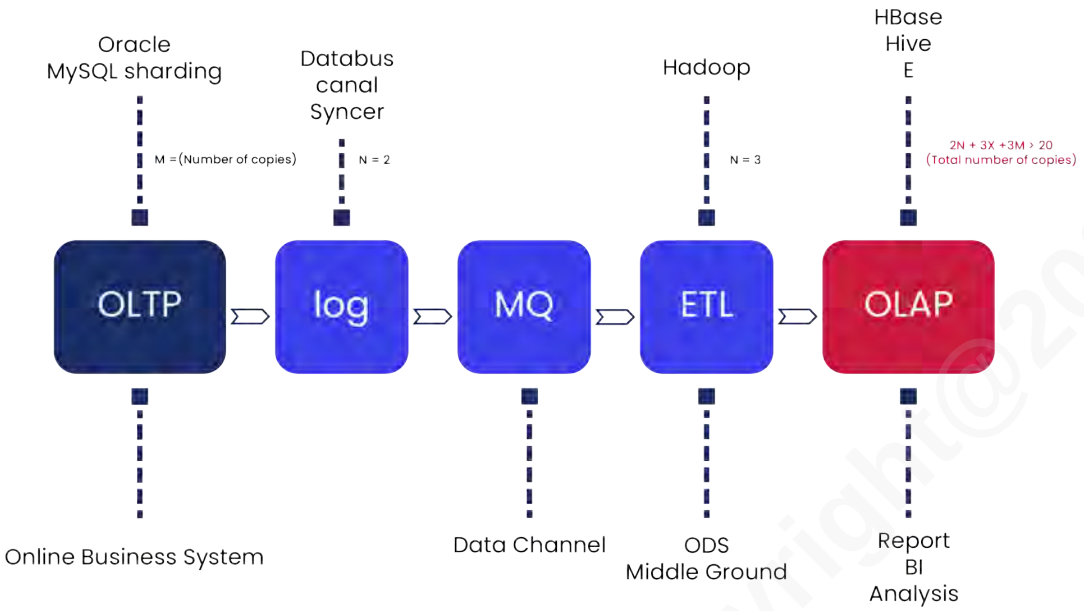
第 7 课:基于分布式数据库 HTAP 数据服务

HTAP 数据服务发展的必然性

在前面的课程，我们提到，场景的多样性会驱动细分技术的发展，而从使用以及业务副本成本的角度，又希望数据服务的统一性。



2005 年，Gartner 提出了 HTAP（Hybrid transactional/analytical processing，在线事务处理/在线分析处理数据库）这一概念，并在 2014 年，Gartner 对 HTAP 数据库给出了明确的定义，HTAP 数据库需要同时支持 OLTP 和 OLAP 场景。基于创新的计算存储框架，在同一份数据上保证事务的同时支持实时分析，省去了费时的 ETL 过程。



从技术发展的角度看，早期的经典关系型数据库，比如 Oracle/SQL Server 本身既可以在线业务（OLTP），又可以进行 OLAP，在这类数据库里有很多 AP 的技术，比如并行计算、物化视图、列存或者 Bitmap 索引等等，前面我们提到数据技术驱动两个关键因素，**数据容量**与**业务创新导致场景多样性**。这两点也导致了 OLTP 与 OLAP 技术开始分道扬镳，对于 TP 业务，更追求了高并发、低延迟，这些主题保留在数据库方向；而 AP 则更关注吞吐量。就像灵活的小汽车和大轮船一样，逐步形成了大数据方向。

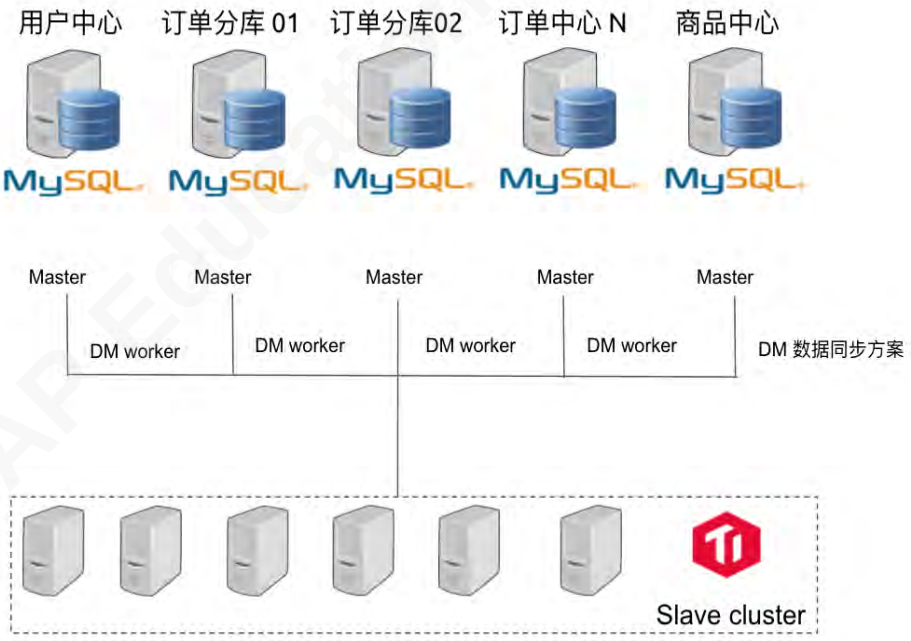
分布式技术的发展逐步解决了数据容量爆炸的问题，而分布式关系型数据库同时满足了 OLTP 的需求，也解决了数据容量的问题，在此基础上，很多传统 AP 的技术可以在分布式架构上进行再融合。实现了更大数据容量的混合数据库（HTAP）。同时业务创新场景多样性，在使用层面模糊了 OLTP/OLAP 的划分，比如业财一体、后台运营、客服后台、大屏展现、用户画像、小报表系统等。从这个角度看，HTAP 又是一个数据服务需求。这个数据服务需求核心诉求是数据服务的统一。

TiDB HTAP 之路

TiDB 被用于“意外”用于数据中台

前面我们说过，可以把 TiDB 比喻成一款大号的 MySQL ，最早 TiDB 是为了解决在线业务的分库分表问题。

为了方便大家数据迁移，我们构建了一套从 MySQL 迁移、合库合表、同步的 Data migration 解决方案。基于此方案，多套 MySQL 业务系统可以实时同步到一个 TiDB Cluster 上，这个 TiDB Cluster 类似大数据里的一个数据汇总层（ODS）。



为什么很多用户选择 TiDB 做为数据中台底座呢？在这个架构下，TiDB 很多天然的特性被用户发现，包括：

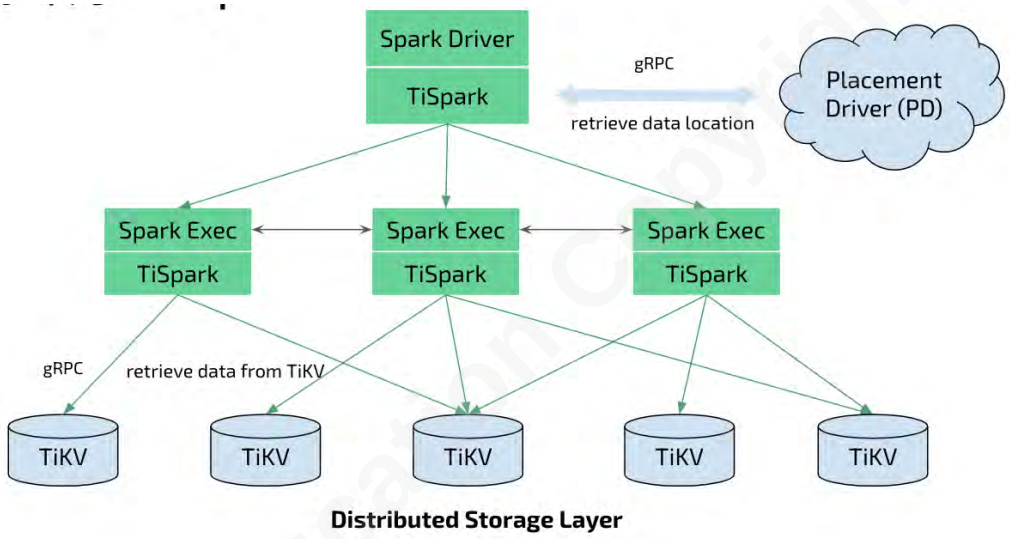
- 海量存储允许多数据源汇聚，数据实时同步。
- 支持标准 SQL，多表关联快速出结果。

- 透明多业务模块、支持分表聚合后可以任务维度查询。
- TiDB 最大下推机制、以及并行 hash join 等算子，决定的 TiDB 在表关联上的优势。

而这些特性很适合诸如后台运营系统、财务报表、大屏展现、用户画像等数据中台业务。

引入 spark 来缓解数据中台算力问题

TiDB 虽然有上面优点，但 TiDB-Server 毕竟是一个面向 OLTP 的业务，对于那些 OLAP 中间结果过大的查询，还是会造成内存超度使用，甚至 OOM 的问题，所以为了满足用户的需求，我们借助了社区的力量，引入大数据 Spark 的生态，让 Spark 识别 TiKV 的数据格式、统计信息、索引，执行器，最终构建了一个跑在 TiKV 上的 Spark 计算引擎，TiSpark 。进而实现了一个分布式的技术平台，在面对大批量数据的报表和重量级 Adhoc 提供了一个方案。



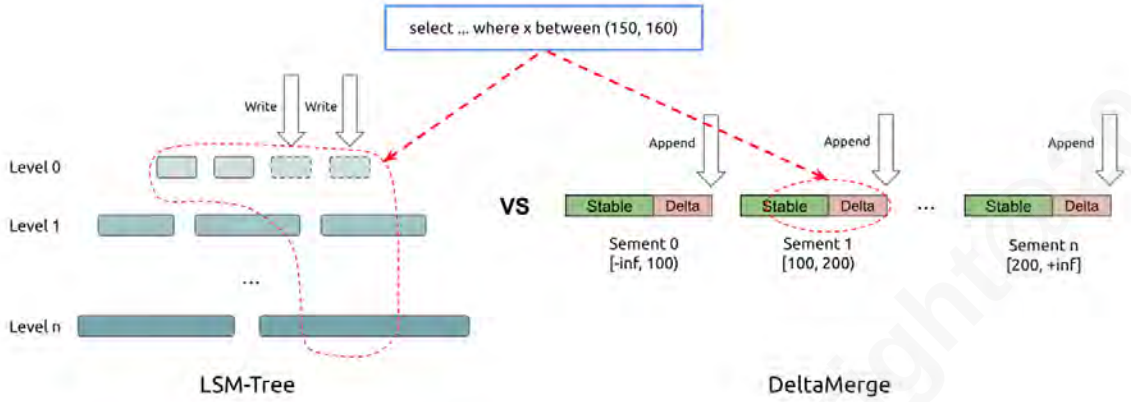
但 Spark 只能提供低并发的重量级查询，在从应用场景，很多中小规模的轻量 AP 查询，也需要高并发、相对低延迟技术能力，在这种场景下，Spark 的技术模型重，资源消耗高的缺点就会暴露。

更重要的 TiDB 本身定位是一个 OLTP 系统，很多 OLTP 查询在这种架构下，是和 TiSpark 共用一套底层的存储系统（TiKV），OLTP 与 OLAP 资源隔离很难通过软件层面彻底解决。

物理隔离是最好的资源隔离

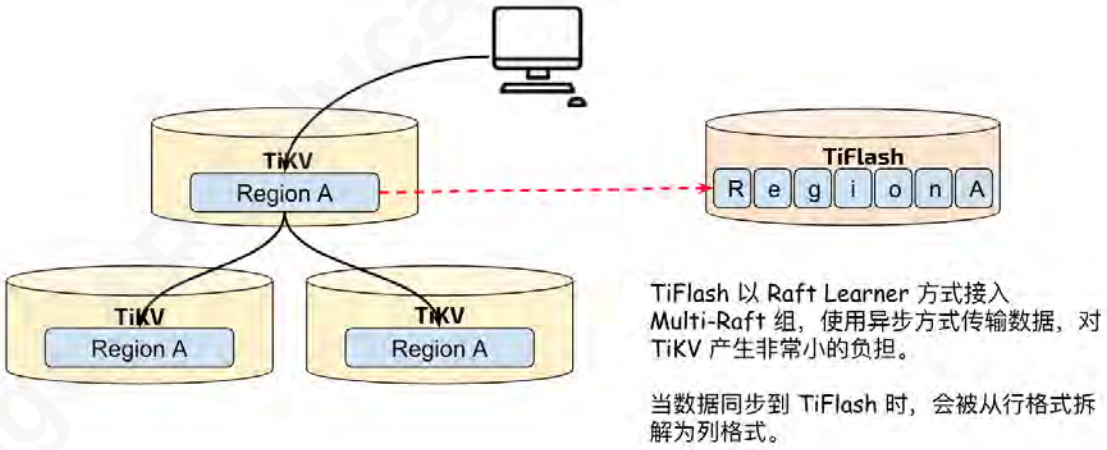
数据库资源隔离的角度看，依次是，数据库软件层、副本调度、容器、虚拟机、物理机，越接近物理机隔离性越好，在传统的 Master-Slave 的架构下，我们经常听到读写分离这个说法，其实也是一个资源隔离的问题。既然隔离，就要有一个单独的副本进行 AP 查询。而列存天然对 OLAP 查询类友好，所以我们选择将这个副本放到一个列式引擎上。

而列式存储引擎，需要按照列为单位进行存储，每个列一个会成立一个单独的对象，这种引擎对批量写入友好，最大的挑战、在于实时更新，所以基于此，我们借鉴 LSM 思想，在列式引擎上引入了 Delta tree 的方式，最终实现了一个支持准实时更新的列式引擎，Tiflash。



行列数据同步，Raft-base 最佳方案

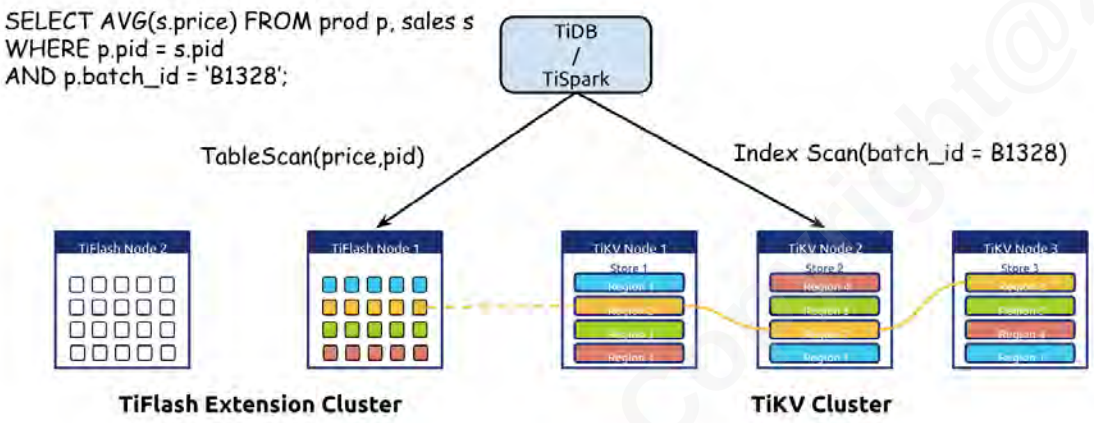
下一步，就是怎么将这个副本同步到这个列式引擎上，最简单的方式就是使用 Binlog 这类数据库日志，但因为我们要强调数据库的实时性，所以我们采用了复制效率更底层更高效的 raft。前面我们提过原有 TiKV 默认三副本，采用的 raft-base 的复制方式，我们对 raft 进行了改造，将一个副本 raft 副本设置为只负责同步，没有投票权的角色，我们叫它 Learner。这样设计的好处是既保障了 Raft Group 的写入效率，又保障了列存副本的极低延迟的异步同步，同时还规避了复制组脑裂问题。



计算统一，TiDB 优化器近水楼台

我们一直强调，对使用者来说，统一与易用是最朴素的需求，所以接下来的工作就是将在技术节点上进行服务的统一，前面说过，我们已经构建了一个支持标准 SQL 的 TiDB-Server，所以接下来的工作，就是将列存的信息暴露给 TiDB-Server，设计新的统计信息规则，与 CBO cost 模型，

最终让 TiDB-Server 的优化器可以通过新的 Cost 模型来自由选择数据寻址路径，形成一个包括行式存储与列式存储执行计划。比如下面的例子：
例如下图中的 SQL 语句，SELECT AVG（s.price） FROM prod p, sales s WHERE p.pid = s.pid AND p.batch_id = 'B1328', TiDB-Server 会根据数据统计信息评估最终只需要计算 sales 表的 price 列的平均值，就没有必要读取其他列的信息。同时 batch_id 这个列，在 TiKV 上有索引，所以，最优的方式在 TiKV 上通过索引过滤，过滤完的数据，再通过 TiFlash 的列存中进行读取和聚合。既节省了 I/O 又降低了传输的网络带宽。



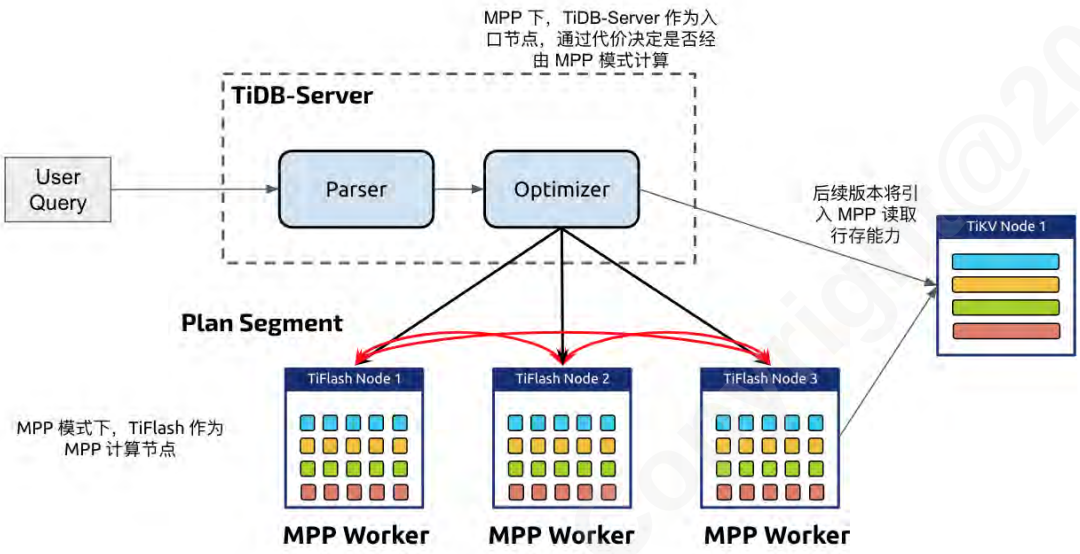
算力再次不匹配，引入 MPP

完成上一步后，下一步是什么？在 OLAP 里，经常会出现多个大表之间的关联。这里面有暴露两个新的问题：

- 我们前面说过分布式架构，一个很重要的优化，就是最大程度的下推算法（在分布式节点上的数据寻址的过程，尽快完成计算与过滤），那之前的架构，在 Join 是推不下去的，具体原因大家可以想想为什么。那么如何实现 Join 下推呢？
- 之前的架构，单一查询总是在某一个具体计算节点完成，如何实现计算节点在单一 SQL 的扩展呢？

在大数据的技术历史上，大家应该听过 MMP 引擎，什么是 MPP 数据库？（Massively Parallel Processing）
MPP 架构是将任务并行的分散到多个服务器和节点上，在每个节点上计算完成后，将各自部分的结果汇总在一起得到最终的结果。
回答我们上面的问题，如果要让一个 join 在多个节点并行执行，那么需要将 join 的两个表的分片在节点分布一致，如果不一致呢？需要通过网络的将分片临时拷贝一份进行计算，这个过程叫 shuffle，所以 MPP 计算模型，本质是通过网络与存储成本来置换计算资源（Trade Off）。来解决大数据量的需要并行计算场景。比如大报表。
所以，我们引入了 MPP 计算框架。如下图：

在下图中，优化器（Optimizer）会根据数据统计信息，决定是否启动 MPP 模式进行计算。如果启动 MPP ，那么会首先在各个 TiFlash 中将多表连接的数据分布一致，也就是 3 个 TiFlash 上面的红色箭头所示，接下来，每个 TiFlash 节点上面的 MPP Worker 会负责表连接在多个节点上的并行执行。最终，将各个表分片的连接结果汇聚到 TiDB-Server 中。



HTAP 下一步探索

最后，我们总结一下，TiDB HTAP 之路已经完成了如下几步：

1. 分布式数据库是在大数据规模下提供 HTAP 的基础。
2. TiDB-Server 最大程度下推算法与 Hash Join 关键算子提供了基础 AP 能力。
3. 借助生态，让 Spark 跑在 TiKV 上。
4. 行列混合引擎，列式引擎提供实时写入能力。
5. 行列引擎采取 Raft-Base replication，解决了数据同步效率。
6. TiDB-Server 听过了一个支持 SQL，可以自动路由行列引擎，统一数据查询服务。
7. MPP 解决 多表 JOIN 场景下的计算节点的扩展性与并行计算问题。

看到 TiDB HTAP 发展路径，我们会发现，这里面既有产品内嵌功能，又有生态的数据的连同。这是两条工程化的思路，接下来，HTAP 会逐步转换成“数据服务统一”的代名词，这两条发展思路都会有不同的探索：

1. 产品内嵌功能的迭代，由一些具体产品来完成 HTAP。
2. 整合多个技术栈与产品，并进行数据的连同。形成服务的 HTAP。

过去 10 年，OLAP 的场景基本上都基于数仓，而流计算的发展，将数仓架构分为几个阶段：

- 最早的批处理（ETL）的离线数仓。
- 批、流结合的 Lambda 架构。

- 流计算为主的 Kappa 架构。

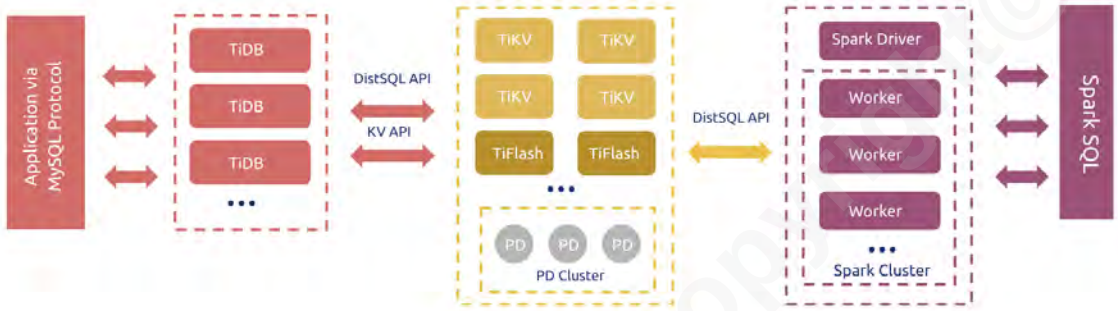
另外一个维度，分布式技术的发展，解决了数据容量、吞吐量的扩展性，在此基础上，很多 OLAP 的技术又可以在更大数据量基础上与 OLTP 进行再融合，比如分区、列式存储、并行计算等。

当这两个发展方向终究碰撞到了一起，流计算在复杂计算中的天然限制，可以在分布式 HTAP 里得到解决。而流计算的实时计算能力，为不同数据技术栈与产品提供了丰富多样数据连通能力；流计算 + 基于分布式的 HTAP 产品，将形成了更有爆发力的 HTAP 数据服务。

第 8 课：TiDB 关键技术创新

在之前的章节，我们阐述了数据技术范畴里的基础技术元素，然后在这些基础技术元素进行各种 Trade Off（选择与平衡），最后分别构建了三个分布式系统；

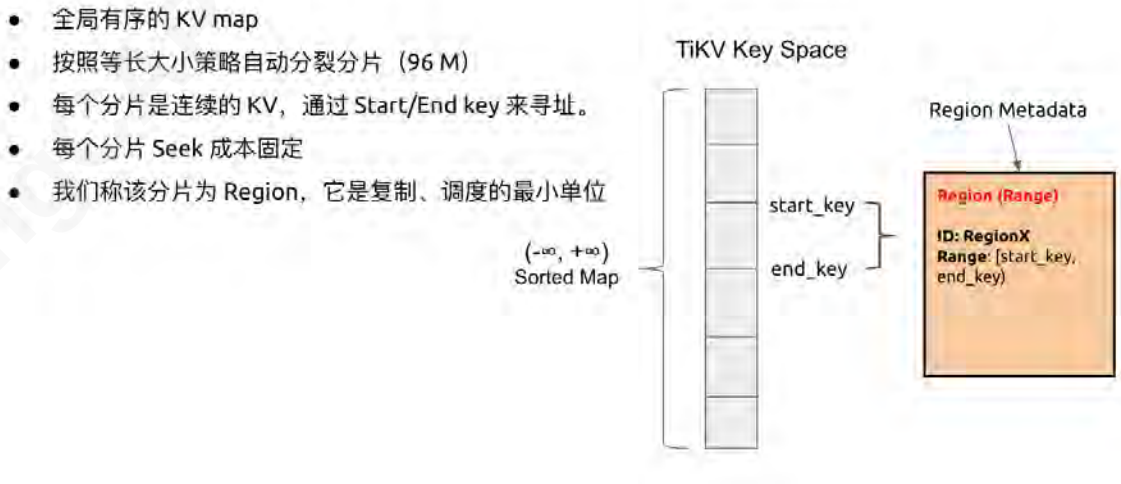
- 分布式的 KV 存储系统
- 分布式 SQL 计算系统
- 分布式的 HTAP 架构系统



在这个章节我们回顾下这三套架构，关键技术与功能创新是什么？

自动分片技术是更细维度弹性的基础

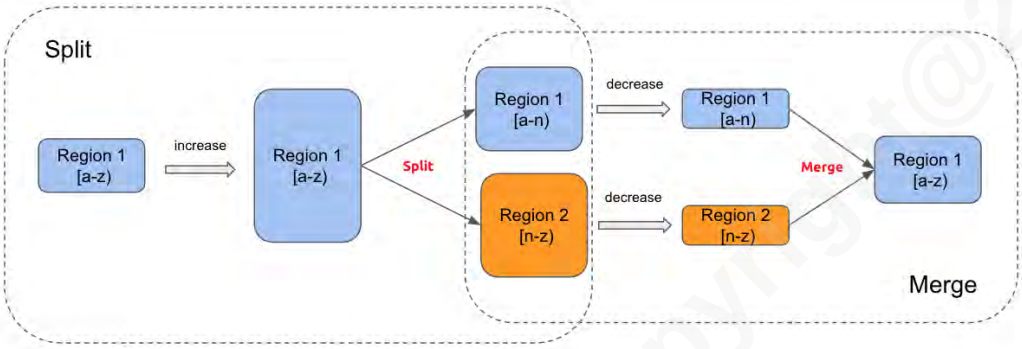
自动分片技术实现了，随着表数据量的增加，会自动分裂出等大小的分片，就像一个细胞的成长后，分裂成两个细胞一样，这种机制将整个系统的弹性颗粒度细化到 96 M，甚至更小，同时保证了每一个分片的 Seek 成本固定。一个现实的场景意义是，无论你的表多大，比如万亿规模，对于 OTLP 场景的点查、小范围查询响应时间都是固定的。



弹性的分片构建成了动态的系统

分片是整个系统的细胞，细胞不只是会分裂，还会自动合并与释放。这种动态的设计，让整个系统变的更有活力。

- 自动 merge
  - 96 MB 自增分片
  - 20 MB 合并分片

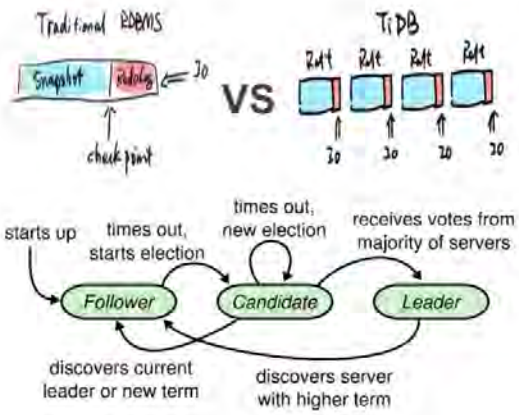


Multi-Raft 将复制组更离散

Raft 是强一致算法，保障了 RPO（Recovery Point Objective）业务系统所能容忍的数据丢失量为 0，这在很多金融级场景里至关重要。

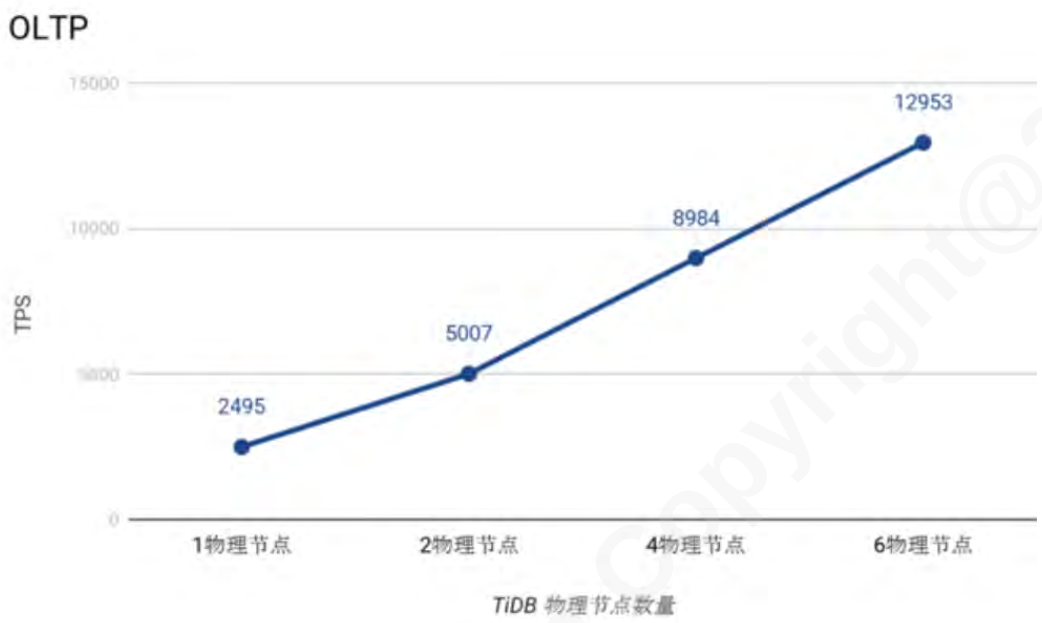
在 TiKV 设计里，我们把自动分片（Region）机制与 Raft 进行了结合，形成了以分片为单位的复制组 （Region base Multi-Raft ），也就是说，一套集群里同时存在几十万个独立的复制组，这种设计大大提供了集群的整体可用性，同时也大大优化了 RTO（Recovery Time Objective）灾难发生到恢复的的时长。

- Raft 、Multi - raft
- leader、follower、learner
- 目前是强主模式、读写在 leader 上
- 4.0 版本开启 follower read



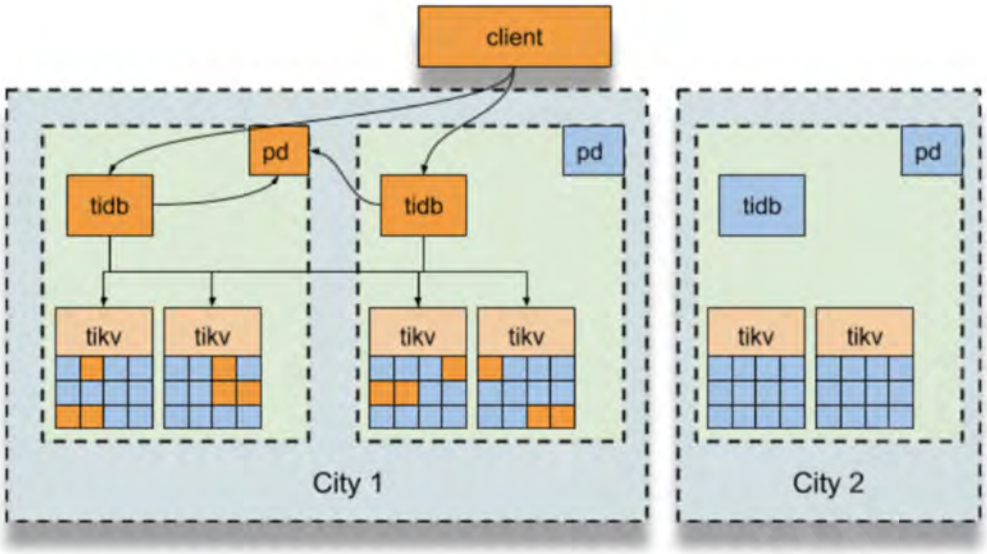
基于 Multi-Raft 实现写入的线性扩张

我们之前提到，在数据库系统里，**写入是最昂贵的**，传统数据库集群写入上限是依赖于硬件，是固定的。Region base Multi-Raft 的机制是集群同时存在几十万个复制组，也就是有几十万个写入点（Leader）,这些 Leader 会自动在物理的存储节点上进行搬迁与平衡，当我们新增一个物理节点时，也就意味着整个集群的写入容量会进行线性增长。



基于 Multi-Raft 实现跨 IDC 单表多节点写入

传统的数据库每个时间点只能有一个写入点 Master，Master 发生故障后，整个集群需要 Failover（主从切换），而 Region base Multi-Raft 的机制，实现了一个表可以同时有多个写入点，TiKV 的调度机制，可以识别单个节点的物理信息，比如 IDC、REC、Host 等（机房、机柜、宿主机等），并进行约束与绑定。实现了一个复制的分片在机房、机柜等维度打散。最终可以实现一张表跨 IDC 多点写入。这在很多金融场景容灾的场景里非常有价值。

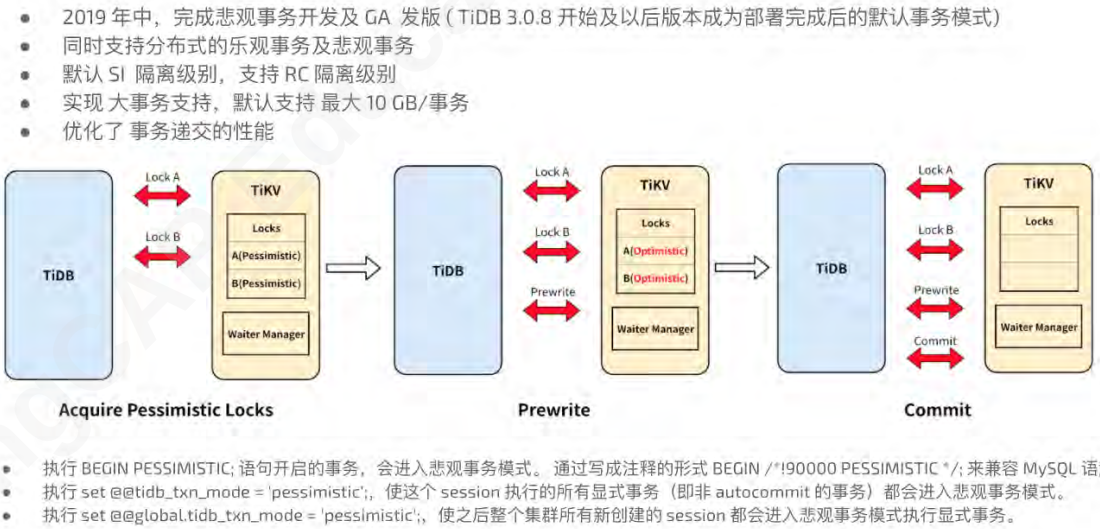


两地三中心 Raft based 容灾方案

去中心化的分布式事务

TiDB 在事务做了很多优化：

- 从结构上，是个去中心化的两阶段提交，这解决了事务能力的扩展性；



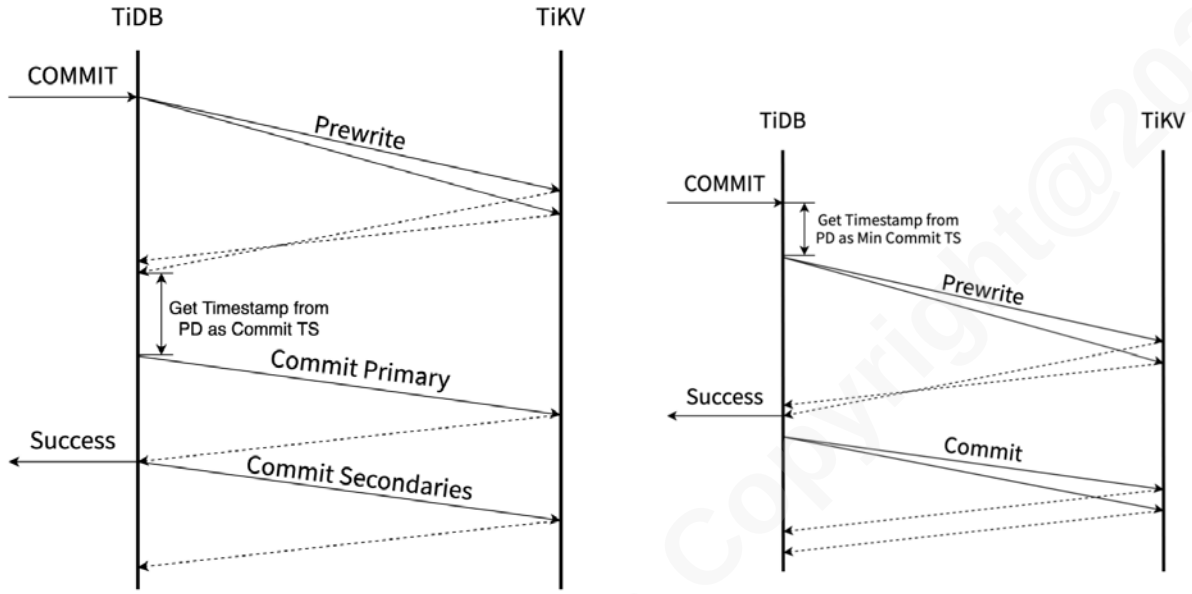
- 在 5.0+ 的版本里，在针对 OLTP 常见的高并发，小数量的写入场景，TiDB 事务在第二阶段提交采取了异步处理方式（Async Commit），变相的实现了 1PC 的效果。大大优化了分布式事务里 2PC 通用的延迟问题。当然这种实现有两个比较大的挑战：
  - 如何确定所有 keys 已被 prewrite。

- 如何确定事务的 Commit TS。

TiDB 解决思路是：

- 将所有事务行的 Key 与事务的 Primary key（状态位）进行索引的 mapping。
- 通过 PD 来保障全局时间递增。

更详细的解决方法在这里不再展开。



Local Read and Geo-partition

Geo-Partition 指多地多活跨地域数据分布，在 TiDB 5.0 中将中央的授时服务改为了分布式授时服务，能够提高场景的数据库性能和降低延迟，可以在多个 IDC 甚至中国，欧洲，美国等地同时提供服务，又可以提供数据安全合规，不出镜的访问场景。 总结下来为：

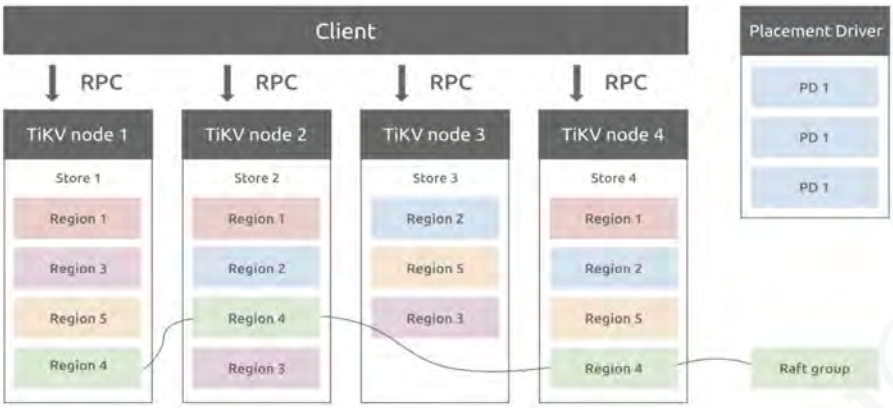
1. 多地部署支持，低访问延时。
2. 数据安全合规，符合数据不出境场景。
3. 支持异地多活容灾。
4. 支持冷热数据分离。

更大数据容量下的 TP 与 AP 融合

扩展性解决了更大的数据容量，在此基础上，很多 OLAP 的技术可以进行在融合，是新更大数据容量的 HTAP；

- TiDB 引入了实时更新的列式引擎，即解决了资源隔离，又提升了 AP 效率。
- 在列存上引入 MPP 模型，实现了 SQL join 的下推与并行处理。
- 通过 Raft-base replication 实现了更时效性。
- 融合大数据生态，比如 TiSpark。

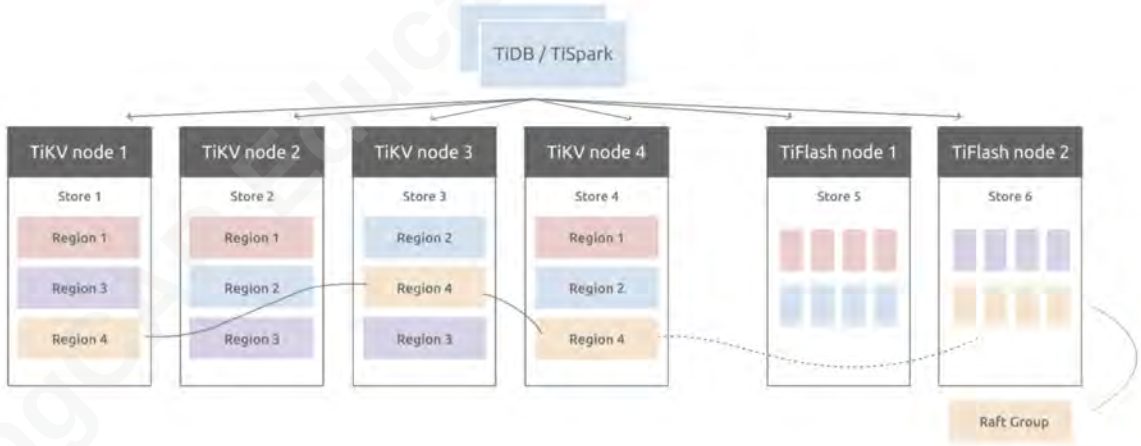
更重要的是 TiDB 这套机制体现的兼容性，可以预期，将来会出现很多 TiXX 的项目。



数据服务的统一

TiDB-Server 是个标准 SQL 引擎，理论上，我们可以构建一个统一的 Proxy 层，通过各种数据接口与下层多种数据存储引擎进行数据交互，但这个方案最大的挑战将来自，多种数据引擎如何保证最佳执行计划。

TiDB 的 CBO 可以采集行列 Cost 模型进行配置，并同步收集不同引擎的统计信息，统一进行最佳执行路径的选择。这是数据服务统一的第一步，可以预期，未来在统一数据服务的方向将会更多创新。



第 9 课：TiDB 典型应用场景及用户案例

产品能力概述

TiDB 设计目标：

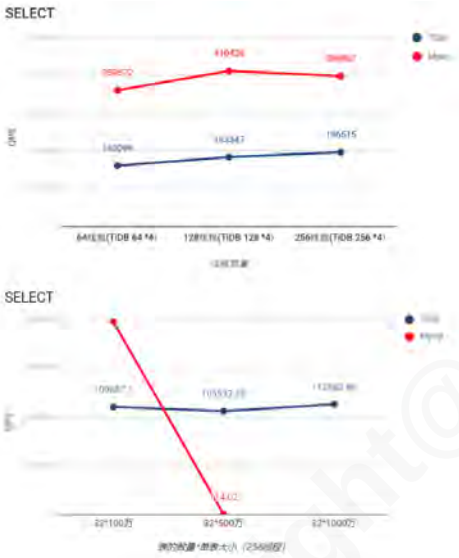
- 横向扩展能力（Scale out）
- 兼容 MySQL
- 完整事务（ACID）+ 关系型模型
- HTAP

典型场景

OLTP Scale

传统 OLTP 面临的数据容量的挑战，包括高并发、大数据量、高可用。

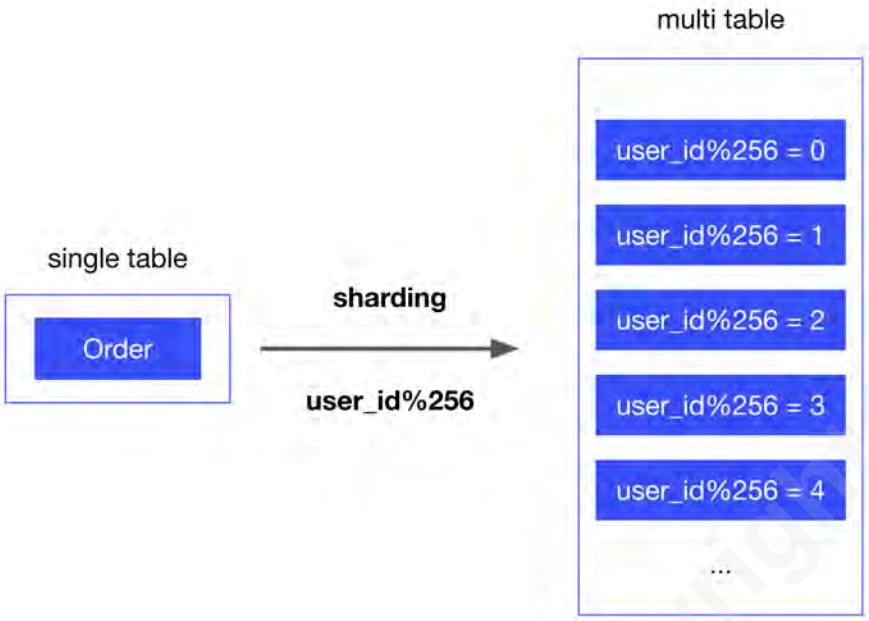
- 高并发（考验计算能力）
- 大数据量（考验存储能力）
- 高可用性（持续服务能力）



传统的 Sharding + Proxy 解决方案：

为什么分表？

以 MySQL 为例，分库分表从拆分阶段上讲，拆分为分表、分库，一般来说是先进行分表，分表的原动力在于 MySQL 单表性能问题，相信大家都听说过类似这样的话，据说 MySQL 单表数据量超过 N 千万、或者表 Size 大于 N 十 G 性能就不行了。这个说法背后的逻辑是数据量超过一定大小，B+Tree 索引的高度就会增加，而每增加一层高度，整个索引扫描就会多一次 IO 。对应的查询响应时间增加。

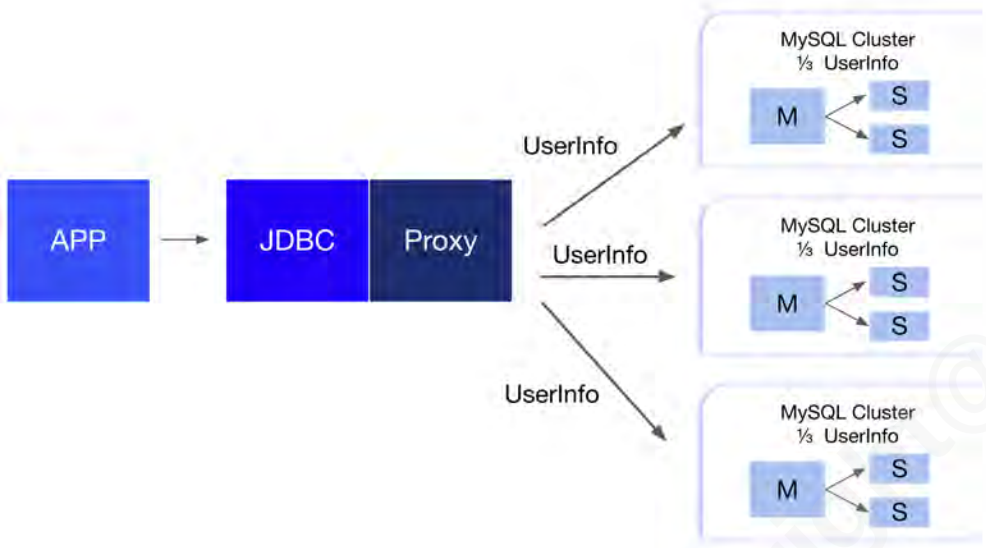


为什么需要分库？

分库主要由于 MySQL 容量上，**数据库的写入是很昂贵的操作**，MySQL 本身有很多优化技术，即使如此，写入也存在放大很多倍的现象。同时 MySQL M-S 的架构虽然天然地支持读流量扩展，但由于 MySQL 从库复制默认采用单线程的 SQL thread 进行 Binlog 顺序重放，这种单线程的从库写入极大地限制整个集群的写入能力，（除非不在意数据延迟，而数据延迟与否直接影响了读流量的可用性）。MySQL 基于组提交的并行复制从某种程度上缓解了这个问题，但本质上写入上限还是非常容易达到（实际业务也就 小几千 的 TPS ）。说到这，目前有一些云 RDS 通过计算与存储分离、log is database 的理念来很大程度解决了写入扩大的问题，但在这之前，更为普遍的解决方案就是把一个集群拆分成 N 个集群，即分库分表（sharding）。为了规避热点问题，绝大多数采用的方法就是 hash 切分，也有极少的范围、或者基于 Mapping 的查询切分。

为什么需要中间件方案

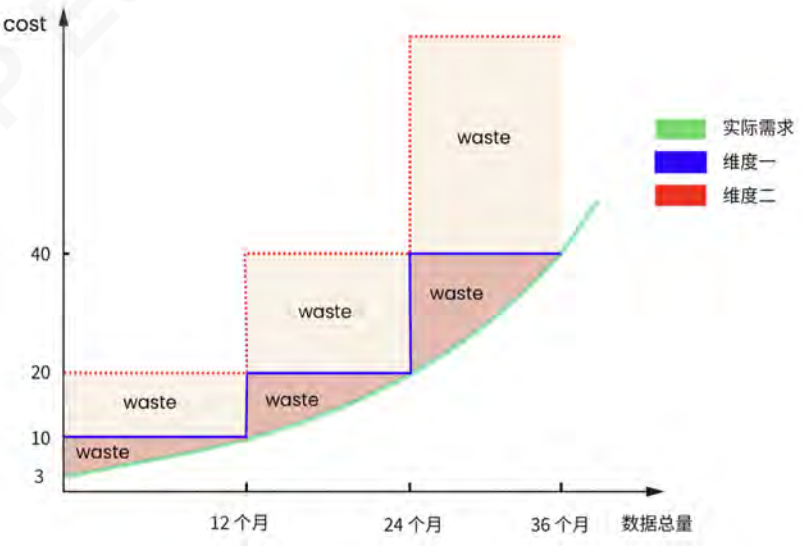
既然做了分表，那数据的分发、路由就需要进行处理，自下而上分为三层，分别 DB 层、中间层、应用层。DB 层实现，简单来说就是把路由信息加入到某个 Metedata 节点，同时加上一些诸如读写分离、HA 整合成一个 DB 服务或者产品，但这种方案实现复杂度非常高，有的逐步演变成了一种新的数据库，更为常见的是在中间层实现，而中间层又根据偏向 DB 还是偏向应用分为 DB proxy 和 JDBC proxy。



TiDB VS Sharding + Proxy

传统的 Sharding + Proxy 有很多成本，以电商场景为例：

- 业务多维度查询支持不友好
  - 分表需要一个选择一个业务维度，一般是按照“用户”维度进行拆分，如果按照“商家”维度查询，需要遍历所有的分表。
- 即使采用了 Hash 算法进行 sharding ，依然会出现分区偏斜
  - 二八原则在多数业务中都是适用的，及 80%的业务流量是由 20%的用户带来的，且这些用户还在随着业务发展不断变化。试想下如果带来业务压力的这 20%用户通过 hash 算法后 有一部分放在了一起那么我们之前所做的 分库分表的努力也就被大大的削弱了。根据墨菲定律这样糟糕的事情是一定会发生的。
- Sharding + Proxy 分表是预拆分，需要提前把很长的空间预留，成本很高

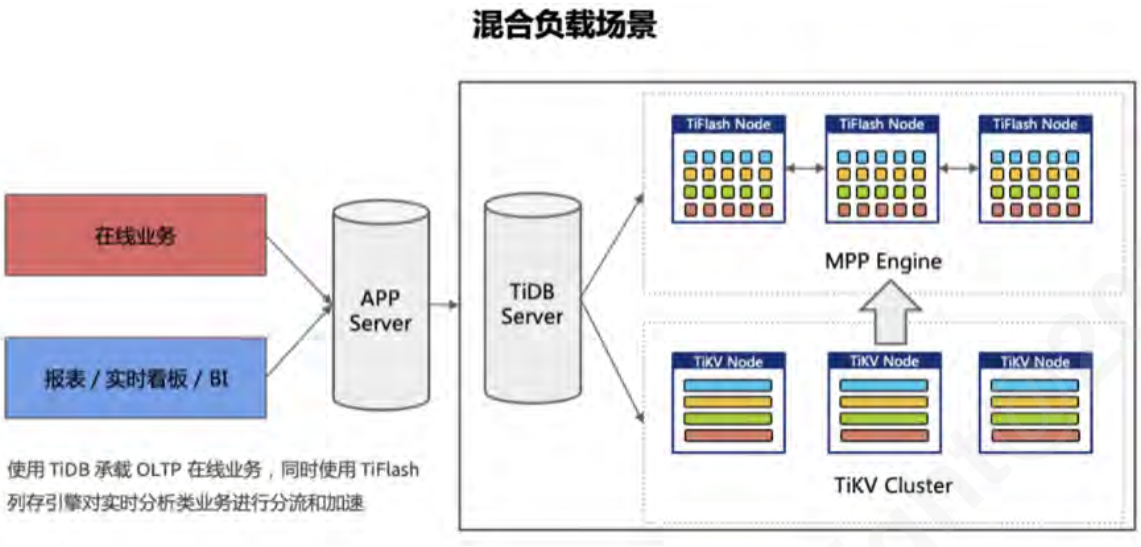


- Resharding 成本
  - 如果一次分了 100 张表，下次拆分只能安装一百的倍数，缺乏弹性。
- 其他
  - 应用开发复杂度变大，全局事务限制、跨分片的关联等等；大大增加开发周期和开发成本
  - 数据 CDC 复杂度增高、数据一致性确认周期变长；往往我们需要进行相关业务统计都要经过一天的等待才能看到前一天的运营数据
  - 运维故障点变多，运维成本大大增加。

类型	传统数据库中间件 / 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

Real-Time HTAP

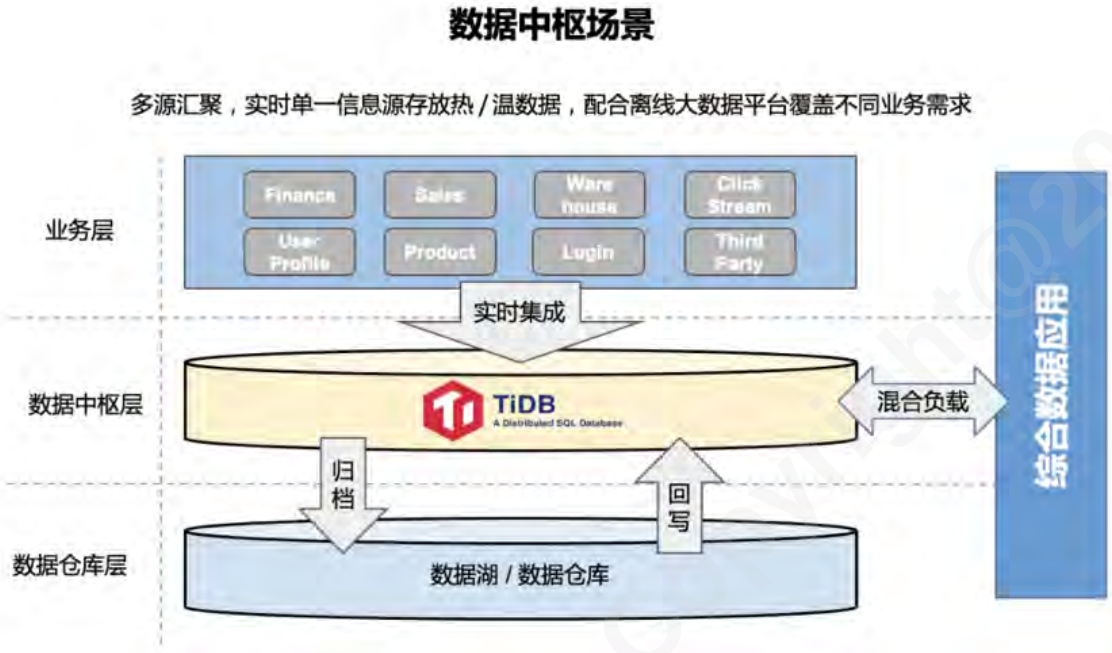
在混合负载场景下，使用 TiDB 可以承载 OLTP 类业务，同时使用 TiFlash 加速实时报表，加速对分析业务的查询响应。从架构上来说，以往需要有不同的数据链路去维护不同数据引擎之间的数据流转和存储，现在对用户而言只需要一个入口，一个 APP Server，不同的业务类型可以不加区分地接入这个 APP Server，APP Server 向 TiDB 发送请求，**TiDB 将不同的业务向不同的引擎进行分流和加速，整套技术架构栈变得非常简洁。**



TiDB 本身是一个带有 OLTP 类属性的 HTAP 数据库，对于任何实时的删改增落地，TiDB 是可以达到实时响应，整条数据链路可以实现秒级查询。TiDB 本身的行列混合属性也意味数据落地不论是明细数据的明细查询，还是对于数据的各种不同维度的实时聚合，都可以很好地响应。

在数据中枢场景中，假设前端有多个业务线，每个业务线都有自己不同的 OLTP 类数据库，例如财务、EPR、销售、仓储数据库，有 Click Stream、有 User Profile、也有产品库，还有用户登录和第三方不同的数据源，这些数据存放在各自的数据库中，可以通过 CDC 的方式或者通过 Kafka 实时集成到 TiDB 里面。这些从不同数据源、不同业务线整合过来的数据，可以放在数据中枢层。**数据中枢层是相对于离线的数仓层来说的一个概念**，首先他只存放一部分时间段内的数据，而数仓可能会放更久远的历史数据；数据中枢层更倾向于是存放温数据和热数据数据，可以提供实时的数据存取和查询服务。相对于大数据和离线数仓层而言，数据中枢可以直接对接数据应用端，

作为一个既可以承接高并发访问，又能够以数据服务的形式来提供全量数据的存取，而离线数仓和数据湖更倾向于离线的方式，通常提供不太新鲜的数据来进行报表与 BI 类查询。



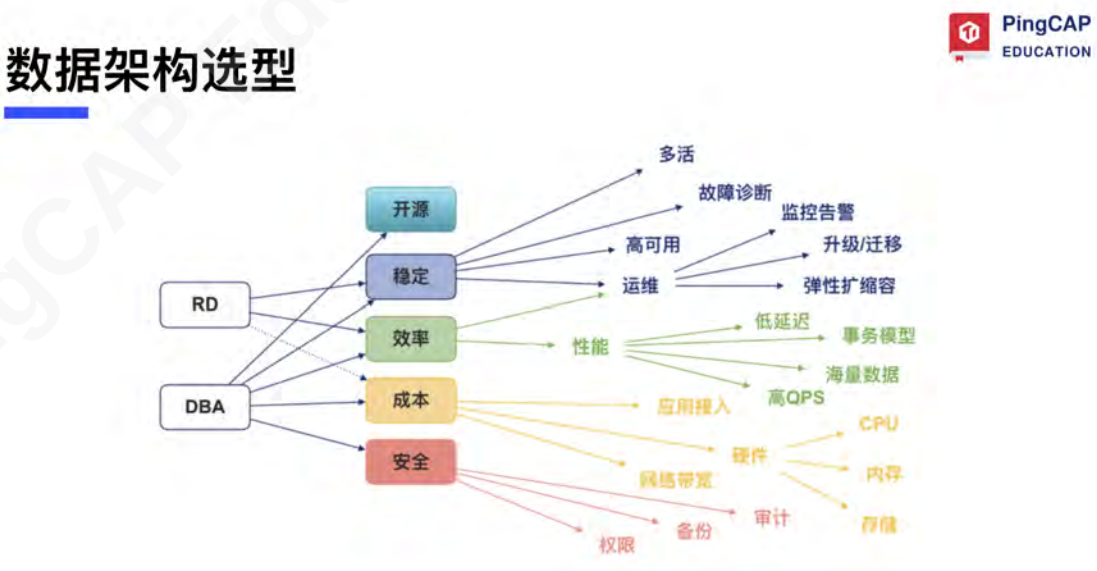
当 TiDB 集成到整个数据平台当中，他充当了一个数据中枢的角色。即使数据平台中已经有了离线数据仓库层和 Hadoop 平台，仍然可以把 TiDB 放在业务层、Hadoop 层，或者在数据仓库层之间，作为提供一个实时数据存储和管理的平台，用来满足越来越多用户对于实时存取与实时分析的需求。

常见用户案例

数据架构选型

对于一个线上的服务来说，要考虑以下几点：

- 稳定。对于任何一个线上服务来说，可以容忍交易稍微慢一点，但一定不可能容忍频繁宕机。稳定是第一要务，脱离了稳定，效率没有任何意义。
- 效率。在系统非常稳定的情况下，速度越快，就意味着用户体验越好。比如外卖下单，秒接单的用户感受一定很好。如果 30 分之后才接单，用户会想到底是系统出了问题还是外卖小哥偷懒。
- 成本。当稳定有了，效率也有了，就要思考花的成本值不值？因为成本降下来才能赚到收益。
- 安全。安全是一个大家都绕不开的问题。但凡做交易，大家都担心自己的数据被泄露。

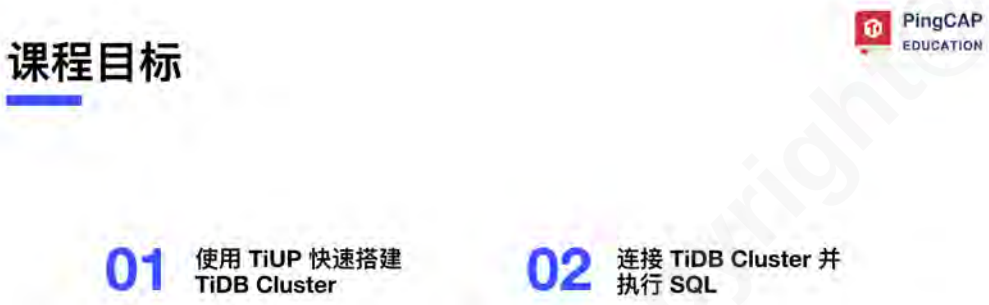


所以在数据库上面，最关注的就是保稳定、提效率、降成本、保安全。除了这四项之外，还有就是开源。在做技术选型时，我们希望这个数据库是个优秀活跃的开源产品。当我遇到一些问题有人能热心的帮助我解答疑问。另外当我想为产品做一些贡献的时候，是可以和这个社区一起迭代成长的。

第 10 课：TiDB 初体验

课程目标

- 使用 TiUP 快速搭建 TiDB Cluster
- 连接 TiDB Cluster 并执行 SQL



TiUP 介绍

TiUP 是 TiDB 4.0 版本引入的集群运维工具，通过 TiUP 可以进行 TiDB 的日常运维工作，包括部署、启动、关闭、销毁、弹性扩缩容和升级 TiDB 集群，以及管理 TiDB 集群参数。
参考文档：https://docs.pingcap.com/zh/tidb/stable/production-deployment-using-tiup

TiUP Playground 介绍

TiUP 的 playground 组件用于部署本地集群。我们本课将通过这个组件为大家演示 TiDB Cluster 的部署，大家也可以在自己的环境中尝试部署一套 TiDB Cluster， 你会发现非常的简单。部署之后，我们会尝试连接到 TiDB Cluster 上，并观察集群中的各个组件。

TiUP Playground 部署本地环境（第一步）

- TiUP Playground 环境确认

基本环境需求：Mac 系统或者 Linux 系统单机，可以连接到外网环境

TiUP Playground 部署本地环境（第二步）

- 下载并安装 TiUP

下载并安装 TiUP：

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

声明全局环境变量：

source <profile 文件绝对路径>

注意：TiUP 安装完成会提示对应的 profile 文件的绝对路径，所以 source 操作需要根据实际位置进行操作。

TiUP Playground 部署本地环境（第三步）

- 启动集群

演示运行最新版本的 TiDB 集群，其中 TiDB、TiKV、PD 和 TiFlash 实例各 1 个：
tiup playground
或者
演示指定 TiDB 版本以及各组件实例个数：
tiup playground v5.0.0 --db 2 --pd 3 --kv 3 —monitor

--monitor 表示同时部署监控组件。
最新版本可以通过执行 tiup list tidb 来查看。

结果：
CLUSTER START SUCCESSFULLY, Enjoy it ^-^
To connect TiDB: mysql --host 127.0.0.1 --port 4000 -u root
To connect TiDB: mysql --host 127.0.0.1 --port 4001 -u root
To view the dashboard: http://127.0.0.1:2379/dashboard
To view the monitor: http://127.0.0.1:9090

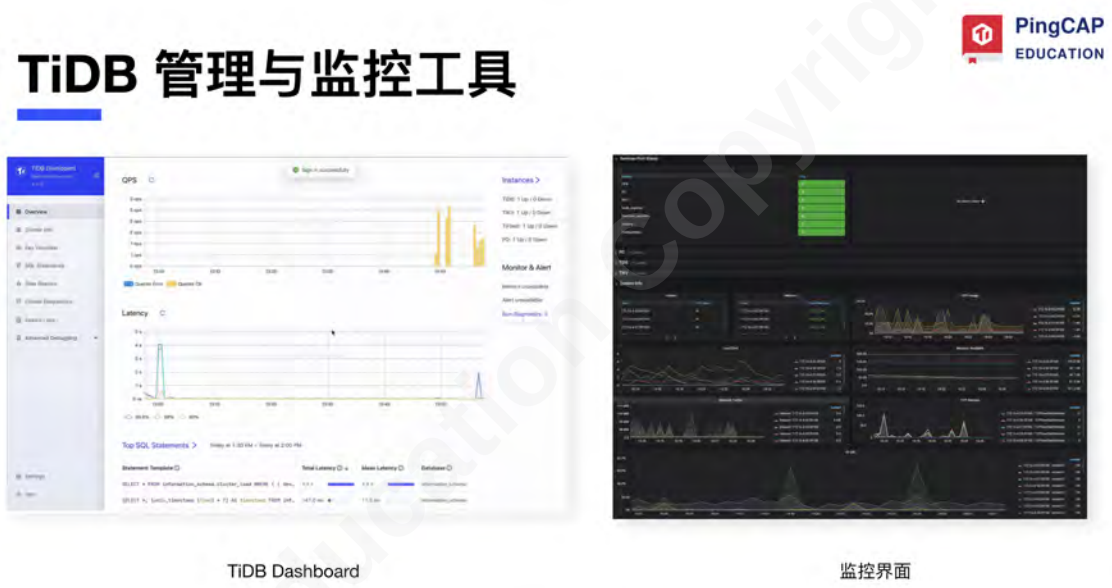
注意：以这种方式执行的 Playground，在运行结束后 TiUP 会清理掉原集群数据，重新执行该命令后会得到一个全新的集群。 若希望持久化数据，可以执行 TiUP 的 --tag 参数：tiup --tag <your-tag> playground ...，详情参考 TiUP 参考手册（ https://docs.pingcap.com/zh/tidb/stable/tiup-reference#-t---tag-string ）。

连接 TiDB Cluster

tiup client
或者
mysql --host 127.0.0.1 --port 4000 -u root

TiDB 管理与监控工具

通过 http://127.0.0.1:9090 访问 TiDB 的 Grafana 管理界面。
通过 http://127.0.0.1:2379/dashboard 访问 TiDB Dashboard 页面，默认用户名为 root，密码为空。
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。
通过 TiDB Dashboard 查看整个集群中 TiDB、TiKV、PD、TiFlash 组件的运行状态及其所在主机的运行状态。



集群环境清理

通过 ctrl + c 停掉进程
执行以下命令：
tiup clean –all