TDSQL
腾讯分布式数据库

TDSQL 核心架构

原理解析

安全可控的金融级分布式数据库

腾讯云TDSQL出品

TDSQL 核心架构与模块技术原理解读

为帮助开发者更好地了解和学习分布式数据库技术，2020 年 3 月，腾讯云数据库、云加社区联合腾讯 TEG 数据库工作组特推出为期 3 个月的国产数据库专题线上技术沙龙《你想了解的国产数据库秘密，都在这！》，邀请数十位鹅厂资深数据库专家深入解读 TDSQL、TBase、CynosDB 三款鹅厂自研数据库的核心架构、技术实现原理和最佳实践等。本专辑为 TDSQL 专题，将为大家带来腾讯自研分布式数据库 TDSQL 核心架构及模块特性的深度分析讲解。

目录

第一章：腾讯分布式数据库 TDSQL 金融级能力的架构原理解读

本章节分为五个部分：

- 产品简介以及适用场景
- TDSQL 架构分析及模块介绍
- 数据一致性保障
- 分布式 TDSQL 实践
- 数据同步与备份

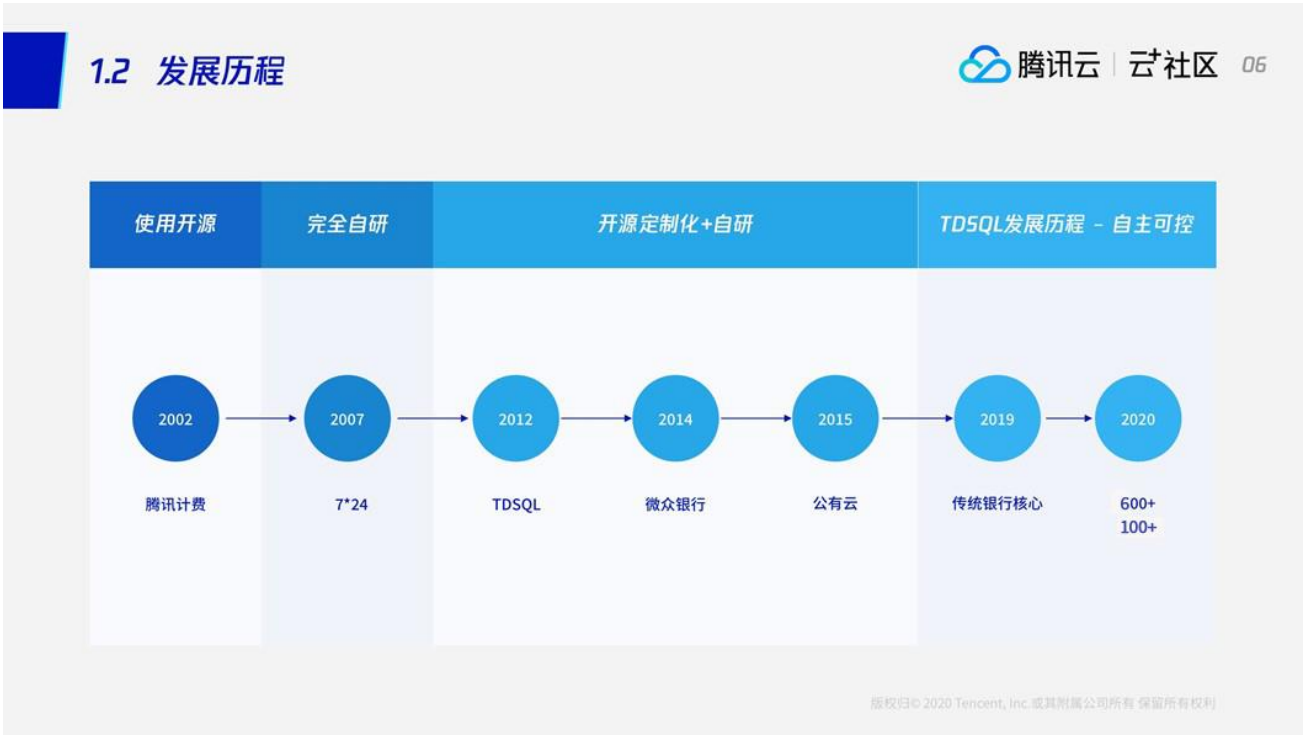
1 TDSQL 是什么：腾讯如何打造一款金融级分布式数据库

TDSQL 是腾讯推出的一款兼容 MySQL 的安全可控、高一致性分布式数据库产品。这里我们强调一点——高度兼容 MySQL，TDSQL 完全兼容 MySQL 协议，并且做到完全安全可控、数据强一致性。第二是 TDSQL 具备分布式的特性，具备一个弹性扩展、高可用的架构。在互联网行业，海量的用户流量场景很常见，如果数据库不具备可伸缩性、可扩展性，是很难应对如：

电商的大型促销，春节抢红包等突增流量的场景，这些其实都是对数据库应对海量用户流量的考验。

目前 TDSQL 已经服务超过 600+的金融政企，行业覆盖银行、保险、证券、政务、互联网金融等各个领域。

TDSQL 最早可以追溯到 2002 年，当时是 TDSQL 的前身，作为腾讯计费平台部的一个数据库服务，当时使用了开源的 MySQL。2002 年-2007 年随着公司业务的发展，腾讯所面临的用户量的压力也越来越大，这个时候我们提出了 7×24 小时不宕机的高可用设计方案，来保证数据库能提供 7×24 小时不间断连续高可用服务。那个时候，腾讯的增值业务日渐成规模，业务对数据也越来越敏感，对数据可用性的要求越来越高，甚至平时还要防备一些像运营商的光纤被挖断等各种各样的异常场景。



在 2007 年-2012 年，这可能是互联网时代从互联网到移动互联网的发展的快速 5 年。当然，公司的业务也是突飞猛进。我们开始把这个高可用的数据库产品化。到 2012 年，TDSQL 的雏形已经形成，并开始作为一款内部产品，在公司内部提供金融级的数据强一致性、可靠性服务。

从 2012 年起，TDSQL 已经在腾讯内部做得已经比较成熟，已经是一个知名的产品了，但一直没有对外做商业化。2014 年恰逢一个很好的机会——微众银行的成立。微众银行做数据库选型的时候关注到了 TDSQL，经过反复测试验证，发现当时的 TDSQL 已经完全具备了微众银行对数据可用性和一致性的要求。借此机会，TDSQL 成功在微众银行投产，成为微众银行唯一的数据库，覆盖了银行的核心业务。

所以说 2014 年，TDSQL 完成了商业化，也实现了私有化部署。2014 年以后，TDSQL 推广到了很多银行、金融机构，这过程中是借鉴了 2014 年 TDSQL 在微众银行成功实施的宝贵的经验。因为在 2014 年微众银行的部署中，我们也踩了很多坑，也认识到在私有化部署环境的各种各样的挑战，并一一攻克了这些挑战。当 2014 年在私有化部署完成之后， 2015 年 TDSQL 上线公有云，继续通过公有云服务打磨自己的产品。

所以从 2012 年作为一个内部产品到 2014 年的私有化部署，再到 2015 年公有云上的部署，TDSQL 已经逐步从一个内部产品逐渐走向行业，成为一个正式对外的商用数据库。从 2015 年到 2019 年，TDSQL 已经推广到许多银行和金融政企。但是很重要的一点是，虽然服务了很多银行、金融客户，但是在银行领域有一块比较难动的地方叫银行的传统核心系统。传统核心系统数据库长期以来一直是被国外的商用数据库所垄断，比如说 ORACLE、DB2 等，像国内分布式数据库是很难介入的。

2018 年，我们关注到张家港银行有更换核心系统的需求，就此建立联系并成功达成合作，最终，2019 年，我们将腾讯这套分布式数据库系统成功应用到了张家港银行的传统核心系统。张家港行也是成为了全国第一家传统核心系统上分布式数据库的银行，而国产分布式数据库不再是只局限于银行的互联网核心、互联网银行等外围系统的尝试，而是真真正正切入到银行系统的心脏—传统核心，这也是国产数据库领域一个具有里程碑意义的事件。

所以在未来，我们也将继续"走出去"，深入到更复杂、更新核心的业务系统，打磨我们的产品。

这是 TDSQL 的发展历程。

2 TDSQL 核心特性：极具挑战的“四高”服务与安全可运维

1. 数据强一致。作为适应于金融场景的数据库，数据强一致性是立命之本，因为数据不能丢、不能错。在金融场景，没有办法去估量——假如错一条数据，到底这条数据是 1 分钱还是 1 个亿，所以数据强一致是 TDSQL 根本特性之一。

2. 金融级高可用。TDSQL 确保 99.999%以上的可用性，并支持跨 IDC、多机房、同城多活的部署方式。TDSQL 最先切入金融场景是因为金融场景的挑战是最大的。中国金融行业受监

管要求最为苛刻，同时也对数据和业务的可用性、可靠性、一致性更是有极高的要求——要求 99.999%的可用性，也就是说这个数据库全年故障的时间不能超过 5 分钟。

3. 高性能、低成本。互联网时代的企业，都是海量业务、海量机器的状态，性能稍微提高 10%，可能就节约成百上千台机器的成本，经济效益巨大。所以高性能、低成本也是 TDSQL 的一个关键特性。

4. 线性水平扩展。因为无论是互联网还是其他企业，随着数字化的发展，比如说出现突增流量，搞个活动等，现在单机的承载量越来越容易凸显出瓶颈。所以我们提出这种水平线性扩展的能力，要求它可进行水平伸缩。可能一台机器的负载、硬盘、机器资源容纳不了，但可以把它拆到多台机器，不需要考虑太多，它可以自动地提高自身吞吐量和负载量。

5. 企业级安全。金融数据是敏感的，一些敏感的金融数据需要在当前数据基础上再做一层更高级别的企业安全防护，比如数据库防火墙，以及透明加密，等等。

6. 便捷的运维。私有化部署中，很多情况下其实他们的网络环境和部署环境跟我们是隔离的，如果银行客户有问题，我们不能第一时间是切入去帮助解决，所以就需要一套完善的配套设施，简单容易上手，可以自动帮用户去定位问题、解决问题，同时也尽量减少运维的复杂度。

3 TDSQL 核心架构

3.1 TDSQL 系统总览

3.2 资源池

上图从下往上看，首先最底层是资源池，属于 IaaS 层服务，可以是物理机，也可以是虚拟机，只要是给 TDSQL 添加机器就好。TDSQL 是在一个机器的资源池上实现了数据库实例的管理。当然，这里推荐的还是物理机——如果增加一层虚拟机服务，无疑在稳定性和性能方面都会引入一些隐患。

3.3 存储节点

从资源池再往上是存储节点。存储节点要强调的是 TDSQL 的两种存储形态：一种是 Noshard 数据库，一种是分布式数据库（也叫 Shard 版 TDSQL）。简单来说，Noshard 就是一个单机版的 TDSQL，在 MySQL 的基础上做了一系列的改造和改良，让它支持 TDSQL 的一系列特性，包括高可用，数据强一致、7×24 小时自动故障切换等。第二种是分布式数据库，具备水平伸缩能力。所以 TDSQL 对外其实呈现了两种形态，呈现一种非分布式形态，一种是分布式的形态。至于这两种形态的区别，或者说什么场景更适合于哪种数据库，后面我们有专门的章节去分析。

3.4 计算节点

再看计算节点。计算节点就是 TDSQL 的计算引擎，做到了计算层和存储层相分离。计算层主要是做一些 SQL 方面的处理，比如词法解、语法解析、SQL 改写等。如果是分布式数据库形态，还要做分布式事务相关的协调，所以我们看到计算层不存储数据，只运行 SQL 方面的实时计算，所以它更偏 CPU 密集型。此外，TDSQL 计算节点还具备 OLAP 的能力，对一些复杂的计算可以进行算法上的优化——什么时候该下推到存储引擎层，什么时候需要在计算层做汇总等，这是计算节点需要做的事情。

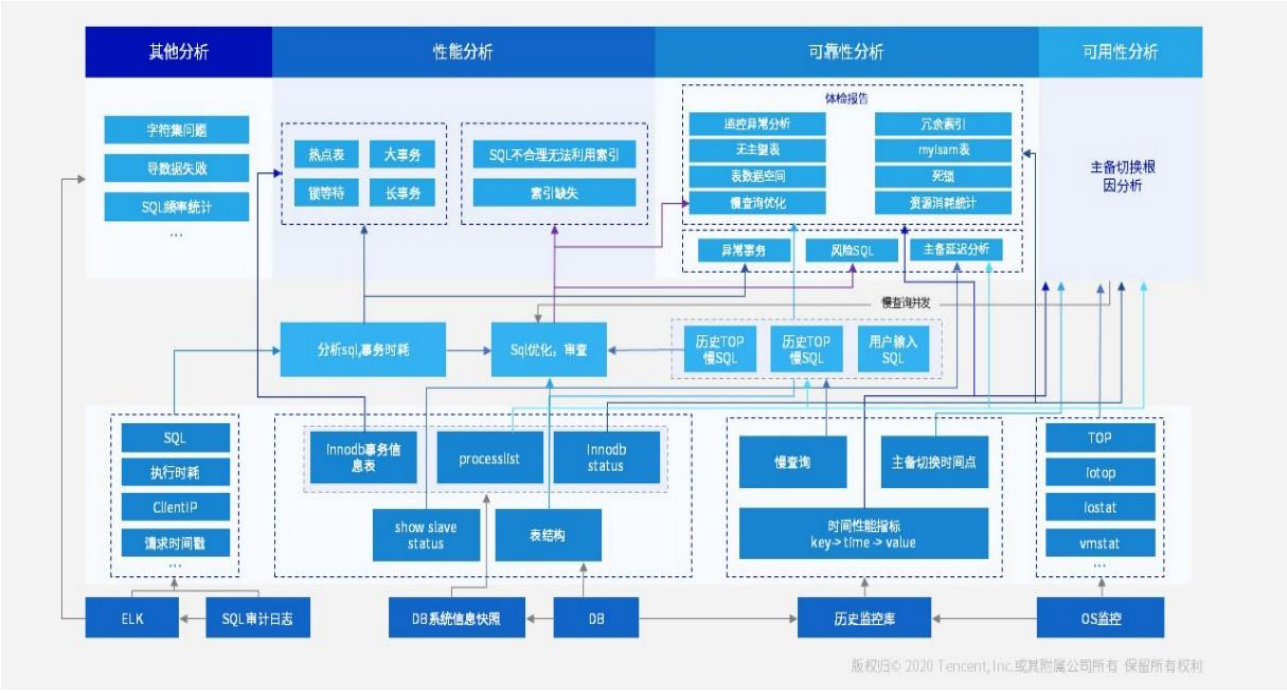
3.5 赤兔运营管理平台

再往上，是赤兔运营管理平台。如果说把下面这一套东西比作一个黑盒，我们希望有一个用户界面可以操纵这个黑盒，这个界面就是赤兔运营管理平台。通过这个平台，DBA 可以操纵 TDSQL 后台黑盒，所以相当于是一套 WEB 管理系统，让所有 DBA 的操作都可以在用户界面上完成，而不需要登陆到后台，不需要关心计算节点是哪个，存储节点是哪个，或者怎么样管

理它，要加一些节点或者减一些节点，或者把这个节点从哪里要迁到哪里……这些都可以通过界面化完成。DBA 操作界面不容易出错，但如果登陆到后台很容易一个误操作，不小心把机器重启了，就可能会造成一定的影响。

3.6 “扁鹊”智能 DBA 平台

有了赤兔之外，为什么还有一个“扁鹊”智能 DBA 平台呢？如果机器发生了故障，或者说哪天磁盘有坏块了，或者是 IO 性能越来越差……SSD 其实有一个衰老的过程，到了后期的话，吞吐量和 IOPS 可能会有一定下降，导致数据库的响应速度变慢。这种情况如果 DBA 要排查，得先去看到是哪一个实例、涉及到哪一台机器、这个机器有什么问题、检测机器的健康状态……这些都是机械性的工作，有了扁鹊智能管理平台，当出现故障的时候就可以自动分析故障的原因，举个例子，可以找出是因为什么导致 SQL 变慢了，或者又是因为什么原因发生了主备切换，突然 IO 异常了或者其他什么原因导致机器故障。



此外，扁鹊智能 DBA 平台还有一个智能诊断系统，可以定期由 DBA 发起对实例进行的诊断。比如有些数据库实例，CPU 常年跑得很高，其实是一些比较差的 SQL 导致的。这个时候扁鹊智能 DBA 系统，可以很方便地到用户实例上做巡检，得到一个健康状况图，并对它进行打分，发现这个实例比如他的 CPU 超用了，需要扩容，但是没有扩容，就会减分；然后其他表的索引没有建好，要减分……以此生成一个诊断报告。所以，有了扁鹊，再加上赤兔运营管理平台，DBA 的工作其实是非常轻松的，可能每天只需要点几下按纽，然后就解决了一系列的麻烦，包括高可用，性能分析，锁分析等，完全把 DBA 从繁杂的工作中解放出来。

此外，我们看到这里其实还有几个小的模块：

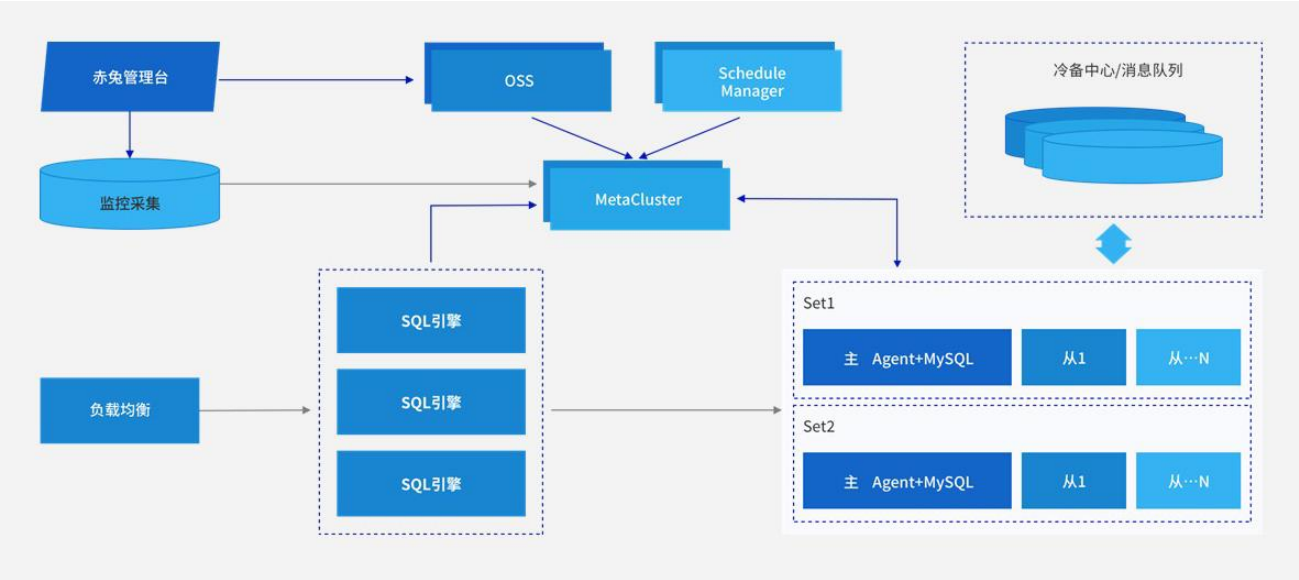
- 调度系统，调度系统主要是负责整体的资源调度，比如说数据库实例的增加删除、过期作废，还有一些容量的调度，即扩容、缩容，还有一些多租户的管理。也就是说这是整个管理台的调度器。

- 另外还有一个备份系统，这个是冷备中心，后面有一个专门的章节去讲，这里就不再赘述。此外，我们还提供了一些服务模块作为辅助，比如审计，还有数据库之间的迁移服务——我们 TDSQL 怎么能够帮助异地数据库迁进来，或者从 TDSQL 再迁出。

- 此外，还包括数据校验、数据订阅、SQL 防火墙、注入检测等安全方面的模块，以及一个辅助模块——帮助我们的 DBA 也好，用户也好，完成一些个性化的丰富的需求。

以上是 TDSQL 系统总览。

4 TDSQL 架构模块及其特性



我们再看一下核心架构：

首先用户的请求通过负载均衡发往 SQL 引擎。然后，SQL 引擎作为计算接入层，根据这个 SQL 的要求从后端的存储节点去取数据。当然，无论是 SQL 引擎还是后端的数据库实例都存在一个元数据来管理调度。举个例子，计算引擎需要拿到一个路由，路由告诉 SQL 引擎，这个 SQL 该发往哪一个后端的数据节点，到底是该发往主节点还是发往备节点。所以我们引入了 MetaCluster(MC)来储存类似于路由这类元数据信息。当然 MC 只是静态的存储元数据，维护和管理这些元数据信息，还需要有一套调度以及接口组件，这里是 OSS、Manager/Scheduler。所以我们这张图可以看到是 TDSQL 整体来说就分为三部分：管理节点、计算节点和存储节点。当然这里还有一个辅助模块，帮助完成一些个性化需求的，比如备份、消息队列，数据迁移工

具等。另外，这里的负载均衡其实不是必需的，用户可以选用自身的硬件负载，也可以用 LVS 软负载，这个负载均衡根据实际的用户场景可自定义。

了解了整体架构以后，我们继续再看一下每个节点的特性是什么、对机器的依赖程度如何，要求机器有哪些特性，等等。

4.1 管理模块：轻松通过 web 界面管理整个数据库后台

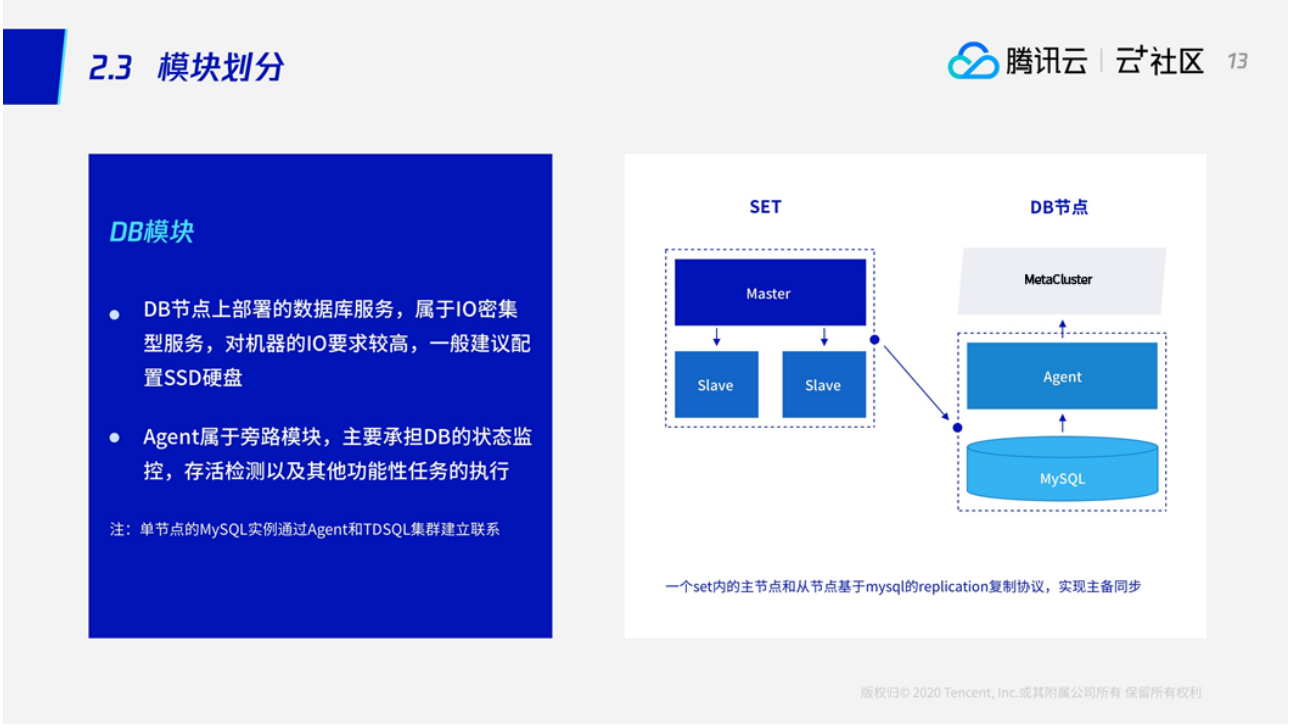
首先，我们要看的是管理模块。作为一个集群只搭建一套的管理模块，一般可以复用一组机器。同时，管理模块对机器的要求相对来说比较低，比如资源紧张的时候，我们用虚拟机就可以代替。在我们内部，一套管理模块承载最大的管理单集群近上万实例。

管理模块包含前文说的几个关键模块：MetaCluster(MC)、Scheduler、Manager、OSS 和监控采集程序、赤兔管理控制台。那么它们是怎么联合工作的呢？首先，DBA 用户在赤兔管理台——这一套 WEB 前台发起一个操作——点了一个按纽，这个按纽可能是对实例进行扩容，这个按纽会把这个 https 的请求转移到 OSS 模块，这个 OSS 模块有点像 web 服务器，它能接收 web 请求，但是它可以把这个转发到 MC。所以，OSS 模块就是一个前端到后台的桥梁，有了 OSS 模块，整个后台的工作模块都可以跟前台、跟 web 界面绑定在一起。

捕捉到这个请求之后，在 MC 上创建一个任务节点，这个任务节点被调度模块捕获，捕获之后就处理任务。处理完任务，再把它的处理结果返回到 MC 上。MC 上的任务被 OSS 捕获，最后也是 https 的请求，去查询这个任务，最后得到一个结果，返回给前端。

这是一个纯异步的过程，但是有了这套管理模块，让我们可以轻松的通过 web 界面去管理整个 TDSQL 的后台。当然，这整个过程都有一个监控采集模块去采集，对整个流程的审计及状态进行获取。

4.2 DB 模块：数据库无损升级

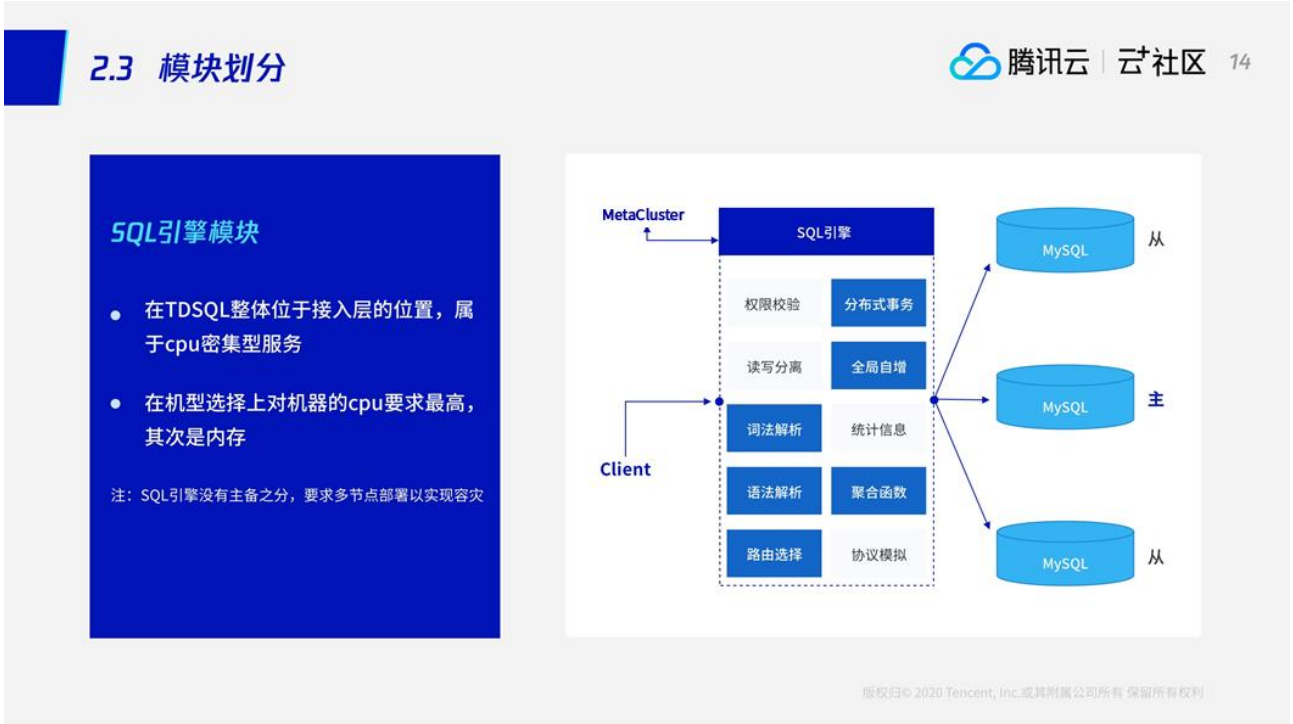


DB 模块，即数据节点，数据存取服务属于 IO 密集型的服务，因此，数据节点也是我们的存储节点，它对 IO 的要求比较高，一般建议配置 SSD 硬盘，最好是 PCI-E 的 SSD。因为对数据库服务来说，CPU 再高，如果 IO 跟不上，仍然是小马拉大车。比如只有 1 千的 IOPS，CPU 根本就跑不起来，用不起来。所以这里一般建议至少 IOPS 要达到 1 万以上。

SET 就是数据库实例，一个 SET 包含数据库的——比如我们默认要求的是一主两备，一个 Master 节点和两个 Slave 节点。当然在 DB 节点上有一个 Agent 的模块。MySQL 在执行中，我们要监控它的行为，以及进行操作。如果把这些东西做到 MySQL 里面为什么不可以呢？这其实存在一个问题，如果对数据节点进行升级，可能就要涉及到重启，一旦重启就影响用户的业务，影响服务。这个时候我们就考虑，在它上面加一个模块 Agent，它来完成对所有集群对 MySQL 的操作，并且上报 MySQL 的状态。有了它之后，对 TDSQL 数据节点的大部分升级，都会转变为对 Agent 的升级，而升级 Agent，对业务没有任何影响，这就实现了无损升级。相

比于 Agent 我们对数据节点 MySQL 不会频繁升级，一般情况下一年、半年都不会动它。这是我们 DB 模块，也是存储节点。

4.3 SQL 引擎模块：分布式复杂 SQL 处理



SQL 引擎处于计算层的位置，本身属于 CPU 密集型，所以我们在选机型上尽量要求 CPU 高一些。其次是内存，作为计算接入层，它要管理链接，如果是大量的短链接或者长链接，非常占内存，所以它对 CPU 和内存的要求比较高。此外，它本身不存储数据，也没有主备之分，所以对硬盘没有太大要求。

QL 引擎首先还是从 MC 上拉取到元数据，作为 SQL 引擎，包括权限校验、读写分离，以及统计信息、协议模拟等相关的操作。SQL 引擎主要负责做词法、语法分析，以及作为查询引擎等工作，而且在分布式的场景下，SQL 引擎复杂的功能性就会凸显，比如要处理分布式事物，还要维护全局自增字段，保证多个数据、多个存储节点共享一个保证全局自增的序列；如果是分布式的话，要限制一些语法，包括词法和语法的解析；还有在一些复杂计算上，它还要做一些 SQL 下推，以及最后数据的聚合。所以 SQL 引擎是作为计算层中一个相对来说比较复杂的模块。

5 TDSQL 金融级特性之：数据强一致性保障

接下来将从 4 个方面介绍 TDSQL 最重要的特性之一——作为金融场景下不可或缺的数据强一致性的保障：

- 主备数据复制方式
- 数据复制比较：TDSQL 主备数据复制方案 VS MySQL 原生方案
- 核心功能：容灾切换，数据强一致、0 丢失 0 出错
- 数据强一致性

5.1 TDSQL 主备数据复制：高性能强同步

首先在讲数据一致性之前，我们先了解一下 MySQL 原生的数据复制的方式。

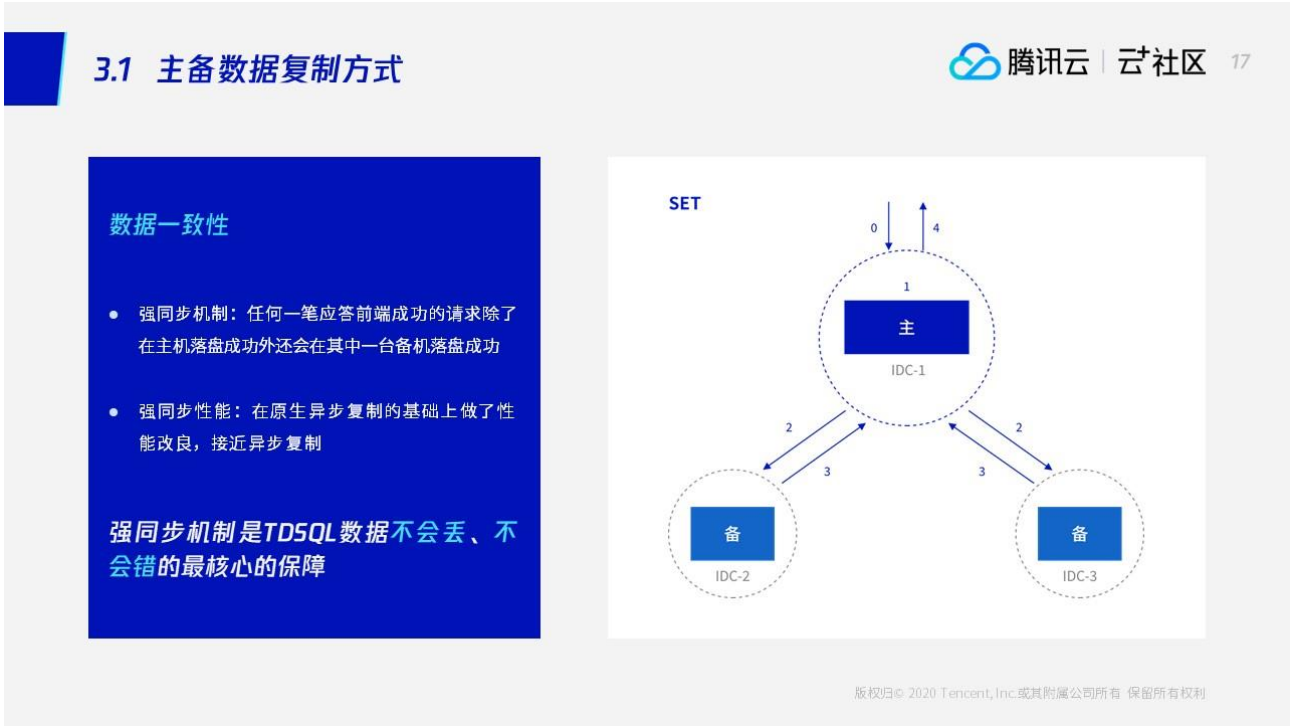
首先第一种是异步复制：主机在不等从机应答直接返回客户端成功。这个在金融场景是不能接受的，这样的话相当于数据是没有多副本保障。

第二种是半同步：主机在一定条件下等备机应答，如果等不到备机应答，它还是会返回业务成功，也就是说它最终还会退化成一个异步的方式，这同样也是金融场景所不能接受的。

除此之外，原生半同步其实是有一个性能方面的缺陷，即在跨 IDC 网络抖动的场景下，请求毛刺现象很严重。所以原生的异步复制和半同步复制都存在一些问题，并不能完全适应金融场景。

TDSQL 引入了基于 raft 协议的强同步复制，主机接收到业务请求后，等待其中一个备机应答成功后才返回客户端成功。比如这张图，我们一主两备下一条业务请求到达了主机之后必须等其中一个备机应答成功，才能返回客户端成功，否则这个请求是不会应答的。所以说，强同步是 TDSQL 最基础的一个特性，是 TDSQL 保证数据不会丢、不会错的关键。

讲到这里可能会有人有疑问：这个强同步其实也不复杂，不就是在半同步的基础上把这个超时时间改成无限大同时应答的备机设置为 1 ？并不是这样的，TDSQL 强同步这里的关键不是在解决备机应答的问题，而是要解决这种增加了等待备机的机制之后，如何能保证高性能、高可靠性。换句话说，如果在原生半同步的基础上不改造性能，仅把超时时间改成无限大的时候，其实跑出来的性能和异步比甚至连异步的一半都达不到。这个在我们看来也是无法接受的。相当于为了数据的一致性牺牲了很大一部分性能。



TDSQL 强同步复制的性能是在原生半同步的基础上做了大量的优化和改进，使得性能基本接近于异步。

所以这里强同步强调的是，实现强同步的同时还具备高性能特性，所以确切地说是一个高性能的强同步。

那么我们如何实现高性能的强同步？我们继续往下看。这里 TDSQL 对 MySQL 主备复制的机制做了改造。首先，我们先引入了一个线程池的模型。

原生的 MySQL 是——一个互联请求是一个线程，这样对操作系统的资源消耗还是很大的：5 千个连接，5 千个线程。那 1 万个连接，1 万个线程，操作系统能扛得住吗？肯定扛不住。而且原生的数据复制的方式，其实串行化比较高，比如说一个用户请求发过来后，等备机应答；这个过程中用户线程在这里是完全不能做事的，只有等备机应答之后，它才能够返回前端。也就是说大量的线程都处于一个等待的状态，这是半同步效率低的根本原因。

引入了线程池模型之后，我们还需要考虑怎么调度线程池。我们希望 MySQL 维护一小部分工作的线程，比如说有 1 万个用户连接，真正干活的可能就那么 100 个、50 个 MySQL 的工作线程。如果是非工作的线程，他在等 IO 应答时可以先去睡眠，并不让它去影响我们的吞吐量。所以 TDSQL 对原生的复制做了改造，除了引入线程池模型，还增加了几组自定义的线程：当一笔用户请求过来之后完成了写操作以及刷新了 binlog，第二步他应该要等备机应答。这个时候被我们新引入的线程组所接管，把这个用户对话保留下来，释放了工作线程，工作线程该干

什么继续干什么，处理其他的用户连接请求。真正备机给了应答之后，再由另外一组线程将它唤醒，回吐到客户端，告诉客户端这个应答成功了。

所以经过这样一个异步化的过程，相当于把之前串行的流程异步化了，这样就达到了接近于异步复制的性能，这就是 TDSQL 在强同步的改造的核心。所以这不仅是一个实现强同步的过程，更是一个接近异步性能的强同步。

再看一下改造之后的性能对比：异步 TPS 大概是 6 万左右，平均时耗小于等于 10 毫秒。再看半同步，明显有三分之二的性能损耗，并且这个时耗波动还是比较大的，比如说 IDC 网络抖动一下。强同步一主两备模式下，首先性能已经接近于异步的性能，此外时耗并没有额外增加。

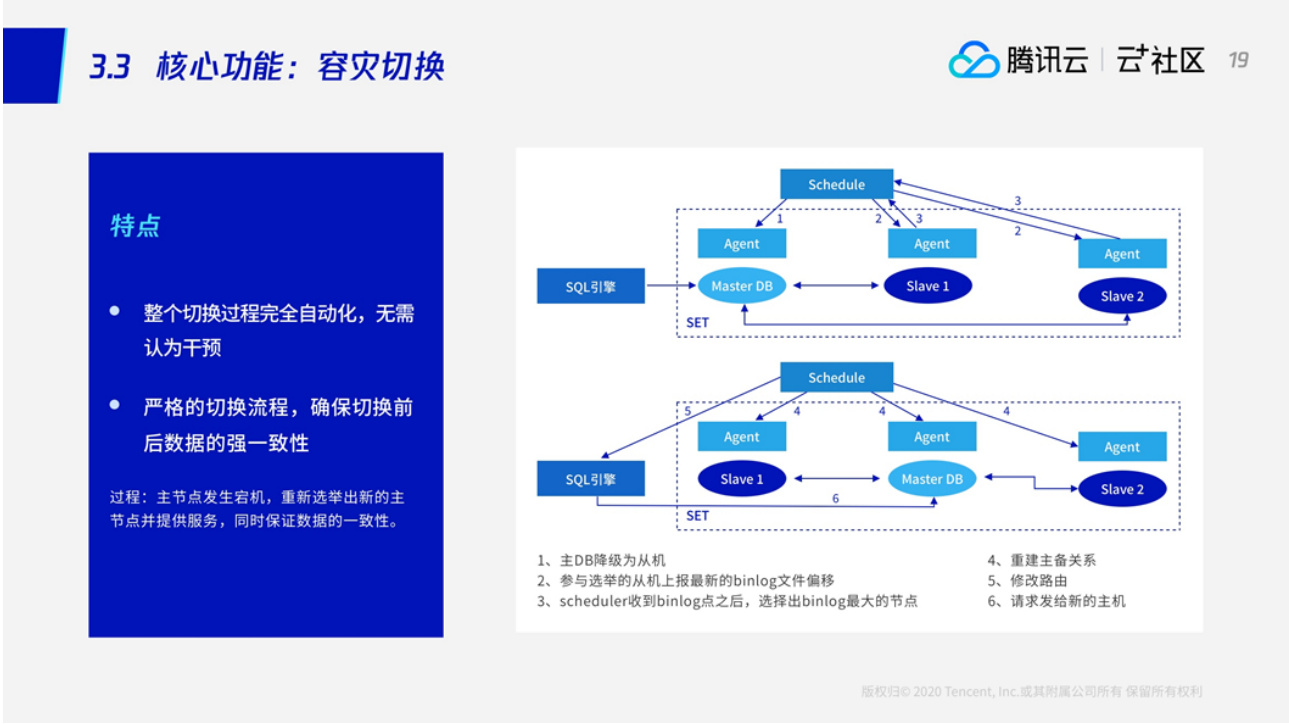
TPS 跟场景相关，数据仅提供横向对比参考

因为一主两备，除非两个机房网络同时抖动，否则的话强同步的时耗不会有明显波动。所以我们看到基于 TDSQL 的强同步实现了数据的多副本又保证了性能。

有了这个多副本保障，又怎么如何实现故障自动切换，数据不丢不错呢？这其实还是需要一套切换选举的流程。这就引出了 TDSQL 的自动容灾切换功能。

5.2 自动容灾切换：数据强一致、0 丢失 0 出错

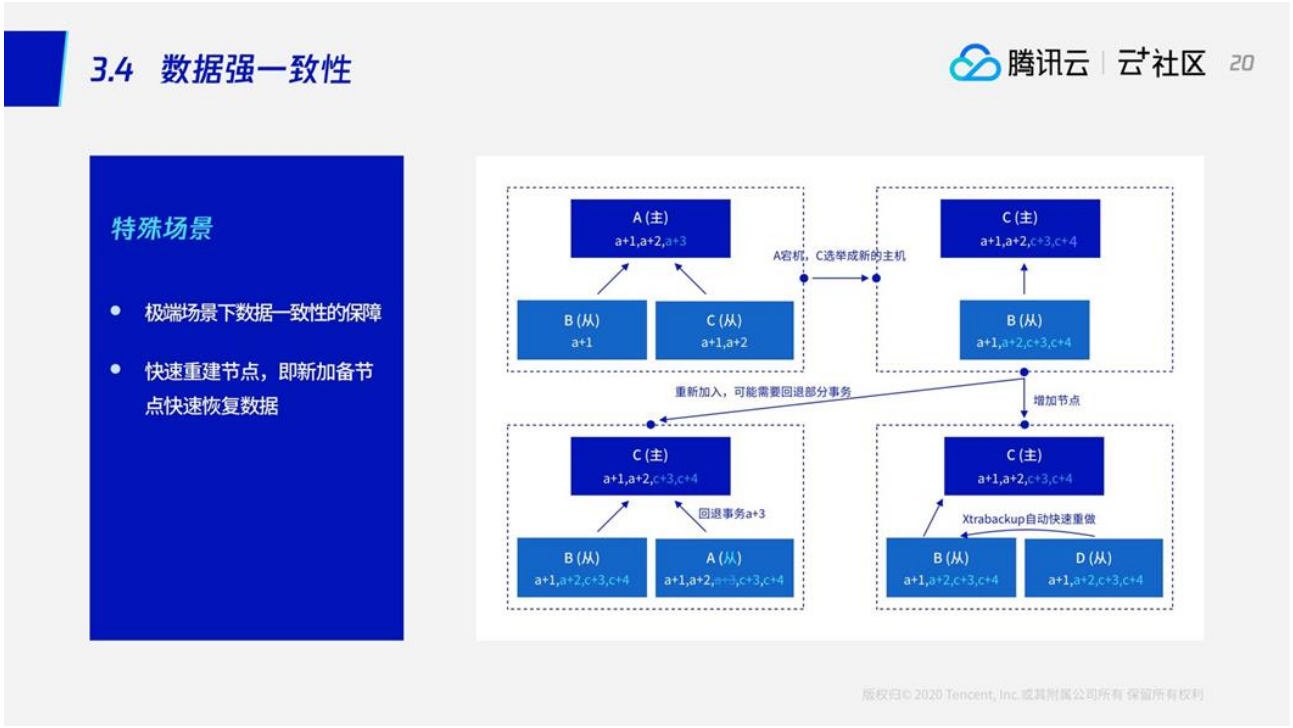
基于强同步特性的基础，TDSQL 自动容灾切换更加容易实现。架构图如下：



SQL 引擎将请求发给主节点，主节点被两个备机所同步，每个节点上都有对应的 Agent 上报当前节点的状态。这时，主节点发生了故障被 Agent 觉察，上报到 MC 被 Scheduler 捕获，然后 Scheduler 首先会把这个主节点进行降级，把它变成 Slave。也就是说此时其实整个集群里面全是 Slave，没有主节点。这个时候另外两个存活的备机上报自己最新的 binlog 点，因为有了强同步的保障，另外两个备机其中之一一定有最新的 binlog，那么两个备机分别上报自己最新的点后，Schedule 就可以清楚的知道哪个节点的数据是最新的，并将这个最新的节点提升成主节点。比如说发现 Slave1 的数据是最新的，这个时候 Schedule 修改路由，把主节点设置为 Slave1，完成了主备的切换。有了前述这个切换机制，保证了整个切换无需人为干预，并且切换前与切换后的数据是完全一致的。因此，正是由于强同步的保证，所以当主机发生故障的时候，我们一定是可以从一个备节点上找到最新数据，把它提成主节点。这就是 TDSQL 容灾切换的整个过程，容灾切换需要建立在强同步的基础上。

5.3 极端场景下的数据一致性保障

聊完了容灾切换，我们再聊一聊故障节点的后续处理事宜。故障处理可能有几种情况：一种是故障以后，它还能活过来。比如说像机房可能突然掉电了，掉完电之后它马上又恢复。或者机器因为硬件原因不小心发生了重启，重启完之后节点是可以被拉起，被拉起之后，我们希望它能够迅速的再加入到集群中，这时数据其实是没有丢没有错的，我们不希望把节点故障之后又重新建一份数据，把之前的数据全抹掉。而是从它最后一次同步的数据为断点，继续续传后面的数据。带着上述问题，我们看一下节点的故障后恢复过程。

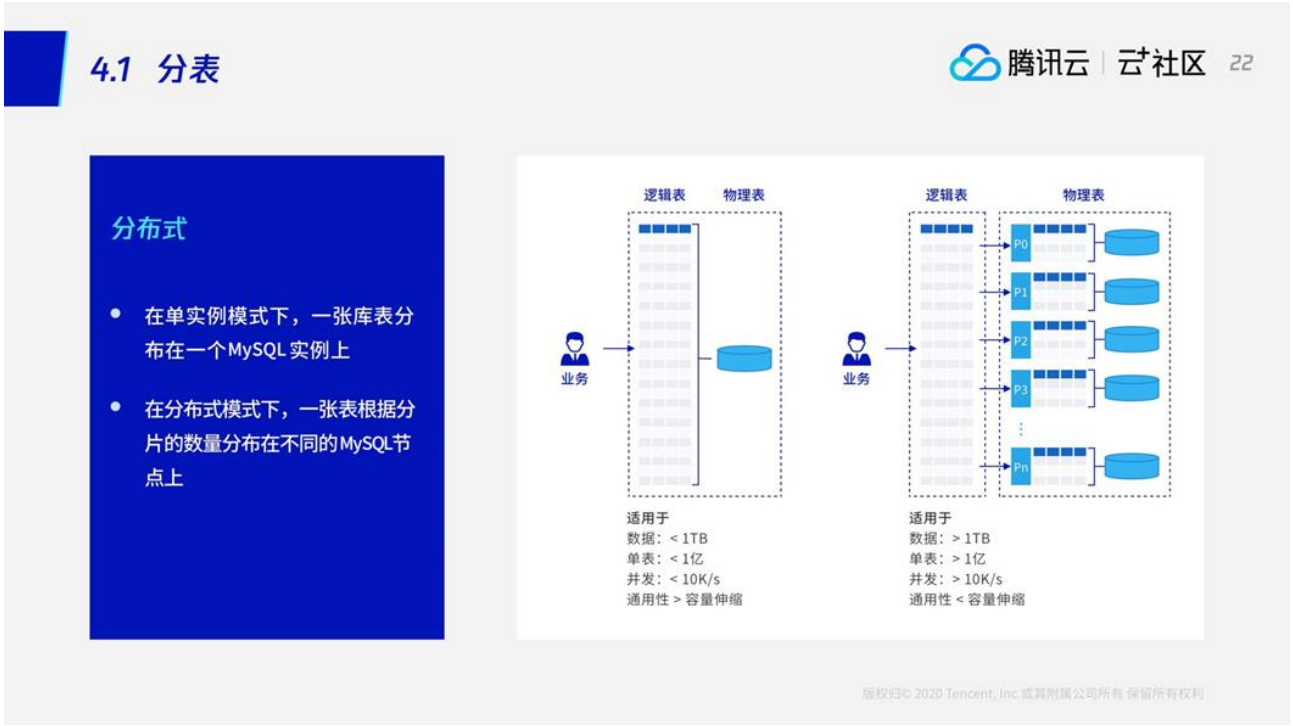


首先我们考虑一种场景，比如说 A 节点作为主节点，B、C 是从，正常去同步 A 节点的数据，A+1、A+2，接下来该同步 A+3。当 A+3 还没有同步到从节点的时候发生了故障，这个时候根据 B、C 节点的数据情况，C 的数据是最新的，因此 C 被选成了主节点，进而 C 继续同步数据到 B。过了一阵，A 节点拉起了，可以重新加入集群。重新加入集群之后，发现它有一笔请求 A+3 还没有来得及被 B、C 节点应答，但已经写入到日志。这个时候其实 A 节点的数据是有问题的，我们需要把这个没有被备机确认的 A+3 的回滚掉，防止它将来再同步给其他的节点。所以这里强调的是一个数据的回退，相当于每一个 Slave 新加入节点的时候，我们都会对它的数据进行检验，将多写的数据回滚。当然刚刚的假设是运气比较好，A 节点还能重启。有些时候 A 节点可能就挂了，这个机器就再也起不来了，我们需要把这个节点换掉，即换一个新机器，比如：加一个 D 节点。我们希望它快速重建数据并且对当前线上业务尽可能无影响。快速重建方面是基于物理拷贝，而且它是从备机上去拉数据，这样它既不会对主节点产生影响，又能够快速把数据重建好。

6 分布式 TDSQL 的实践

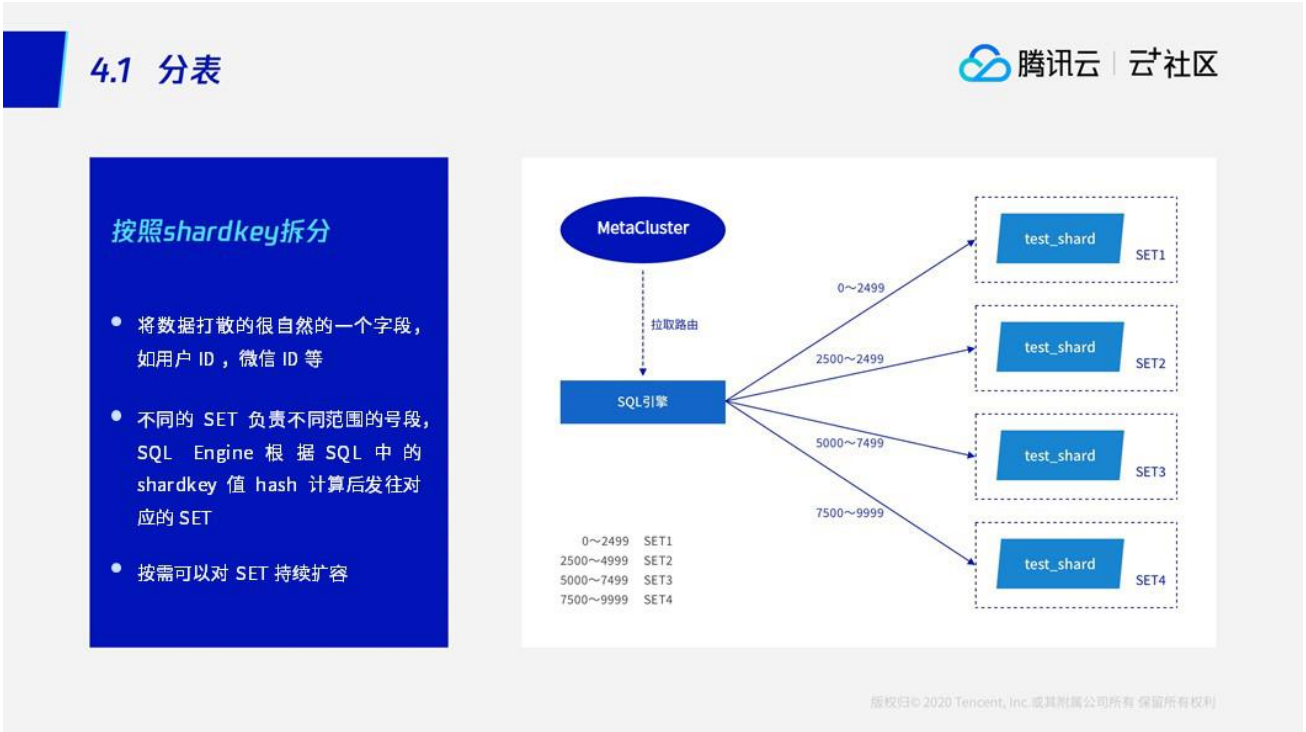
前面是关于 TDSQL 的高可用、强一致的介绍，这些作为金融场景是一个必备的特性，还没有涉及到分布式，当涉及到分布式时就开启了 TDSQL 的另外一种形态。接下来我们将介绍分布式 TDSQL 的实践，了解分布式 TDSQL 跟单节点的 TDSQL 有什么不同，以及这种分布式架构下又是如何实现一系列的保障，同时如何做到对业务透明、对业务无感知。

6.1 分表

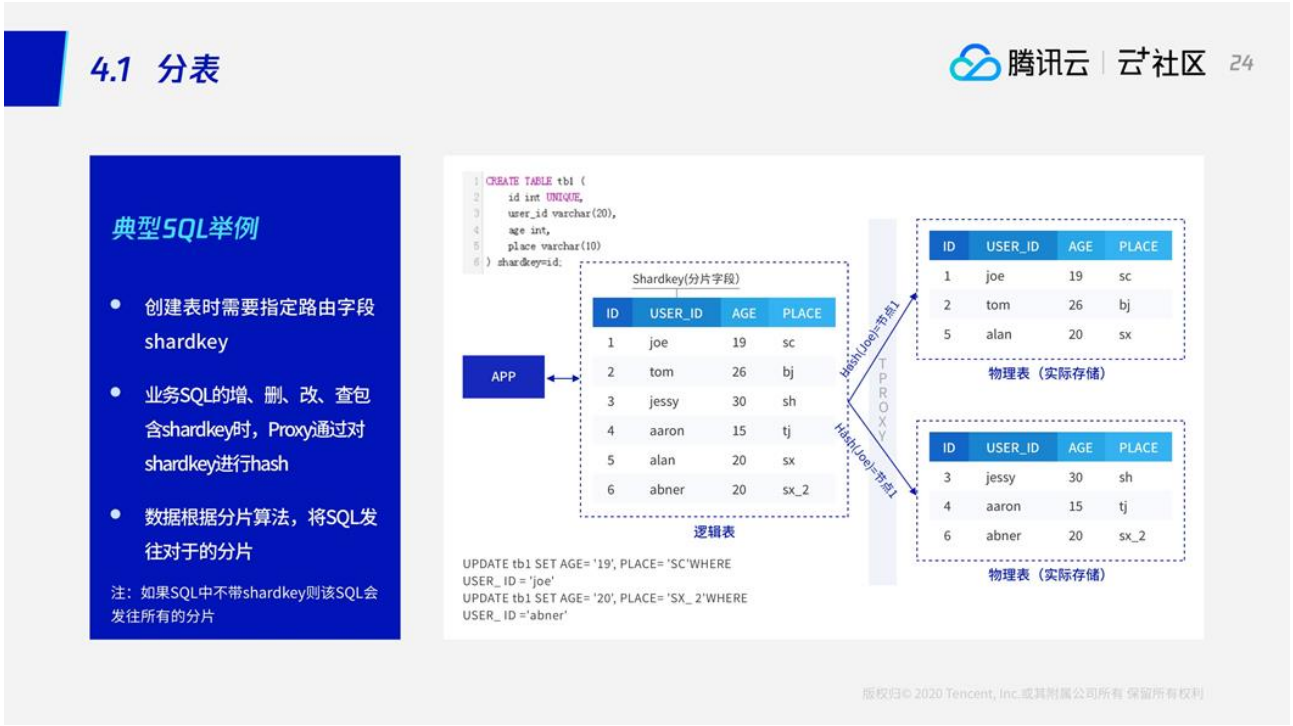


分表，当在单机模式下，用户看到的一张逻辑表，其实也是一张物理表，存储在一个物理节点（物理机）上。在分布式形态下，用户看到的逻辑表的实际物理存储可能是被打散分布到不同的物理节点上。所以 TDSQL 分表的目标，希望做到对业务完全透明，比如业务只看到一个完整的逻辑表，他并不知道这些表其实已经被 TDSQL 均匀拆分到各个物理节点上。比如：之前可能数据都在一台机器上，现在这些数据平均分布在了 5 台机器上，但用户却丝毫没有觉察，这是 TDSQL 要实现的一个目标——在用户看来是完全的一张逻辑表，实际上它是在后台打散了的。

这个表在后台如何去打散，如何去分布呢？我们希望对用户做到透明，做到屏蔽，让他不关心数据分布的细节。怎么将这个数据分布和打散呢？这就引出了一个概念：shardkey——是 TDSQL 的分片关键字，也就是说 TDSQL 会根据 shardkey 字段将这个数据去分散。

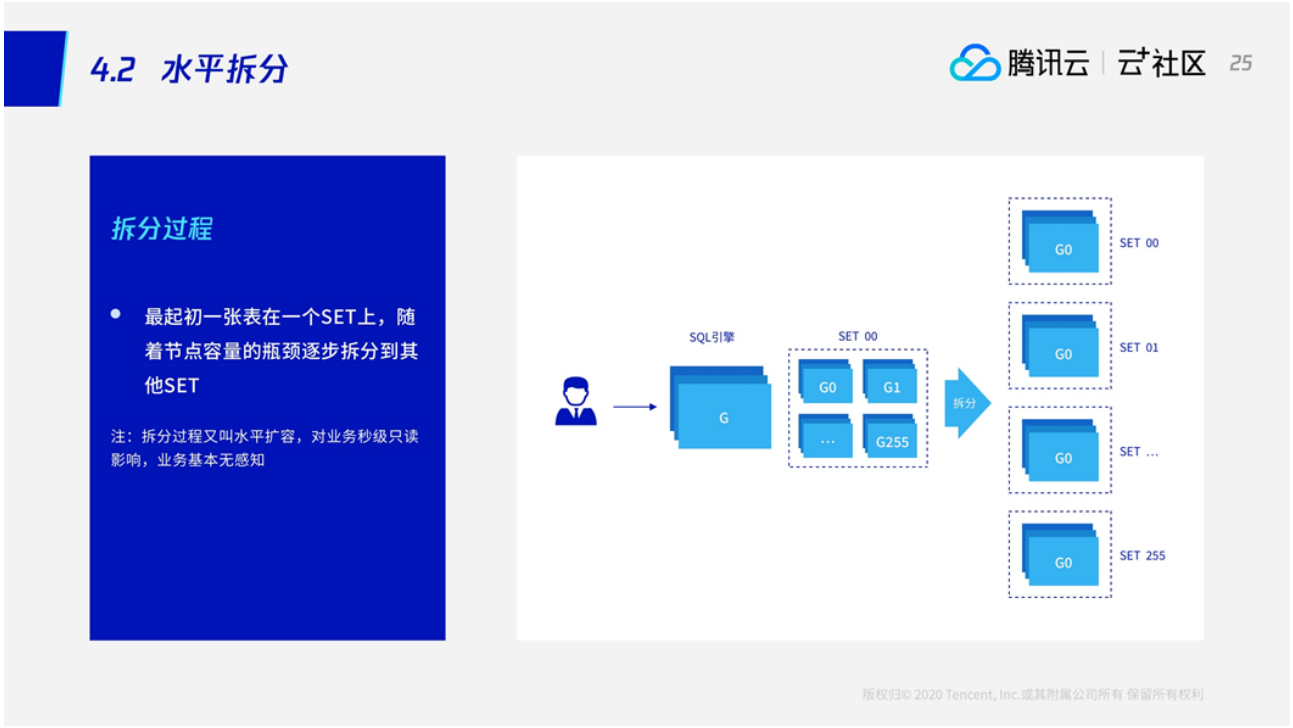


我们认为，shardkey 是一个很自然的字段，自然地通过一个字段去将数据打散。举个例子，腾讯内部我们喜欢用 QQ 号作为一个 shardkey，通过 QQ 号自动把数据打散，或者微信号；而一些银行类的客户，更喜欢用一些客户号、身份证号以及银行卡号，作为 shardkey。也就是说通过一个字段自然而然把这个数据分散开来。我们认为引入 shardkey 后并不会增加额外的工作，因为首先用户是最了解自己得数据的，知道自己的数据按照什么字段均匀分布最佳，同时给用户自主选择分片关键字的权利，有助于从全局角度实现分布式数据库的全局性能最佳。所以这里可能有些人会想，是不是主键是最好的或者尽可能地分散？没错，确实是这样的，作为 TDSQL 的分片关键字越分散越好，要求是主键或者是唯一索引的一部分。确定了这个分片关键字后，TDSQL 就可以根据这个分片关键字将数据均匀分散开来。比如这张图，我们按照一个字段做了分片之后，将 1 万条数据均匀分布在了四个节点上。



具体的使用上，举个例子，我们创建了 TB1 这个表，这里有若干个字段，比如说 ID，从这个名字上来看就应该知道它是一个不唯一的，或者可以说是一个比较分散的值。我们看到这里，以“ID”作为分配关键字，这样六条数据就均匀分散到了两个分片上。当然，数据均匀分散之后，我们要求 SQL 在发往这边的都需要带上 shardkey，也就是说发到这里之后可以根据对应的 shardkey 发往对应的分片。如果不带这个 shardkey 的话，它不知道发给哪个分片，就发给了所有分片。所以强调通过这样的改善，我们要求尽可能 SQL 要带上 shardkey。带上 shardkey 的话，就实现了 SQL 的路由分发到对应的分片。

6.2 水平拆分



对于分布式来说，可能最初我们所有的数据都在一个节点上。当一个节点出现了性能瓶颈，需要将数据拆分，这时对我们 TDSQL 来说非常简单，在界面上的一个按钮：即一键扩容，它就

可以将这个数据自动拆分。拆分的过程也比较容易理解，其实就是一个数据的拷贝和搬迁过程，因为数据本身是可以按照一半一半这样的划分的。比如最先是这么一份数据，我们需要拆成两份，需要把它的下半部分数据拷到另外一个节点上。这个原理也比较简单，就是一个数据的拷贝，这里强调的是在拷贝的过程中，其实业务是不受任何影响的。最终业务只会最终有一个秒级冻结。

为什么叫秒级冻结？因为，最后一步，数据分布到两个节点上涉及到一个路由信息变更，比如原来的路由信息要发到这个分片，现在改了之后需要按照划分，上半部分要发给一个分片，下半部分发给另一个分片。我们在改路由的这个过程中，希望这个数据是没有写入相对静止的。

当然改路由也是毫秒级别完成，所以数据拆分时，真正最后对业务的影响只有不到 1s，并且只有在最后改路由的冻结阶段才会触发。

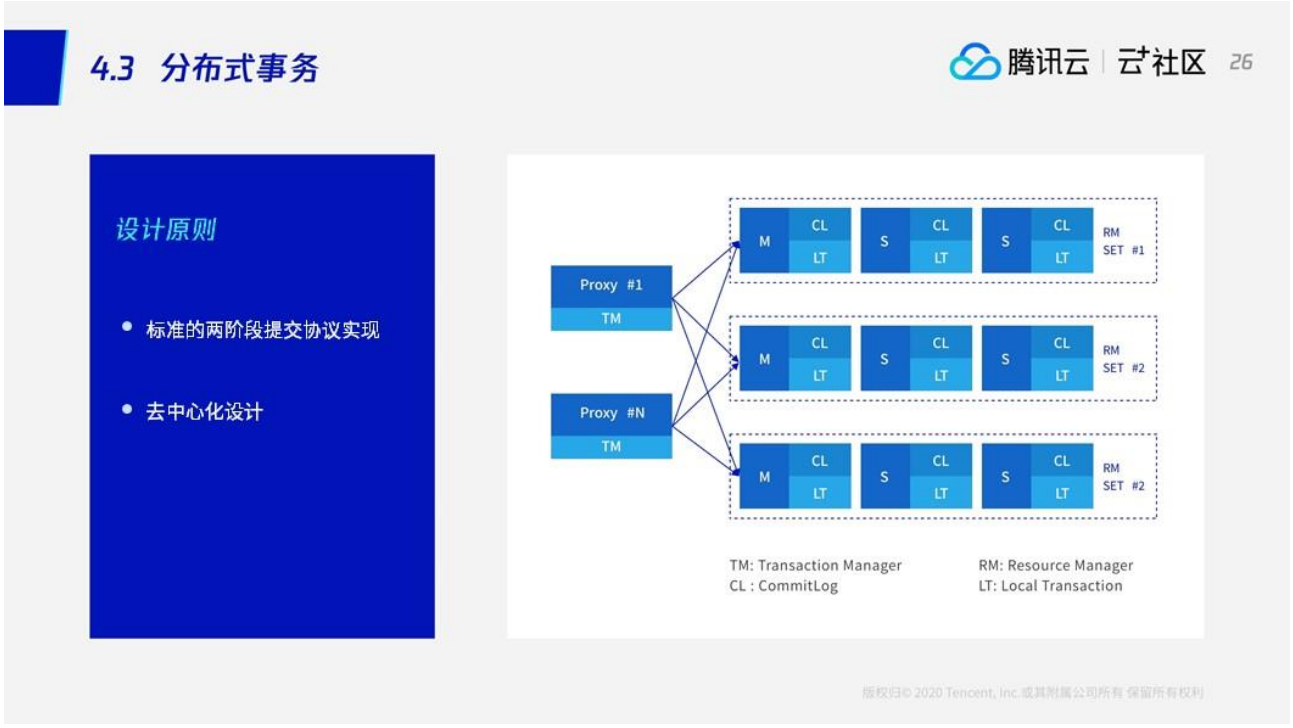
讲完数据拆分，我们开始切入分布式里面最难解决的这个问题，分布式事务。

6.3 健壮的分布式事务

单节点的事务是很好解决的，但是在分布式场景下想解决分布式事务还是存在一定的困难性，它需要考虑各种各样复杂的场景。

其实分布式事务实现不难，但首要是保证它的健壮性和可靠性，能应对各种各样的复杂场景。比如说涉及到分布式事务的时候，有主备切换、节点宕机……在各种容灾的测试环境下，如何保证数据总帐是平的，不会多一分钱也不会少一分钱，这是分布式事务需要考虑的。

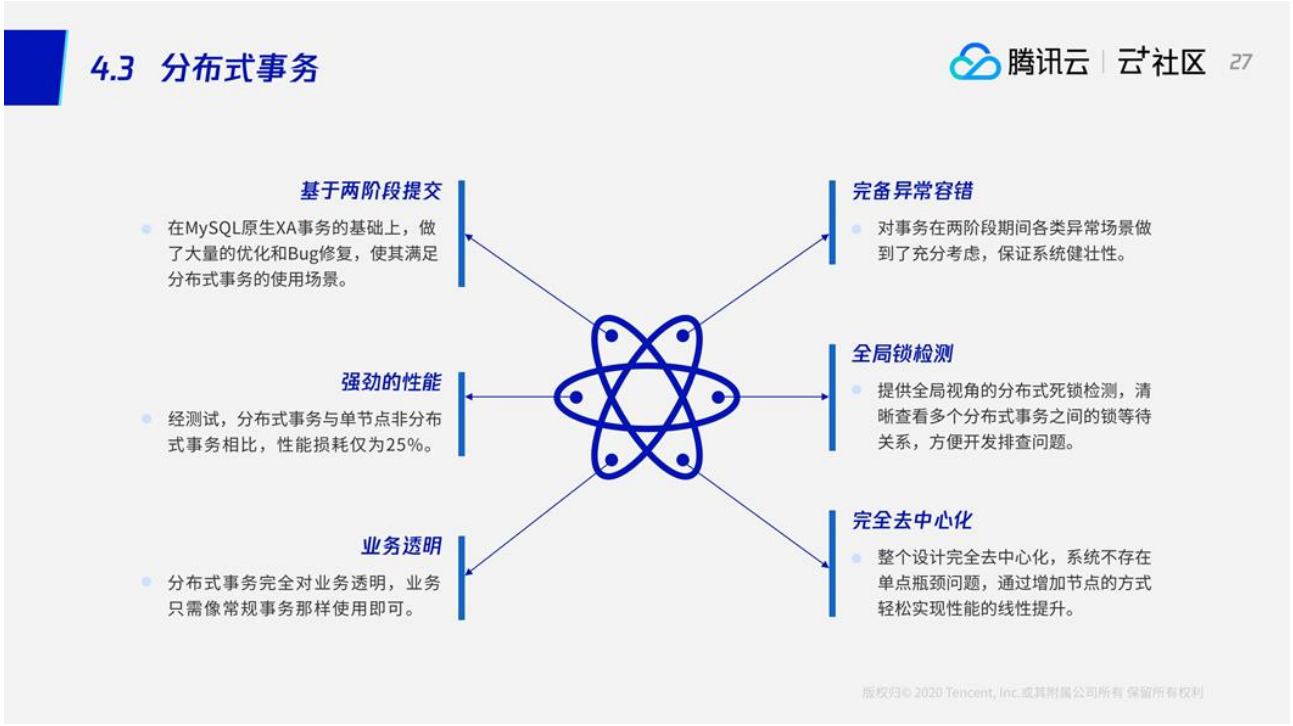
TDSQL 分布式事务基于拆的标准两阶段提交实现，这也是业内比较通用的方法。我们看到 SQL 引擎作为分布式事务的发起者，联合各个资源节点共同完成分布式事务的处理。



分布式事务也是根据 shardkey 来判断，具体来说，对于 SQL 引擎读发起一个事务，比如第一条 SQL 是改用户 ID 为 A 的用户信息表。第二条 SQL 是插入一个用户 ID 为 A 的流水表，这两张表都以用户 ID 作为 shardkey。我们发现这两条 SQL 都是发往一个分片，虽然是一个开启的事务，但是发现它并没有走分布式事务，它实际还是限制在单个分片里面走了一个单节点的事

务。当然如果涉及到转帐：比如从 A 帐户转到 B 帐户，正好 A 帐户在第一个分片，B 帐户是第二个分片，这样就涉及到一个分布式事务，需要 SQL 引擎完成整个分布式事务处理。

分布式事务是一个去中心化的设计，无论是 SQL 引擎还是后端的数据节点，其实都是具备高可用的同时支持线性扩展的设计。分布式事务比较复杂，单独讲的话可能能讲一门课，这里面涉及的内容非常多，比如两级段提交过程中有哪些异常场景，失败怎么处理，超时怎么处理，怎么样保证事务最终的一致性等等。这里不再赘述，我们总结一下：



- 基于两阶段提交。我们在 MySQL 原生 XA 事务的基础上做了大量的优化和 BUG 修复。比如说原生的 XA 在主备切换时会发生数据不一致和丢失，TDSQL 在这个基础上做了大量的修复，让 XA 事务能够保证数据一致性。

- 强劲的性能。起初我们引进原生分布式事务的时候，分布式事务的性能还达不到单节点的一半。当然经过一系列的优化调优，最后我们的性能损耗是 25%，也就是说它能达到单节点 75%的性能。
- 对业务透明。因为对业务来说其实根本无需关心到底是分布式还是非分布式，仅需要按照正常业务开启一个事务使用即可。
- 完备的异常容错。分布式事务是否健壮也需要考虑容错性的能力。
- 全局的锁检测。对于分布式环境下锁检测也是不可或缺的。TDSQL 提供全局视角的分布式死锁检测，可清晰查看多个分布式事务之间的锁等待关系。
- 完全去中心化。无论是 SQL 引擎还是数据节点，都是支持高可用并且能够线性扩展。

以上是 TDSQL 分布式事务的总结。如果说用户要求保持跟 MySQL 的高度兼容性，那可能 Noshard 版 TDSQL 更适合。但是如果对于用户来说，单节点已经达到资源的瓶颈，没有办法在单节点下做数据重分布或者扩容，那必须选择 Shard 模式。但是在 Shard 模式下，我们对 SQL 有一定的约束和限制，后面会有专门的一门课去讲分布式 TDSQL 对 SQL 是如何约束的。

我们看到无论是 Noshard 还是 Shard，都具备高可用、数据强一致、自动容灾的能力。

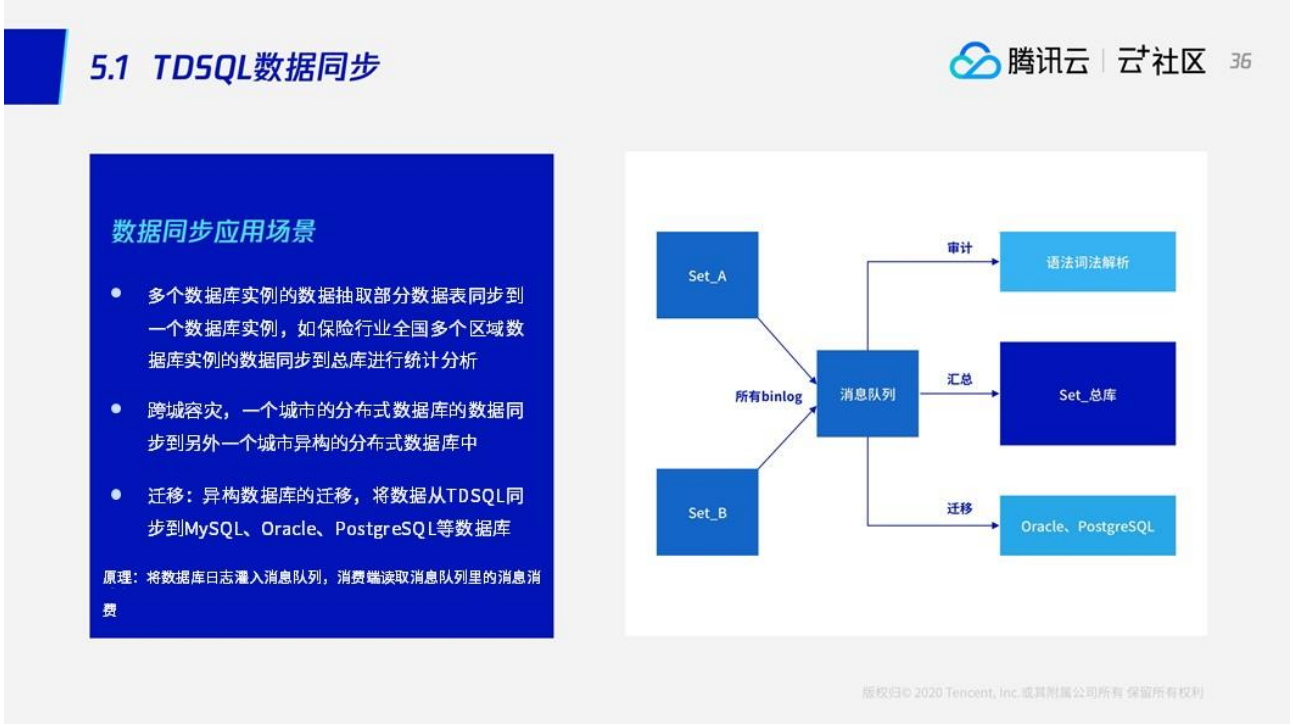
同时 TDSQL 也支持 Noshard 到 Shard 的迁移，可能早期我们规划的 Noshard 还可以承载业务压力，但是随着业务的突增已经撑不住了，这个时候需要全部迁到 Shard，那么 TDSQL 也有完善的工具帮助用户快速进行数据迁移。

7 TDSQL 数据同步和备份

接下来是 TDSQL 另外一个辅助特性：数据同步和备份。

7.1 TDSQL 数据同步组件

数据同步的重点分为三个场景：



第一个场景是数据汇总。比如，多个数据库实例的数据同步到一个数据库实例，如保险行业用户多喜欢将全国多个区域数据库实例的数据同步到全国总库进行统计分析。

第二是跨城的容灾。跨城容灾，一般一个城市的分布式数据库的数据需要同步到另外一个城市异构的分布式数据库中做灾备。有的时候我们要做异构数据库的跨城容灾，比如说主城是一个十六节点的数据库，它非常庞大。但是备城，可能我们基于成本考虑，选用的设备数量、机房都要差一些。比如灾备实例只有两个物理分片。一个是两分片数据库实例，而另外一个时十六分片的数据库实例。从十六分片同步到；两分片，这是一个异构的数据库的同步，这时候我们就需要利用数据同步这个组件。

第三是迁移。异构数据库的迁移，将数据从 TDSQL 同步到 MySQL、Oracle、PostgreSQL 等数据库。当然，从 TDSQL 到 TDSQL 是一种同步方式，更有一种是 TDSQL 到其他异构数据

库。举个例子，张家港农商行核心系统需要从传统的国外商用数据库替换为 TDSQL，可能还是需要做一定的风险的防范。最终我们提供了一套用 Oracle 作为 TDSQL 灾备示例的方案，通过数据同步组件，将 TDSQL 的数据准实时同步到 Oracle。假如在极端情况下需要将业务切到 Oracle，我们也是有这个能力的。

当然数据迁移也体现了 TDSQL 开放的思想，既然允许用户将数据迁移到 TDSQL，如果有一天用户可能觉得 TDSQL 不是很好，觉得有更好的产品可以替代它，TDSQL 支持用户把数据迁走。

7.2 TDSQL 数据备份

TDSQL 支持在线的实时热备，同时这个备份是基于备机上做的备份，备份支持镜像和 binlog 的备份。镜像又支持物理镜像和逻辑镜像（也叫物理备份和逻辑备份）。

物理备份的好处是速度快，直接操纵物理文件，缺点是只能备份整个数据库实例，无法选择指定库表。逻辑备份的好处是通过 SQL 的方式备份，它可以选择单个库表备份，但是如果对整个实例备份效率不及物理备份。比如说有 1T 的数据，只有 100 兆是我的关键的数据，如果为了节省存储空间就没有必要用物理备份，就用逻辑备份，只备份我们关心的库表。

有了物理备份和逻辑备份之后，基于数据的镜像，再结合 binlog 轻松实现数据的定点恢复。对于 binlog 的备份 TDSQL 的 Agent 模块完成准实时的异步备份。比如说我们每天的 0 点备份镜像，同时各个时间段的 binlog 准实时备份。当需要恢复到一个早上 6 点的数据，利用 0 点的数据镜像再结合 0 点到 6 点的 binlog，即可完成 6 点的数据恢复。

备份是在备机上做不会影响主机。整个备份过程也是有监控有告警，实现整个备份过程的追踪。

Q&A

Q：TDSQL 1.0 版本没有 SQL 引擎模块吧?

A：最早在 2002 年的时候我们使用单机版 MySQL 作为数据存取服务，而后衍生出了 TDSQL，这种计算存储相分离的架构，进而引入 SQL 引擎。

Q：请问存储节点的 MySQL 是使用官方原生的么？

A：TDSQL 在原生 MySQL 基础上做了大量调优，如线程池、强同步的优化，以及安全限制，分布式事务 XA 优化等等。

Q：银行核心要做到分库分表，开发的聚合查询如何实现?

A：SQL 引擎屏蔽了分表的细节，让业务在逻辑上看到的和单节点模式一下一样，仍然是一张独立的库表。此外，SQL 引擎会自动做数据聚合，业务开发不需要关心。

Q：a+3，如果掉失了，B 和 C 节点都没有同步过来，怎么办？A 机器已经已经无法恢复。

A： a+3 如果没有被 B,C 确认，即不满足被多数派确认，是不会应答给业务成功的，最终会以超时的错误返回给业务。如果 A 机器无法恢复，这时新加入节点会通过物理拷贝方式最快速度"克隆"出一个备节点继续代替 A 节点提供服务。

Q：Shard 版本的可以通过 pt 工具，或者 gh-ost 加字段不？会有什么限制不?

A：TDSQL 管理台提供 online ddl 的功能，会自动对多分片做原子性变更，不需要业务再用第三方工具；分布式 TDSQL 在做 DDL 的时候不允许调整 shardkey 字段，比如原来以 id 作为 shardkey，现在要调整成 name 作为 shardkey，这样是不允许的。

Q：TDSQL Shard 算法有几种？建表语句必须需要修改语法吗?

A：TDSQL shard 算法对业务屏蔽，即基于 MySQL 分区表做的 hash 拆分（该算法不允许用户修改），这么做也是为了对业务屏蔽 TDSQL 分表细节。这里并不是限制用户只能做基于 hash 的分区，用户可以在 TDSQL-shard 的基础上做二级分区（比如：按照日期、时间）。建表以及使用方面的语法，后文有关于分布式开发的详细介绍。

Q：谁来解决 MC 的可靠性?

A：一方面，MC 自身做了高可用跨机房部署，同时奇数个 MC 部署当发生故障时只要剩余存活 MC 数量大于集群 MC 的一半，MC 就可以继续提供服务；另外一方面，就算 MC 节点全部宕机，各个模块自身对 MC 也不是强依赖的，即 MC 在不工作的情况下，数据库实例的正常读写请求不会受到任何影响，只是不能处理切换、扩容等调度相关的触发式操作。

Q：前文讲两种模式，如果是用 Shard 模式，应用层对 Sql 语法有要求吗?

A：兼容 MySQL 99%的 SQL 语法，shard 模式与 noshard 模式最主要的区别是 shardkey 的引入。引入 shardkey 之后，为了发挥出 shard 模式下的性能优势，建议所有 sql 都带 shardkey 访问，同时在 shard 模式下，一些数据库的高级特性如：存储过程、触发器、视图等特性会受到一定的使用限制。更详细的内容后面会有专门的分布式开发的课程会专门介绍。

Q：分支到总行数据同步汇总， Oralce 同步到 TDSQL 是双向都支持么?

A：支持双向同步。这里的支持是有条件的：从 TDSQL 到 Oracle 是可以做到准实时同步，但是从 Oracle 到 TDSQL 目前还无法做到准实时同步，后续会支持。

Q：分布式表和非分布式表如何做 join?

A：分布式 TDSQL 下存在两种表分别是大表和小表，大表就是 shard 表（分布式表），小表又叫广播表，会冗余到所有数据节点上。分布式表和非分布式表做 join 时，由于分布式表所在的物理节点上存在非分布式表的一份冗余，因而可以在单个数据节点上完成 join。

Q：是否支持自建私有云？如果公有云，成本会不会上升？

A：支持私有云的，TDSQL 大部分的银行客户都是采用私有云部署模式，并和外网隔离。公有云的成本相比私有云明显是前者更低，像私有云需要自建机房，搭建光纤设备，但是好处是独占物理硬件资源。公有云的话都是和公有云上的其他用户公用一套 IDC、网络环境。

Q：是否支持 K8S 部署？

A：TDSQL 自带了一键部署解决方案，不依赖 K8S。

Q：分局分表支持大表关联查询吗？

A：支持。

第二章：破解分布式数据库的高可用难题：TDSQL 高可用方案实现

TDSQL 是腾讯推出的金融级分布式数据库。在可用性和数据一致性方面，基于自主研发的强同步复制协议，在保证数据的跨 IDC 双副本的同时，具备较高的性能。在强同步复制的基础

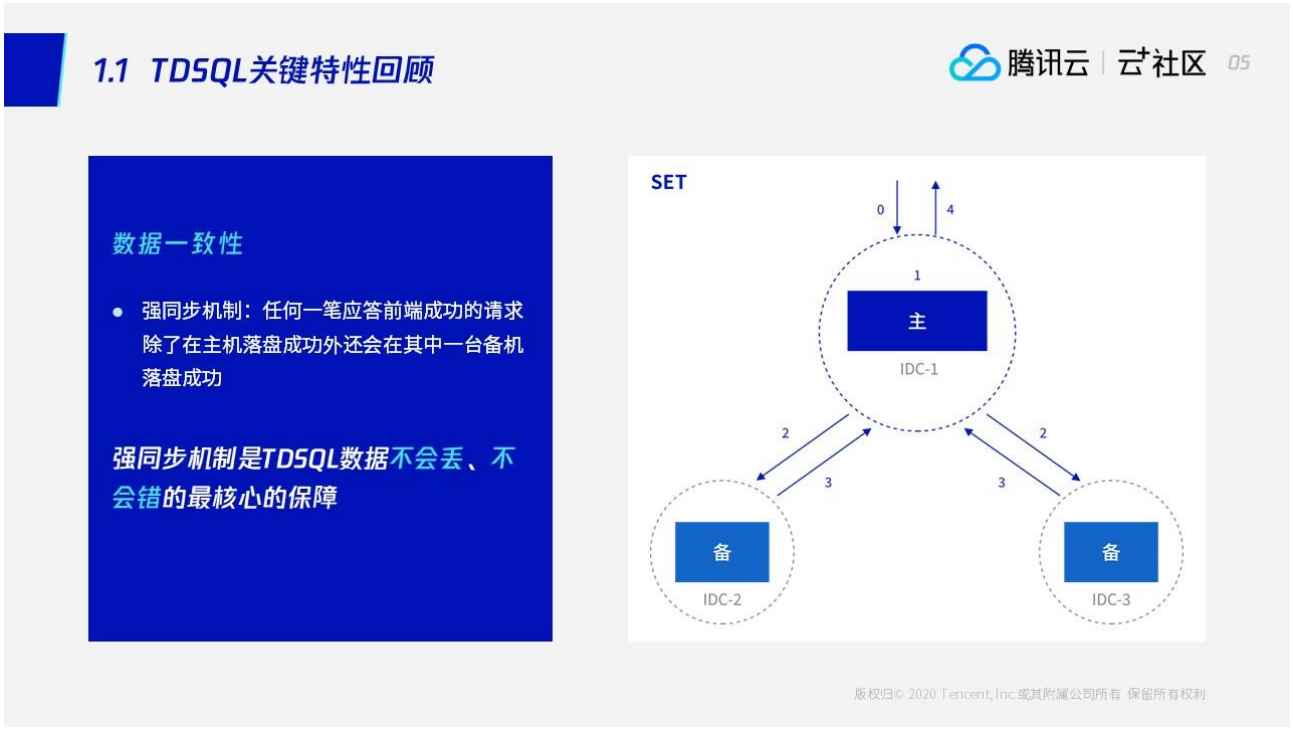
上，TDSQL 又实现了一套自动化容灾切换方案，保证切换前后数据零丢失，为业务提供 7×24 小时连续高可用服务。

事实上，不光是数据库，任何对可用性有较高要求的系统都需要具备高可用的部署架构。本章节将介绍 TDSQL 的几种典型部署架构，以及各种架构的优缺点。在实际生产实践中，会有各种各样的资源条件限制，比如同城有一个机房和有两个机房的容灾效果就完全不一样；再比如虽然有两个机房，但是这两个机房的规格相差比较大，有的机房可能网络链路比较好，而有的机房可能网络链路比较差，等等……所以，如何在有限的资源条件下搭建一套性价比高或者说效能比高 TDSQL，是我们这次分享要探讨的主要内容。

在正式切入正题前，我们先回顾一下 TDSQL 的核心特性以及整体架构。核心特性方面，本章重点将聚焦在“金融级高可用”。TDSQL 如何做到 99.999%以上的可用性呢？所谓“五个九”的高

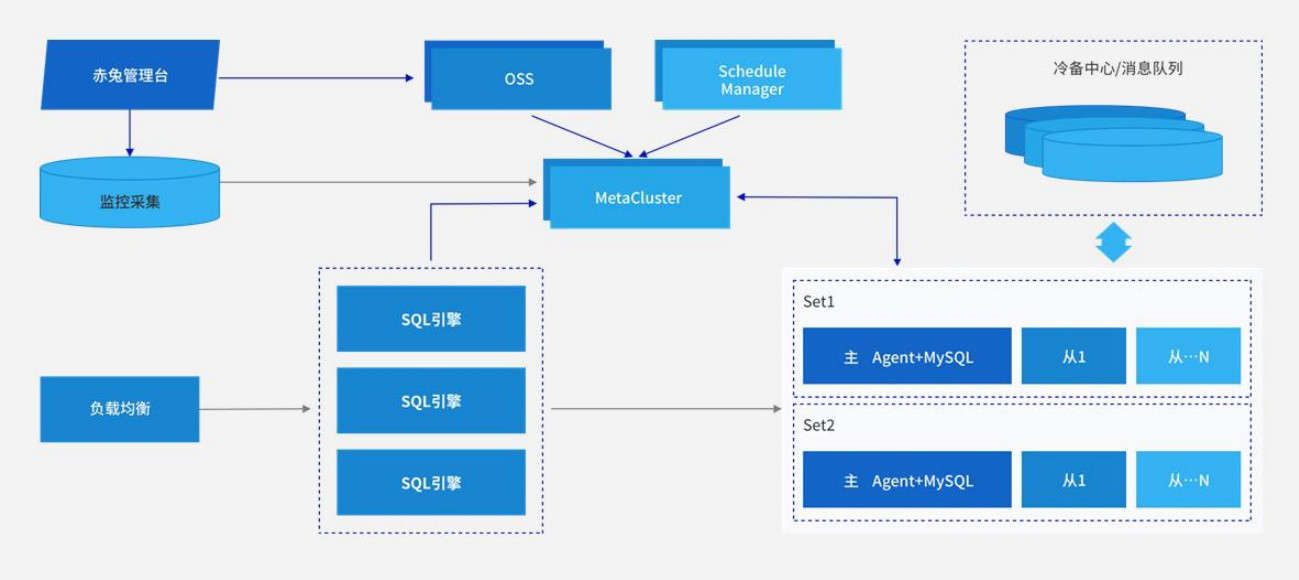
可用意味着：全年不可用时间不可超过 5 分钟。我们知道，故障是一种无法避免的现象，同时故障也是分级的，从软件故障，操作系统故障，再到机器重启、机架掉电……这是一个灾难级别从低到高的过程，对于金融级数据库需要考虑和应对更高级别的故障场景，如：整个机房掉电甚至机房所在的城市发生地震、爆炸等自然灾害。如果发生这类故障，我们的系统首先能否保证数据不丢，其次在保证数据不丢的前提下需要多久恢复服务，这都是金融级高可用数据库需要考虑的问题。

1 TDSQL 数据库一致性：强同步机制是最核心的保障



TDSQL 的关键特性——强同步机制，它是 TDSQL 保证数据不会丢、不会错的关键，并且相比于 MySQL 的半同步复制，TDSQL 强同步复制的性能更是接近于异步。

那么这个高性能强同步是如何工作的呢？强同步复制要求任何一笔应答业务成功的请求，除了在主机落盘成功以外，还需在至少一个备机落盘成功。所以我们看到，一个请求到了主机之后，立刻发送发给备机，当两台备机中的一台应答成功之后主机才能应答业务成功。也就是说任何成功应答给前端业务的请求一定会有两个副本，一份在主节点上，另外一份在备节点上。所以，强同步是数据多副本的关键性保障。TDSQL 通过引入了线程池模型与灵活的调度机制，将工作线程异步化，实现了对强同步的性能大幅提升，改造后其吞吐量接近于异步。



TDSQL 核心架构

看完强同步，我们接下来再回顾一下 TDSQL 的核心架构：负载均衡是业务的入口，业务请求经过负载均衡到达 SQL 引擎模块，SQL 引擎再将 SQL 转发到后端数据节点。图中的上半部分是集群的管理调度模块，作为集群的总控制器承担着资源管理、故障调度等工作。所以，TDSQL

整体分为：计算层、数据存储层，以及集群管理节点。集群管理节点我们之前强调过，一个集群部署一套就行，一般是 3 个、5 个、7 个这样的奇数个部署。为什么是奇数呢？因为当发生灾难的时候，奇数个能够形成选举，话句话说，比如 3 个节点部署在 3 个机房，其中有 1 个机房故障的话，而另外两个机房可以互通并且都连不上第三个机房，就可以对第三个机房故障达成共识将其踢除。我们可以把三个机房的管理模块理解为集群的三颗大脑，这组大脑坏掉一个后，如果剩余的大脑能够在数量上达到初始数量的一半就可以继续提供服务反之则不行。

2 高可用集群的部署实践

以上是对 TDSQL 一些核心特性的回顾，接下来我们看一下各个模块在机型上的选择。对于计算与存储相分离的分布式架构数据库，我们该如何选择机器？我们知道，要想让一个 IT 系统发挥出最大的价值，就是要尽可能地榨干机器的资源，让这些资源物有所值效能比高。如果这个机器 CPU 跑得很满，但是 IO 没有什么负载，或者说内存配 128 个 G 但实际上只用了 2 个 G。这种就属于效能比非常低的部署，无法发挥出机器以及系统的整体性能。

首先是 LVS 模块，首先作为接入层它不是一个 TDSQL 的内部组件，TDSQL 的 SQL 引擎兼容不同的负载均衡，比如软件负载 LVS、硬件负载 L5 等。作为接入层的负载均衡一般都是 CPU 集型服务，因为它需要维护管理大量链接请求，相对较为耗 CPU 和内存。所以这里推荐的配置中 CPU 比较高，16vCPU，32G 内存，一定是万兆网口。到了今天，网卡设备的成本已经非常低了，所以一般都给装配万兆网卡。而 vCPU 这里强调的是逻辑核 16 核就可以（可能实际

物理核只有 8 个），因为我们大部分程序都是多线程的。

我们再看计算节点。如果集群规模较小，同时资源相对比较紧张，计算节点可以跟存储节点复用机器，因为存储节点的机型基本能够满足计算节点的需求，一个 16vCPU，32G+内存，万兆网口。

说完了接入层和计算层，下面我们再看看数据存储层。存储节点负责数据存取，属于 IO 密集型服务，建议用 PCI-E 的 SSD，并且需要独站物理机。对于数据库来说，我们推荐部署在真实的物理机上，相比虚拟机更为稳定。此外，有条件的建议再做一层 Raid0，让数据节点的读写能力更为强劲。有些同学肯定会问，数据节点为什么不做一个 Raid5、Raid10 而直接做 Raid0。因为 TDSQL 本身已经是一主多从的架构，甚至可以加更多从机，在磁盘阵列这里我们就没必要继续做冗余，无限冗余只会让效能比降低。作为数据节点，这里推荐的配置是 32vCPU，64G 内存。数据节点采用的 Innodb 引擎是一个优先使用缓存的引擎，也就是说大内存对性能的提升具有显著的作用。所以，这里推荐大内存，万兆网，PCI-E 的 SSD 的机型。

接下来是管理节点：推荐配置是 8 核 CPU、16G 内存，万兆网口。管理节点任务比较轻，一个集群也只需要少量的管理节点。如果说没有物理机用虚拟机也可以，配置相对于前面的计算、存储节点明显可以低一些。

备份节点的话，硬盘越大越好，它主要负责存储冷数据，用普通的 SATA 盘就可以。

以上机器的型号不用太关注，是腾讯内部的一个编号，没有什么实际意义。这里简单做一个总

结，计算节点依赖 CPU 和内存，对磁盘没有太多要求。存储节点虽然对 CPU 内存也要求较高，但它更强调 IO 的强劲（需要 PCI-E 的 SSD）。

通过刚刚的介绍希望可以帮助大家进一步加深对 TDSQL，尤其是这种计算存储相分离的数据库的认识。从机型配置的解读我们可以清楚的了解，怎样的机型搭配能够让系统的性能发挥最佳。

总结起来，如果机型正确搭配，业务也按照规范使用，那么就可以轻松发挥出数据库强劲的性能，即以较低的运营成本获取较高的业务支撑能力。海量的业务都是伴随着现金流，比如说广告、游戏、电商，一套低成本的系统如果在满负荷工作下轻松高效处理这类业务请求，带来的经济效应是比较可观的。

3 跨城跨机房级别容灾部署方案

第三部分，开始切入我们的正题，这一章我们将介绍比较典型的几种部署方案，那些耳熟能详的名词：同城三中心，两地三中心，异地多活，同城双活分别代表什么意义，或者说能带来什么样的效果呢。接下来就为大家一一将这些问题的答案揭开。

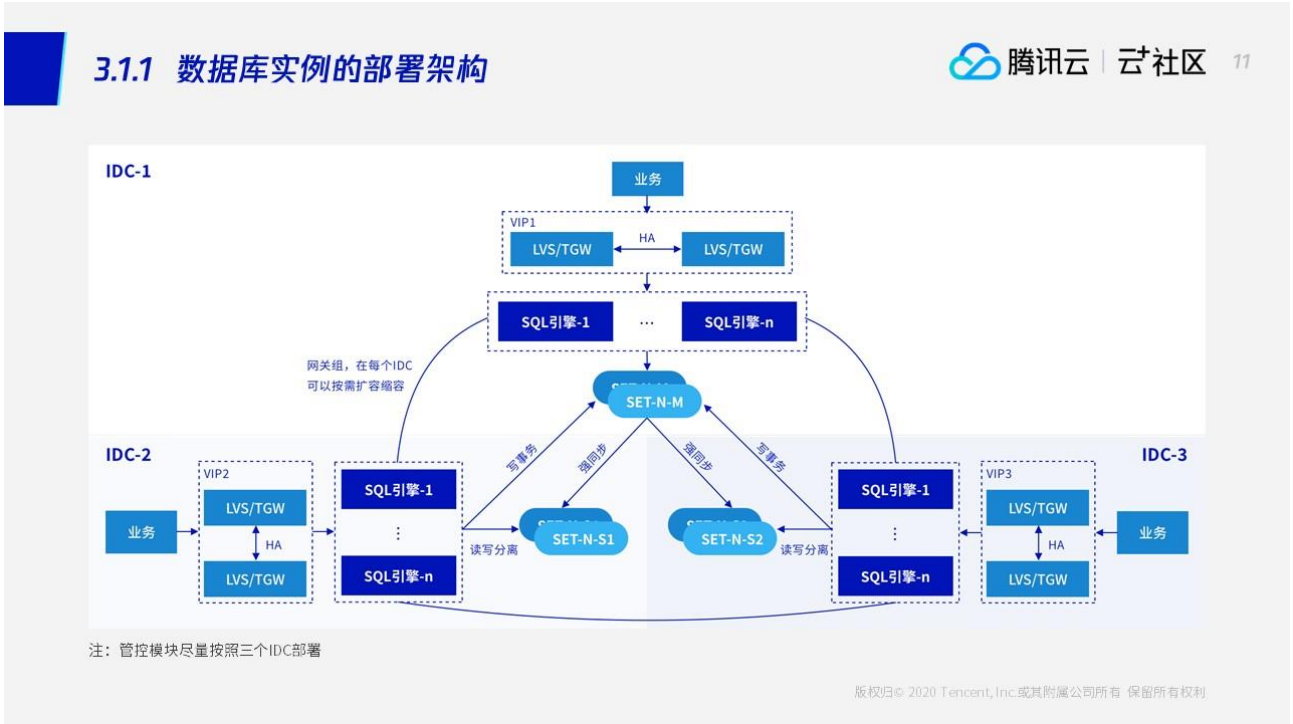

腾讯云 | 云社区
09
第三章：部署方案
3.1 同城三中心架构
3.2 同城单中心架构
3.3 两地三中心架构
4.4 两中心架构
版权归© 2020 Tencent, Inc.或其附属公司所有 保留所有权利

3.1 “同城三中心”架构


3.1 同城三中心架构
腾讯云 | 云社区
10
每个数据库实例采用三个节点的模式，分布式在3个IDC
每个IDC提供两台做高可用的LVS负载均衡系统
每个IDC内的业务访问本IDC的LVS对应VIP
版权归© 2020 Tencent, Inc.或其附属公司所有 保留所有权利

第一部分是同城三中心架构。同城三中心顾名思义：在一个城市有 A、B、C 三个机房，TDSQL 仍采用"一主两备"结构，很显然我们需要将三个数据节点分别部署在三个机房，其中主节点在一个机房，两个备节点分别部署在另外两个机房。

每个 IDC 提供两台高可用的 LV5 做负载均衡系统。为什么每台 IDC 都要放一个 LV5？因为每个 IDC 有自己的业务，对应都需要有一个独立的负载接入。从接入层看三个机房是一个相对平行的对等架构，三个机房都放了自身的业务，可能第一个机房支撑的是一个全国大区的业务，第二个机房是另外一个大区，对等就是这样的含义。这种架构比较简单，整体也比较清晰。



我们再看架构图，刚刚说了它是一个对称的结构。从上往下首先看业务，每个机房可能都部署有业务系统，每个机房的业务通过 LVS 负载接入访问到 TDSQL 的 SQL 引擎。由于在同一机房需要部署多个 SQL 引擎提供高可用服务，而对于业务来说更希望屏蔽后端多个 SQL 引擎的访

问方式，所以这里引入一层 LVS 接入层，业务只需访问负载均衡的 VIP 即可。当请求到达 SQL 引擎后，根据路由信息将 SQL 发到 master 或者 slave 节点，最后返回业务数据。我们再看数据节点，一主两备分别部署在三个机房，任何一个机房故障，master 节点都可以切换到另外两个机房中的一个。同城三中心架构下，从计算层到存储层都不存在单点，做到了高可用容灾。任意一个机房的故障都不会造成数据丢失，同时在 TDSQL 一致性切换机制的保证下，能够在 30s 内完成故障节点的切换。

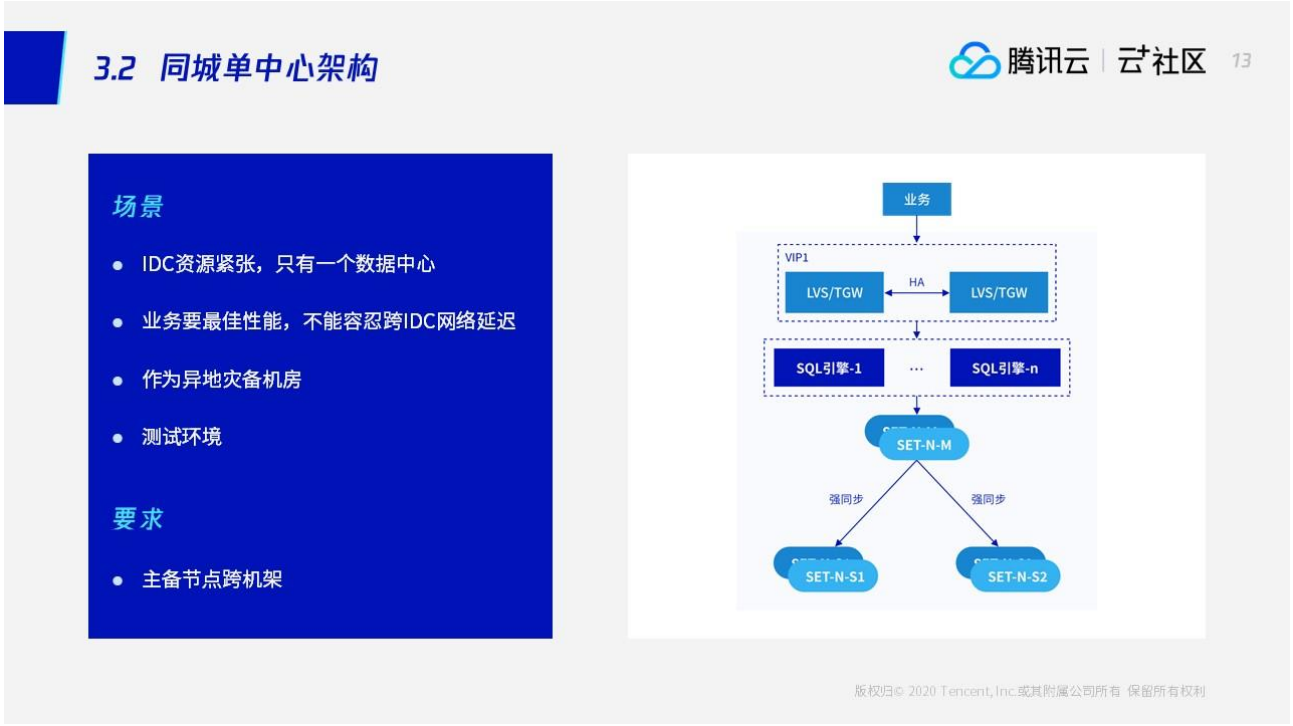
这个图里面没有画出管理节点，我们刚刚也说了管理节点可以看做整个集群的大脑，负责判断当前的全局态势。三个机房很明显需要部署三颗大脑，之所以是"三"刚刚也讲过，当其中一个大脑出问题的时候，另外两个可以形成多数派，完成相互投票确认将故障大脑踢除。

3.2 "同城单中心"架构

"同城单中心"架构的场景有几种情况：

第一种场景是 IDC 资源比较紧张，只有一个数据中心。这种场景下做不到跨机房部署，只能按照跨机架方式部署，当主节点故障或者主节点所在的机架故障，能够自动切换。

第二种场景是业务追求极致的性能，甚至不能容忍跨 IDC 网络延迟。虽然现在的机房之间都是光纤网络，相隔 50km 的两个机房之间的网络延迟也只有不到 1ms。但有些特殊的业务甚至无法忍受 1 毫秒的延迟。这种情况下我们只能将主备部署在同一机房。

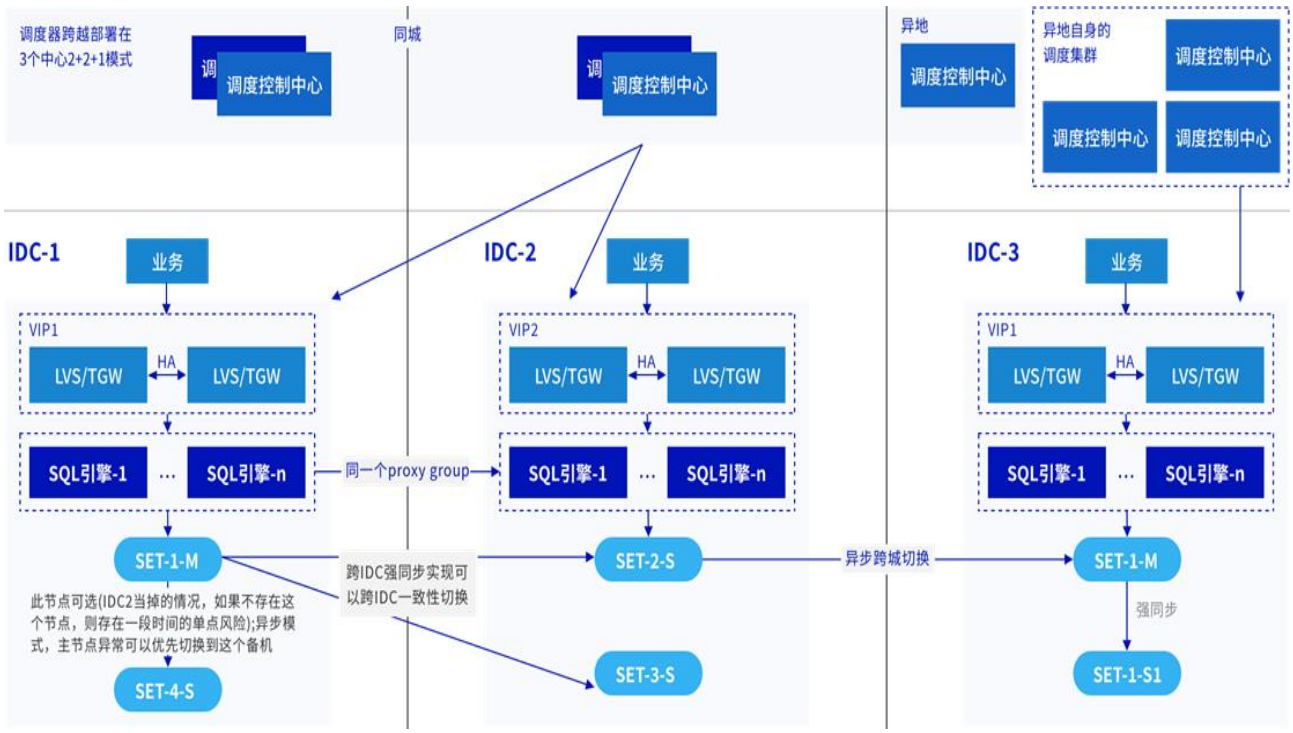


第三类是作为异地灾备机房。作为灾备储存一般也没有实际业务访问，更多的可能是做备份归档，因此对它的资源投入比较有限。

第四种是作为测试环境，在此不展开。

3.3 “两地三中心”架构

“两地三中心”架构，它是银行常见的部署方式，更是监管要求的基本部署架构。这种架构通过同城两个数据中心加异地一个数据中心，在较低的成本下，提供较好的可用性和数据一致性。在节点异常、IDC 异常都能做到自动切换，非常适合金融场景，是 TDSQL 重点推荐的部署方式。



“两地三中心”数据库实例的部署架构

部署方式方面，我们从上往下看，分别是同城两个机房和异地一个灾备机房。最上层是集群的大脑管理模块，分别跨三个机房部署。

管理模块的部署可以采用“2+2+1”也可以“1+1+1”的方式。我们知道，如果按照“2+2+1”的部署方式，当第一个机房故障时，还剩下“2+1”颗大脑，“2+1”比 5 的一半要多，剩下的“2+1”形成多数派将故障节点踢掉，同时继续提供服务。

此外，我们看到数据节点采用的是“一主三备”的模式，并且是跨机房强同步，同机房异步。为什么同机房这里是异步，不能做强同步？如果同机房是强同步的话，由于它和主节点距离上要比另外两个跨机房的备节点近（IDC1、IDC2 之间相隔了平均至少 50 公里），业务每次发送给主节点的请求都是这个同机房的强同步节点率先应答，最新的数据永远都只会落到同机房的备

节点。而我们希望数据的两个副本应该位于相隔 50km 以上的不同机房，这样才能保证跨机房主备切换时数据能够保持一致。

可能有人会问，这个 IDC1 配置的异步节点和不放没有区别。这里解释一下为什么有了这个异步节点后更好。我们考虑一种情况，当备机房 IDC2 发生了故障，备机房里面的两个节点全部宕机，IDC 1 这里的 master 节点就成单点了。此时，如果开启强同步，由于没有备机应答，主节点依然无办法提供服务；但如果关闭强同步继续提供服务，数据存在单点风险，如果这时主节点发生软件硬件故障，数据就再也无法找回。一个比较好的方案是：给 IDC1 增加了一个跨机架的异步节点，当 IDC2 挂掉之后，这个异步节点会提升为强同步。这样在只剩下一个机房的前提下，我们仍然能够保证一个跨机架的副本，降低主机的单点风险。

看完主城两个机房，我们再看看异地灾备机房。作为异地灾备机房，一般是和主节点相隔 500 公里以上，延迟在 10ms 以上的机房。在这样的网络条件下，灾备节点和主城之间只能采用异步复制的方式同步数据，因而异地灾备节点承担的更多是备份的职责，日常不会有太多正式业务访问。虽然表面上看有点花瓶，没有它却也万万不行。假如有一天发生了城市级别故障，灾备实例仍可以为我们挽回 99%以上的数据。正是由于灾备节点和主节点的这种异步弱关系，才允许我们的灾备实例在备城是一个独立部署的单元。

异地灾备机房除了作为异步数据备份外，另外一个重要的职责是：当主城的一个机房故障，通过和主城另外一个正常的机房形成多数派，将故障机房踢掉进而完成主备切换。部署在异地的这个大脑，在大部分时间都不参与主城的事宜，只有在主城的一个机房发生故障时才介入。正常情况下，主城的模块访问主城的大脑，备城的模块访问备城的大脑，不会交叉访问导致延迟

过高的问题。

3.4 “两中心”架构

“两中心”架构，具体来说，同城只有两个机房，根据我们上一个 PPT 的经验，在两机房部署 TDSQL 需要按照同机房异步，跨机房强同步的方式部署。因而采用四节点的模式，分布式在 2 个 IDC。

然而“两中心”架构有个需要权衡的地方是，只有部署在备机房而且故障的不是备中心，才能实现自动跨 IDC 容灾。但如果是备中心故障，事实上，在同机房异步，跨机房强同步的方式下，不管是部署在主机房还是备机房，假如发生故障，无法顺利完成多数派选举以及自动故障切换，要么强同步节点无法被提升形成多数派，要么多数派随机房故障而故障，需要人工介入。因而在高可用要求的场景下，一般更为推荐“两地三中心”等 7*24 小时高可用部署架构。

3.5 标准化高可用部署方案总结

最后总结一下本章节：

- 首先，对于跨城市容灾一般建议在异地搭建独立的集群模式，通过异步复制实现同步。主城和备成可以采用不同的部署方式，如主城一主三备，备城一主一备的方式灵活自由组合。

- 现网运营的最佳方案是同城三中心加一个异地灾备中心，其次是金融行业标准的两地三中心的架构。这两种架构都能轻松实现数据中心异常自动切换。

- 如果只有两个数据中心，做不到任意数据中心异常都能自动切换，需要一些权衡。

事实上，不光是对于数据库系统，任何做一种高可用系统都需要做基于部署架构方面的考量。

Q&A

Q：同城主备切换一次多长时间?

A：30 秒以内。

Q：两地三中心的主城是不是设置成级联?

A：这个问题非常好，如果从主城的视角看，这显然是一个级联关系，数据先由主城的 master 同步到主城的 slave，再通过主城的 slave 同步到备城的 master，一层层向下传递数据。

Q：请教一下强同步会等 SQL 回放吗?

A：不会等，只要 IO 线程拉到数据即可。因为基于行格式的 binlog 是具备幂等写的，我们通过大量的案例证明它是可靠的。此外，增加了 apply 反而会使得平均时耗的上升和吞吐量的下降。最后，假如 apply 有问题，TDSQL 的监控平台会立刻识别并告警，提醒 DBA 确认处理。

Q：备机只存 binlog 不回放，性能上跟得上主吗?

A：备机拉取 binlog 和回放 binlog 是两组不同的线程，分别叫 IO 线程和 SQL 线程，并且两组线程互不干扰。IO 线程只负责到 master 上下载 binlog，SQL 线程只回放拉取到本地的 binlog。上一个问题说的是强同步机制不等待回放，并不是说到备机的 binlog 不会被回放。

Q：同城三中心写存储节点都在 IDC1，那么在 IDC2 的业务延迟是不是很大?

A：同城机房现在都是光纤传输，时耗基本都是做到 1 毫秒以下，完全没必要担心这种访问时耗。当然，如果机房设施比较陈旧，或者相隔距离间的网络链路极为不稳定，为了追求卓越性能可能需要牺牲一部分容灾能力。

Q：一主两备，SQL 引擎做成故障切换有 VIP 方式吗?

A：当然有，多个 SQL 引擎绑定负载均衡设备，业务通过 VIP 方式访问 TDSQL，当 SQL 引擎故障后负载均衡会自动将其踢掉。

Q：这样不是三个业务各自写一个库吗?

A：不是的，三个业务都写到主库。SQL 引擎都会路由到主库，一主两备。TDSQL 强调任何一个时刻只有一个主提供服务，备机只提供读服务不提供写服务。

Q：同城多副本，多 SET 对同城 IDC 之间网络要求有什么?

A：5 毫秒以内的延迟。

Q：如果两个强同步主从可以设置其中一个返回?

A：TDSQL 强同步默认机制就是等待一台强同步的备机应答。

Q：中间一个节点挂了，异地节点会不会自动连接到主节点?

A：当然会。

Q：强同步和半同步复制相比的优势是什么?

A：强同步跟半同步复制相比，最直观的理解可能有人会问，强同步不就是把半同步的超时时间改成无限吗？其实不是这样的，TDSQL 强同步这里的关键不是在解决备机应答的问题，而是要解决这种增加了等待备机的机制之后，如何能保证高性能、高可靠性。换句话说，如果在原生半同步的基础上不改造性能，仅把超时时间改成无限大的时候，其实跑出来的性能和异步比甚至连异步的一半都达不到。这个在我们看来也是无法接受的。相当于为了数据的一致性牺牲了很大一部分性能。

TDSQL 强同步复制的性能是在原生半同步的基础上做了大量的优化和改进，使得性能基本接近于异步，并且能实现数据零丢失——多副本，这是 DSQL 强同步的一个特点。上一期的直播我们介绍了，TDSQL 如何实现高性能强同步的。比如经过一系列的线程异步化，并且引入了线程池模型，并增加一个线程调度的优化等。

Q：仲裁协议用的哪一个?

A：多数派选举。

第三章：亿级流量场景下的平滑扩容：TDSQL的水平扩容方案实践

本章主要包含这四部分：

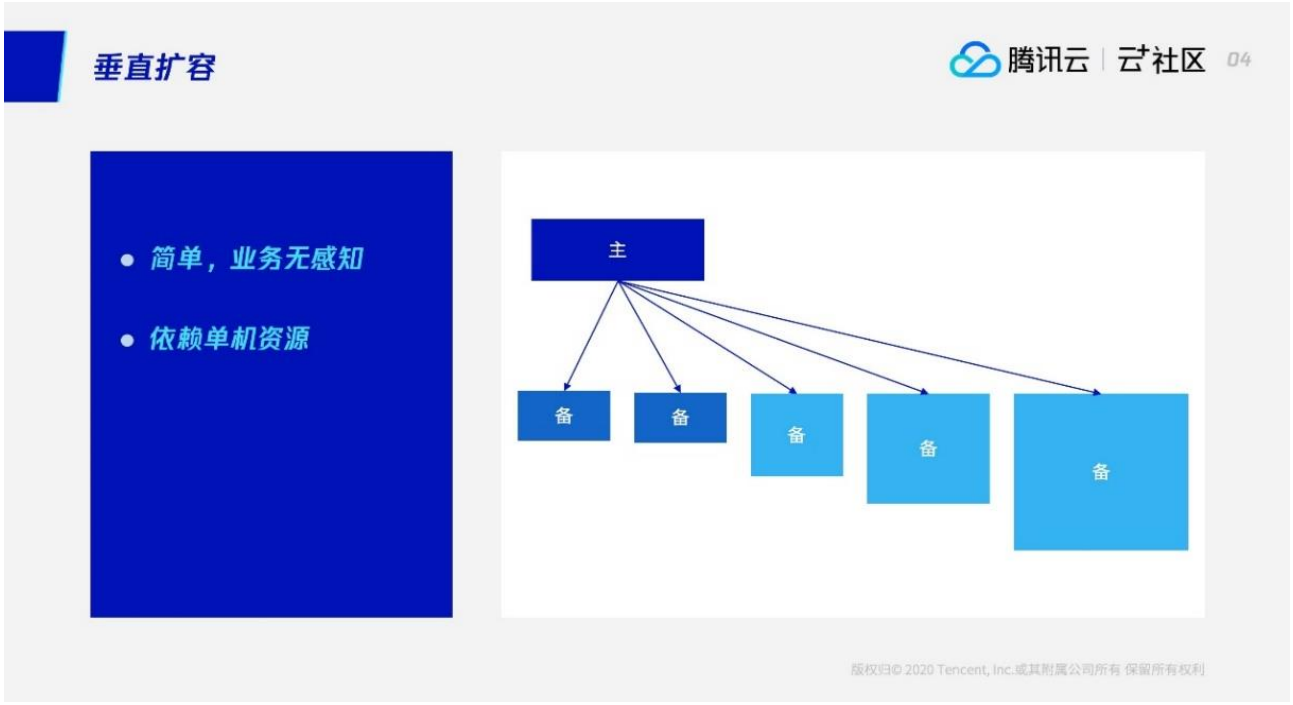
- 第一部分首先介绍水平扩容的背景，主要介绍为什么要水平扩容，主要跟垂直扩容进行对比，以及讲一下一般我们水平扩容会碰到的问题。
- 第二部分会简单介绍 TDSQL 如何做水平扩容，让大家有一个直观的印象。
- 第三部分会详细介绍 TDSQL 水平扩容背后的设计原理，主要会跟第一部分进行对应，看一下 TDSQL 如何解决一般水平扩容碰到的问题。
- 第四部分会介绍实践中的案例。

1 数据库水平扩容的背景和挑战

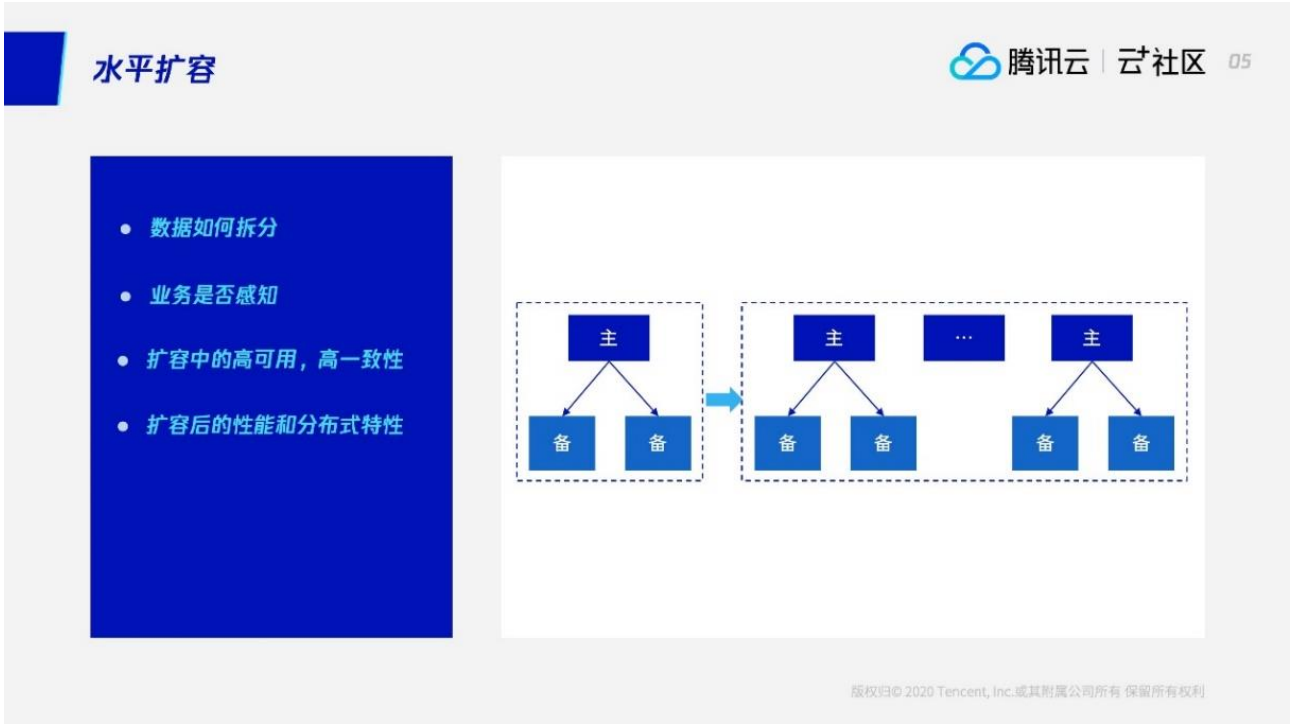
首先我们看水平扩容的背景。扩容的原因其实非常直观，一般来说主要是随着业务的访问量，或者是需要的规模扩大，而现有的容量或者性能满足不了业务的需求，主要表现在 TPS、QPS 不够或者时延超过了业务的容忍范围，或者是现有的容量不能满足要求了，后者主要是指磁盘或者网络带宽。一般碰到这种问题，我们就要扩容。扩容来说，其实比较常见的就是两种方式，一种是垂直扩容，一种是水平扩容。这两种有不同的特点，优缺点其实也非常明显。

1.1 水平扩容 VS 垂直扩容

首先我们看一下垂直扩容。垂直扩容，主要是提高机器的配置，或者提高实例的配置。因为，我们知道，大家在云上购买一个数据库或者购买一个实例，其实是按需分配的，就是说对用户而言，可能当前的业务量不大，只需要两个 CPU 或者是几 G 的内存；而随着业务的增长，他可能需要对这个实例进行扩容，那么他可能当前就需要 20 个 CPU，或者是 40G 的内存。这个时候，在云上我们是可以通过对资源的控制来动态地调整，让它满足业务的需求——就是说可以在同一台机器上动态增加 CPU。这个扩容的极限就是——当整台机器的 CPU 和内存都给它，如果发现还不够的话，就需要准备更好的机器来进行扩容。这个在 MySQL 里面可以通过主备切换：通过先选好一台备机，然后进行数据同步；等数据同步完成以后，进行主备切换，这样就能利用到现在比较好的那台机器。大家可以看到，这整个过程当中，对业务来说基本上没有什么影响——进行主备切换，如果换 IP 的话，其实是通过前端的或者 VIP 的方式，对业务来说基本上没有什么影响。那么它一个最大的不好的地方就是，它依赖于单机资源：你可以给它提供一个更好的机器，从而满足一定量的要求。而随着业务更加快速的发展，你会发现现在能提供的最好的机器，可能还是满足不了，相当于扩不下去了。因此，垂直扩容最大的缺点就是，它依赖于单机的资源。



跟垂直扩容对比，另外一种方式我们叫水平扩容。水平扩容最大的优点是解决了垂直扩容的问题——理论上水平扩容可以进行无限扩容，它可以通过增加机器的方式来动态适应业务的需求。

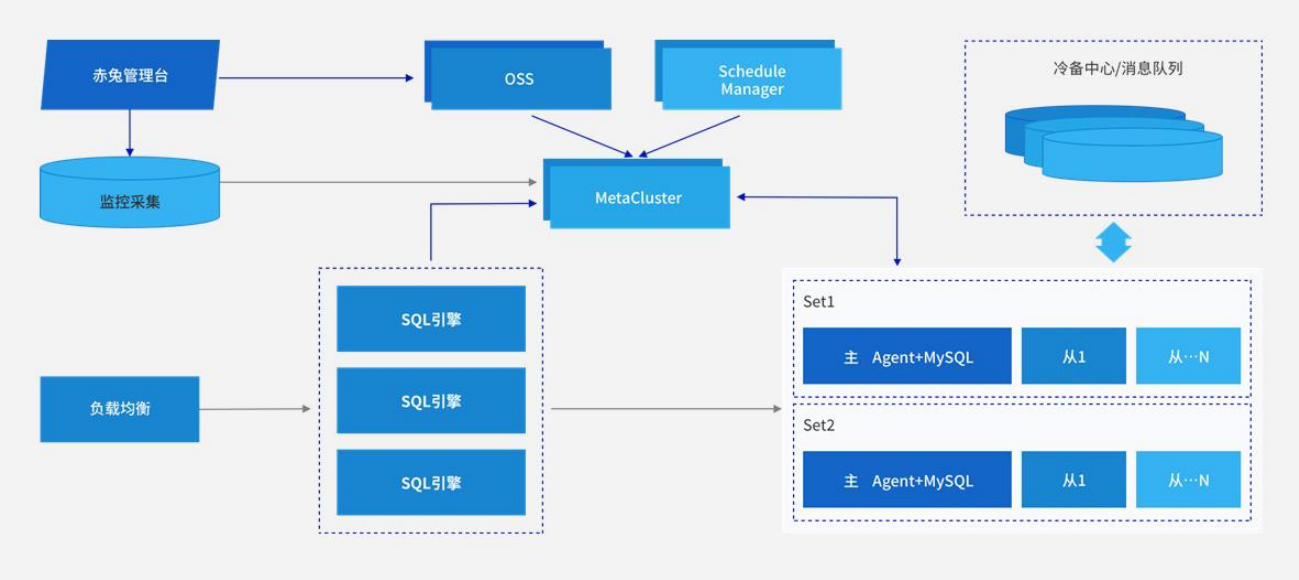


水平扩容和垂直扩容相比，它可以解决垂直扩容的问题，但是会引入一些其他的问题。因为水平扩容比垂直扩容更加复杂，下面我们分析下可能遇见的问题，以及后面我们会介绍 TDSQL 的解决方案：

- 首先，在垂直扩容里面，系统经过扩容以后，其实数据总体来说还是存在一个节点，一主多备架构中，备机上也存储着所有数据。而水平扩容过程中数据会进行拆分，面临的第一个问题是，数据如何进行拆分？因为如果拆分不好，当出现热点数据时，可能结果就是，即使已经把数据拆分成很多份了，但是存储热点数据的单独节点会成为性能瓶颈。
- 第二点，在整个水平扩容过程中，会涉及到数据的搬迁、路由的改变。那么整个过程中能否做到对业务没有感知？或者是它对业务的侵入性大概有多少？
- 第三，在整个扩过程中，因为刚才有这么多步骤，如果其中一步失败了，如何能够进行回滚？同时，在整个扩容过程中，如何能保证切换过程中数据高一致性？
- 再者，在扩容以后，由于数据拆分到了各个节点，如何能保证扩容后的性能？因为理论上来说，我们是希望我随着机器的增加，性能也能做到线性提升，这是理想的状态。实际上在整个水平扩容的过程中，不同的架构或者不同的方式，对性能影响是比较大的。有时候会发现，可能扩容了很多，机器已经增加了，但是性能却很难做到线性扩展。
- 同样的，当数据已经拆分成多份，我们如何继续保证数据库分布式的特性？在单机架构下，数据存储一份，类似 MySQL 支持本地做到原子性——可以保证在一个事物中数据要么全部成功，要么全部失败。在分布式架构里，原子性则只能保证在单点里面数据是一致性的。因此，从全局来说，由于数据现在跨节点了，那么在跨节点过程中怎么保证全局的一致性，怎么保证在多个节点上数据要么全部写成功，要么全部回滚？这个就会涉及到分布式事务。

所以大家可以看到，水平扩容的优点很明显，它解决了垂直扩容机器的限制。但是它更复杂，引入了更多的问题。接下来大家带着这些问题，下面我会介绍 TDSQL 如何进行水平扩容，它又是如何解决刚才说的这些问题的。

2　TDSQL 水平扩容实践

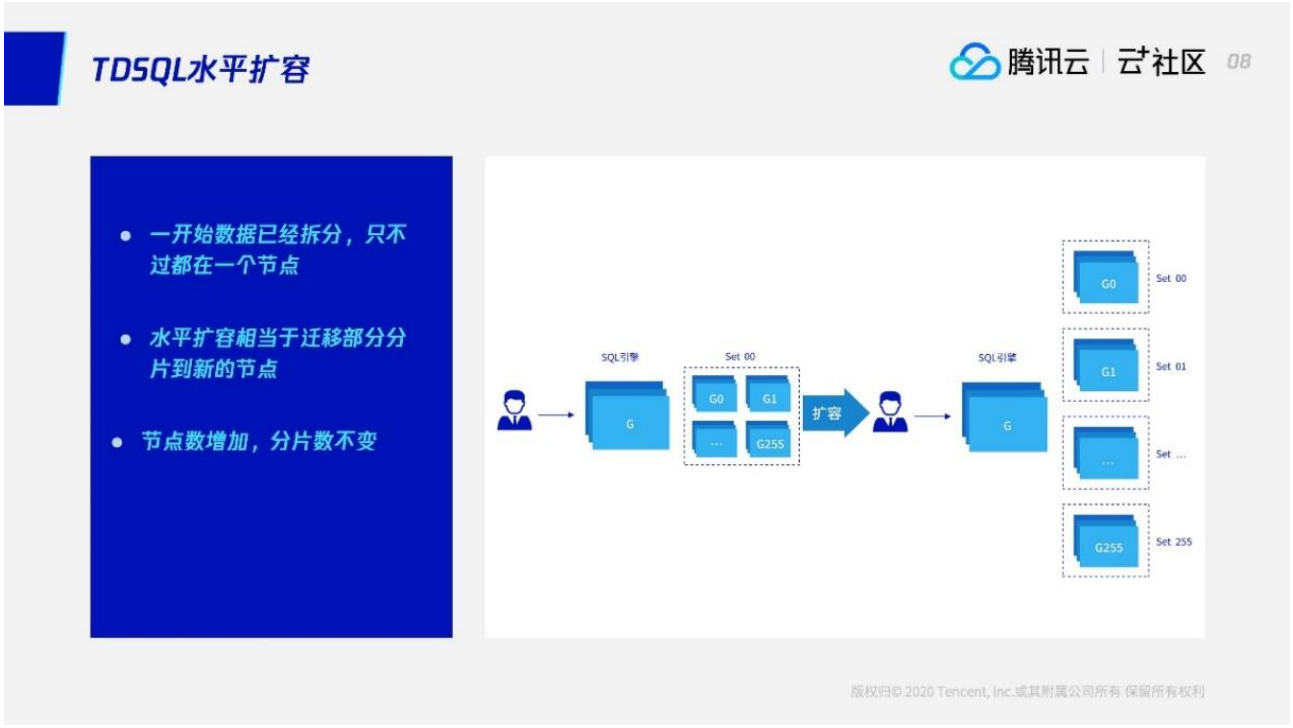


首先我们看一下 TDSQL 的架构。TDSQL 简单来说包含几部分：

- 第一部分是 SQL 引擎层：主要是作为接入端，屏蔽整个 TDSQL 后端的数据存储细节。对业务来说，业务访问的是 SQL 引擎层。
- 接下来是由多个 SET 组成的数据存储层：分布式数据库中，数据存储在各个节点上，每个 SET 我们当做一个数据单元。它可以是一主两备或者一主多备，这个根据业务需要来部署。

有些业务场景对数据安全性要求很高，可以一主三备或者一主四备都可以。这个是数据存储。

- 还有一个是 Scheduler 模块，主要负责整个系统集群的监控、控制。在系统进行扩容或者主备切换时，Scheduler 模块相当于是整个系统的大脑一样的控制模块。对业务来说其实只关注 SQL 引擎层，不需要关注 Scheduler，不需要关注数据是怎么跨节点，怎么分成多少个节点等，这些对业务来说是无感知的。



整个扩容流程大家可以看一下：一开始数据都放在一个 Set 上，也就是在一个节点里面。那么扩容其实就是会把数据扩容到——这里面有 256 个 Set，会扩容到 256 台机器上。整个扩容大家可以看到有几个要点：

- 一开始虽然数据是在一个节点上，在一台机器上，但是其实数据已经进行了拆分，图示的

这个例子来说是已经拆分成了 256 份。

- 水平扩容，简单来说就是把这些分片迁移到其他的 Set 上，也就是其他的节点机器上，这样就可以增加机器来为提供系统性能。

总结起来，数据一开始已经完成切分，扩容过程相当于把分片迁到新的节点。整个扩容过程中，节点数是增加的，可以从 1 扩到 2 扩到 3，甚至扩到最后可以到 256，但是分片数是不变的。一开始 256 个分片在一个节点上，扩成两个节点的话，有可能是每 128 个分片在一个节点上；扩到最后，可以扩到 256 个节点上，数据在 256 台机器，每台机器负责其中的一个分片。因此整个扩容简单来说就是搬迁分片。具体细节下文将继续深入介绍。

在私有云或者是公有云上，对整个扩容 TDSQL 提供了一个统一的前台页面，用户在使用的过程中非常方便。

我们看一下这个例子。现在这个案例中有两个 Set，也就是两个节点，每一个节点负责一部分的路由，第一个节点负责 0-31，另一个名字是 3，负责的路由信息是 32-63。如果要进行扩容，在前台页面上我们会有一个“添加 Set”的按钮，点一下“添加 Set”，就会弹出一个对话框，里面默认会自动选择之前的一个配置，用户可以自己自定义。

此外，因为扩容要进行路由切换，我们可以手动选择一个时间，可以自动切换，也可以由业务判断业务的实际情况，人工操作路由的切换。这些都可以根据业务的需要进行设置。

第一步创建好以后，刚才说大脑模块会负责分配各种资源，以及初始化，并进行数据同步的整个逻辑。最后，大家会看到，本来第一个节点——原来是两个节点，现在已经变成三个节点了。扩容之前，第一个节点负责是 0-31，现在它只负责 0-15，另外一部分路由由新的节点来负责。所以整个过程，大家可以看到，通过网页上点一下就可以快速地从两个节点添加到三个节点—

—我们还可以继续添加 Set，继续根据业务的需要进行一键扩容。

3 TDSQL 水平扩容背后的设计原理

接下来将介绍，第一章提到的水平扩容会遇到的一些问题，我们是如何来解决这些问题的。事实上，这些问题是不管在哪个系统做水平扩容，都需要解决的。

3.1 设计原理：分区键选择如何兼顾兼容性与性能

刚才提到，水平扩容第一个问题是数据如何进行拆分。因为数据拆分是第一步，这个会影响到后续整个使用过程。对 TDSQL 来说，数据拆分的逻辑放到一个创建表的语法里面。需要业务去指定 shardkey"等于某个字段"——业务在设计表结构时需要选择一个字段作为分区键，这样的话 TDSQL 会根据这个分区键做数据的拆分，而访问的话会根据分区键做数据的聚合。我们是希望业务在设计表结构的时候能够参与进来，指定一个字段作为 shardkey。这样一来，兼容性与性能都能做到很好的平衡。

其实我们也可以做到用户创建表的时候不指定 shardkey，由我们底层这边随机选择一个键做数据的拆分，但这个会影响后续的使用效率，比如不能特别好地发挥分布式数据库的使用性能。我们认为，业务层如果在设计表结构时能有少量参与的话，可以带来非常大的性能优势，让兼容性和性能得到平衡。除此之外，如果由业务来选择 shardkey——分区键，在业务设计表结构的时候，我们可以看到多个表，可以选择相关的那一列作为 shardkey，这样可以保证数据拆

分时，相关的数据是放在同一个节点上的，这样可以避免很多分布式情况下的跨节点的数据交互。

我们在创建表的时候，分区表是我们最常用的，它把数据拆分到各个节点上。此外，其实我们提供了另外两种——总共会提供三种类型的表，背后的主要思考是为了性能，就是说通过将global表这类数据是全量在各个节点上的表——一开始大家会看到，数据全量在各个节点上，就相当于是没有分布式的特性，没有水平拆分的特性，但其实这种表，我们一般会用在数据量比较小、改动比较少的一些配置表中，通过数据的冗余来保证后续访问，特别是在操作的时候能够尽量避免跨节点的数据交互。其他方面，shardkey 来说，我们会根据 user 做一个 Hash，这个好处是我们的数据会比较均衡地分布在各个节点上，来保证数据不会有热点。

3.2 设计原理：扩容中的高可用和高可靠性

刚才也提到，因为整个扩容过程的流程会比较复杂，那么整个扩容过程能否保证高可用或者高可靠性，以及对业务的感知是怎么样的，TDSQL 是如何实现的呢?

数据同步

第一步是数据同步阶段。假设我们现在有两个 Set，然后我们发现其中一个 SET 现在磁盘容量已经比较危险了，比如可能达到 80%以上了，这个时候要对它进行扩容，我们首先会新建一个实例，通过拷贝镜像，新建实例，新建同步关系。建立同步的过程对业务无感知，而这个过程是实时的同步。

数据校验

第二阶段，则是持续地追平数据，同时持续地进行数据校验。这个过程可能会持续一段时间，对于两个同步之间的延时差无限接近时——比如我们定一个 5 秒的阈值，当我们发现已经追到 5 秒之内时，这个时候我们会进入第三个阶段——路由更新阶段。

路由更新

路由更新阶段当中，首先我们会冻结写请求，这个时候如果业务有写过来的话，我们会拒掉，让业务过两秒钟再重试，这个时候对业务其实是有秒级的影响。但是这个时间会非常短，冻结写请求之后，第三个实例同步的时候很快就会发现数据全部追上来，并且校验也没问题，这个时候我们会修改路由，同时进行相关原子操作，在底层做到存储层分区屏蔽，这样就能保证 SQL 接入层在假如路由来不及更新的时数据也不会写错。因为底层做了改变，分区已经屏蔽了。这样就可以保证数据的一致性。路由一旦更新好以后，第三个 SET 就可以接收用户的请求，这个时候大家可以发现，第一个 SET 和第三个 SET 因为建立了同步，所以它们两个是拥有全量数

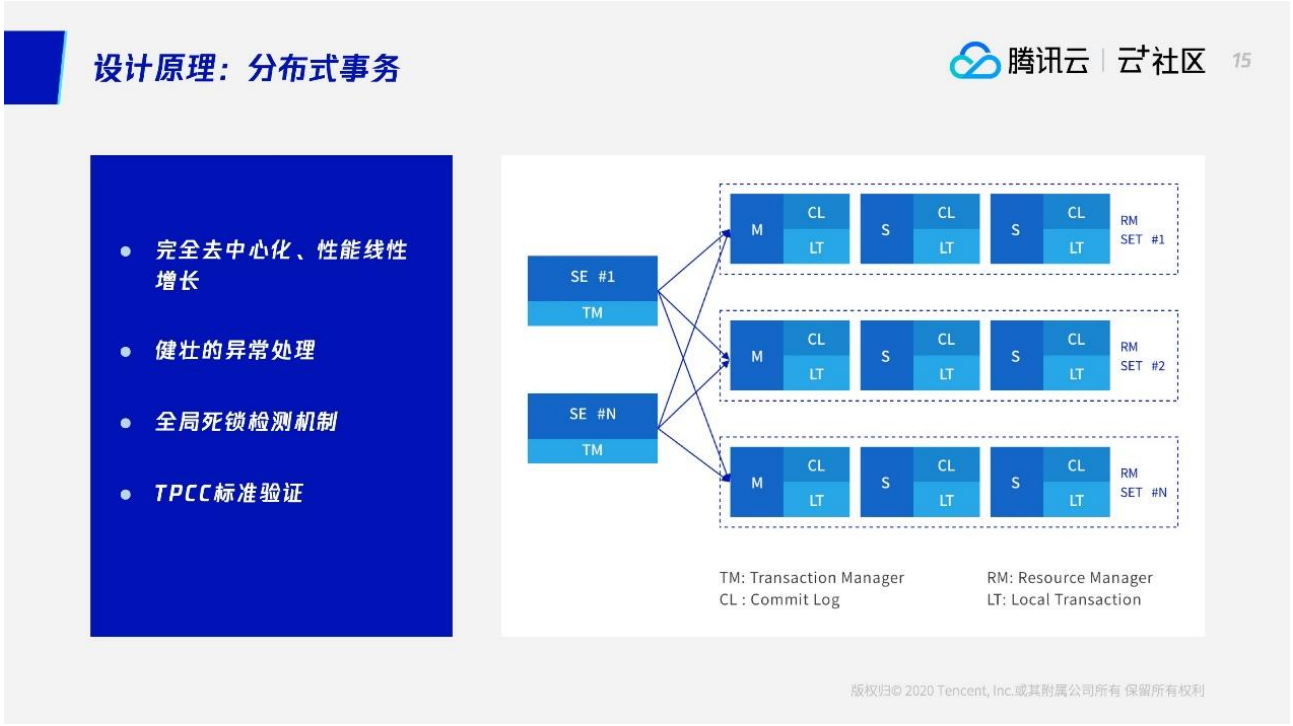
据的。

删除冗余数据

最后一步则需要把这些冗余数据删掉。删冗余数据用的是延迟删除，保证删除过程中可以慢慢删，也不会造成比较大的 IO 波动，影响现网的业务。整个删除过程中，我们做了分区屏蔽，同时会在 SQL 引擎层会做 SQL 的改写，来保证当我们在底层虽然有冗余数据，但用户来查的时候即使是一个全扫描，我们也能保证不会多一些数据。

可以看到整个扩容流程，数据同步，还有校验和删除冗余这几个阶段，时间耗费相对来说会比较长，因为要建同步的话，如果数据量比较大，整个拷贝镜像或者是追 binlog 这段时间相对比较长。但是这几个阶段对业务其实没有任何影响，业务根本就没感知到现在新加了一个同步关系。那么假如在建立同步关系时发现有问题，或者新建备机时出问题了，也完全可以再换一个备机，或者是经过重试，这个对业务没有影响。路由更新阶段，理论上对业务写请求难以避免会造成秒级的影响，但我们会将这个影响时间窗口期控制在非常短，因为本身冻结写请求是需要保证同步已经在 5 秒之内这样一个比较小的阈值，同步到这个阶段以后，我们才能发起路由更新操作。同时，我们对存储层做了分区屏蔽来保证多个模块之间，如果有更新不同时也不会有数据错乱的问题。这是一个我们如何保证扩容中的高可用跟高可靠性的，整个扩容对业务影响非常小的原理过程。

3.3 设计原理：分布式事务



刚才讲的是扩容阶段大概的流程，以及 TDSQL 是如何解决问题的。接下来我们再看扩容完成以后，如何解决刚才说的水平扩容以后带来的问题。首先是分布式事务。

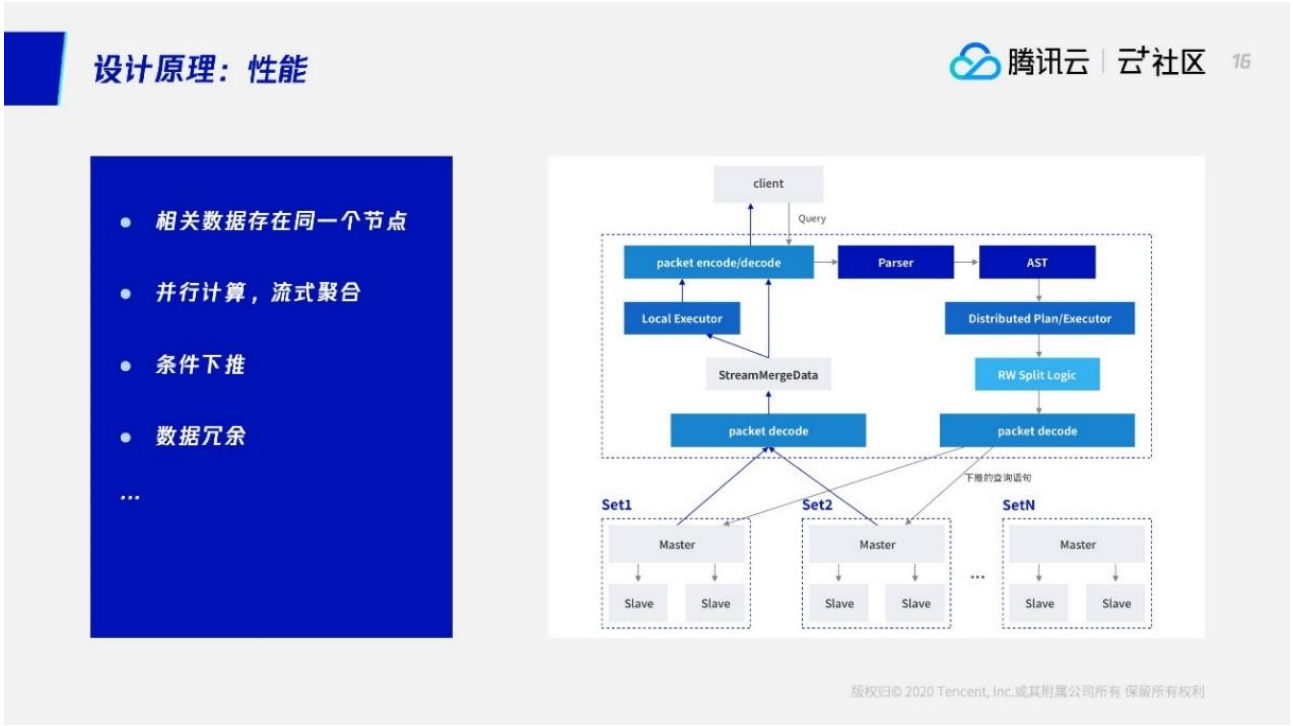
原子性、去中心化、性能线性增长

扩容以后，数据是跨节点了，系统本来只有一个节点，现在跨节点的话，如何保证数据的原子性，这个我们基于两阶段提交，然后实现了分布式事务。整个处理逻辑对业务来说是完全屏蔽了背后的复杂性，对业务来说使用分布式数据库就跟使用单机 MySQL 一样。如果业务这条 SQL 只访问一个节点，那用普通的事务就可以；如果发现用户的一条 SQL 或者一个事务操作了多个节点，我们会用两阶段提交。到最后会通过记日志来保证整个分布式事务的原子性。同时我们对整个分布式事务在实现过程中做到完全去中心化，可以通过多个 SQL 来做 TM，性能也可实现线性增长。除此之外，我们也做了大量的各种各样的异常验证机制，有非常健壮的异常处理和全局的试错机制，并且我们也通过了 TPCC 的标准验证。

3.4 设计原理：如何实现扩容中性能线性增长

对于水平扩容来说，数据拆分到多个节点后主要带来两个问题：一个是刚才说事务原子性的问题，这个通过分布式事务来解决；还有一个就是性能。

垂直扩容中一般是通过更换更好的 CPU 或者类似的方法，来实现性能线性增加。水平扩容而言，因为数据拆分到多个节点上去，如何才能很好地利用起拆分下去的各个节点，进行并行计算，，真正把水平分布式数据库的优势发挥出来，需要大量的操作、大量的优化措施。TDSQL 做了这样一些优化措施。

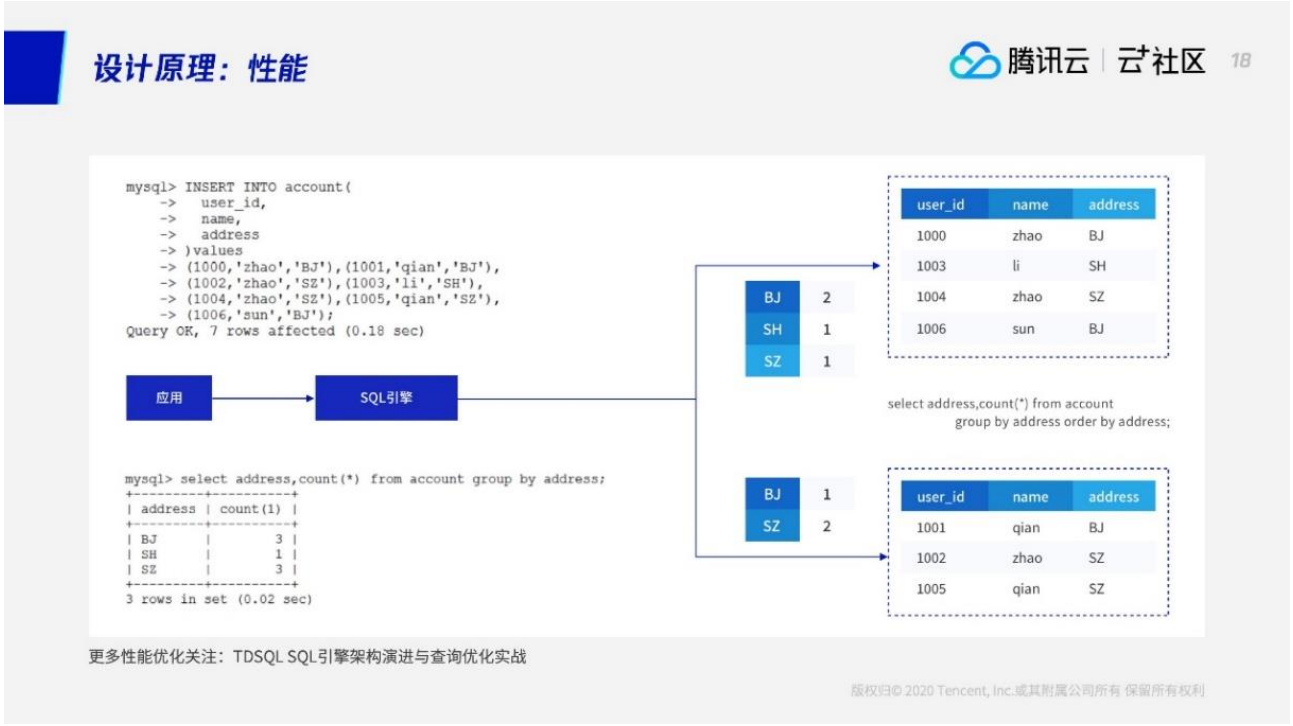


一是相关数据存在同一个节点上。建表结构的时候，我们希望业务能参与进来一部分，在设计表结构的时候指定相关的一些键作为 shardkey，这样我们就能保证后端的相关数据是在一个节点上的。如果对这些数据进行联合查询就不需要跨节点。

同样，我们通过并行计算、流式聚合来实现性能提升——我们把 SQL 拆分分发到各个后台的节点，然后通过每个节点并行计算，计算好以后再通过 SQL 引擎来做二次聚合，然后返回给用户。而为了减少从后端把数据拉到 SQL，减少数据的一个拉取的话，我们会做一些下推的查询——把更多的条件下推到 DB 上。此外我们也做了数据冗余，通过数据冗余保证尽量减少跨节点的数据交互。

我们简单看一个聚合——TDSQL 是如何做到水平扩容以后，对业务基本无感知，使用方式跟使用单机 MySQL 一样的。对业务来说，假设有 7 条数据，业务不用管这个表具体数据是存在一个节点还是多个节点，只需要插 7 条数据。系统会根据传过来的 SQL 进行语法解析，并自动把这条数据进行改写。

7 条数据的话，系统会根据分区键计算，发现这 4 个要发到第一个节点，另外 3 个发到第二个节点，然后进行改写，改写好之后插入这些数据。对用户来说，就是执行了这么一条，但是跨节点了，我们这边会用到两阶段提交，从而变成多条 SQL，进而保证一旦有问题两边会同时回滚。



数据插录完以后，用户如果要做一些查询——事实上用户不知道数据是拆分的，对他来说就是一个完整的表，他用类似聚合函数等进行查询。同样，这条 SQL 也会进行改写，系统会把这条 SQL 发到两个节点上，同时加一些平均函数，进行相应的转换。到了各个节点，系统会先做数据聚合，到这边再一次做聚合。增加这个步骤的好处是，这边过来的话，我们可以通过做一个聚合，相当于在这里不需要缓存太多的数据，并且做到一个流式计算，避免出现一次性消耗太多内存的情况。

对于比较复杂的一些 SQL，比如多表或者是更多的子查询，大家有兴趣的话可以关注我们后面的分享——SQL 引擎架构和引擎查询实战。

以上第三章我们比较详细地介绍了 TDSQL 整个水平扩容的一些原理，比如数据如何进行拆分，水平扩容实践，以及如何解决扩容过程中的问题，同样也介绍水平扩容以后带来的一些问题，

TDSQL 是如何解决的。

4 水平扩容实践案例

第四章，我们简单来介绍一些实践和案例。

4.1 实践：如何选择分区键

刚才我们说，我们希望在创建表的时候业务参与进行表结构设计的时候，能考虑一下分区键的选择。如何选择分区键呢？这里根据几种类型来简单介绍一下。

如果是面向用户的互联网应用，我们可以用用户对应的字段，比如用户 ID，来做分区键。这样保证在拥有大量用户时，可以根据用户 ID 将数据拆分到各个后端节点。

游戏类应用，业务的逻辑主体是玩家，我们可以通过玩家对应的字段；电商应用的话，可以根据买家或者卖家的一些字段来作为分区键。物联网的则可以通过比如设备的 ID 作为分区键。选择分区键总体来说就是要做到对于数据能比较好地做进行拆分，避免最后出现漏点。也就是说，通过这个分区键选择这个字段，可以让数据比较均衡地分散到各个节点。访问方面，当有比较多 SQL 请求的时候，其实是带有分区键条件的。因为只有在这种情况下，才能更好地发挥分布式的优势——如果是条件里面带分区键，那这条 SQL 可以直接录入到某一个节点上；如果没有带分区键，就意味着需要把这条 SQL 发到后端所有节点上。

这个大家可以看到，如果水平扩容到更多——从一个节点扩到 256 个节点，那某一条 SQL 写不好的话，可能需要做 256 个节点全部的数据的聚合，这时性能就不会很好。

总结来说，我们希望业务在创建表，在设计表结构的时候尽量参与进来。因为不管是聚合函数或者是各种事务的操作，其实对业务基本上属于无感知，而业务这时参与则意味着能够换来很大的性能提升。

4.2 实践：什么时候扩容？

我们什么时候扩容？在 TDSQL 里面，我们会有大量的监控数据，对于各个模块我们在本地会

监控整个系统的运行状态，机器上也会有各种日志上报信息。基于这些信息，我们可以决定什么时候进行扩容。

简单来说，比如磁盘——如果发现数据磁盘使用率太高，这个时候可以进行扩容；或者 SQL 请求，或者 CPU 使用率接近 100%了——目前基本假如达到 80%使用率就要进行扩容。还有一种情况是，可能现在这个时候其实请求量比较少，资源使用比较充足，但如果业务提前告诉你，某个时候将进行一个活动，这个活动到时候请求量会增长好几倍，这个时候我们也可以提前完成扩容。

下面再看几个云上的集群案例。这个大家看到，这个集群有 4 个 SET，每个 SET 负责一部分的 shardkey，这个路由信息是 0-127，意思是它最后能扩到 128 个节点，所以能扩 128 倍。

这个“128”可以由初始化的业务预估。因为如果池子太大的话，的确最后可以扩到几千台，但是数据将比较散了。事实上今天每台机器性能已经非常好，不需要几千台的规格。

这是另外一个集群——它的节点数会多一点，有 8 个节点，每个节点也负责一部分的路由信息。这个数字只有 64，所以这个最后可以扩到 64 个节点。这个是云上的相关例子。

Q&A：

Q：没扩容之前的 SET 里面的表都是分区表，问一下是不是分区表？

A：是的，在没扩容之前，相当于在这个，简单说我们现在就一个节点，那么我们告诉他 256，这个值我们在进行初始化的时候就定下来的。而且这个值集群初始化以后就不会再变了。假设我们这个集群定了一个值是 256——因为他可能认为这个数据量后面会非常非常大，可以定 256。这个时候，数据都在一个节点上。这个时候用户，按照我们刚才的语法创建了一个表，这个表在底层其实是分成 256 份的。所以他即使没有进行扩容，它的数据是 256 份。再创建另外一个表，也是 256 份。用户可能创建两个表，但是每个表的底层我们有 256 个分区的，扩容就相当于分区把它迁到其他的地方去。

Q：各个节点的备份文件做恢复时如何保证彼此之间的一致性？

A：各个节点之间没有相互关系，各个节点自己负责一部分的路由号段，只存储部分数据，水平扩容只负责一部分数据，它们之间的备份其实是没有相互的关系，所以这个备份其实是之间不相关的。每个节点我们可能有一主两备，这个其实是我们有强同步机制，在复制的时候来保证数据强一致性。大家可以参考之前的分享里面会比较详细介绍 TDSQL 在单个节点里面 TDSQL 一主多备架构是如何保证数据的强一致性的。

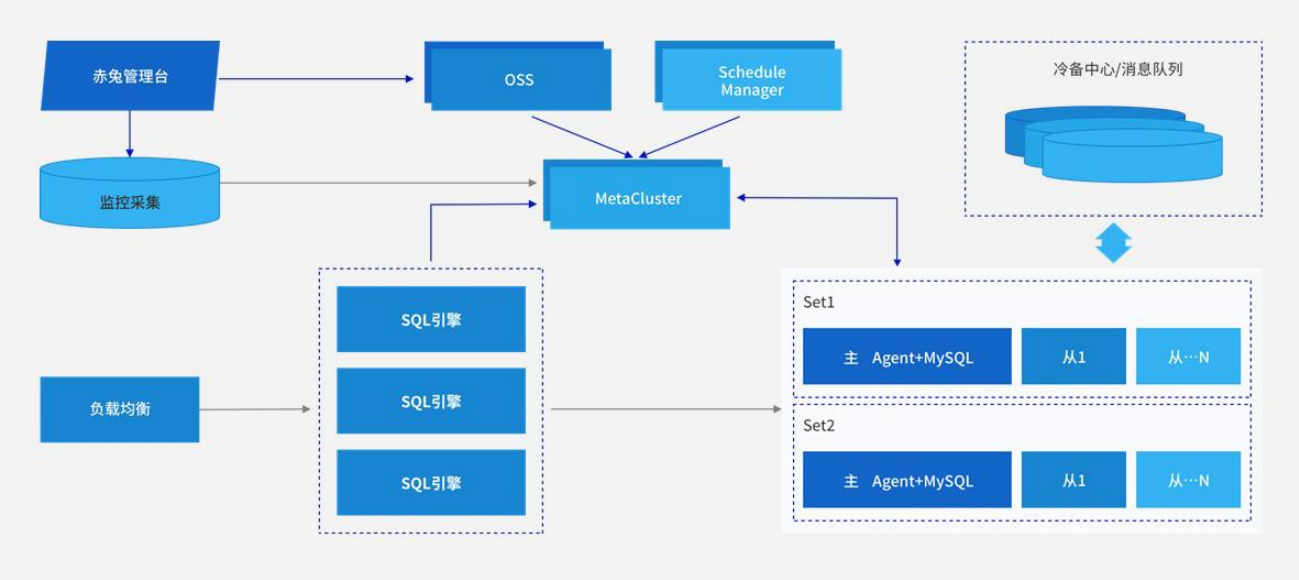
Q：两阶段在协调的时候能避免单点故障吗？

A：首先在两阶段提交的时候，我们是用 SQL 引擎做事务的协调，这个是单个的事务。如果其他的连接发过来，可以拿其他的 SQL 引擎做事务协调期。而且每个 SQL 引擎是做到无状态的，可以进行水平扩展。所以这个其实是不会有太多的故障，我们可以根据性能随机扩展的，可以做到性能的线性增长，没有中心化。日志这些都是被打散的，记日志也会记到 TDSQL 后端的数据节点里面，一主多备，内部保证强一致性，不会有单点故障。

第四章：亿级并发丝毫不虚，TDSQL-SQL 引擎是如何炼成的？

本章节内容分为四个部分，分别是：TDSQL 简介、SQL 引擎简介、SQL 引擎查询处理和最佳实践。

1 TDSQL 分库分表策略



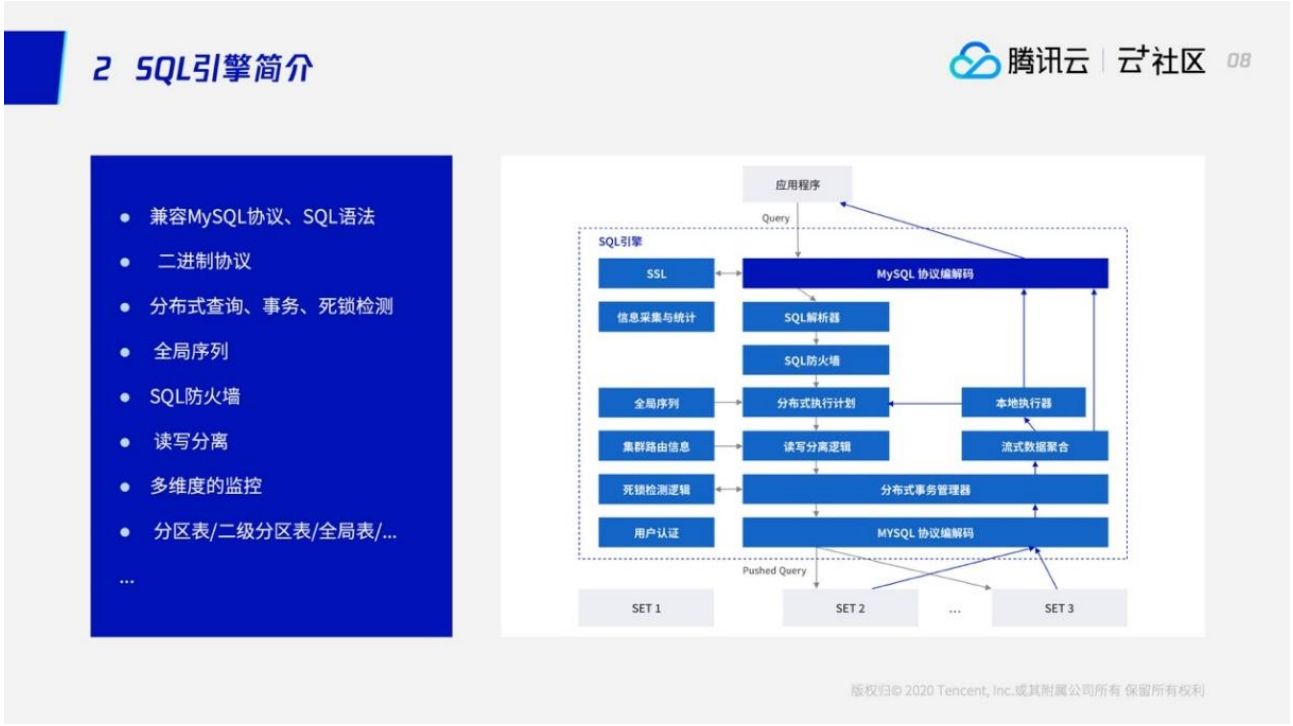
TDSQL 是 Shared-Nothing 架构的分布式数据库，这张图是 TDSQL 的核心架构图，在这个架构图中包含了三个重要组件，分别是 SQL 引擎、MetaCluster 和后端 SET。其中每个 SET 是由一主多备构成的高可用复制单元，在一个 TDSQL 中往往包含多个这样的 SET。每一个 SET 负责存储分布式表的部分分片信息。SQL 引擎负责处理业务发过来的 SQL 请求，然后对 SQL 进行拆分，发送给各个 SET 进行处理，并对每个 SET 返回的结果进行聚合，得到最终的结果。

在 TDSQL 中，每个 SET 都会负责一段连续的哈希；在 TDSQL 中建立的每一个分布式表，SQL 引擎都要求用户指定其中的一个列为分区键，SQL 引擎通过计算这个分区键的哈希值，将这个表的每一行数据都映射到这个哈希空间，由对应的 SET 进行存储。哈希空间与 SET 之间的对应关系，我们叫做路由表。这个路由信息会存储在 MetaCluster 里。当对整个集群进行缩容或扩容时，每个 SET 对应的哈希空间也会发生变化，对应的数据也会进行搬迁。SQL 引擎通过监听 MC 来感知集群的变化。

在建表的时候，SQL 引擎通过关键字让用户在建表的时候指定其中的一个列为分区键。在这个例子中，用户建立一张表，其中指定 ID 为 shardkey，也就是说 SQL 引擎通过计算分区键 ID 的哈希值，将这个表的数据进行打散，均匀的分布在各个 SET 上。

2 TDSQL-SQL 引擎：如何优雅处理海量 SQL 逻辑

前面我们回顾了一下 TDSQL 的整体架构，以及 Sharding 策略。这里我们再介绍一下 SQL 引擎。



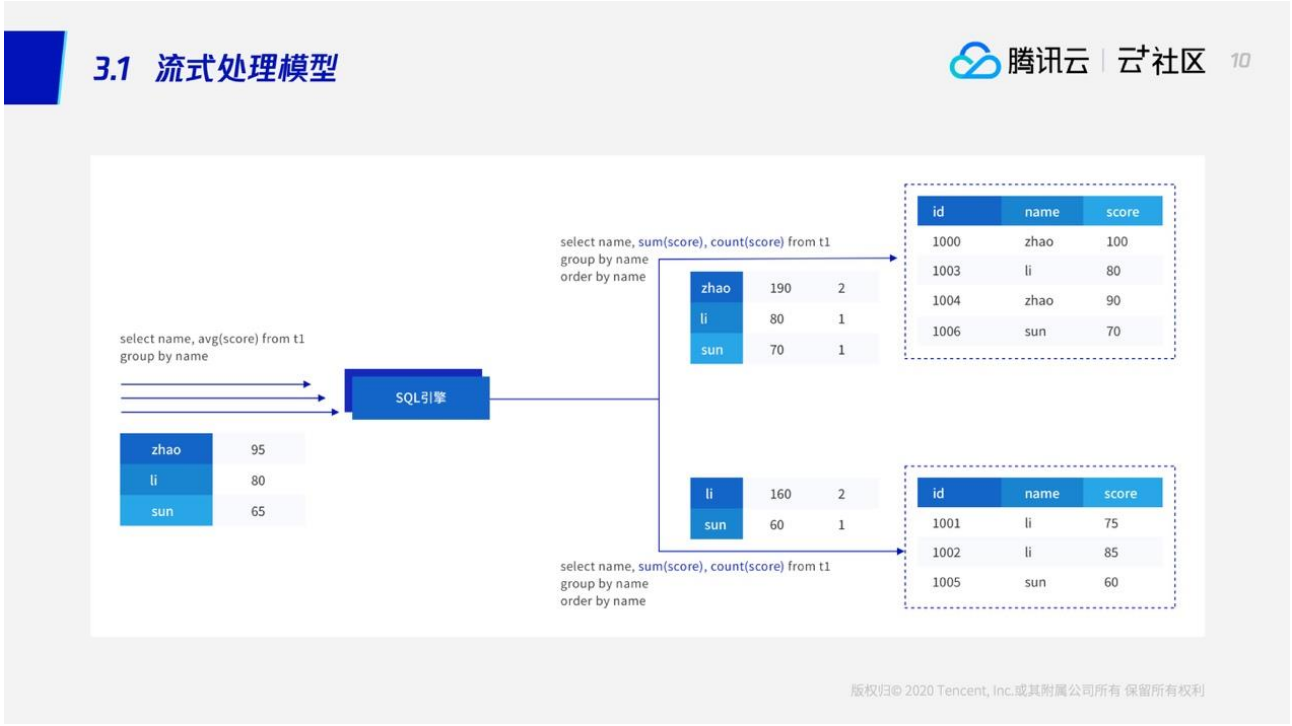
目前 TDSQL-SQL 引擎已经能够支持绝大多数的 MySQL 语法，分布式查询、事务和死锁检测。应用程序端可以像使用单机数据库一样，通过 SQL 引擎来使用整个 TDSQL。除了这些比较基本的功能，SQL 引擎还支持一些比较高级的功能，例如 SQL 防火墙、读写分离等等。我们这里通过一条 SQL 的大体执行路径来看一下这些功能之间的关系：应用程序通过 MySQL 客户端向 SQL 引擎发送了一条 SQL，SQL 引擎通过协议解析，从数据包中得到这条 SQL，并对这条 SQL 进行语法解析，语法解析以后我们就得到一棵抽象的语法树，如果应用配置了 SQL 防火墙，我们还会匹配防火墙的规则，判断这条 SQL 能否执行。如果能够执行的话，接着我们会构建这条 SQL 的分布式执行计划——在这个计划里面，我们描述了由哪些 SET 参与查询的执行，每个 SET 应该执行什么样的逻辑，SQL 引擎又进行怎样的聚合，两者之间怎么进行协调。得到分布式执行计划以后，我们将这个计划再转换成 SQL 的形式发送给对应的 SET 执行。接着我们执行读写分离算法，从每个 SET 中挑选出最适合访问的实例，然后就可以向这个实例发送 SQL 了。我们收到每个 SET 反馈的响应以后，对返回的结果进行聚合。这里的聚合逻辑

主要是执行分布式执行计划分配给 SQL 引擎的一些任务。如果是一条简单的查询，我们通过流式聚合就可以得出最终的结果。并将结果反馈给应用；如果是一条比较复杂的查询，往往需要拆分成多个阶段来执行。这个时候就需要构建下一个阶段的分布式执行计划，然后再执行这个计划。经过多次执行，所有的阶段全部执行完毕之后，最终在本地的执行器中进行最终的计算和聚合，从而得到一个最终的结果。

3 TDSQL-SQL 引擎查询处理模型：如何实现高性能 SQL 引擎查询

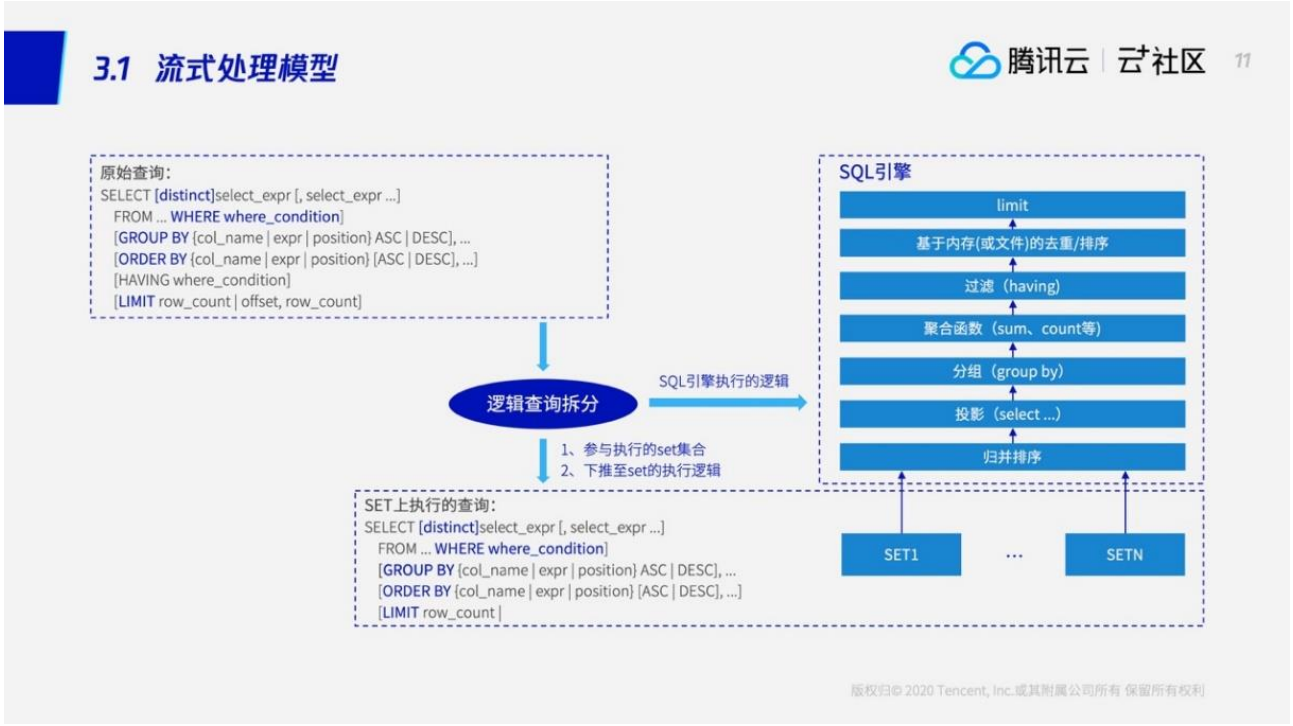
我们这里已经介绍了 SQL 引擎的一个大体的功能，这里我们再具体介绍 SQL 引擎是怎么处理各类查询的。我们将查询分成两类，一类是 Select 查询，一类是更新操作，例如 Delete、Update、Insert 等等。对 Select 查询的话，我们有两种处理方式：一种是流式处理模型，一种是通用处理模型。流式处理模型主要是处理单表上的查询，而通用处理模型是对流式处理模型的弥补，负责处理分布式的跨节点查询。

3.1 流式处理模型原理及典型案例详解



我们先介绍一下流式处理模型。这里有一个简单的例子。

在这个例子里面用户访问的是一个日志表，这个日志表里面有三个列：其中 ID 是日志的 ID，name 是用户名，score 是用户的历史游戏得分。一个用户会有多个游戏得分，并且他的游戏得分的记录会分散在各个 SET 上。业务想要获取每个用户的平均分，SQL 引擎不能直接把 SQL 发送给各个 SET，所以他需要计算出每个用户的总得分和这个用户在这张表上的一个总记录数，然后用两者的商来求得每个用户的平均分——具体来说 SQL 引擎要求每个 SET 返回，每个用户在每个 SET 上的一个局部的总得分和总记录数。接着 SQL 引擎将每个 SET 上返回的结果再进行聚合，得到每个用户的总得分和总记录数，然后再用两者相除。

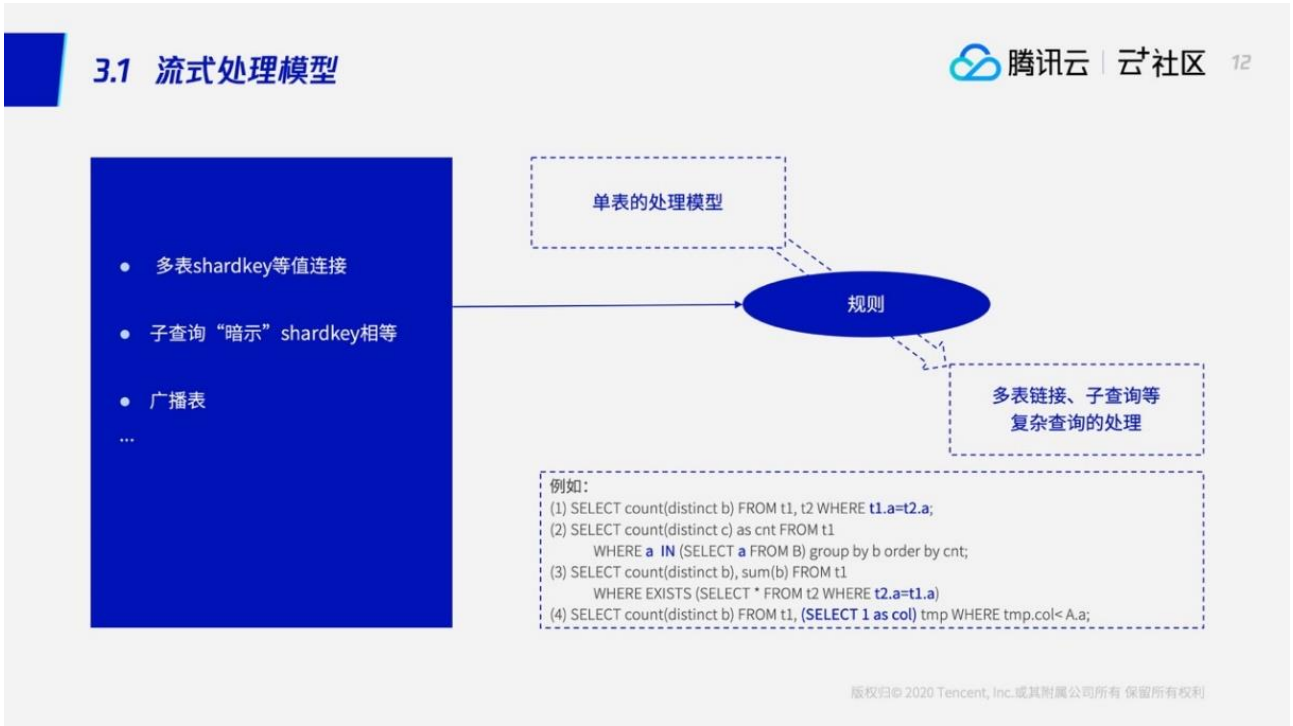


我们发现，SET 执行的 SQL 里面有一个 order by，这个是做什么的呢。我们假设没有这个 order by，那么某一个用户张三，SET1 反馈的第一条记录是张三的局部得分，SQL 引擎拿到这个张三的结果以后，无法直接计算出张三这个用户的平均分，他还需要知道张三在 SET2 上的得分。但是 SET2 可能在最后一条记录才返回张三在 SET2 的局部的得分，所以 SQL 引擎不得不进行等待操作，等待 SET2 返回所有的结果，而这个等待操作会非常影响性能。如果我们增加了这个 order by，SET1 和 SET2 返回的结果都是按名字有序地进行，SQL 引擎进行一个简单的 Merge 就可以得到一个全局有序的结果。对张三这个用户来说，这个有序的结果里面，他在 SET1 和 SET2 上的局部得分就是相邻的，所以 SQL 引擎不需要等待 SET1 和 SET2 反馈的所有结果，他直接就能计算出张三的平均分。这个引擎计算出张三的平均分以后，立马就可以将张三的得分再反馈给业务。

在这个例子中可以看到，SET1 和 SET2 不停地向 SQL 引擎返回数据，SQL 引擎随即对返回的

结果进行聚合，并将聚合结果立即返回给这里。整个过程就像流水一样，所以我们将这个过程称之为流式处理模型。流式处理模型最核心的地方就是对 SQL 进行拆分——拆分成两部分，一个是 SET 执行的部分，以及在 SQL 引擎执行的部分。就拿刚刚的例子来说，SET 进行一个局部的聚合，计算出每个用户在 SET 上的局部的总得分和总记录数。SQL 引擎进行一个全区的累加和聚合，得到每个用户的总得分和总记录数。进行拆分以后，流式处理模型还会添加一些额外的操作，将两者衔接起来——例如刚刚例子中的 order by，这样一来我们就得到了一个流式处理的执行计划。当然除了这个以外，我们还会进行一些优化，将原本分配给 SQL 引擎的一些操作进行下推，例如模型中的 Limit 和 distinct，其实它应该在 SQL 引擎上执行的。我们通过把这些下推，能够较大地减少 SET 返回给 SQL 引擎的数据量，从而也就提升了整个执行的性能。

刚才我们一直强调流式处理模型是针对单表处理的，事实上它还能够处理其他的场景。



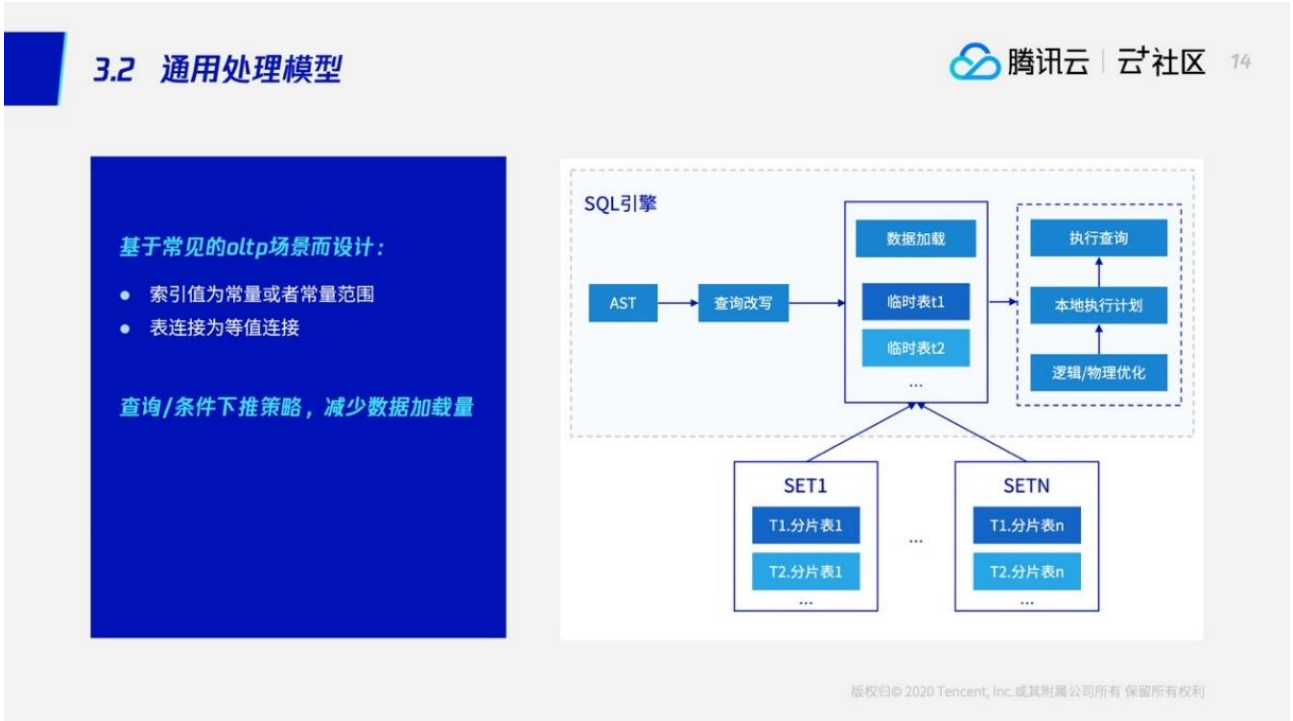
例如这个例子 1 中的表，两张表的 shardkey 相等，因此我们可以把 t1、t2 当成一个整体，当成一个整体之后我们就可以用刚刚介绍的查询拆分构建执行计划，并用流式处理模型进行处理了。对子查询的话，类似这里的 IN，a 是 T1 和 B1 的 shardkey，这个子查询同样暗含着 T1 的 shardkey 等于 B 的 shardkey，我们同样将 T1 和这个子查询当成一个整体，套用刚刚介绍的查询拆分和流式处理模型进行处理。当然，并不是所有的查询都能够满足这样的一个情况，如果两个表不是 shardkey 相等的话，这个时候我们就需要用后面的通用处理模型进行处理了。但是通用处理模型的性能比不上流式处理模型，那怎样才能够将流式处理模型进行进一步推广呢？TDSQL 就引入了一个广播表——广播表的意思就是说将一些表的数据全量备份在每个节点上，这样一来原本 T1、T2 的关联查询，他们必须用通用处理模型进行处理，这个时候我们也可以使用流式处理模型进行处理了。

从刚刚的介绍，我们可以看到流式处理模型是以 pipeline 的方式执行，SET 不断返回数据给

SQL 引擎，SQL 引擎不停地对返回的结果进行聚合，并将聚合的结果立即返回给应用。所以说在这种处理方式下，SQL 引擎的内存开销是非常低的。同时在执行的过程中，各个 SET 是并发执行，这也保证了流式处理模型的性能、并行性。流式处理模型最核心的一方就是拆分和下推——通过智能的拆分和下推，我们将大量的计算下推给 SET 执行，这样就大大减少了 SQL 引擎需要从 SET 获取的数据。流式处理模型虽然要求比较严苛，但是我们通过合理的选择 shardkey 以及引入广播表，事实证明流式处理模型其实能够满足绝大多数的 OLTP 业务。

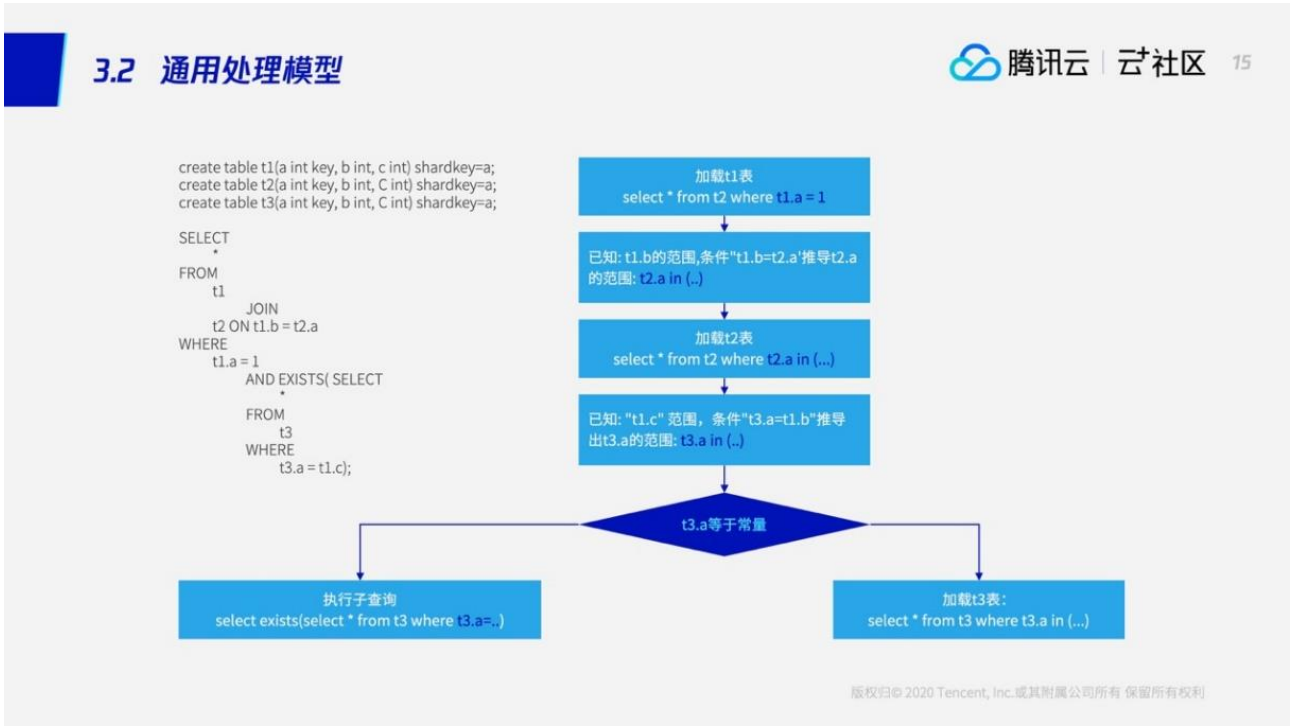
3.2 通用处理模型：跨节点分布式查询优化的利器

接下来我们介绍一下 SQL 引擎的通用处理模型。



通用处理模型主要是针对分布式的跨节点查询。说到分布式的跨节点查询，大家首先想到的肯定是 Spark、Greenplum 这样的大数据处理引擎。在这些引擎中，他们执行查询的时候往往需要在节点之间进行大规模的数据搬迁，因为 TDSQL 主要针对 OLTP 业务，所以在生产上，TDSQL 往往运行着大量的交易 SQL，进行大规模数据搬迁肯定是不现实的。所以，TDSQL 基于常见的 OLTP 场景而设计了自己的分布式处理查询框架。在这个模型下，SQL 引擎对 SQL 进行语法解析以后，就将实际参与查询的数据加载到 SQL 引擎，然后在本地执行这个查询。在我们所设想的 OLTP 场景下，用户常常为一个表的索引指定一个常量或者常量的范围，表之间的连接也基本上为等值连接。在这样的一个场景下，SQL 引擎通过查询下推和条件下推大大降低了需要加载的数据量，从而使得这个方案变得切实可行。

这里我们以一个例子来讲解通用数据模型是怎样处理一条 SQL 的。



在这个例子中一共有三张表，每个表中的 shardkey 我们都设置为 A。SQL 先将 T1、T2 进行一个等值连接，然后 where 条件指定了 T1.A 是一个常量；在这里还有一个子查询——这个子查询是一个相关子查询，它引用了外层的 T1.C。在执行这条查询的时候，SQL 引擎首先选择一个需要加载的表，在这里我们选择了 T1.A，因为它包含了一个可以下推条件，而且条件是一个主键。我们加载完 T1 以后，就可以知道 T1.B 和 T1.C 是两个列的范围了，根据这个连接的连接条件，我们可以推算出 T2.A 的范围。根据推算出来的 T2.A 的范围我们继续加载 T2。这样一来，就可以将上层涉及的两个表 T1、T2 都加载到本地。接着我们处理内层的子查询，这个子查询是一个相关子查询，如果我们在加载 T2 的时候发现 T1.C 其实是一个常量，这个时候就可以将这个相关子查询转换成一个非相关子查询，并将这个子查询提取出来进行独立计算。计算以后我们将它的结果来替换这个子查询；如果它不是一个常量的话，我们就利用这个条件推断出 T3.A 的范围，再用推算出来的这个条件去加载 T3，这样一来，这条 SQL 所设计的所有的表的数据，我们都加载到了 SQL 引擎，接下来就像 MySQL 一样执行这条查询就可以了。

通过这个例子我们可以看到，SQL 引擎通过条件下推，并利用加载的数据推算出新的条件，然后利用新的条件再去过滤我们需要加载的表，这样可以大大降低需要加载的数据量。对于子查询的话，如果我们发现它能够转换成非相关词查询，这个时候我们还会进行子查询的下推。当然具体优化的技术还有很多，这里我们就不再一一列举。

我们将优化和工程实践中所使用的一些技术进行一个分类，主要包括逻辑优化、条件下推和隔离。

- 逻辑优化：主要是对 SQL 的结构进行优化，使它变得更加容易处理。例如我们将左/右连接转换成内连接。如果它能够转换——在原来的逻辑下需要先加载左连接的左表，然后才能加载左连接的右表。最后转换以后，我们就可以消除这种限制，能够生成多种加载方式，这样的话我们就可以有更好的一个优化的空间。
- 条件下推：是指用一些手段将 SQL 中的条件提取出来，或者说我们利用已知的一些信息推断出新的条件，用这些推断出来的条件来过滤需要加载的数据。同时我们还可以通调整加载的顺序，例如刚刚例子中我们优先加载了 T1，利用优先加载小表的策略同样能够起到降低需要加载的数据量的作用。
- 隔离：TDSQL 主要针对 OLTP 业务，所以它线上会运行着大量的交易 SQL，这个时候如果应用发送过来的是一个复杂的查询、是一个 OLAP 类的查询的话，往往需要加载很大的数据量。而且执行的时候也会有很大的 CPU 开销，为了不影响这些正在运行的 OLTP 业务，我们需要进行隔离。比如说我们将比较耗时的操作，例如把数据加载过来，然后缓存在本

地的临时表里，以及最终要执行查询的时候，也将这个比较耗时的操作放在后台异步执行，这样就可以对 OLTP 业务进行隔离，避免影响在线的一个交易。

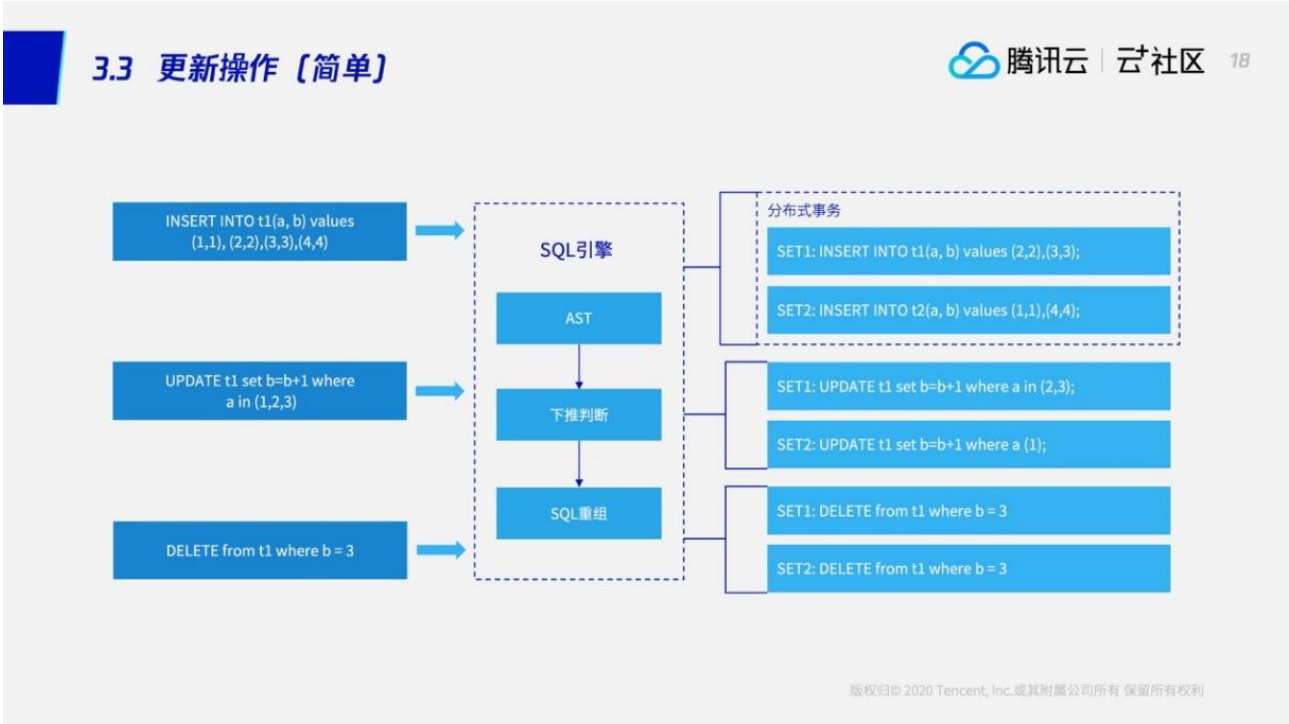
3.2.1 通用处理模型总结

通过前面我们可以看到通用处理模型的 SQL 兼容性是非常好的，因为我们只需要将数据加载到本地，然后用传统的方法执行这个查询就可以了。然后我们对 OLTP 场景有很多的优化策略，通过这些策略，我们大大降低了需要加载的数据量。同时，这个通用处理模型也可以作为 OLAP 来用，如果当做 OLAP 来用的话，我们会将数据加载和执行这两个比较耗时的动作在后台异步执行。

3.3 TDSQL-SQL 引擎更新操作原理

前面我们讲解了 SQL 引擎如何处理 Select 的查询。这里我们再简单介绍一下 SQL 引擎是如何处理更新操作的。

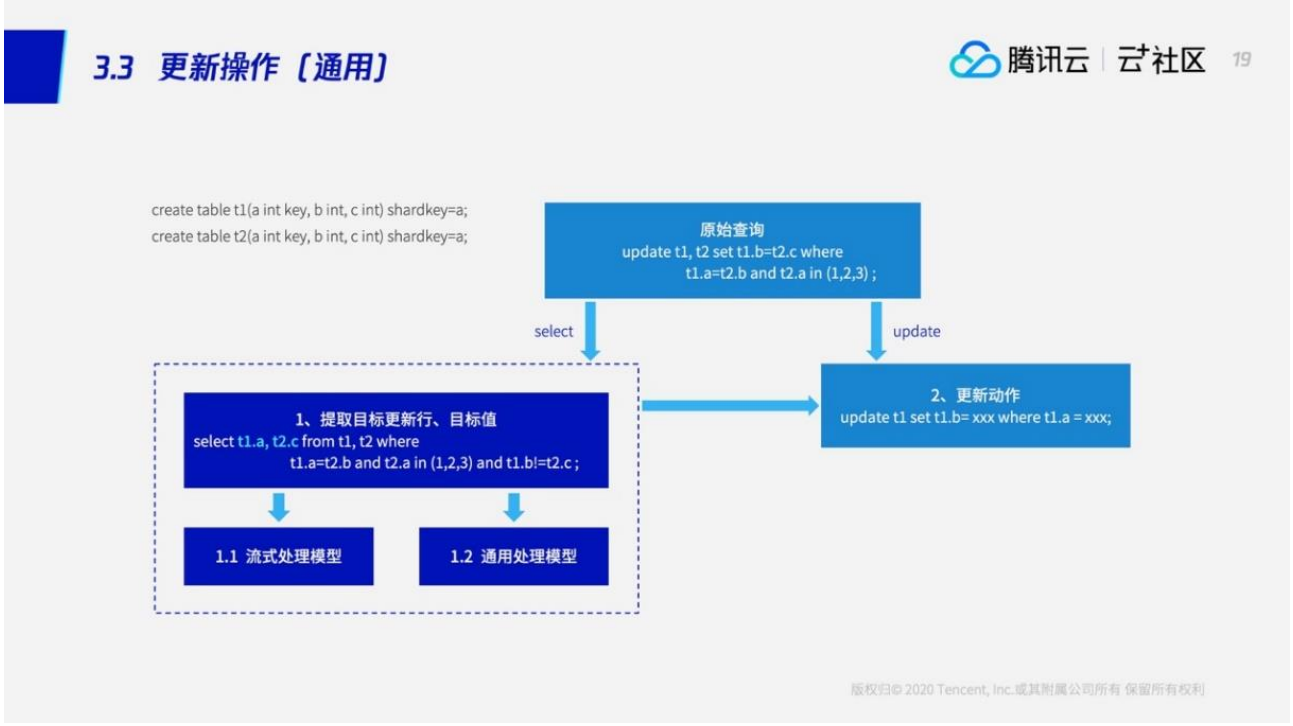
3.3.1 简单更新操作



对于一些简单的更新操作，例如这里的 Insert、Update 语句，我们根据 SQL 中的 shardkey 对这些 SQL 进行拆分。例如 INSERT，它插入这些数据，我们根据它的 shardkey 计算出 2、3 是需要插入到 SET1 的；1 和 4 是需要插入到 SET2 的。我们将这些需要插入的数据进行拆分以后，再构建对应的 Insert 语句分别发送给 SET1 和 SET2 进行执行。如果一条更新语句没有带

shardkey，例如这里的 Delete 语句，这个时候我们就需要将这条更新操作广播给所有的 SET 执行。如果一条更新语句更新了多个 SET，我们就会使用分布式事务来保证这个更新操作的原子性。

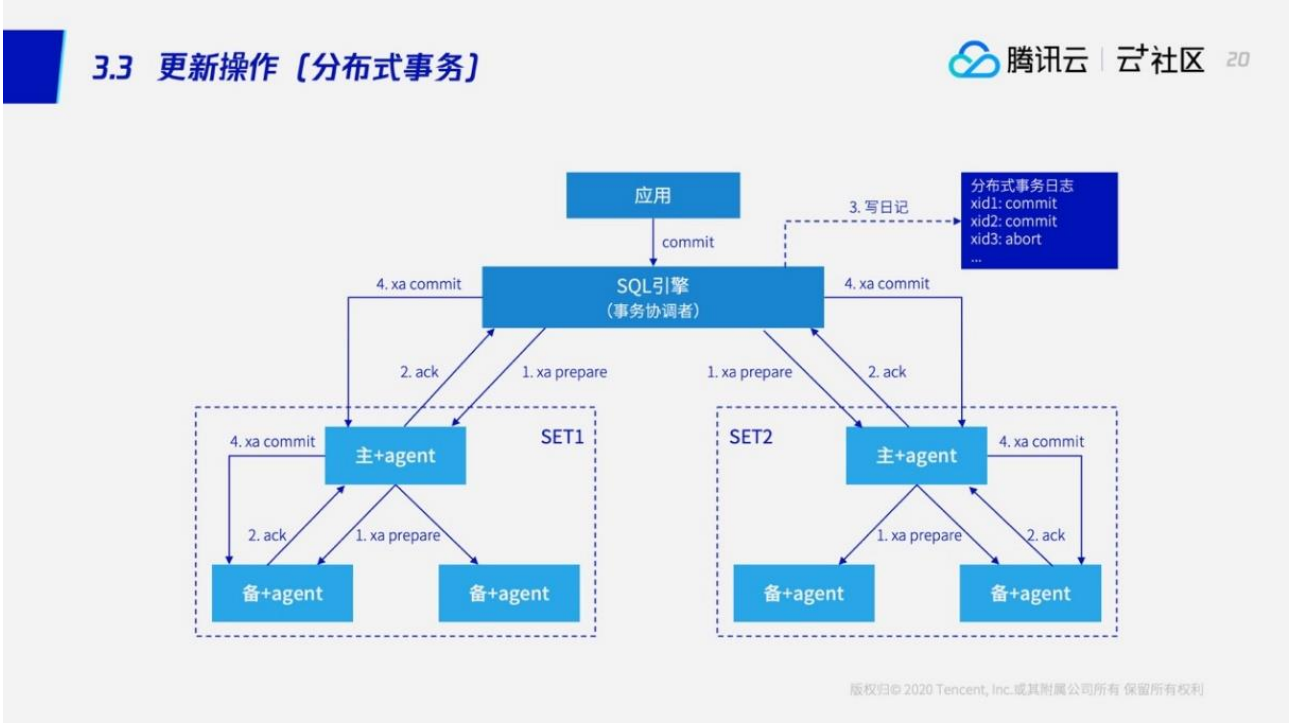
3.3.2 复杂更新操作



而对于一个比较复杂的更新操作，例如这个例子里面的联合更新 SQL。对于这样一个复杂的 SQL，SQL 引擎会将它拆分成两个部分：一部分是 Select，一部分是对应的更新部分。对于这个语句来说就是一条 Update。而 Select 的话，它主要的目的是提取出那些需要被更新行的主键，以及我们需要设置的新值。获取这两个关键信息以后，我们会构建对应的更新操作，因为这个例子里面就是一条 Update 语句——在这个语句里面我们指定了需要被更新行的主键，同

时我们也指定了需要被更新的列和它的新值；然后将这条 Update 语句发送给对应的 SET 执行。对于这条 Select 语句的话，我们就看它的类型，看它是否 shardkey 相等——如果 shardkey 相等的话，我们就将由流式处理模型先处理；如果不相等，我们就用通用处理模型先处理。

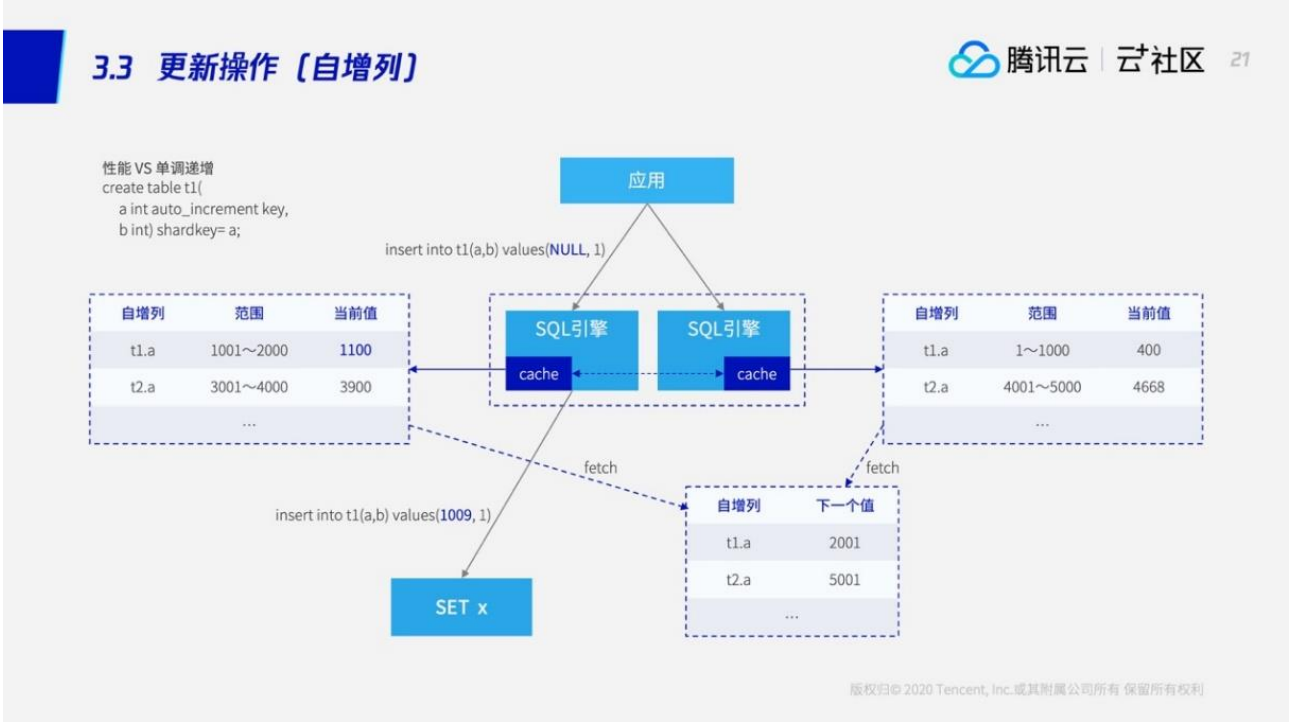
3.3.3 更新操作中的分布式事务



前面我们多次提到分布式事务，在前面的课程中，我的同事也对分布式事务做了一些讲解。这里我们再简要回顾一下。SQL 引擎分布式事务所使用的算法是两阶段提交，对应用来说其实不需要关心具体的实现细节，对应用来说就是简单的 Commit——当业务发送 Commit 到 SQL 引擎的时候，SQL 引擎就会向所有的参与者发送 prepare，接下来每个收到 prepare 的主机将这个事务的状态设置成 prepare 状态，并将这个事务所产生的 binlog 全部发送给备机。备机

收到以后就会再向主返回一个应答，主再向 SQL 引擎返回一个应答，SQL 引擎收到所有参与者的应答之后就会做一个决策。当所有的参与者都应答成功，它就会做一个提交的决策，并将这个决策写入到分布式的事务日志中。一旦写成功的话，我们就可以直接将这个事务提交；而当发生任何一种故障，我们都可以根据这个全局的分布式事务日志进行提交或者回滚。

3.3.4 更新操作自增列优化原理



使用数据库的过程中，我们经常会使用自增列。SQL 引擎也提供了这个功能，当应用创建表的时候，如果指定了一个自增列，SQL 引擎就向一个全局的元数据表中插入一个记录，这个记录包含了 shardkey 的名称，以及下一个可以使用的值。当应用后续再发送 INSERT 语句过来的时候，SQL 引擎就会对记录中这个值进行加 1，然后再用原来的值来填充这条 INSERT 语句，

再将这条填充后的 Insert 语句进行拆分并发送到对应的 SET 执行。如果对每一条 Insert 语句我们都去访问这一条记录，这一条记录必然会成为一个热点，继而对整个系统的稳定性和性能都会产生很大的影响。所以 SQL 引擎丢弃了自增的属性，只保留了它全区唯一的属性。SQL 引擎为了做到这一点，在每次获取自增列值的时候会一次性获取多个，然后再缓存到本地。当应用发送 Insert 语句过来，就直接从本地的 cache 中拿出一个值来填充这个 Insert ，避免热点的产生，从而也提升了 Insert 语句的性能。当然，如果用户非得要使用递增值，SQL 引擎还提供序列功能，通过序列来满足单调递增的属性。

4 TDQL-SQL 引擎查询最佳实践

前面我们介绍了 SQL 引擎是如何处理各类查询的，接下来是我们的最佳实践。在这一章节我们会介绍选择广播表的基本原则；SQL 引擎提供的基本命令等，通过这些命令我们可以看到一条 SQL 具体的执行细节。

4.1 广播表

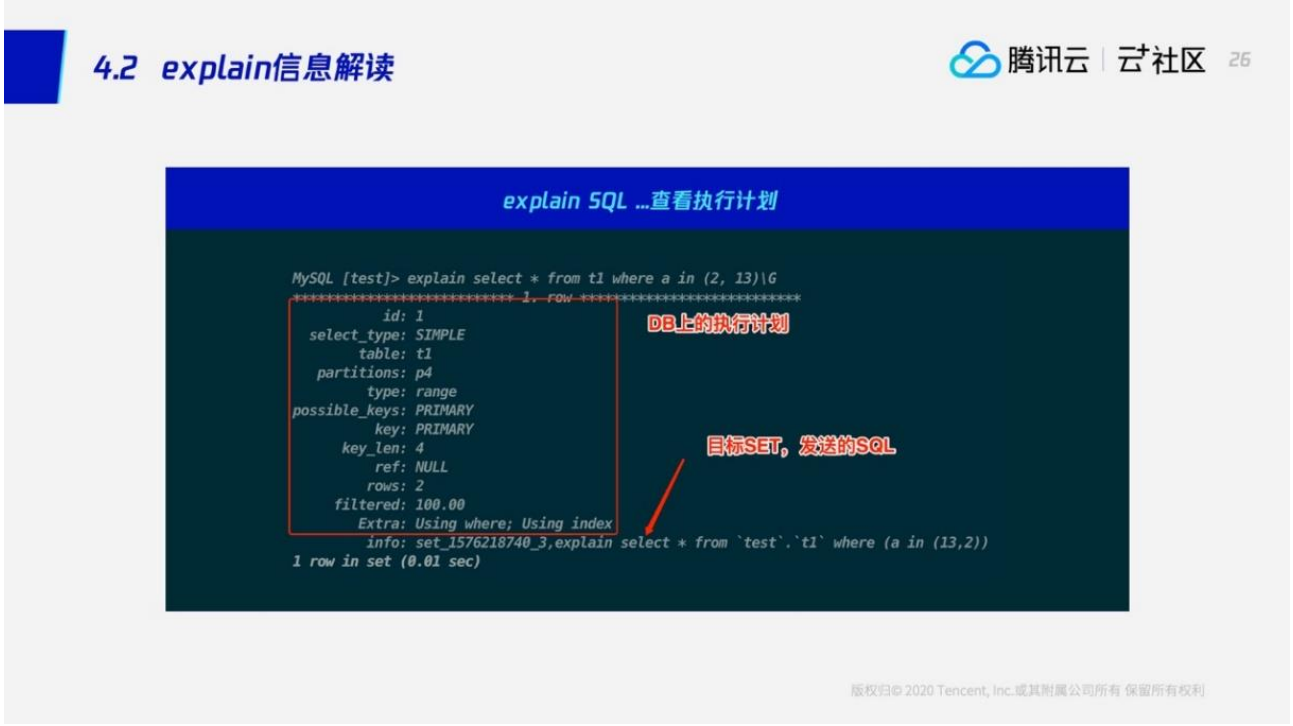
使用广播表可以让一条 SQL 使用流式处理的方式进行处理，流式处理的性能往往是比较高的。那么什么样的一个表适合作为广播表呢？

对于广播表，SQL 引擎需要将这个表全量备份到所有的 SET 上，这会产生空间上的消耗；同

时对广播表的每一次更新，都需要使用分布式事务，从而保证这个更新操作的一致性。因此，我们优先要选取的表，会希望它的数据量比较小，来减少对空间的占用；同时我们希望它的更新频率比较低，这样能够降低对更新操作性能的影响。我们使用广播表主要是为了优化连接查询，所以说我们选择的广播表本身，它应该经常被访问，当满足这两个条件，我们就可以建议用户将这样的一个表设置成广播表。

4.2 explain 信息解读

在使用 MySQL 的时候，我们经常使用的命令就是 explain，通过 explain 我们可以看到查询的执行计划，看到这条 SQL 是否使用了某个索引，或者说多表连接的顺序是否符合预期，SQL 引擎也提供了这样一个功能。



刚才我们说过一条 select 语句的执行方式有两种：一种是流式处理模型，一种是通用处理模型。对流式处理模型来说，SQL 引擎会拆分成两部分，一个是 SET 执行部分和 SQL 引擎执行部分。如果 SQL 引擎执行的逻辑比较简单，这个时候 explain 所产生的结果就是 SET 上的一个实际执行计划，例如对这条 SQL 来说，其实 SQL 引擎并没有做什么比较大的计算，所以它产生的结果其实是 MySQL 上的执行计划。SQL 引擎还会追加一些额外的信息，例如可以看到提示用户这条 SQL 是发给哪个 SET 的，实际发给 SET 的是怎样的一条 SQL。

如果 SQL 引擎对 SQL 在进行流式处理的过程中进行了比较复杂的聚合，例如——对这条 SQL 来说，SQL 引擎需要计算出 count（distinct　b），这个时候 explain 产生的结果就跟前面不一样，explain 会告诉用户 DB 上执行的是什么样的查询，SQL 引擎进行怎样的聚合。

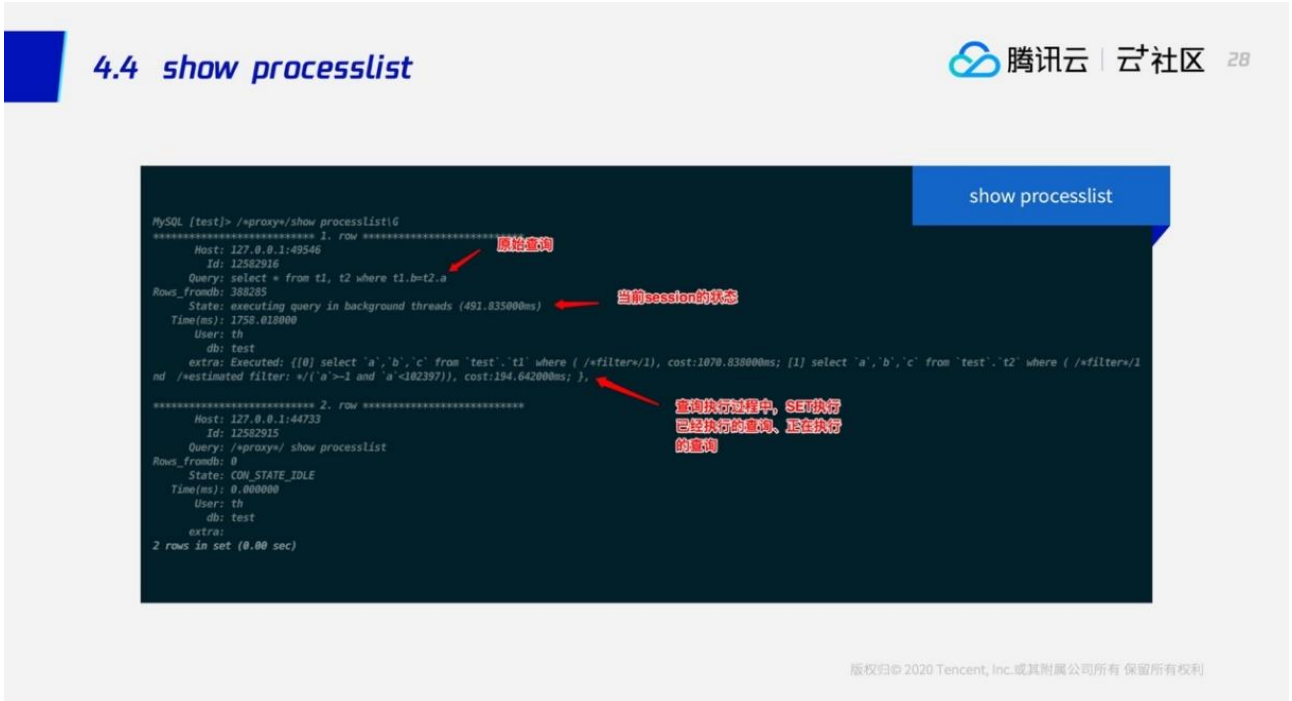
这个例子里面，我们可以看到 DB 上执行的是扫描查询，并且下推了 distinct，在 DB 部分进行了初步去重。如果想要知道下推部分的执行计划，我们还提供了透传命令——通过透传命令我们可以看到这条 SQL 在每个 SET 上的具体执行计划。图片显示，为了执行这个计划，每个 SET 使用的临时表，并且进行了排序。

对于使用通用处理模型处理的查询，explain 展示的信息又会不一样。前面我们说过在通用处理模型下，我们需要对表进行加载，因此 explain 输出的结果就是告诉我们，SQL 引擎到底使用了什么条件去加载数据。对于推导出来条件，explain 会添加一个前缀/*esteimated*/加以提示。如果子查询可以直接下推，explain 同样也会展示出来。在展示怎么样加载数据和怎么样下推查询之后，explain 会展示最终 SQL 引擎要执行的查询。

4.3 trace：展示 SQL 各个阶段的耗时

当然光知道一条 SQL 是怎么执行，有时候还是不够的的。因为影响一条 SQL 的执行效率可能还跟当前的环境、负载有关系。所以我们想知道一条 SQL 在实际执行过程中各个阶段的时耗，SQL 引擎就提供了这样一个强大的功能——trace。当你在 SQL 的前面加一个 trace 前缀的时候，SQL 引擎会执行这条查询并记录在执行过程中各个阶段的时耗。例如在这个例子里面，这条 SQL 是以使用通用处理模型进行处理的，所以在展示的结果里面，我们可以清晰地看到，加载 T1 时，从 SET1 和 SET2 加载数据的时耗，以及 SET1 和 SET2 返回的数据量。通过这些信息，我们就可以看到在数据加载过程中的具体细节。如果时耗比较大的话，我们就再去翻看前面的 EXPLAIN 信息，看看它的下推是否正常，下推之后是否能够使用到索引。通过这些信息我们就可以清楚的看到一条 SQL 在各个阶段所占用的时间，结合前面介绍的的 explain 命令，就可以很方便地定位到性能上的瓶颈了。

4.4 show processlist



另外我们在 MySQL 中经常使用的一个命令还有 show processlist，这个命令可以看到当前性能中正在运行的一些查询，SQL 引擎也同样提供了这个命令。通过这条命令可以看到系统中当前正在运行的查询，通过观察查询执行时从后端 DB 加载的数据量，以及时耗，我们可以方便地定位到这个系统中哪条查询是资源的消耗大户。show processlist 输出中还有一个关键的字段，extra，因为 SQL 引擎会对用户发送过来的 SQL 进行拆解，实际发送给 DB 的 SQL 跟原生的 SQL 已经不太一样，extra 会展示这些被拆分以后得到的 SQL 及其实际执行的时间。这样我们就可以将 DB 上的 SQL 和当前正在运行的业务 SQL 结合起来，如果我们发现 DB 上被某条 SQL 占用了大部分资源，我们就可以通过这些信息将其与一条业务的 SQL 进行一个关联。

当然，如果是 DB 出现慢查询的时候，你刚好在电脑旁边的话，就可以用刚刚说的 show pracesslist 进行查看。如果你不在电脑旁边，当回到电脑旁边的时候，这个高峰期已经过去了，那怎么办？SQL 引擎会将应用发过来的 SQL，以及拆分过的 SQL 记录到日志中。通过日志分

析工具，例如这个例子，可以看到当时 SQL 引擎对这条 SQL 的拆分，以及每条被拆分的 SQL 在哪个时间点发送到了哪一个 SET，同时还展示每条 SQL 的影响行数及返回的数据量。通过日志分析工具，我们可以还原出现问题的时候当时的现场，从而进一步对业务的 SQL 进行分析和优化。

Q&A

Q：SQL 引擎是自己开发的还是用的 MySQL 的引擎?

A：SQL 引擎是我们自己开发的。

Q：如果做数据库从 Oracle 换成 TDSQL，对应用来说需要的变更很大吗?

A：TDSQL 与 Oracle 在语法上会有一些差异，我们的一些客户也是从 Oracle 迁移到 TDSQL 来的，实践证明，只需要一点点改造，将部分 Oracle 的 SQL 替换成与之相对应的 TDSQL 语法即可。

Q：如果业务单表数量小的情况是不是不推荐使用这种分库分表的方式？请问随着业务的发展，当数量扩展到多少时，数据库推荐分库分表?

A：数据量比较小的时候自然不需要使用分库分表。当数据量在可预期的时间范围内，它可能会超过单机能够承受的范围，比如说当可能有上 T 的数据时，就需要考虑分库分表了。另外，当一台机器的计算能力满足不了业务的话，也可以使用分库分表，SQL 引擎通过将计算进行下推，使得计算能力随着节点数而线性增长。所以说我们当出现容量或者计算能力上单机无法满

足的情况时，我们就可以使用分库分表。

Q：这个 SQL 引擎类似于 Oracle 优化器或 sharding？开发者不用过度关注吧？

A：你用 SQL 引擎的话，使用 TDSQL 可以像使用单机 MySQL 一样使用 SQL 引擎，一般来说你是不需要过度关注的。

Q：参数在每个 SET 上执行的时候也是走事务吗？不然每个 SET 的原子性没法保证吧？

A：是的，通过 SQL 引擎开启一个事务的时候，SQL 引擎也会在每个 SET 上开启相应的子事务。

Q：分布式事务是基于 MySQL 原生的两阶段提交，不是基于 BASE，两阶段提交性能会不会很差？

A：分布式事务的话，我们是基于原生的 MySQL 所提供的 XA 功能，但是在真正上线之前，其实我们对它做了很大的一个改造，就是说其实跟开源差别比较大。在实际使用过程中，两阶段提交会有一些性能影响，但是性能影响不是特别大，基本上影响可能在 20%左右。

Q：Squence 的实现只是保证递增，不保证连续性吗？如果可以实现单调递增，具体的实现逻辑是怎么样的？

A：Squence 的话，是可以保证连递增的，每次获取新值都需要访问元数据表。

Q：开发者应该需要关注 TDSQL 的语法吗？比如说建表的时候需要指定 shardkey 之类的。

A：前面的课程已经讲解过。如果建表的时候没有指定 shardkey——TDSQL 会自己选择一个

shardkey，但是该功能默认是关闭的。为什么呢？如果你没有参与到选择 shardkey 的过程，TDSQL 没法保证这个 shardkey 的选择是最合适的，因为 TDSQL 不知道业务逻辑是怎么访问这张表的。

第五章：银行核心海量数据无损迁移：TDSQL 数据库多源异构迁移方案

关于 TDSQL 异构数据同步与迁移能力的建设以及应用方面的整个内容分四个部分：

- 一是异构数据库方面包括数据分发迁移同步的背景——我们为什么要发展这一块的能力以及现在这部分服务的基本架构；
- 二是 TDSQL 异构迁移能力有哪些比较好的特性，以及在实现这些特性的过程中的难点问题和我们提出的特色的解决方案；
- 三是结合 TDSQL 现在在国产数据库的一些推广以及应用的经验，我们针对在异构数据迁移或者同步的领域场景最佳实践，也介绍一些好的用法和场景;
- 四是针对本章节内容进行总结。

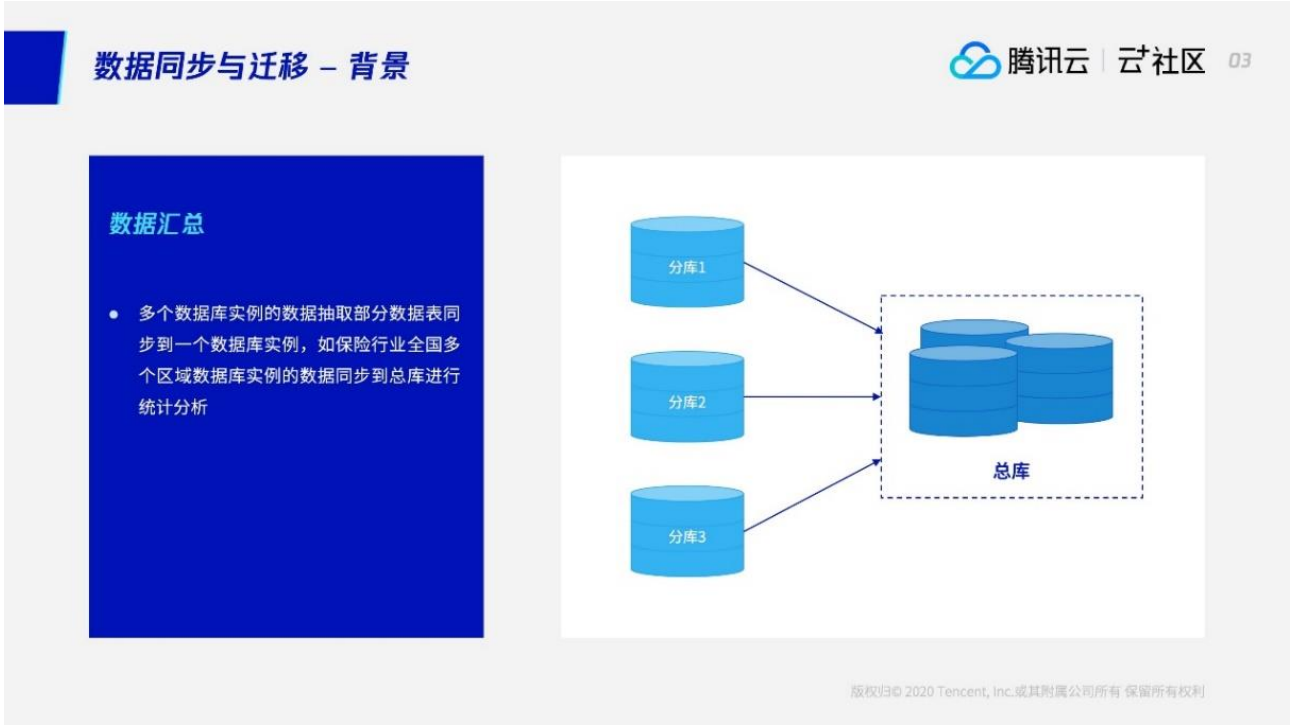
事实上，作为国产自研的成熟的分布式数据库产品，TDSQL 对内稳定支撑腾讯海量计费业务，对外开放 5 年来也通过云服务为微众银行等超过 600 家金融政企机构提供高性能、高可用、高可靠、强一致的分布式数据库服务。TDSQL 崇尚良性的竞争，也给予客户强信任的保障：

TDSQL 具备开放的架构，不仅支持安全快速的数据库数据迁入，同样支持异构数据库迁出。

从客户的需求角度出发，持续打磨产品，是我们一贯的原则。当然，除了支持数据库迁移，多源异构迁移方案也支撑数据汇总、分发等业务场景，这也是 TDSQL 具备完善的产品服务体系的体现。

1 TDSQL 异构数据迁移分发的背景及架构方案

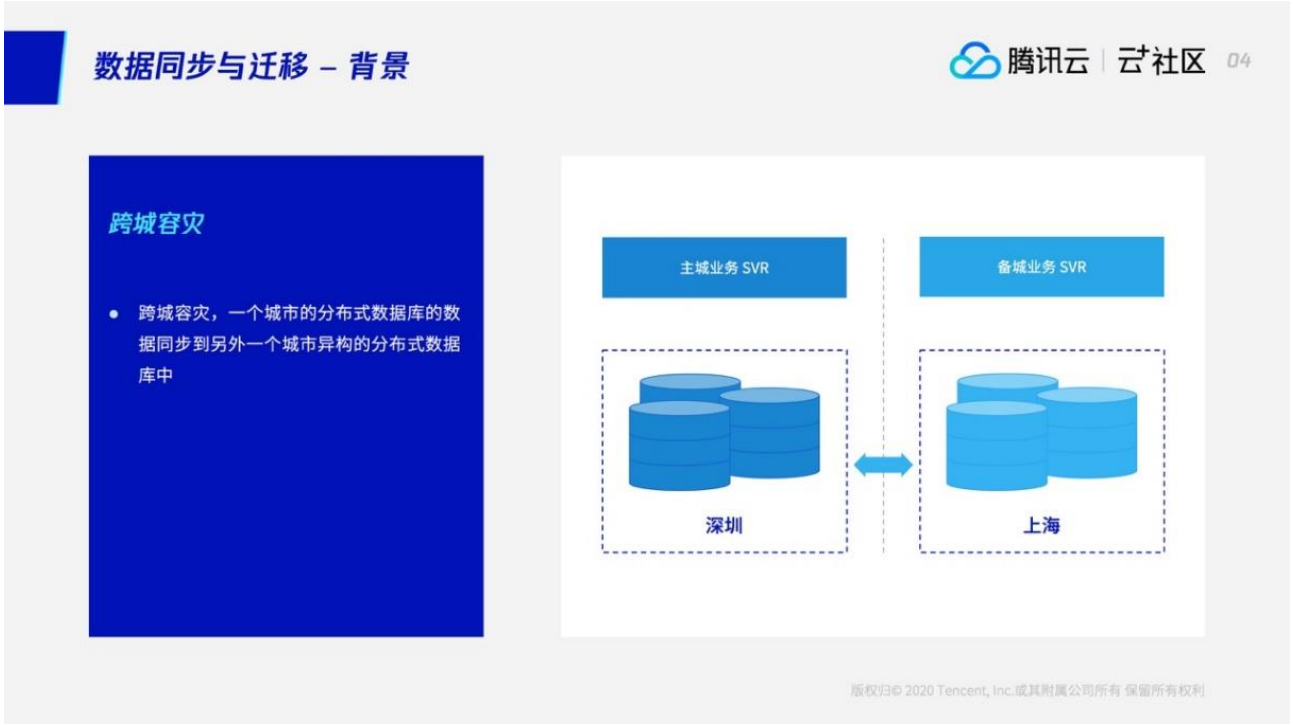
1.1 TDSQL 异构数据迁移方案的场景



TDSQL 作为一个金融级数据库，面对的更多是金融级场景以及金融机构客户，金融机构往往

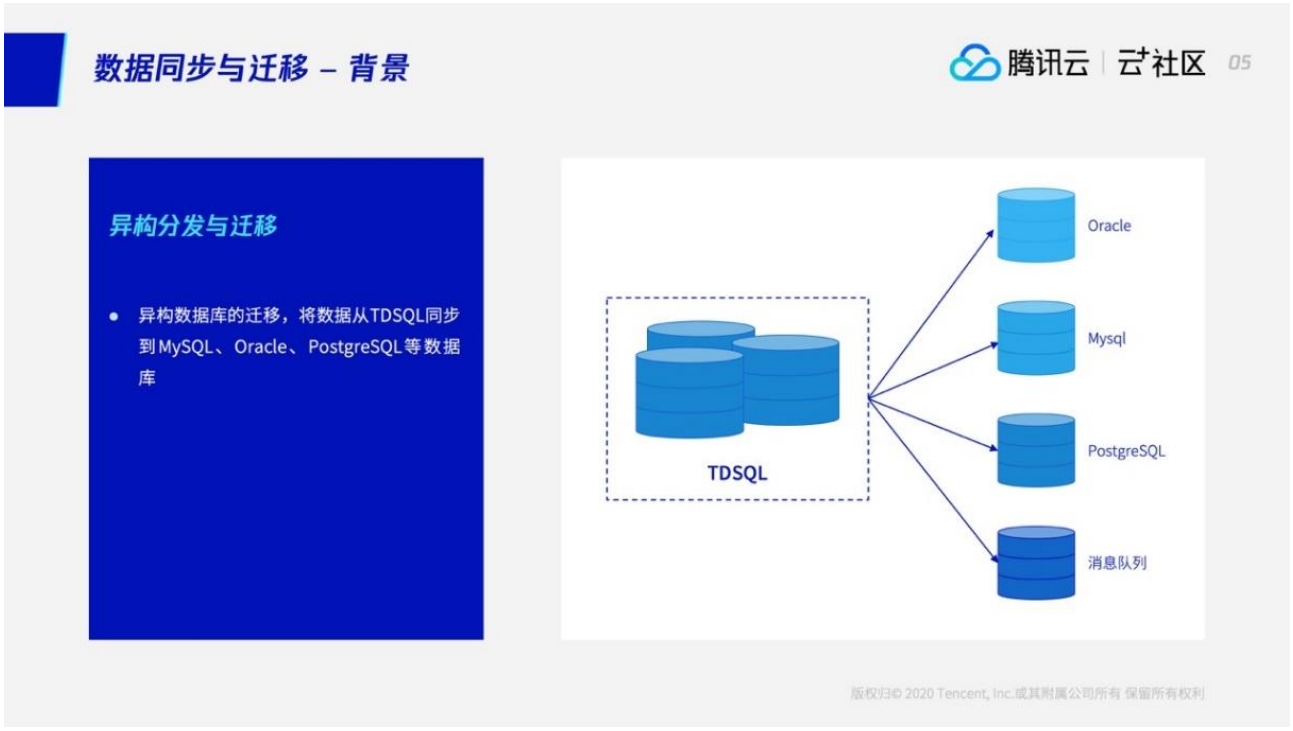
有一些比较特殊的需求，比如说保险行业，他们基于 TDSQL 构建业务时会进行一些业务划分，或者基于水平扩展的要求，数据会落在多个分库上等，而有时又需要把多个区域或者多个分库上的数据汇总到一个总库上进行统计分析。

TDSQL 作为数据层的服务必须具备高性能、准确可靠的将数据实时汇总的能力。也就是说，TDSQL 遇到的第一个数据库迁移场景需求就是要支持高速准确进行数据汇总的能力。



二是来自跨城容灾场景的需求。举个例子，腾讯内部金融级业务有跨城容灾的需求，这也是大型互联网公司中常见的容灾级别要求，我们要求业务具备城市级别自动快速容灾切换的能力。比如说在深圳和上海分别有一套数据库并且支撑了相关业务 SVR，我们就需要在这两个城市间的数据库 DB 之间实现数据实时同步，这样有两个好处：1、这一套实时同步的东西可以在城市级切换的时候快速地将业务切换备城；2、这一套实时同步做到足够好，比如说实时性更好或

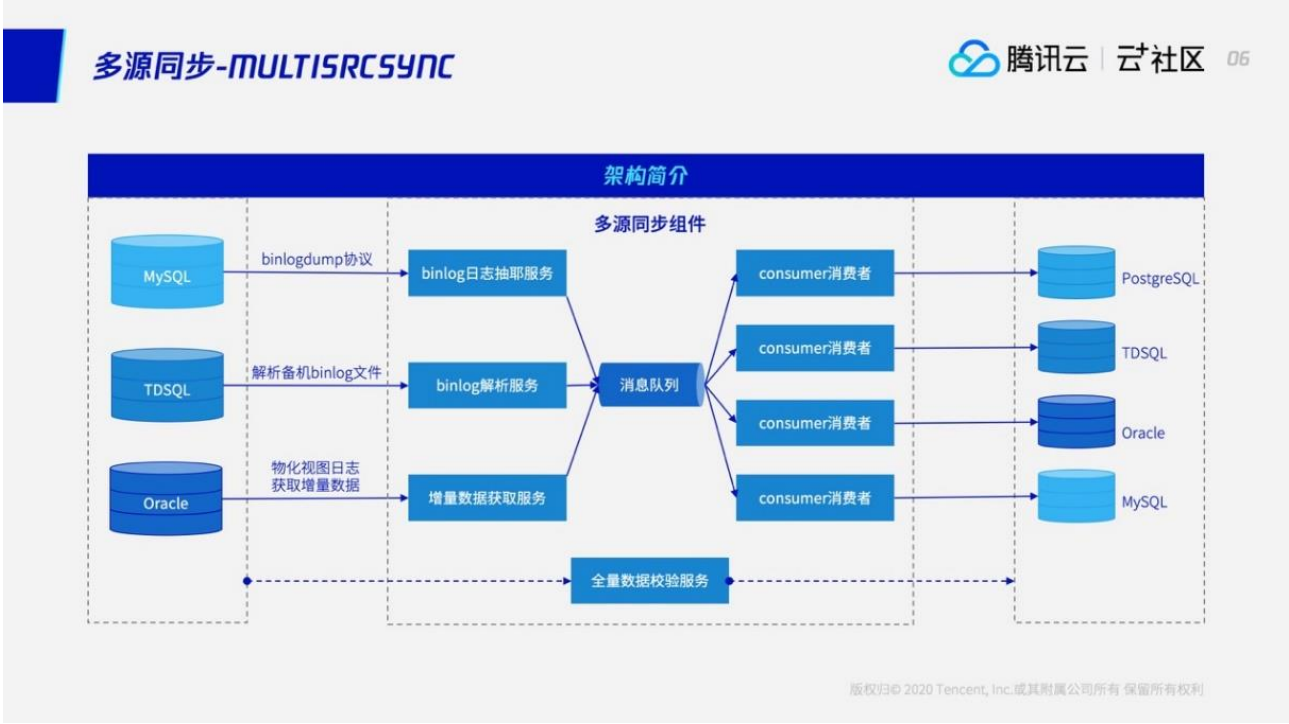
者说数据准确度是完全没有问题的情况下，我们可以做到业务的分流，比如有部分的业务是在主城的 SVR 上，当我们认为主城 SVR 业务量过大或者说压力太大时候，也可以切换一部分业务流量到备城的 SVR 上，实现两个城市之间数据层的数据和数据同步。我们遇到第二个需求就是在跨城容灾或者跨城业务分流、跨城数据同步上我们需要 DB 侧有这样的能力提供给业务使用。



三是异构的数据分发和迁移。TDSQL 作为一个金融级数据库，对外是非常开放的架构，我们支持将数据以各种各样满足业务的方式同步到外面的平台，比如当有一些业务需要在 Oracle 上跑一些比较老的业务或请求等等；也有一些业务需要把数据同步到消息队列给下游业务使用，比如大数据平台、其他的检索之类，我们可以通过消息队列把数据抽出来。金融机构往往需要数据不仅能够存进来而且能够很好地按照要求分发出去，供下游业务使用，这也是我们遇到比较重要的需求。

针对上面提到的三种场景——数据汇总、跨城容灾、异构数据库间数据的分发和迁移 TDSQL 针对这些需求构建出一套叫做多源同步的系统。现在介绍一下多源同步的系统是通过什么样的架构来满足我们提到的三个需求的。

1.2 开放的架构：TDSQL 多源同步方案架构解读

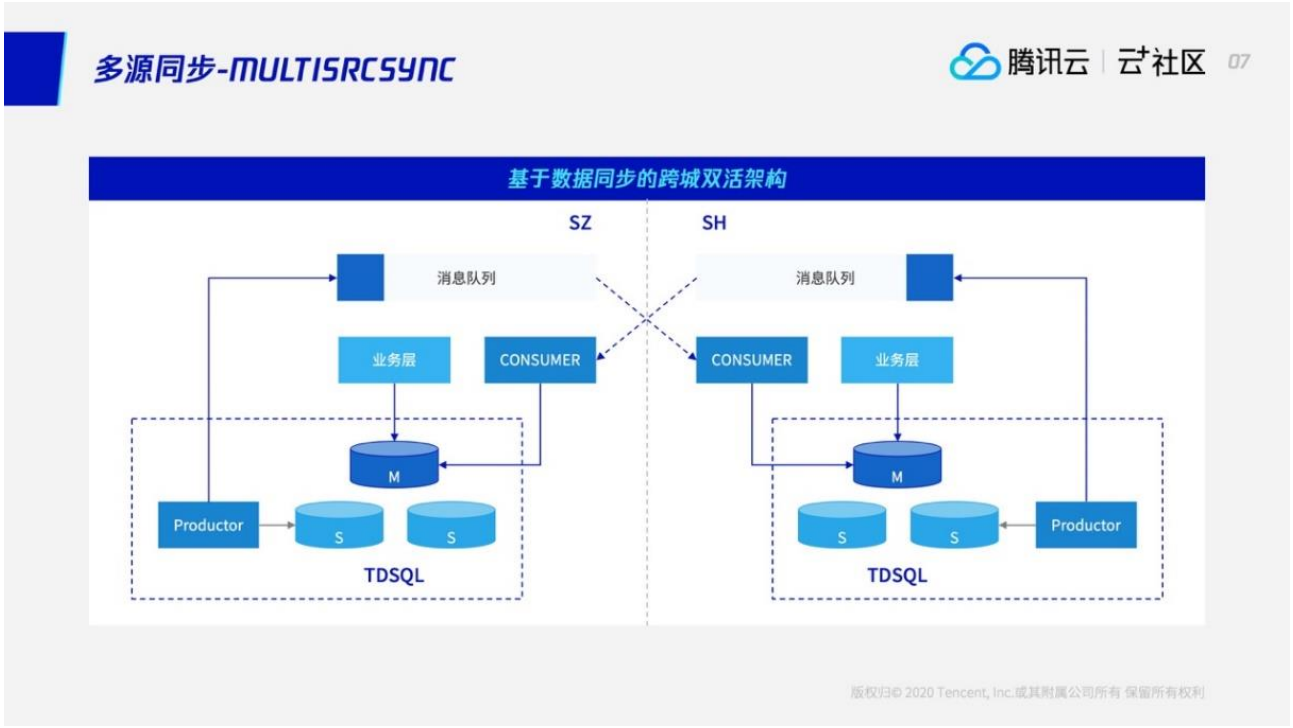


从图上可以看到有三个组成部分：一是原数据的抽取；二是中间的存储——这是一个消息队列，三是目标实例。这是一个非常典型的 CDC 架构，通过获取源端数据源的增量数据，通过消息队列，下游消费的逻辑将数据进行分发。从左边看到，这一套多源同步，源端支持 MySQL，就是对 MySQL 系列的 DB 可以获取它的增量数据；还有 Oracle。。抽取完数据增量，比如说 binlog 日志或者增量数据获取服务抽取到的数据，我们会以一个中间格式存放到消息队列里

面；存到消息队列以后，我们自己实现了一个消费逻辑——叫做 consumer 消费者，它可以实现将这一套存到消息队列的数据按照不同的需求以及不同的目标端类型将数据推送到下游。

目前 TDSQL 多源同步方案支持的目标端类型有下面这几种： PG， TDSQL， Oracle，还有一部分就是 MySQL，另外还可以将增量数据再推往另外一个消息队列，比如说有一些业务可能需要将这套增量的数据推往业务使用的队列组件里面，我们也是支持这样做的。

最后，在这一套同步的数据链路过程中，我们有一个数据校验的服务，包括两个方面：一是增量校验，含义就是会实时校验这一笔数据从源端抽取，到它的增量变化，再到写到目标端之后，这笔数据落库落得准不准确，是不是在正确的目标上写下这笔数据；二是存量校验，可能是一些定时定期去跑批，比如说定期对源和目标的数据进行整体的校验，我们能够主动及时地发现整个数据通路上的问题和错误。



结合我们刚刚说的需求，基于数据同步的跨城双活架构，也是腾讯内部现在在使用的架构。基于数据同步的跨城双活架构是这样的形式：

首先左边和右边代表不同的城市，这里举例左边是深圳，右边是上海。从图上可以看到，TDSQL 在 SZ 这套实例会将业务不断写入的增量数据源源不断地写入本城的消息队列里面。对城的 SH 也会将自己业务访问的增量数据源源不断写到消息队列里面，同时在各个城市有一套自己的消费服务，这套消费服务会拉取对端的增量数据，也就是说会拉对城的消息队列里面的增量数据进行重放，这样就实现了两套基于数据同步的一套跨城双活。这个双活是有前提条件的——就是两套业务在 SZ 和 SH 同时写的时候，它的访问主键一定是分离，在这一套逻辑下面没有办法做到同时对同一条主键进行修改。我们基于跨城的这套双活架构也是要基于主键分离的做法。

2 TDSQL 多源同步方案的挑战和特性

2.1 要求与挑战

介绍完整体架构，我们继续深入拆解下，这套架构所面对的业务场景，都有哪些要求？在这些要求实现的过程中是有哪些难点，并且针对这些难点我们是怎么处理的？以下将介绍这其中的特性、难点、解决方案：

一是高性能：对实时性要求比较高的业务对数据同步的速率有比较高的要求，比如说秒级别等等。但无论如何，在这个互联网时代，这套数据同步要快，不能说加了这套数据同步、异构分发的逻辑以后，它同步的速度非常慢，这肯定是不可以。

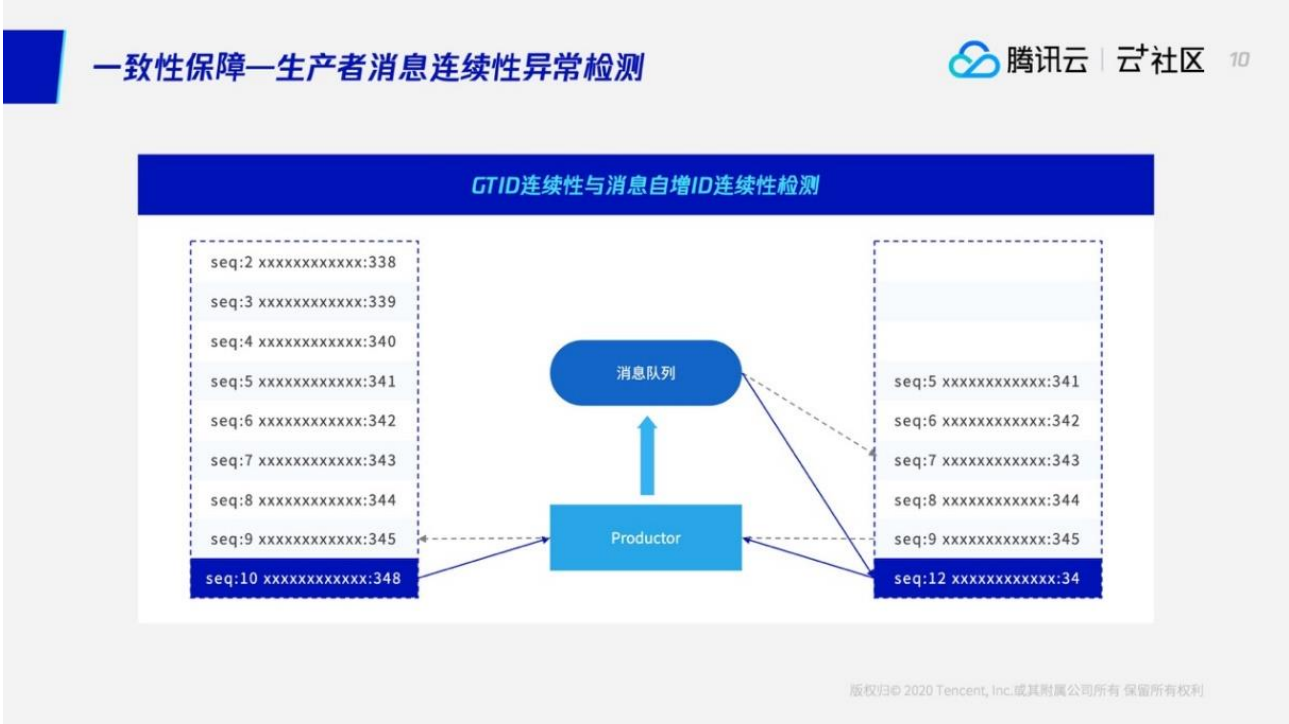
二是数据强一致性：在快的基础上，同步的数据一定要准。这套数据同系统，分发的系统把数据从源端抽出来，往里面写的过程中，需要做到原来写出来是什么样的，目标重放就是什么样的，两边的数据一致性一定要有保证，这里面就包含了我们如何规避在抽取链路、重放链路这两个数据链路上的错误；二是如何保证在异常情况下写入的数据一定是对的。

三是服务高可用：这一套同步服务，一定是高可用的，体现在两个方面：1、灾难的情况下，本身消费者的服务能够在假如机器出现一些不可恢复的故障时能够及时地感知并且自动迁移和切换；2、要应对本身常规的扩容——垂直扩容或者水平扩容的伸缩性需求，这也是我们比较

强调，这一套同步服务要能够兼容各种灾难情况和常规的运维场景下各种各样的要求来做到服务的高可用。

2.2 一致性保障

2.2.1 自动化消息连续性检测



从上面的架构图我们可以看出来，整个数据链路比较长，它要先把增量数据拿到，写到消息队列里面去；再从消息队列里面消费出来。生产者这一套服务做的事情就是首先要拿到增量数据，二是要正确地把拿到的增量数据准确地投递到消息队列里面，这里面有两个问题：1、如何判断我拿到的消息——本身的增量数据，是对的；2、我如何确定写到消息队列里面，消息队列

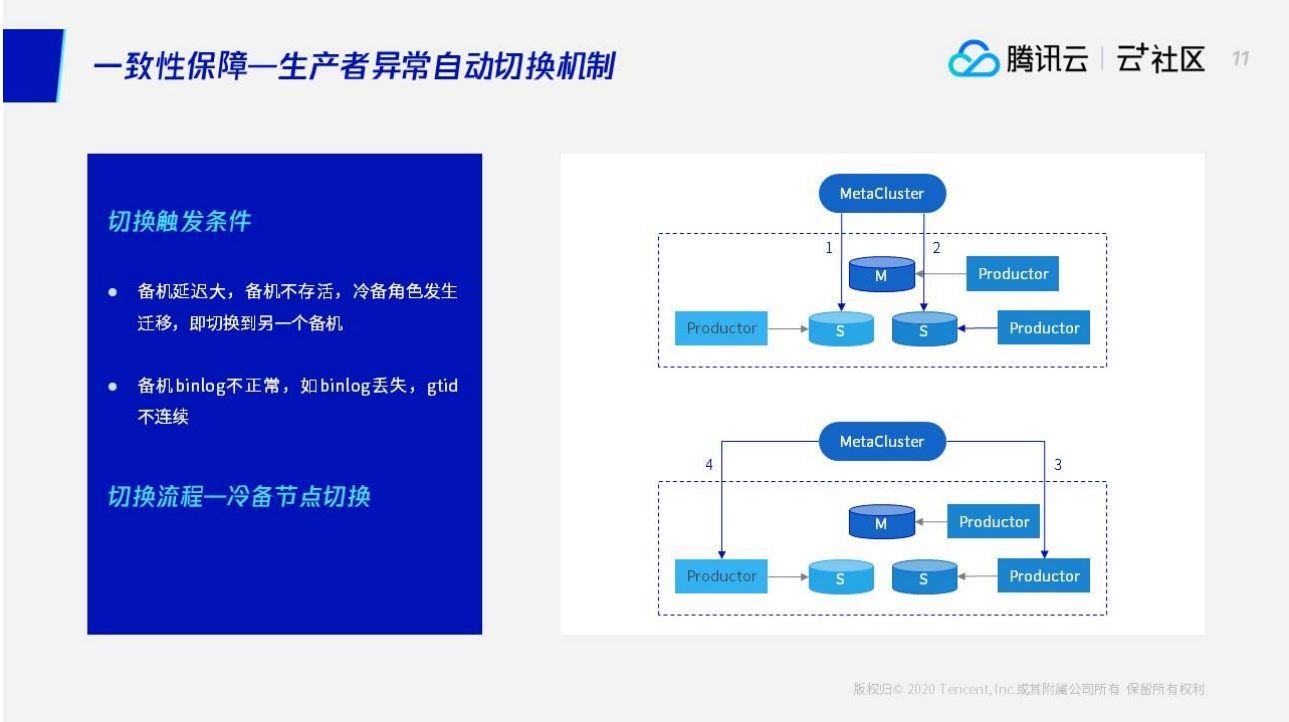
存的也是对的。

这就衍生出来两个方式：一是拿到增量消息的时候，我们会根据 GTID 的特性，检测拿到消息的 GTID 的连续性，保证拿到这套增量数据的东西一定是准的，比如说 GTID 上一个拿到的是 345，下一个如果拿到的是 348，这个时候系统会认为现在拿到这一条 GTID 跟上一个并不连续，并不连续的情况下我们就要进行容错处理，比如会向主机补偿或者向其他的节点切换补偿等。总结来说，TDSQL 在拿增量消息这部分，是具备连续性检测的能力，保证拿到的数据一定是准确、连续的。

二是系统如何保证写到队列里面的数据一定是准？在写到队列过程中有可能出现重复、乱序等情况，TDSQL 多源同步方案采用的策略是——利用消息队列本身在写消息的回调通知的特性，我们在将消息推到消息队列的时候，会给每个消息赋予一个连续递增的序列号，通过消息队列回调的写入消息来确定系统写入的消息是不是有序的。举个例子，我们按[5,6,7,8,9]这样的顺序向消息队列生产一部分消息（写），届时收到的消息回调序号也应该是[5,6,7,8,9]；当接收完 9 号这条消息回调的时候，下一条如果收到的回调序号是 12、或者 11，那么就会认为从 9 号往后的消息队列消息不是一个有序的消息，这时系统会重新从 9 这条序号往后的消息重新上报消息队列，最终保证写到消息队列里面的数据是没有空洞并且是连续递增的。这是生产者服务在消息连续性异常检测方面我们提供的两种机制。

2.2.2 异常自动切换机制

以上介绍的机制可以保障多源同步、异构迁移中如何检测到错误。那么，检测到错误之后如何处理呢？以下就介绍生产者异常自动切换的机制、切换的条件。

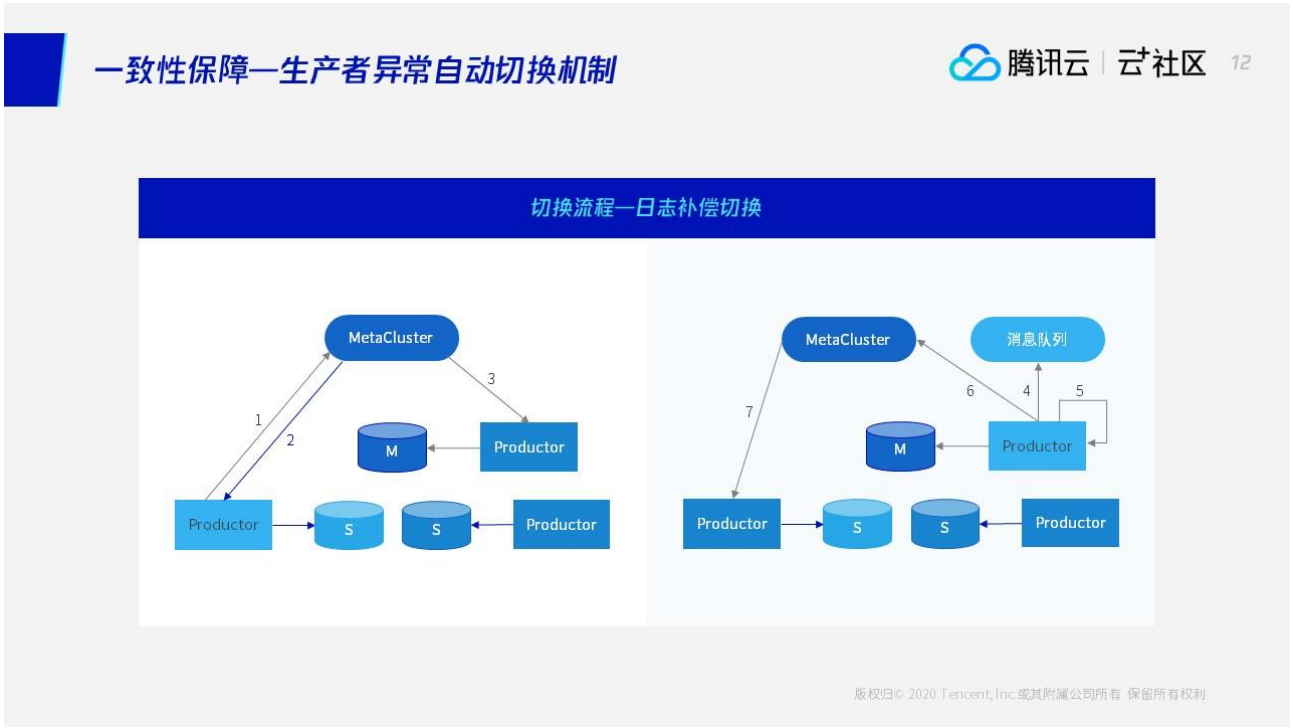


这里面都以 TDSQL 的实践为例：获取增量日志必须要在一个合适的 TDSQL 角色上处理，TDSQL 本身是一个一主多备的分布式数据库集群，在选择获取数据库增量日志的角色上我们选择从备机上获取。

选择一个合适的备机对增量数据获取来说是非常重要的。当获取增量日志的备机的延迟比较大，或者这个备机本身不存活，或者这个冷备发生了迁移（什么是冷备？离主机的距离最近，或者跟主机的差距最小的备机，我们叫它冷备），这个时候系统就会将解析日志生产者的服务切换到另外一个节点，整个切换流程通过 MetaCluster 服务协调。也就是说当工作的数据库节点本身的状态发生跃迁之后，其他节点生产者服务就会通过 MetaCluster 来感知到状态的跃迁，并

且适当地启动自己的服务——从一台备机状态跃迁到另外一台备机，而跃迁前的备机的生产者服务会停掉，新的备机生产者服务会自动拉起来。这就是它的切换流程——通过 MetaCluster 进行下发协调。

切换是基于什么样的触发条件？现在解析到的这台备机本身状态是正常的，比如延迟没问题，存活性也正常，冷备角色一直没有发生变化，但是发现它的 binlog 不连续。当我们检测到拿到的这套 binlog 是不连续的时候，就可以认为这里面可能会出现 binlog 的丢失，这个时候就要发起补偿的操作。怎么补偿呢，通过这个流程给大家介绍一下。



当系统发现解析到这套 GTID 不连续了，就会向 MC 注册一个节点。举个例子，系统现在已经发现拿到的 binlog 不连续了，于是注册一个补偿节点，包含着"向主机补偿"这样的信号。当主机检测到有这样一个补偿节点时，会将日志解析的角色接管过来并开始工作。

接下来，我们如何确定主机从哪里开始解析日志？我们会从消息队列上读取最后一条消息——最后一条消息包含 GTID 的信息。这时主机就会把这条消息对应的 GTID 转化成本地的 binlog 文件名和偏移量开始解析。

主机的补偿需要持续多长时间？持续一个文件处理的流程。当主机补偿到解析的所在文件结束以后就会退出主机补偿的流程，并将这个角色通过 MetaCluster 重新下放给备机的生产流服务，而备机的生产流服务接到这个请求以后会重新从消息队列上拉取上一次主机补偿的日志中最后一套消息。如果说找到了对应的 GTID，并且往下解析的时候没有发现不连续的情况，这一套补偿流程就算结束，备机会继续在自己的角色上持续地进行增量数据生产。

如果发现从消息队列拉下来的主机补偿日志最后一条本机找不到，就说明这个主机的补偿不完整，有可能备机缺了两三个文件，这个时候会持续向主机进行补偿，通过注册 MetaCluster 节点的方式一直重复这个流程，直到主机补偿完成，备机在接管角色的时候能够连续顺利地接着解析，本身的日志才认为这套补偿流程已经完全结束——这就是一个通过不断向主补偿日志的方式来进行异常的切换的流程。

2.2.3 幂等重放机制

介绍完生产这套链路之后介绍一下下游的链路——消费，消费的链路中怎么保证数据一致性？首先回顾一下刚刚提到的生产端的数据一致性保障——生产端在实现消息生产的时候实现的

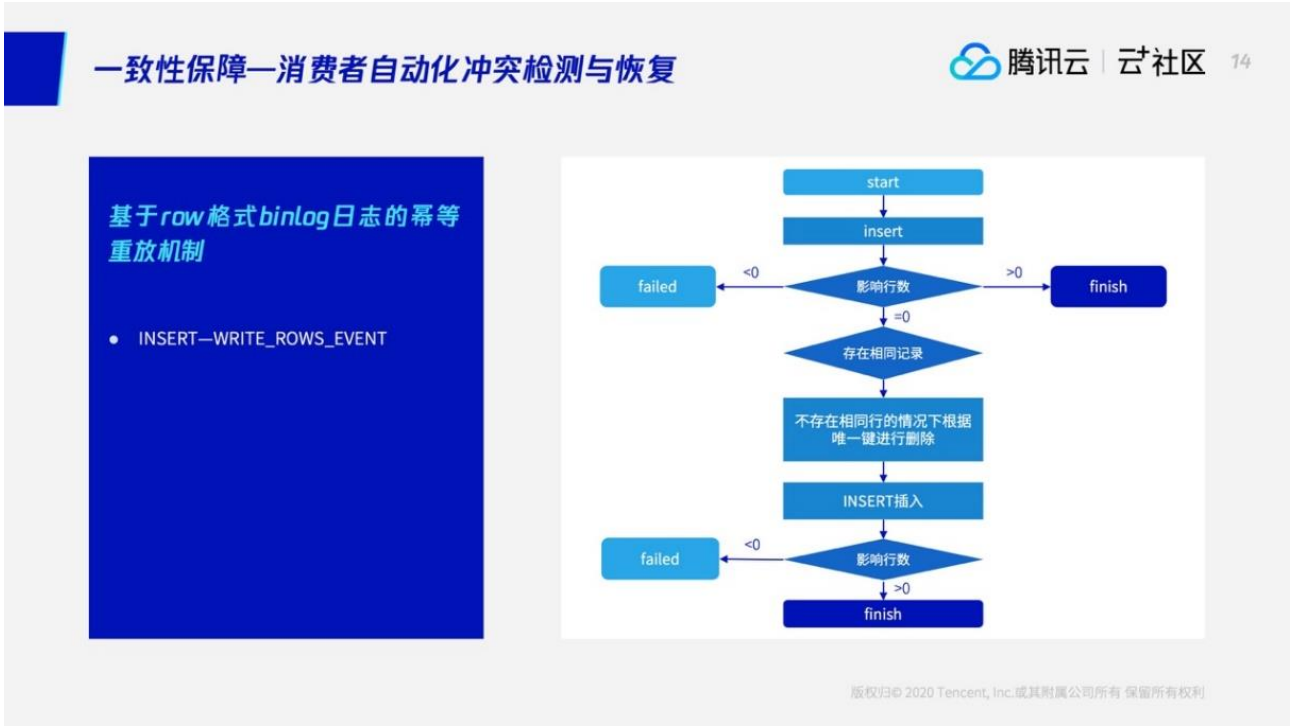
是一种 at-lease-once 的模式进行消息生产，这里面就要求消费服务必须能够确地处理消息重复这个问题，也就是说我们要支持所谓的幂等逻辑。

支持幂等之后有什么好处？在 binlog 是连续无空洞的前提下，支持幂等机制的消费服务可以从任意一个时间点重放 binlog 消息，当重放结束以后目标的数据会达到最终一致，这就是消费链路实现幂等的动机和优势。这个机制实现的难点在于要绝对的可靠——重放一定是要百分百没有问题，准确无误。

基于这样的要求以及上游数据写到消息队列里面的现状，TDSQL 以此为设计的原则，实现保证按照 binlog 事件本身的意图对目标实例进行修改。什么叫做按照 binlog 事件的意图去对目标进行修改呢？

增量数据无非就是三个方面：一是 insert 的写入，二是更新，三是删除。

insert 写入



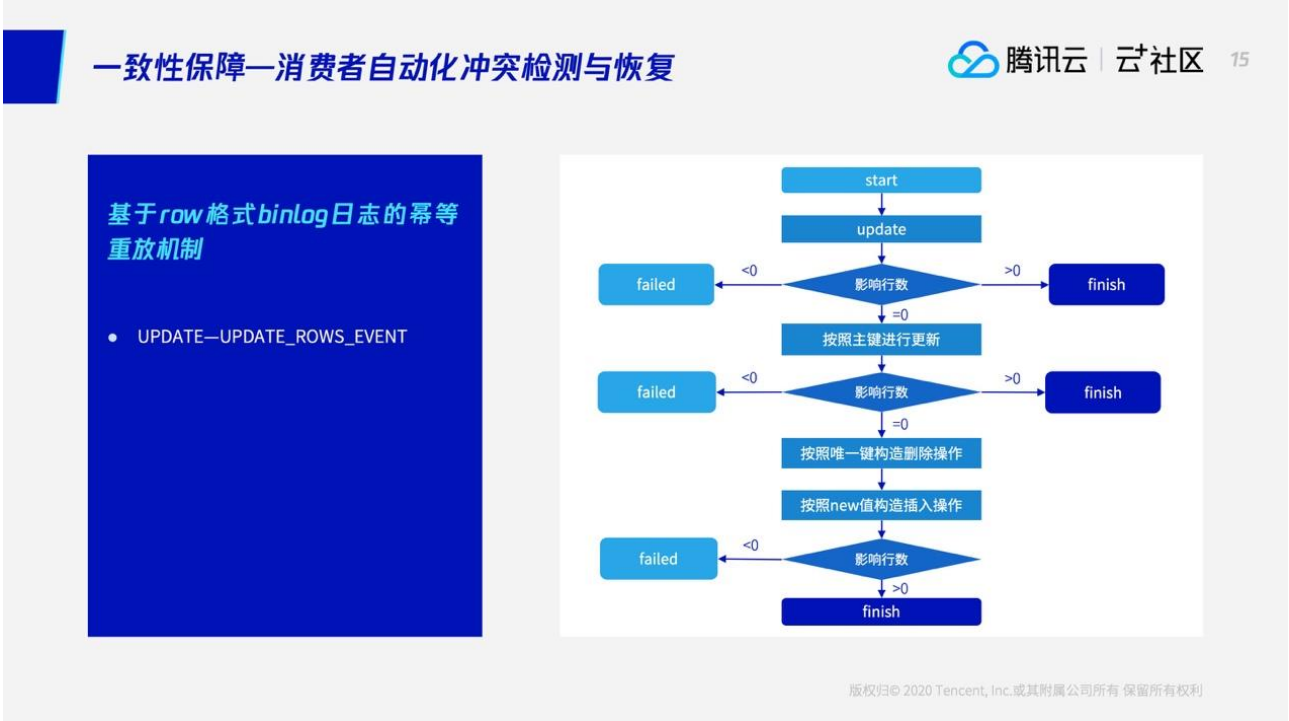
在写入的时候我们是如何做到 insert 事件幂等呢？一个 start 进来我们要重放 insert：

当它的影响行数大于 0，我们就认为这套 insert 执行成功；如果执行失败，我们认为它可能有一些报错，比如说语法错误或者目标的字段过小，并进行重试的逻辑。

当影响行数等于 0，则判定可能会出现主键冲突——insert 失败影响行数为 0，这里面唯一的可能就是出现了冲突。出现主键冲突的时候这个时候怎么处理？insert 这一条数据发生的时候意图是什么？在 insert 之前 DB 里面是没有 insert 这条数据，而当这条 insert 发生之后，DB 里面是有的——按照本身的意图来做，意味着如果发生了主键冲突或者影响

行数等于 0 的情况，里面存在一个相同的记录，这个时候系统会按照 insert 本身的值拼一个 delete 操作。这条 delete 操作下去后，就能保证在这条 insert 写入之前，目标里面是没有这条数据的；当我这条 delete 做完之后，再把这条 insert 进行插入。这个时候如果影响行数大一点，可认为这条 insert 被按照本身的意图做完了。其实也就是说，要保证这条 insert 做的时候，只有当前这一条数据——这就是 insert 本身的幂等。

更新



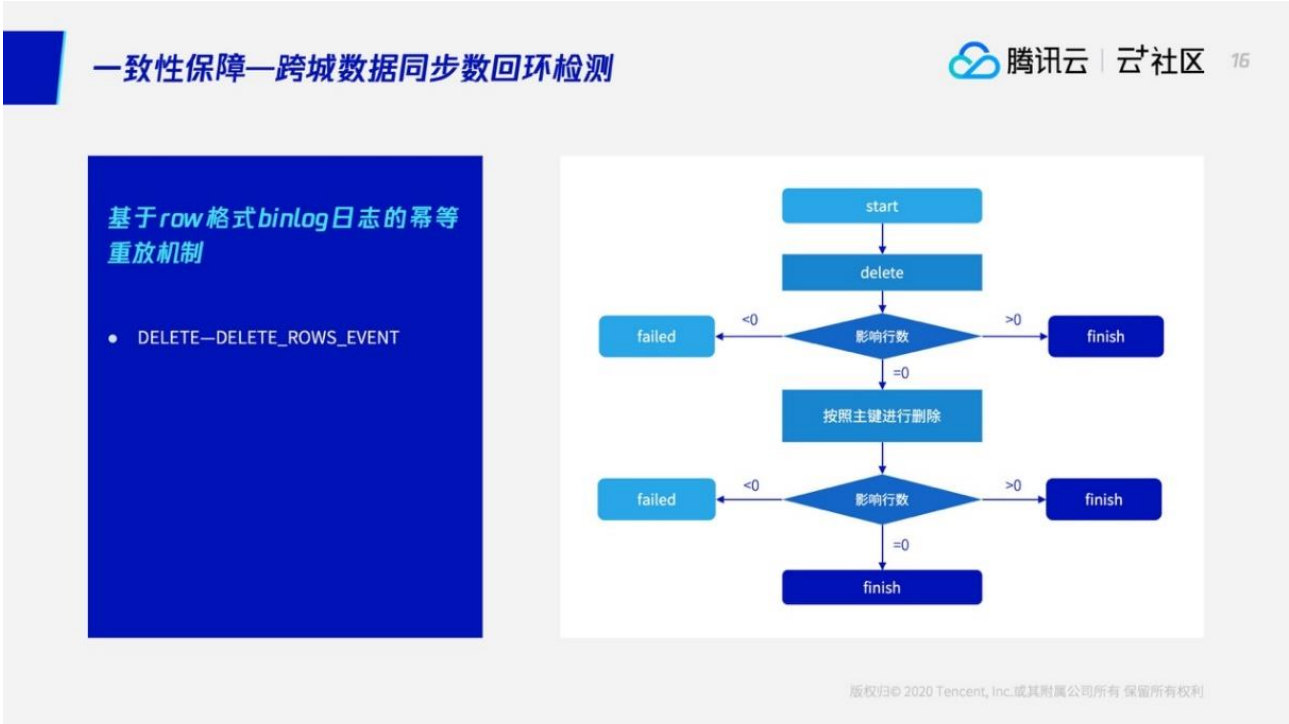
更新：首先一条 update，如果影响行数大于 0，可判定这条执行是正常的。如果小于 0，则意味着可能出现一些执行错误，比如语法有问题或者字段长度有问题。

如果它的影响行数等于 0，有两种可能：一是没有匹配到——进行 update 时是按照全字段进行匹配的，这一行改之前和改之后所有的字段都在这条消息里面，原始更新也会按照所有字段

来去拼装，没有匹配到则意味着某些字段没有匹配到，这个时候会按主键更新——也就是匹配到这些值可能是全字段，执行更新的操作。

如果按照主键更新操作，影响行数还是 0 的话，则可以判定为出现了主键操作的冲突。这个时候系统就会思考一下，这条 update 它的语义是什么——update 的语义是指这条 update 执行完以后，目标库里面第一个是没有改之前的值，第二个是有且只有改之后的值，所以我们按照这个语义做接下来的操作，按照所有的唯一键去构造一个删除的操作，操作完了以后再按照 update 里面改后的，构造一条插入操作，将这条插入操作写入目标 DB——如果影响行数大于 0，实际可认为这条 update 就是按照它本身的意图对目标实例进行了修改。

删除



我们来看一下删除的过程。相对于 update 来说简单多了。这个过程中，delete 结束后大于 0

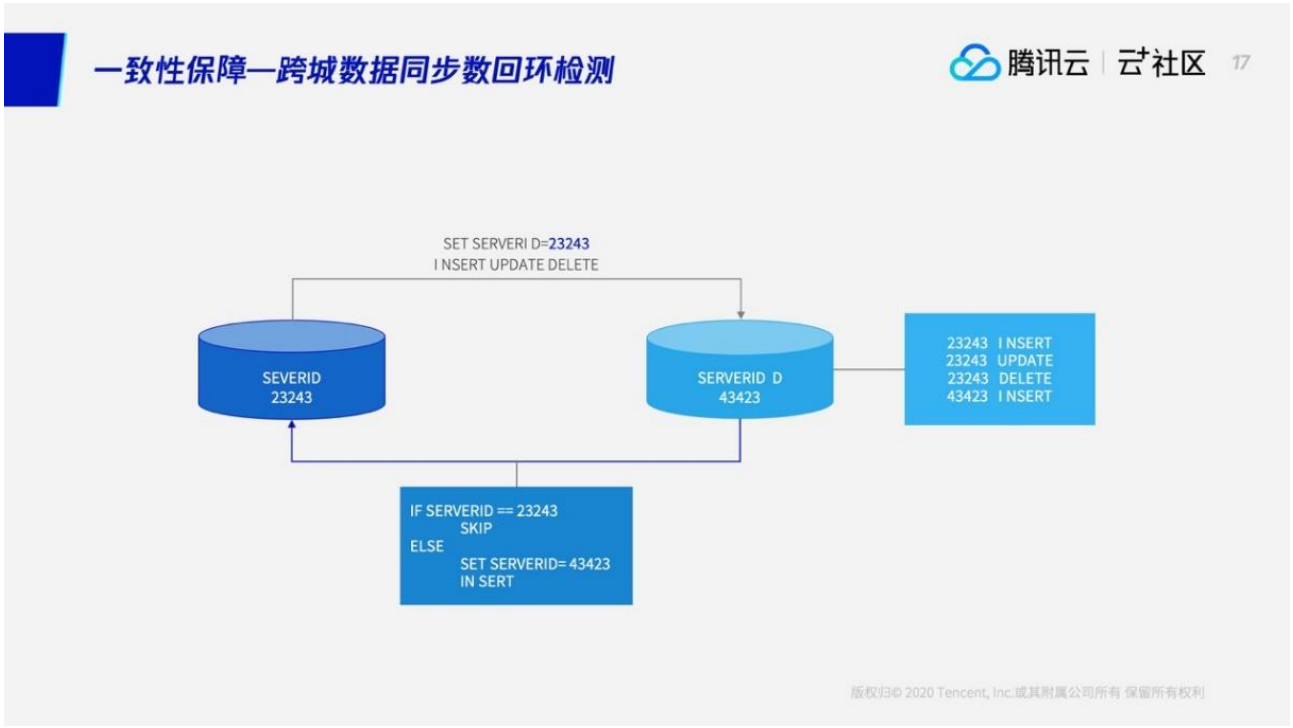
就成功；小于 0 就是失败；等于 0 的时候我们认为它可能没有匹配到行，这个时候我就按照主键操作——因为删除的操作最终的结果就是目标一定没有了当前删除的消息主键所标识的这一行——这条操作完成后，DB 里一定没有这行数据，因此仅仅是按照主键进行删除就可以了。这个时候如果影响行数大于 0，则删除成功。如果等于 0，就认为按照主键去匹配，本身删除不到，匹配不到——意思是本身目标就没有这条要删的主键所标识的数据——所以实际上它的结果跟要做完删除的结果，影响是一样，也就结束这一条删除的幂等。

回顾三种类型的时候，我们比较关注这条数据在执行前后的状态，它执行前是什么样的，执行后是什么样的，我们在重放这条消息的时候，严格按照这个来做，insert 就是执行前没有这条数据，执行后有这条数据，如果遇到冲突就先删除后 insert，update 执行后它的结果，一定没有改之前的值，有且只有改之后的值，删除也是一样，目标里面一定没有主键所标识的这一行在目标实例里面，我们按照这个逻辑设置幂等的流程，就是这样的过程。

2.2.4 跨城数据同步如何规避数据回环

接下来看一下在跨城数据同步如何规避数据回环。跨城的架构中，本城的一套数据实例会把增量数据写到消息队列，对城会有一个服务从消息队列里面把这个数据拉出来——对城的消息也会落到对城的消息队列，本城有一些消费服务会把这些数据拉过来，也就是说数据具有一个环路。如果不做回环检测和规避，比如插入这一条数据，这条数据的目标又插入了，并且也落了一个日志，做了这个日志又写回来，这相当于同一个主键的数据来来回回在写，这样会把数据写脏。

我们是如何来规避跨城数据同步的回环，以及对它进行检测的？TDSQL 结合 DB 内核的改造，通过 SERVERID 来规避数据在跨城双活数据同步架构里面的回环问题。



假如说左右两端是两个 DB，这两个 DB 对应的 SERVER ID 不一样：一个是 23243，一个是 43423。现在有一条叫做 insert 的数据写入目标，写入目标之后会设置当前这个 SERVER ID 跟原来的 SERVER ID 一样——23243 的 ID。这条 insert 落到 DB 里面，会记录成它的对应日志的 SERVER ID 23243，而不是记录它本身备城的 43423 的 ID。

当消费服务拿到增量日志——拿到的这条日志所对应的 SERVER ID 跟目标 DB 的 SERVER ID 是一样时，则认为拿到这条日志一定是目标写过来的日志，然后执行跳过的操作。只有当拿到的这条日志对应的 SERVER ID 跟目标的 SERVER ID 不匹配的时候，才会把这条数据写到目标里面去。这样一来，才能保证只有是真正业务访问到源端的 DB，并落下来的那条日志，才会

被成功写入到目标上——这就形成一个通过 SERVER ID 将环路里面的数据过滤的机制。

2.3 高性能保障

2.3.1 有序消息并发重放

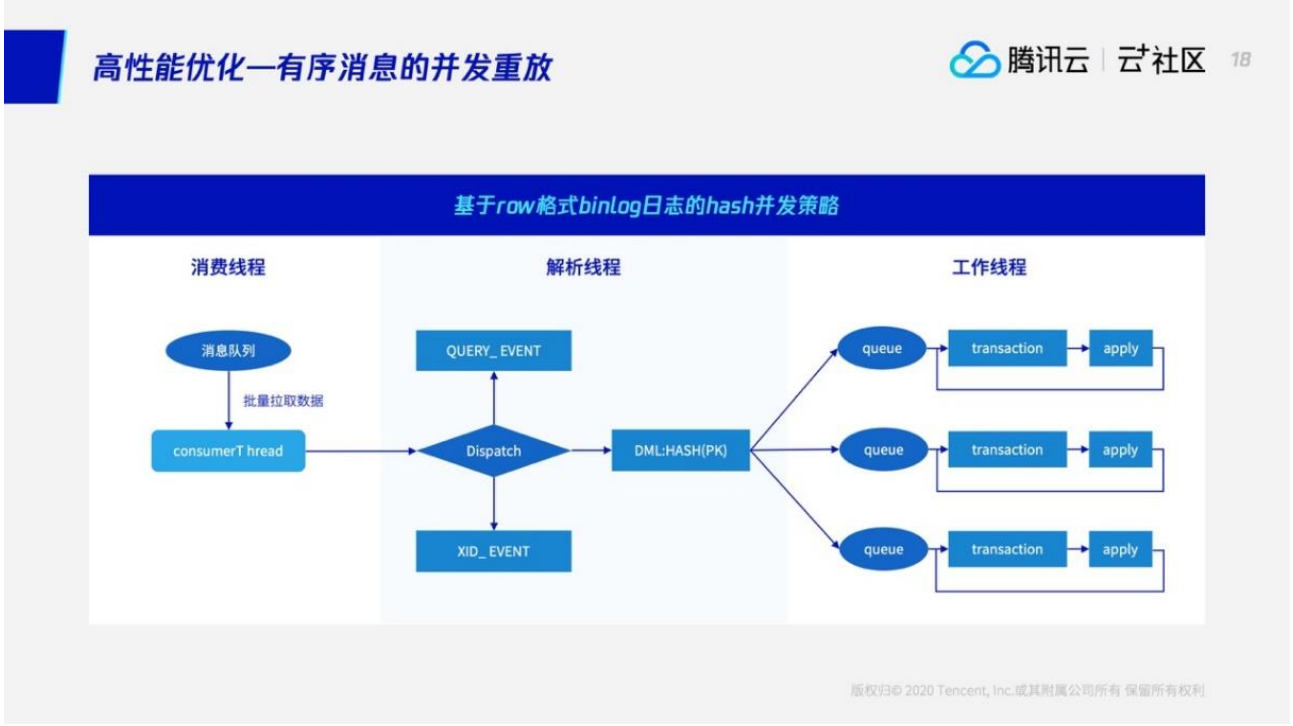
现在介绍一下关于高性能的优化实现。

MySQL 本身在落日志的时候是有序的消息，就是说 binlog 是有序的。如果按照 binlog 的数据来重放，是没有问题的——按照一个事务一个事务进行串行解析。但这会带来一个问题——就是慢。

对于这个问题，TDSQL 采取对有序消息进行并发重放来提升数据同步的效率。采取通过基于 row 格式 binlog 日志的 hash 并发策略来实现。

这个 hash 策略就是根据表名和主键来做：首先从消息队列拿到数据之后，系统会进行派发，派发过程中根据消息里面的主键和表名进行 hash，将消息 hash 到不同的工作队列。

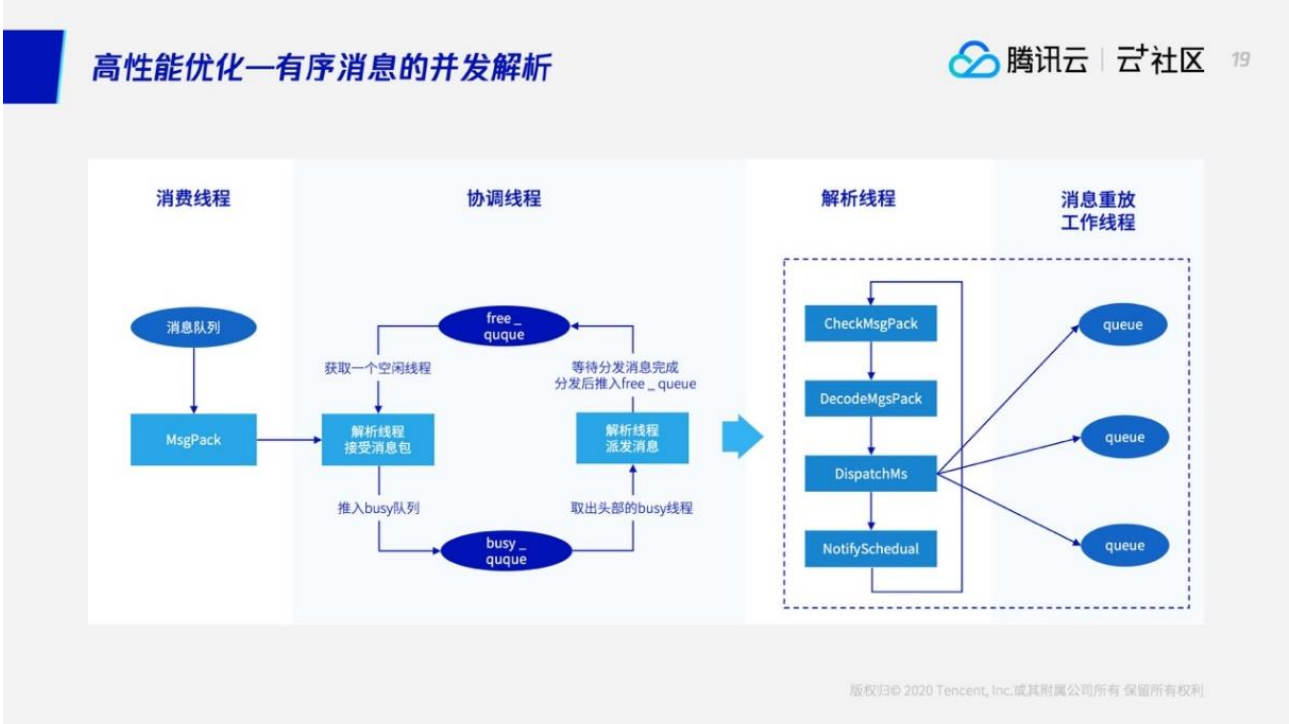
这样的 hash 策略有一个什么样的结果？相同表的同一行操作的序列一定会被划分在同一个工作现场，只要保证对某一张表其中固定一行的操作是串行有序的，就认为这套数据在并发重放结束之后数据是最终一致的。



总结而言，TDSQL 多源同步并发策略就是按照主键和表名进行 hash，保证每一个表里面的固定一行的操作序列在同一个工作队列里面串行化。并发的数据同步和串行数据同步区别就是一致性的问题，可以看到这种并发策略相当于把事务打散。这里面并发重放的时候就会产生事务一致性的问题，有可能会非常小的几率读到中间状态，在数据同步速率有保障的情况下说不会出现这种问题的。如果说这个业务本身对事务一致性要求非常严格——当然我们现在还没有碰到这样的场景。这就是一个有序消息的并发重放。

2.3.2 有序消息并发解析

以上是消费端性能优化的过程，首先就是要写得更快，通过各种优化把 hash 并发到多个现场去写。那么写完之后，消费端的性能瓶颈在哪里？在解析上。



大家如果有印象的话，我们写到消息队列里面的数据是中间格式——json 格式。json 格式需要一个解析过程。当我们解决了重放性能瓶颈之后，原始的消息包拿到后解析的过程又变成性能优化的瓶颈。针对这个问题 TDSQL 同样做了有序消息的并发解析优化。

并发解析的策略就是，维持一个线程池。从消息队列上拉下来的这条消息，本身是一个原始没有解析的包，当拉下来这条消息包时会从这个池子里面捞一个空闲队列，并把这个包给空闲的线程，这个线程拿到这个包以后就开始解析，当它拿到包这一刻就会进入到另外一个 busy 队列里面。它在忙队列里面会不停地解析拿到的这些原始消息，解析完之后会有一个协调线程，从忙队列面不断把解析线程摘出来唤醒，把解析后的消息再并发地分往后面的工作线程。源源不断从消息队列拉消息，拉完之后就把这些没有解析的消息分给一组线程去解析，这一组线程在解析的时候——虽然解析是并发的，但在被唤醒派发的时候有一个出队的操作——也就是

派发是按照顺序派发——这就做到有序消息的并发解析：通过一个忙队列、一个闲暇队列，两个协调线程把整个流程串起来，这样基本解决了在 json 解析上的瓶颈。

2.4 高可用保障：多机容灾保护

2.4.1 多机容灾保护

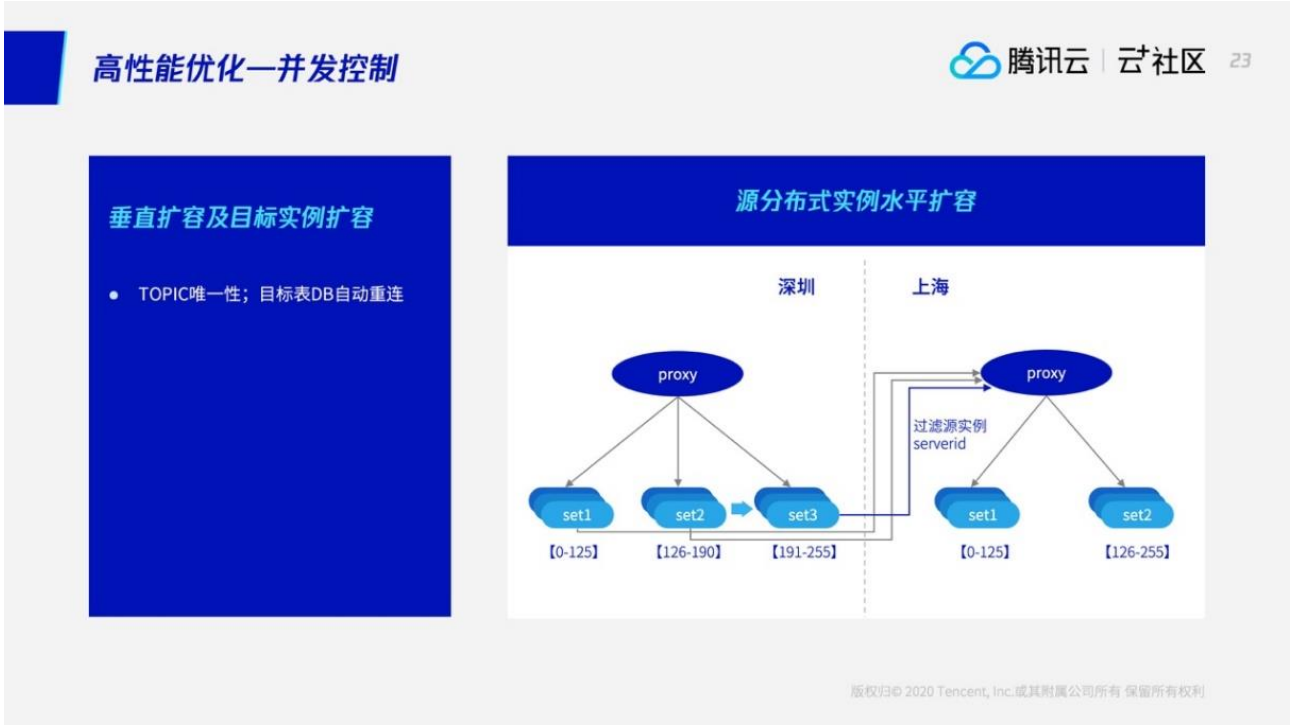
现在介绍一下消费者高可用保障。

消费者服务本身无状态，所有的任务下发通过 MetaCluster 实现，可以通过多台机器去部署同步服务，这套容灾机制通过 manager 进程来实现，也就是说当整个机器掉电，运营这个机器

的 consumer 已经不存活，这个时候这些 consumer 在 MetaCluster 上的存活节点的失效就会被其他机器的 manager 节点感知——认为另外一些机器的 consumer 已经不存活，这个时候就会把任务接管过来，并且在自己机器上重新拉起这些服务。

二是我们要做到同一个数据同步的链路不能在两台机器上同时拉起，这是一个互斥的要求。高可用机制会通过一些像唯一标识或者当前的分派节点做到，同一个数据同步的任务在被拉起的时候一定是发生在不同的机器上来实现漂移与互斥的操作，这个多机容灾保护总体上就是通过 MetaCluster 和监控的进程，比如说 manager 这样的服务进行协调完成，保证在机器级别灾难或者其他灾难情况下这些任务能够在十秒以内成功迁移到其他的存活节点上。

2.4.2 扩容场景的高可用设计



可用性一方面在灾难情况下需要保证服务可用，另一方面则是在扩容等数据库常规运营场景下保证这个数据同步有效且不会中断。

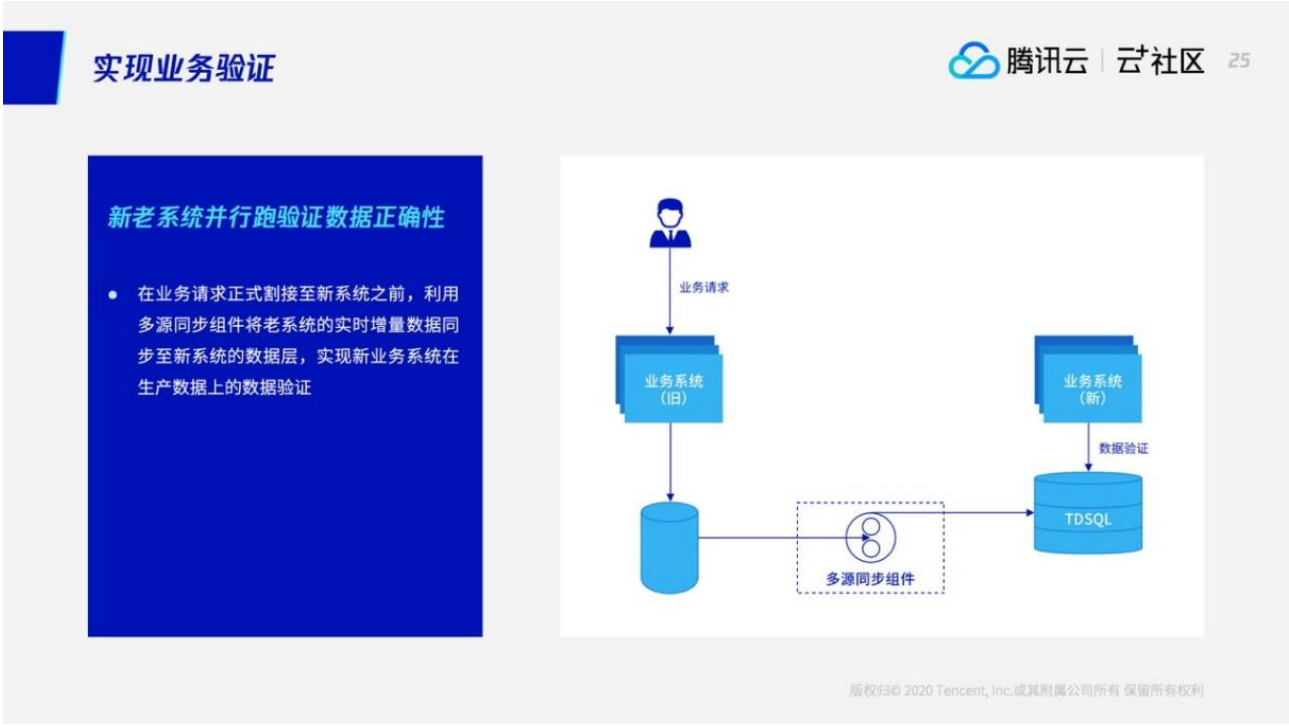
首先来说一下为什么在扩容的场景下，有可能造成数据同步异常？以垂直扩容来说，是相当于重新买了一套实例，然后经过数据的搬迁来实现数据同步的。这个时候我们是通过 TOPIC 唯一性来保证服务可用。扩容中从一个实例迁移到另外一个实例的时候，两个实例之间关系是什么？它们会往同一个消息队列上 TOPIC 去打增量数据。新实例打增量数据的起始点是什么？生产者在工作的时候会从消息队列上拿起始点，上一个服务结束的位置就是这个服务开始的位置。

关于水平扩容，则是新扩出一个 set 来，然后建立数据同步，对重复的分区进行切割和删除。如果现在有两套分布式实例进行数据同步，比如源端有一个分布式实例，这里面对应会有两个同步任务写到目标上，如果对其中一个分片进行水平拆分之后，就会拆出另外一个实例来，这个实例在拆分中有一个数据同步的过程，这个过程会产生问题——在 set 3 binlog 里面，会有一些 set2 上写的数据，并且 SERVER ID 跟 set 2 一样，如果单纯对 set 3 新扩出来的分片创建一个数据同步任务，将数据写到目标上的话，我们认为这里面可能会把 SET2 已经写进去的部分数据重复。这个 TDSQL 的数据同步服务针对水平扩容的这个场景也是实现了高可用保障，比如我们会针对扩容前的 SERVER ID 进行过滤，过滤水平扩容前 set 的原实例的 SERVERID，这个跟跨城的回环操作是比较类似的。通过这样的方式，来保证新创建出来的这部分增量数据开始往目标上写的时候，一定是这套扩容流程已经结束了，并且是有真的业务数据写到新扩的分片上来，不会出现同样的数据反复写两遍的情况。

3 TDSQL 多源同步金融级应用场景和最佳实践

上面我们解释了这个模块的特性、难点、解决的方式，现在介绍这些应用场景以及案例，包括 TDSQL 在多个客户场景中的最佳实践。

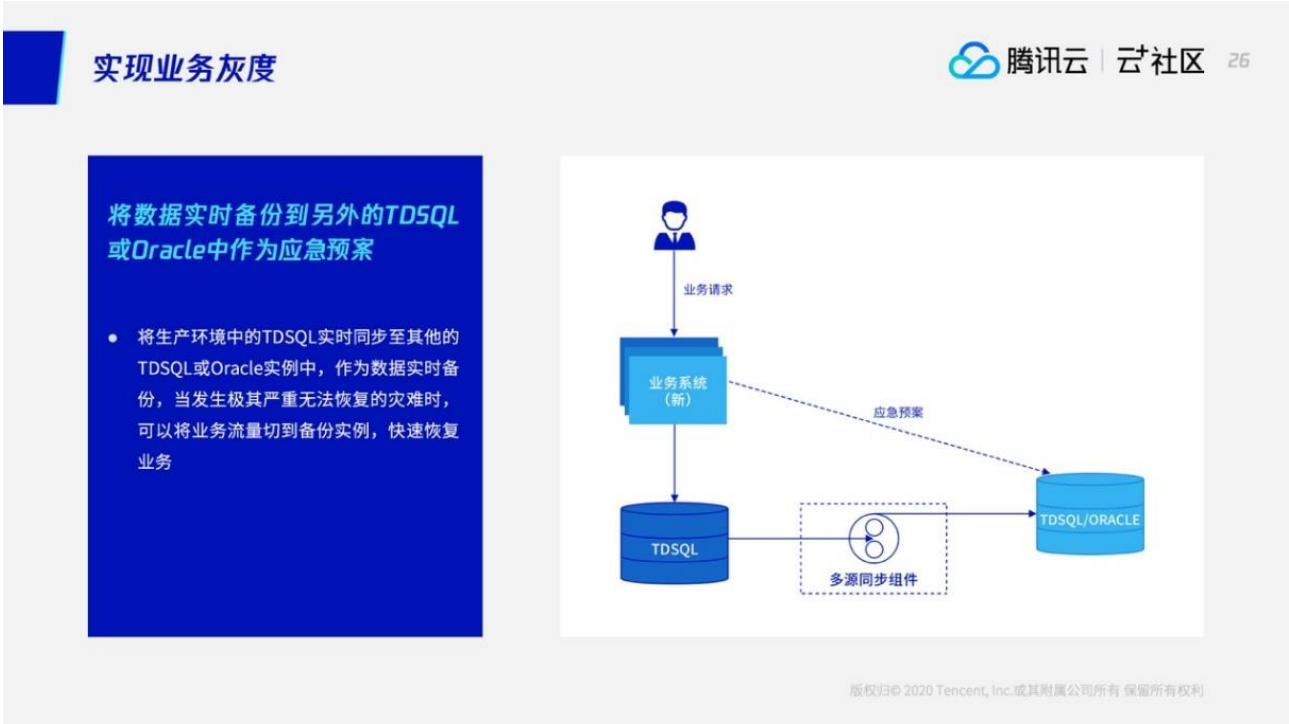
3.1 实现业务验证



关于实现业务的验证，比如可以对两个 DB 进行实时的数据同步。新的业务系统升级时，不可能直接把新的业务系统放到老的 DB 上直接跑，这时可以把新的业务系统先落到新的 DB 上做相关业务验证，或者在异构数据库的 DB 层变更上，把数据先同步到新的 DB 上来做业务上新

老系统并行跑的验证——一方面是保证了原先业务系统的安全性，另一方面也可以让业务切割更加方便，因为数据已经实时同步了。

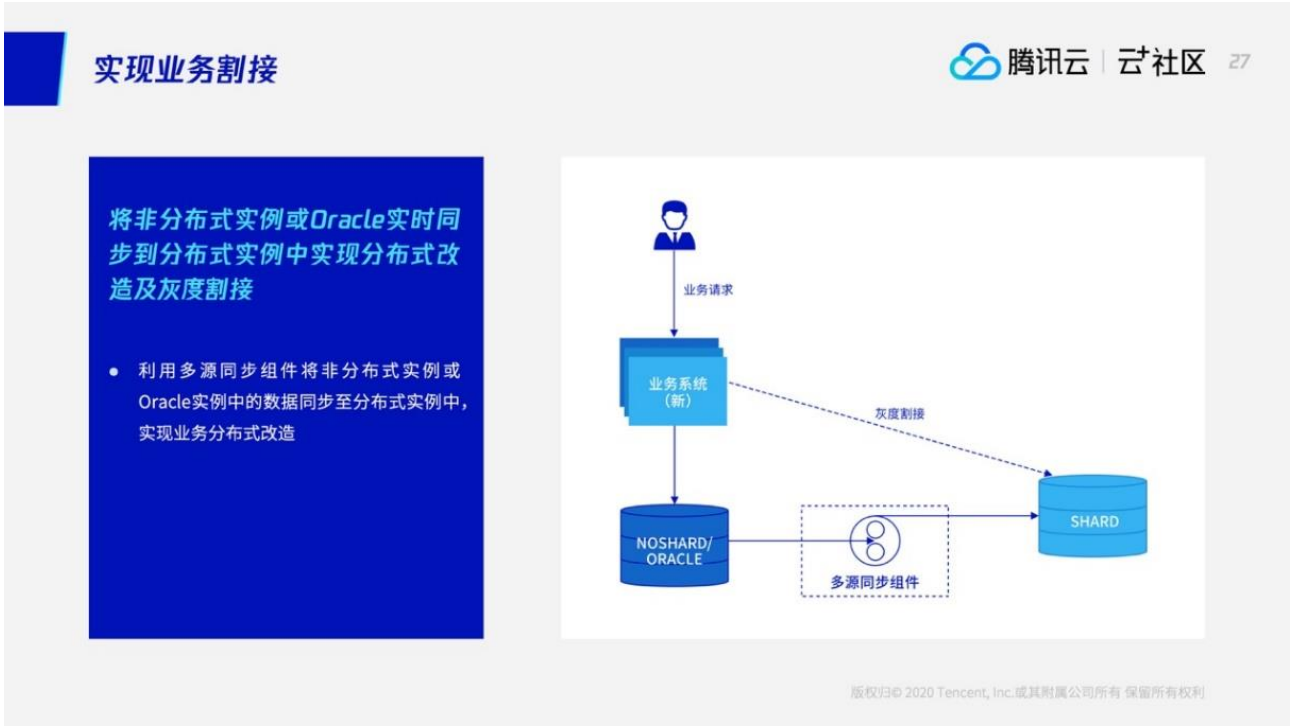
3.2 实现业务灰度



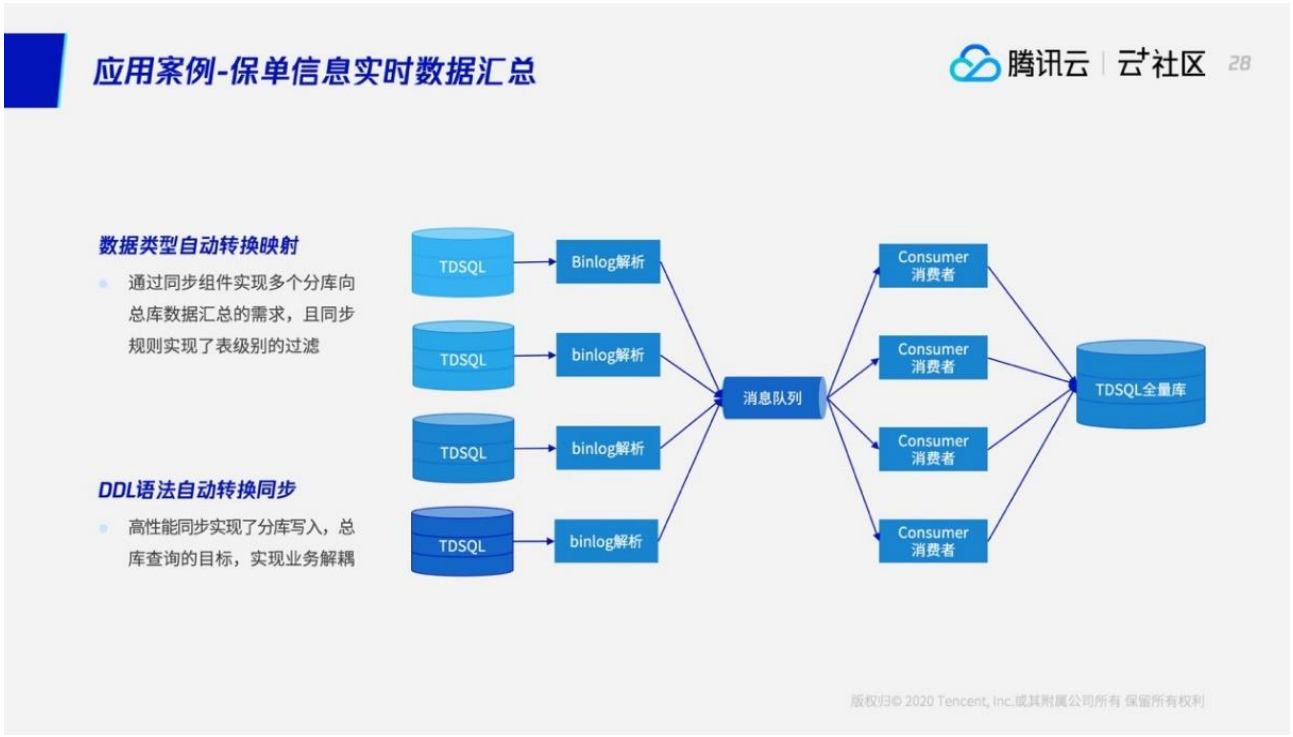
二是业务的灰度，以张家港农商银行的实践为例，在核心系统上线的过程中，我们把数据通过主键同步到 TDSQL 或者 Oracle 上，主库如果发生了一些比较小概率的灾难性实践，这时可以将这个业务系统迅速地切入到备库上，可以是 TDSQL 也可以是 PG、Oracle，相当于是实时的数据备份来形成备份的 DB，走备库上把这些业务拉起来在备份的 DB 上跑起来。

3.3 实现业务割接

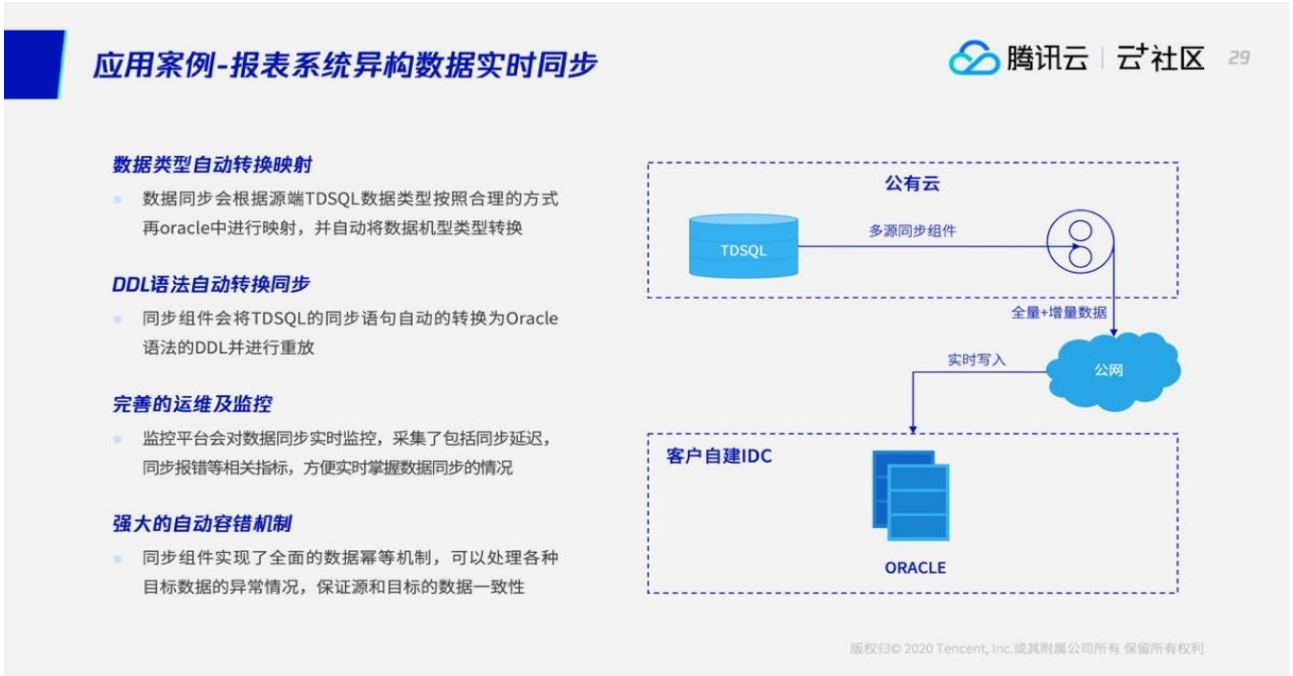
三是在实现分布式改造、进行业务割接过程中，可以将单实例的操作同步到分布式的实例上——这个过程先将数据通过多源同步组件同步到分布式的实例上，之后将业务的流量逐渐地从单实例往分布式实例上切，同时在分布式实例上也可以去相关业务验证，这也是我们的应用场景。



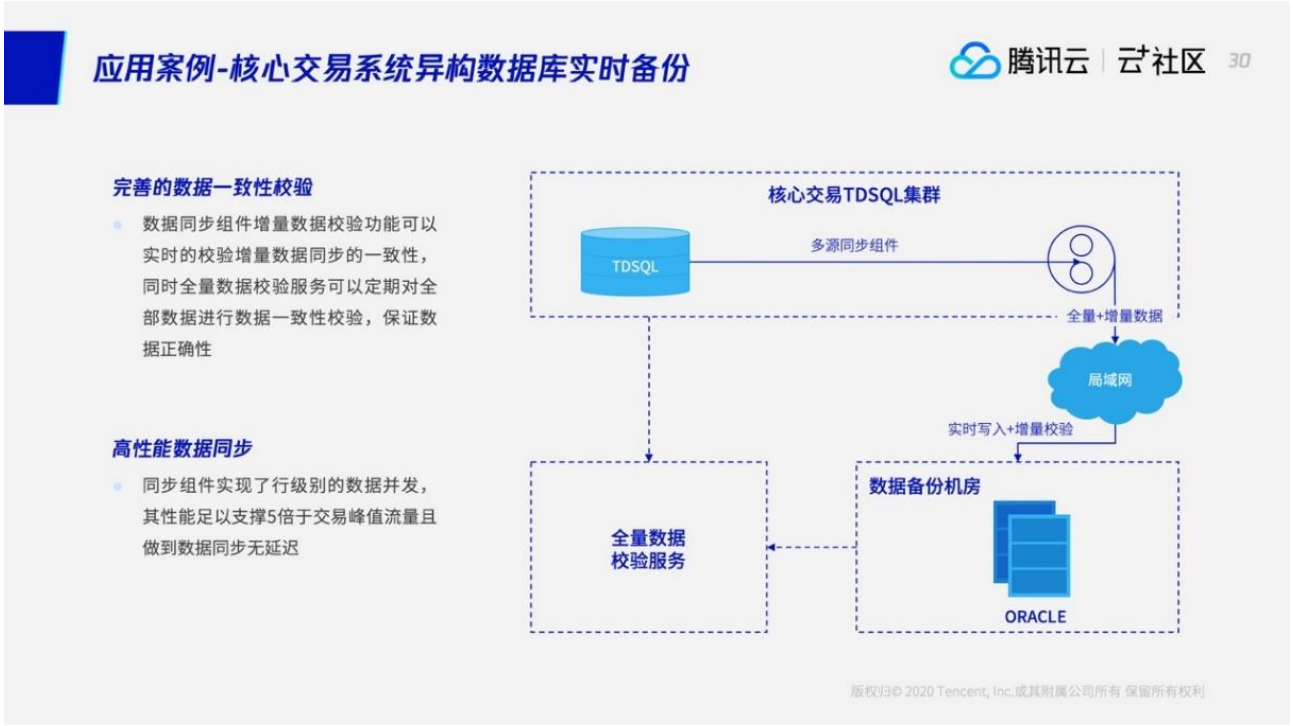
3.4 金融级最佳案例实践



1、我们可以通过多源同步对业务进行分布式的改造，将数据直接通过实时的同步将单实例往分布式的架构上迁移。比如说保险客户通过多个分库、多个分片区或者多个单的业务、逻辑上的划分，把这些数据通过 TDSQL 这套服务同步到存量库里面。



2、公有云上，TDSQL 的实例是通过公网实时写入自建的 IDC 里面，不管是 Oracle 还是 TDSQL——写到 Oracle 也支持。TDSQL 多源同步方案可以直接把 MySQL 的 DDL 转换成 Oracle 可以兼容的 DDL，实现云上的生产业务在跑的同时，本身之前 IDC 离线业务的老旧业务系统也可以在自建 IDC 的老实例上运行。



3、在张家港行实践中，核心交易集群是 TDSQL，数据同步通过内部的局域网，将存量和增量数据，写入到备份机房，同时也通过全量的数据校验服务保证数据源、目标是完全一致的来做风险控制。当核心交易系统如果出现一些小概率不可恢复的灾难时候，系统可以在短时间内将交易的服务全部切换到备份机房的 Oracle 上。

总结

以上介绍 TDSQL 对外分发解耦，数据分发、迁移、同步的能力，承载这部分能力的模块叫做多源同步模块，在应对金融级别或者金融场景客户的对外解耦、迁移的需求时候，所衍生出来的，高一致、高性能、高可用这"三高"的特性，并且介绍了针对这些特性我们是如何通过技术手段来实现的。

Q&A

Q：全量检测的效率怎么样?

A：单表的话全量校验一分钟可以校验 5 个 G 的数据，但是表和表之间本身是可以并发，也就是说单表一分钟 5 个 G，但是可以多个表并发去跑，这个上限就是机器本身的上限，比如说网

卡。这套全量校验也是通过主键去把数据值拉出来走内存去做 MD5。这套效率目前来看还可以，但是校验速度也不适宜过快，本身它去拉取数据时候对数据库也是有一些影响的，我们本身做校验的时候也会在业务低峰或者晚上去做。

Q：如何实现抽取 binlog 到消息队列不丢事务？

A：这就是前面介绍的，系统去做 GTID 连续性检测，我们知道在记录 binlog 的时候，GTID 一定是连续的，如果不连续则可判定认为它丢了，继而会到主机上补偿。就是通过这样的方式保证我们拿到的数据一定是没问题的。

Q：DDL 同步吗？

A：DDL 同步。因为 DDL 同步是会进行一次语法解析，解析出来相关的操作的，比如要改的哪些表、哪些字段，哪些类型需要改，针对目标的不同类型去做类型的转换，将这个 DDL 重放到目标上。

Q：原抽取和目标回放支持按条件抽取、按条件回放吗？

A：抽取我们支持按白名单去抽。为什么要支持白名单抽？我们原先的策略是全量上报到消息队列，有的时候会带来一些问题，比如说业务去清理一些历史的表，比如说一些流水的大表可能去做日志的删除等等，会批量去做一些操作，这个时候会产生一些疯狂往消息队列上打一些业务并不关心、同步并不关心的数据，这个时候我们也支持源端配白名单的方式，我抽取哪些库表的数据。目标重放更灵活一些，目标重放的时候是通过一个同步规则去配，同步规则本身支持精确匹配，同时也支持政策匹配。精确匹配就是一个表同步到另外一个表，精确匹配支持表名的变换，比如说我在原实例上表名是 A 表，同步到 B 实例上表名是 B 表，这里面强调的

是表名可以不一样，但是它的表结构是要一样的。也支持我可以匹配源端多个表同步到目标的一张表里面，也可以支持汇总的方式，就是表名在映射这一块也是比较灵活的。

第六章：DBA 上班也能轻松喝咖啡，数据库“自动驾驶”技术全解密

前言

“赤兔”平台是 TDSQL 提供的产品服务之一，它从管理员视角提供 TDSQL 的全部运维功能和上百项数据库状态监控指标的展示，让数据库管理员日常 90%以上的操作均可通过界面化完成，同时更方便定位排查问题。扁鹊系统是 TDSQL 面向云市场推出的一款针对数据库性能/故障等问题的自动化分析并为用户提供优化/解决方案的产品，它提供包括数据采集、实时检测、自动处理、性能检测与健康评估、SQL 性能分析、业务诊断等多种智能工具的集合。

在数据库日常应用中，常常会面对如存储容量和性能需求急速增长带来的性能等异常问题。对于引起数据库异常的问题 SQL，这时扁鹊可以通过一键诊断分析，帮助用户快速进行智能检测和分析，快速将问题定位，同时给出优化建议。在扁鹊的帮助下，DBA 可以从日常繁杂

的数据库运维工作中解脱出来。在支持腾讯会议需求量暴涨，数据库遇到性能问题过程中，扁鹊智能运维曾帮助 DBA 快速在亿条 SQL 中定位到了问题 SQL，并提供优化意见，将数据库的性能问题及时扼杀在萌芽当中。系统显示，经过优化，99%的 SQL 都消除了性能瓶颈。

“扁鹊”在迭代演进过程中沉淀了腾讯数据库实践中积累的海量运维规则知识库，可帮助 DBA 迅速排查日常常见异常，而结合腾讯云海量数据+机器学习能力，扁鹊可对未知问题进行主动分析检测，并告知客户，尽可能将大部分异常在发生之前就发出预警，将风险降到最低，提升 DB 持续可靠地服务各种不同业务场景的特性能力。

“赤兔”和“扁鹊”这一套组合拳既满足高星级业务的精细化运维，又能轻松应对大量的普通数据库运维需求，更好地帮助用户降低运维成本。

1 TDSQL 自动化运营体系

第一章节主要括三部分内容，一方面是对 DBA 日常工作进行梳理。第二介绍 TDSQL 自动化运营平台“赤兔”。第三介绍数据库后台如何实现自动化运营，介绍背后原理性的东西，帮助大家理解 TDSQL 是如何怎么实现自动化运营的。

1.1 DBA 的日常工作复杂繁琐？赤兔一键搞定

大家看到这个图会非常烦燥，这是日常 DBA 要处理各种问题：比如一个业务要上线，需要创造一个实例；比如机器要扩容，需要紧急申请权限；不小心弄错数据，需要赶紧回档；或者是

表结构需要变更.....这张图里就是 DBA 日常要面临的复杂量大的问题处理。从中我们也可以总结出两个特点：第一个每个 DBA 对于处理这些都会有一套自己的逻辑。比如我做一个 DDL，可能这个 DDL 可以这么操作，另外一个 DDL 要用另外一个工具，这些操作的方法不够标准而且不够稳定；也许今天用的这个方法明天用就不灵了——它的稳定性难以保证。

第二，我们做这个事情需要很多后台操作来配合，后台操作对于 DBA 来说非常频繁。一方面可能习惯于这种操作，但是这个方式带来的风险会比较大，我们敲下一条命令的时候有没有感觉很心慌，它的后果是什么？以往没有统一的平台来保证做这个事情是非常安全的。

所以 TDSQL 致力于解决这两个难点：

- 第一个是流程要标准化；
- 另一方面是效率提升。

我们不需要后台操作，而是前台点一点就可以解决 DBA 的大部分事务。

DBA 日常工作大致可以分为两类：一类是日常类事务，另一类是故障类事务。日常类就是平时业务的各种需求，DBA 通过一些脚本、自动化工具可以做的事情；故障类就是比如我们遇到一个难题，“数据库连不上”，“数据库特别慢”，需要看一下为什么。

我们简单看一下日常类。除了刚才提到的创建实例、DDL 在线变更，还有实例下线，这个虽然不常用，但是数据库运营较长一段时间中也是可能遇到的。此外还包括业务停运了将数据库清

掉然后存放其他数据，还有参数调整和调优——我觉得这个超时时间可能设得不太合理，业务侧要这么设置更改，等等。而扩容更是在所难免——业务数据量特别大，或者请求量特别多，实例撑不住了就要扩容。还有读写分离，还有重做备机、备份手工切换、在线 SQL……

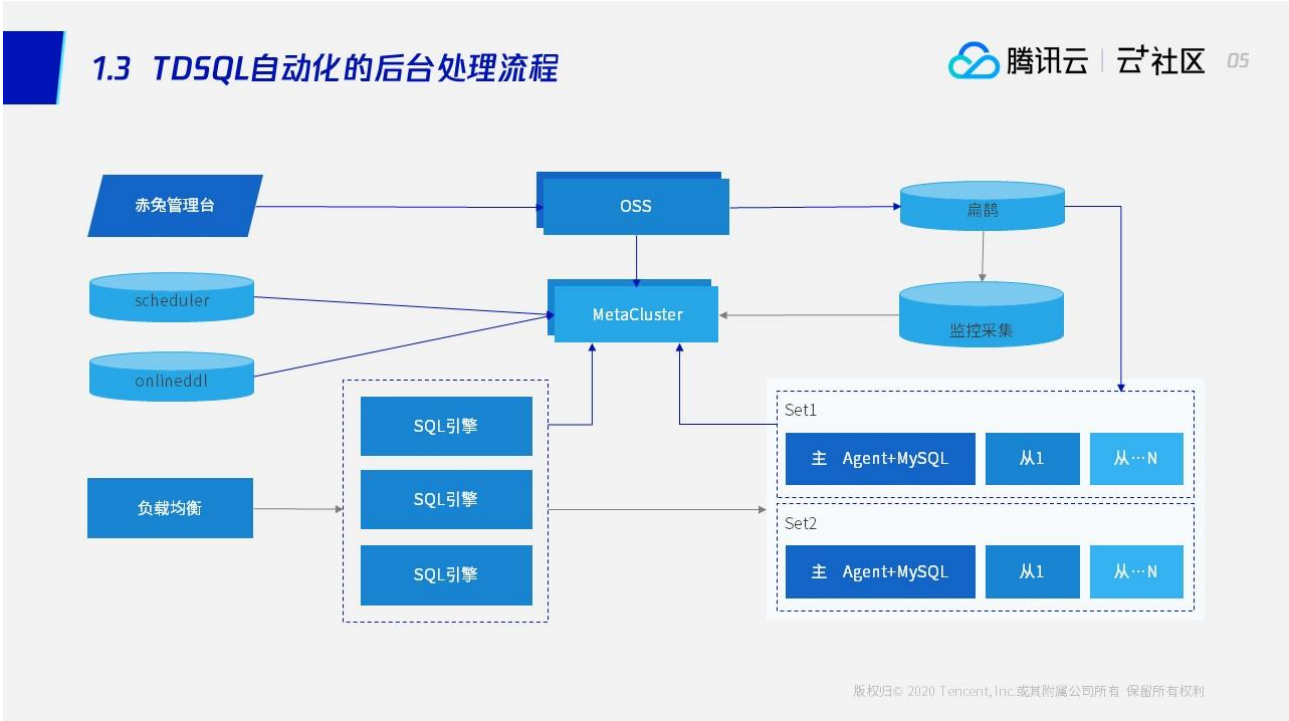
故障类指的是问题诊断类型：业务说 DB 连不上，帮忙看一下为什么；还有监控预警——如果 DB 有异常我们要自动发现这个问题，不然非常被动。系统分析——可能数据库运行慢了，但是还不影响使用性，看一下能不能有优化的空间；自动容灾切换就是保证用户的高可用；数据回档防止数据误删；日常巡检是针对性地发现一些潜在问题。

以上大概介绍了一些事务，但是这些并不是说完全代表 TDSQL 自动化运营平台支持的功能，这只是简单列举了几个比较重要的案例。言归正传，这些事情 TDSQL 是怎么自动化完成的？TDSQL 把这些功能呈现在数据库运营平台"赤兔"上，所有用户包括 DBA，在赤兔上就可以完

成以上所有的操作，而且全是自动化的；赤兔又会和 TDSQL 打通，赤兔会把流程以任务的形式发给 TDSQL，TDSQL 集群会自动帮用户完成这个事情，并且反馈给赤兔，然后用户就可以看到这个操作的结果。

所以赤兔平台整个流程就是把每一项日常类事务、故障分析类事务封装成一个个自动化流程，然后打通用户的需求和 TDSQL 后台操作的流程，帮助大家能够利用平台高效安全地解决日常工作中遇到的问题。

1.2 TDSQL-赤兔自动化处理原理



接下来介绍 TDSQL 后台如何完成自动化流程处理，包括介绍其中的自动化处理流程以及核心

的模块。这个架构图我们依次看一下：

用户的请求在赤兔平台以任务的形式发给后台的 OSS（OSS 是 TDSQL 对外的接口），OSS 又会做一个分发，把一部分的任务写在 MetaCluster 里面。MetaCluster 写成任务之后左边有 scheduler 和 onlineddl 两个模块会监控监听各自的任务节点，有新的任务就进行处理。

OSS 把问题分析类型的任务会直接发到扁鹊——TDSQL 自动化运营体系中的智能分析平台。扁鹊负责问题智能分析，它利用监控采集的数据——比如 DBA 的状态、主备切换等 TDSQL 监控数据，以及扁鹊还会实际访问数据节点，采集它认为需要支撑分析实例的数据。扁鹊在 TDQSL 框架里是一个新的模块。另外还有一个模块是 onlineddl，它专门独立处理 DDL 请求。除了这两种请求类型，其他的任务基本由 scheduler 模块完成。

右下角有一条线是 Agent——每个节点都由 Agent 进程模块来监控甚至完成辅助性动作，scheduler 无法直接接触所以 Agent 也会辅助完成刚才提到的流程。它也是通过 MetaCluster 获取自己需要的信息进行处理，处理完之后响应反馈给 scheduler 甚至其他模块。

所以这个图有三个核心模块来处理日常的流程：一个是扁鹊，这个是问题分析模块。onlineddl 处理 DDL 请求，以及 TDSQL 各个模块后面的角色以及任务的处理。TDSQL 自动化运营流程和框架就介绍到这里。

2 TDSQL 日常事务处理自动化

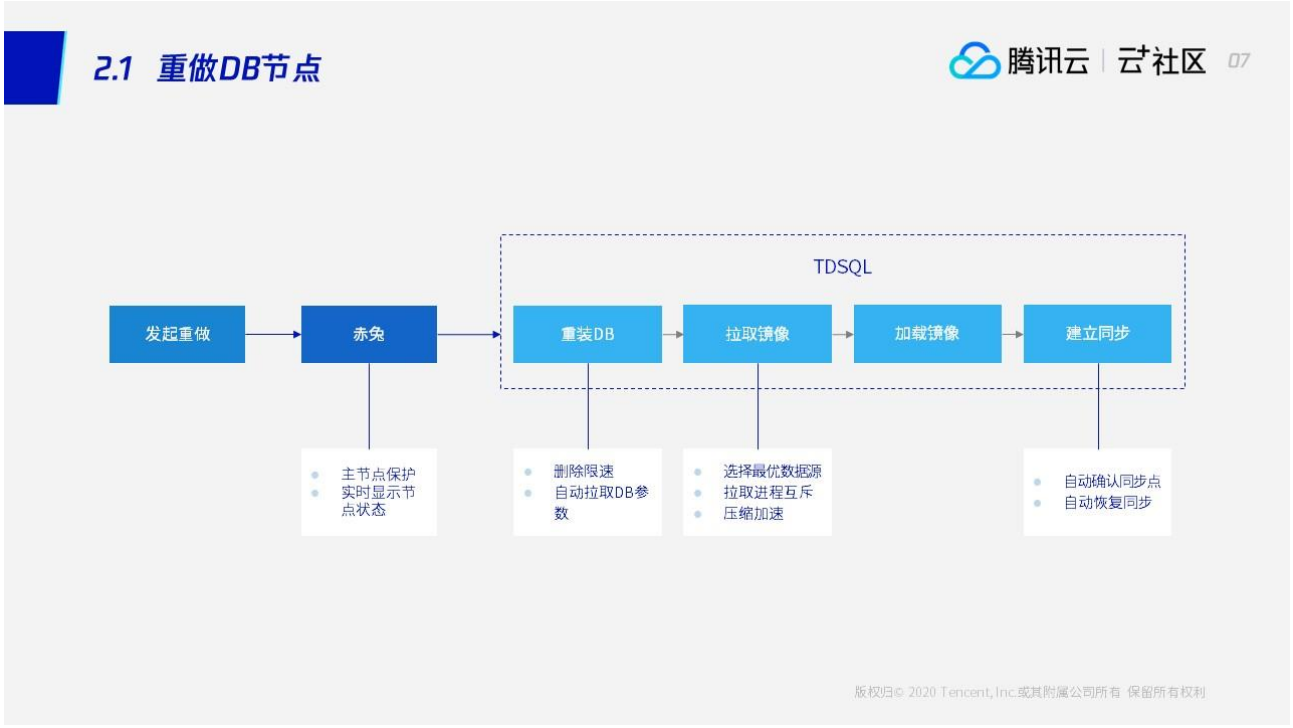
刚才讲过我们把 DBA 任务分成两类，一类是日常类，一类是故障类。先讲日常类的处理自动化，这个章节会选取两个日常非常常见的场景分析，分享 TDSQL 是如何解决这些问题的。

第一是重做 DB 节点功能，第二个是在线 DDL 功能，第三是 TDSQL 在自动化流程里如何做安全保障。

2.1 重做 DB 节点功能

第一节重做 DB 节点。大家可能会想为什么重做 DB 节点？这个场景比较常见，虽然它不是每天都发生，但是它隔一段时间就会发生，而且这个事情也是比较重要的。比如机器故障了机器修复可能需要一点时间，机器修复之后需要重新加入集群，数据可能就丢失；或者数据非常旧，已经追不上了，我们需要对这个数据节点进行重做；另外，如果卡顿无法恢复了我们就需要对数据节点进行重做，恢复节点的功能。

这个怎么做呢？我们可以看到这样一张图：



首先系统会发起重做流程——这个流程在赤兔上进行完成，赤兔会把任务发给 TDSQL 集群。TDSQL 集群针对这个任务有四个步骤：

1. 为什么要重做 DB 节点呢？因为机器上可能残留一些数据，我们需要清掉，删除限速。

2. 拉取镜像。无论是逻辑上还是物理上，都需要拉过来到节点上，作为数据的基准。
3. 然后是加载镜像。
4. 最后是恢复同步。

可能大家在日常的运维或者是处理事情的时候都是这个流程，它基本比较符合大家的习惯。不同的是可能以往较多的是手工处理，而赤兔在这里一键就可以发下去。

整个流程做了非常好的优化：

- 赤兔提供了主节点保护，因为假如是 1 主 2 备架构，为了防止出错，系统限制了不能直接在主节点实施重做。此外还形成了实时节点信息，例如这个节点是不是故障状态？延迟多少？……通过这个优化我们可以实时判断是不是正确的节点。
- 再看重装 DB 步骤。第一步会删除限速，而且这个数据往往是非常大的，几百 G 甚至上 T，所以我们要控制速度，如果快速删除会导致 IO 较高，而且一台机器上是多租户的架构，可能会影响其他实例的正常运行，所以要限速删除。另一方面，删除之后要把数据进程重新安装，安装的时候会自动拉取 DB 参数。因为 DB 在运行过程中可能很多参数已经被改动，安装之后的参数要保持和原来的参数一致，所以安装过程要自动拉取。而且，有时候参数列表会很长有十几个，手动操作是一个容易失误且工作量极大的事情。
- 另外，拉取镜像步骤，这是耗时最长而且是比较重要的一步，这里面做了三个优化：第一是选择最优的数据源，比如像一主几备的情况下，每个备机都有延迟状况，我们可能会选择延迟最小的，这个数据是最新的，如果是一备的情况则优先选择备机。而这个过程可能会对业务的读写有影响，所以要选择最优的数据。第二拉取进程——比

如同时做了很多流程不能同时拉取，一个是网卡流量会跑满，另外是由于有大量数据写入，就是 IO 负载比较重。所以要互斥，这样影响是最小的。第三是做压缩加速——这个地方主要是在于数据源，系统会把数据拉取的镜像进行优化压缩，压缩之后传到做的节点上；这样做的好处一方面是减轻网络的压力，压缩比大约是三分之一到四分之一。同时，我们可以加速，毕竟传输比较小，比如压缩四分之一传 100 兆就是传过去 400 兆的数据，对提升效率非常有帮助。

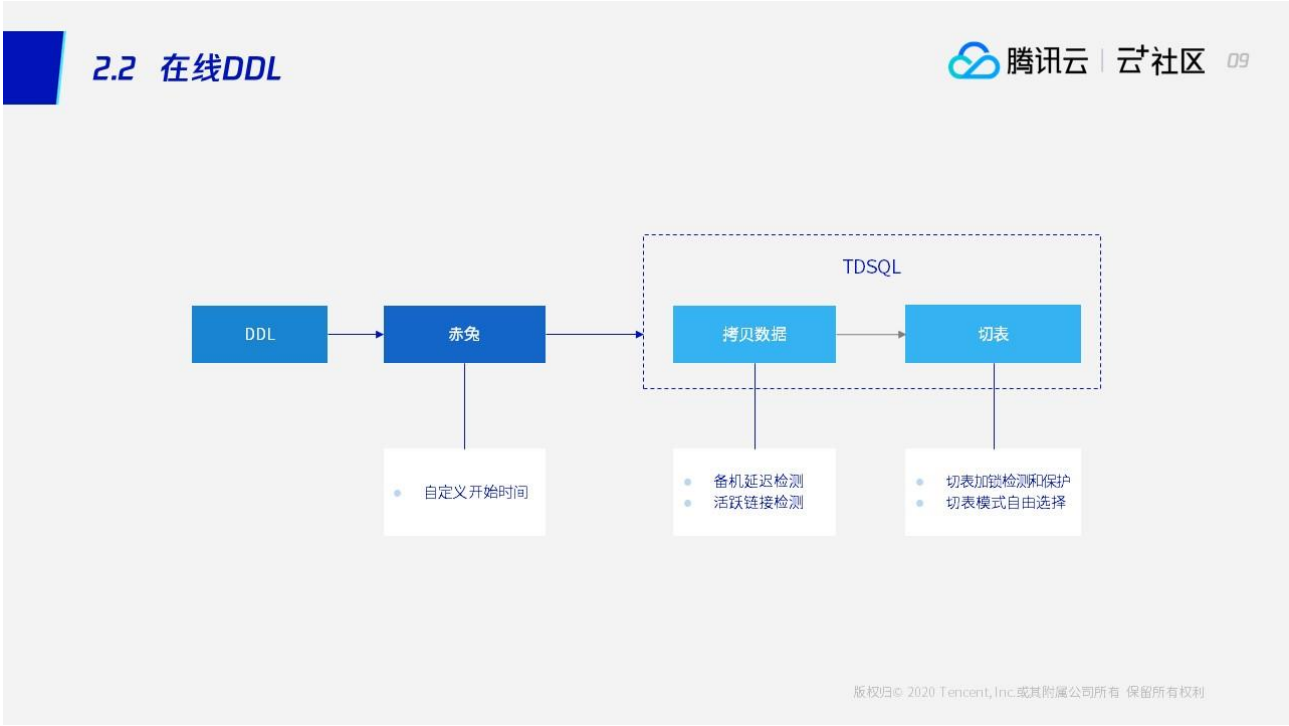
- 最后步骤是建立同步，这里主要是确认同步点以及恢复同步，这里 TDSQL 会帮助你自动去做。

所以大家看到，整个流程对用户来说只需要在赤兔上点击"发起重做"就可以自动完成整个流程，不需要过度介入。

上面的图是一个案例实录：可以看到 DB 节点有三个，重做节点就在右下角，点“重做备机”按钮进入流程，这个页面可以实时显示两个备节点状态，我们可以看到第一个备节点延迟非常大，这个就是我们需要重做的异常节点，防止大家误操作选错节点。提交完之后过一段时间会告诉你“重做成功”。整个流程就结束了。

2.2 在线 DDL

我们再看在线 DDL 功能。



操作 DDL 非常常见的应用，尤其是在业务变化非常频繁、表结构频繁变化的场景中。为什么要提在线 DDL？如果是面对一个普通的小表，那么可以直接做 DDL，但是如果面对的是一个

数据量比较大的表，比如几十 G 甚至几百 G，要做一个表结构变更怎么办？这个时候很有可能影响业务请求，所以我们提出要做在线 DDL。

在赤兔上，在线 DDL 也非常简单：我们在赤兔上提交请求，然后传输到 TDSQL 模块实施，一共两步—熟悉数据库的应该比较了解，这两个步骤一个负责拷贝数据，随后表数据同步完之后再进行切表——新老表进行切换。

TDSQL 在这个流程里做了哪些事情？赤兔可以自定义 DDL 的开始时间。那为什么要做这个事情？DDL 虽然是在线，但是也会涉及到拷贝数据，特别是在业务负载比较高的时候会对业务有影响，我们希望在业务不繁忙的时候——业务一天里的周期一般是固定的，即在业务低谷的时候做这个事情，因此平台支持可以自定义时间，比如白天发起任务，晚上一点钟业务低估时再正式运行任务。

拷贝数据。刚才提了拷贝数据可能会对业务有时耗影响，所以 TDSQL 会检测这两个指标：备机延迟检测与活跃链接检测——超过了会暂停，直到恢复正常时再继续。这两个指标在前台也可以自定义，不过系统有推荐的默认值，一般不需要更改。

另外是切表流程。这个流程涉及到新老表的切换——把新表切成老表的名字。

切表流程涉及两个功能应用：切表加锁检测和保护，以及切表模式自由选择。第一个，日常中我们很常见的场景是，切表前有一个大的事务在访问这张表，查询了半个小时还没有跑出来。这个时候要做切表操作就获取不到元数据锁，同时又阻塞了后面的业务请求，后面的业务请求

会等前面的切表流程才能继续。所以 TDSQL 根据这个场景做了一个切表加锁保护——就是说，我们在知道要切表之前，要先看一下请求里有没有这表的大查询，有的话就暂时不切，先让它完成，我们不会把它直接杀掉。开始切的话时间非常短，对用户最多影响一秒钟。如果正要发生切表时，正好有个请求抢在前面让切表无法执行，那系统就自动超时，不影响后面的业务 SQL。回到数据同步状态隔一段时间又会发起切表，而且间隔时间会越来越长，直到可以完成整个 DDL 操作。

另一个自由选择功能意味着，切表模式可以选择手动和自动切表：自动完成也有加锁保护；手动切表就是拷贝数据完成之后不立即切表，而是 DBA 可以手工在前台发起切表操作。

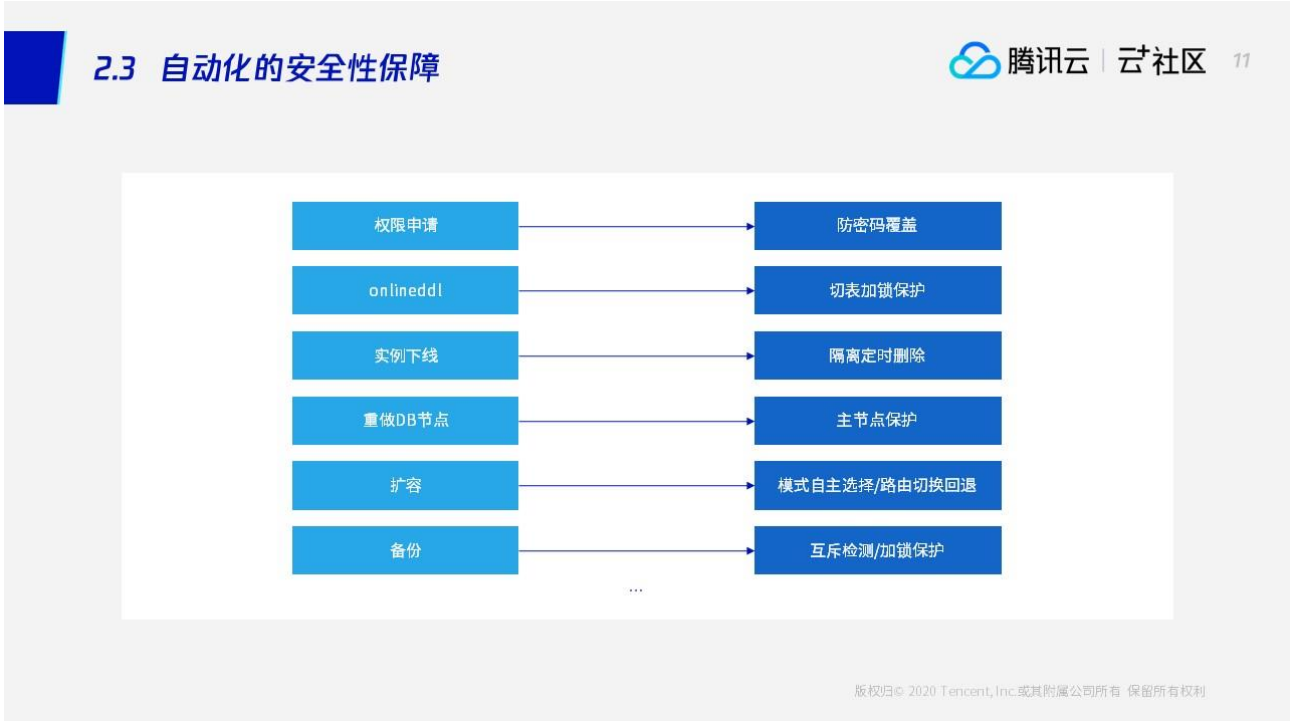
我们看一个在线 DDL 的例子：

左图就是在线 DDL 页面，通过页面可以看到表结构，大家点击“编辑”按钮就可以修改字段和索引。右图的页面里面我们可以看到，刚才提到的参数设置都可以自定义，也可以选择默认值，点击“确认”后整个过程自动完成。如果选择手动切表，可以选择合适的时间完成切表操作。

2.3 TDSQL 自动化运营体系的安全性保障

我们看一下安全保护机制，流程自动化了，我们更要保证这里面每一个流程都是安全合理的。

安全性保障不仅限于这些，PPT 里选取了部分常见的场景。



- 权限申请非常常见：如果有用户已申请了密码 A，而且程序已经使用这个账户在运行

中，如果再申请这个用户而使用不同密码，系统会自动检查，所以不让老的密码被覆盖掉。

- 第二类是 onlineddl 自动保护。
- 第三个是实例下线，这个我们做了一个隔离定时删除功能。我们可以先进行隔离，隔离之后访问数据库的请求都会被拒掉，但是隔离状态是可以及时恢复的，所以我们相当于放在一个回收站里，数据还没有清掉，但业务访问不了。过一段时间定时清理，比如 7 天之后（这个时间长度是可以自定义的）没有业务反馈，则可以清掉，安全下线。
- 重做 DB 节点，在这个环节 TDSQL 提供了主节点保护的功能机制。
- 扩容，会涉及到 TDSQL 扩容：把数据切换到另一个数据节点，新数据节点在做数据的过程中是没有影响的，因为业务的请求还是访问老的实例节点，但是在最后一步路由切换，是在扩容流程里唯一会对业务有影响的，因此 TDSQL 对这个流程做了保护——可以自主选择切换模式，就是可以手动切换或者自动切换。手动切换业务可以实时观察，有问题可以及时反馈；路由切换可能要经过几个步骤，中间流程如果有失败会自动回滚，不对业务有什么影响，所以也是对扩容的保护。
- 备份：不需要干预，TDSQL 会自动备份，备份过程中也有互斥检测。最后也会有加锁机制，虽然比较短，但是有业务请求比较长的也会加锁失败，这里加锁时间如果超过一定时间会自动停止备份，备份会择机再次发起，不对业务有影响。也就是说，备份不影响备机的正常运行。

总结来说，TDSQL 对自动化运营提供了很多安全性保障措施，保证每一个流程在关键节点，特别是在流程里可能会对业务的请求、访问有影响的环节，都做了很多的保障。这个也是

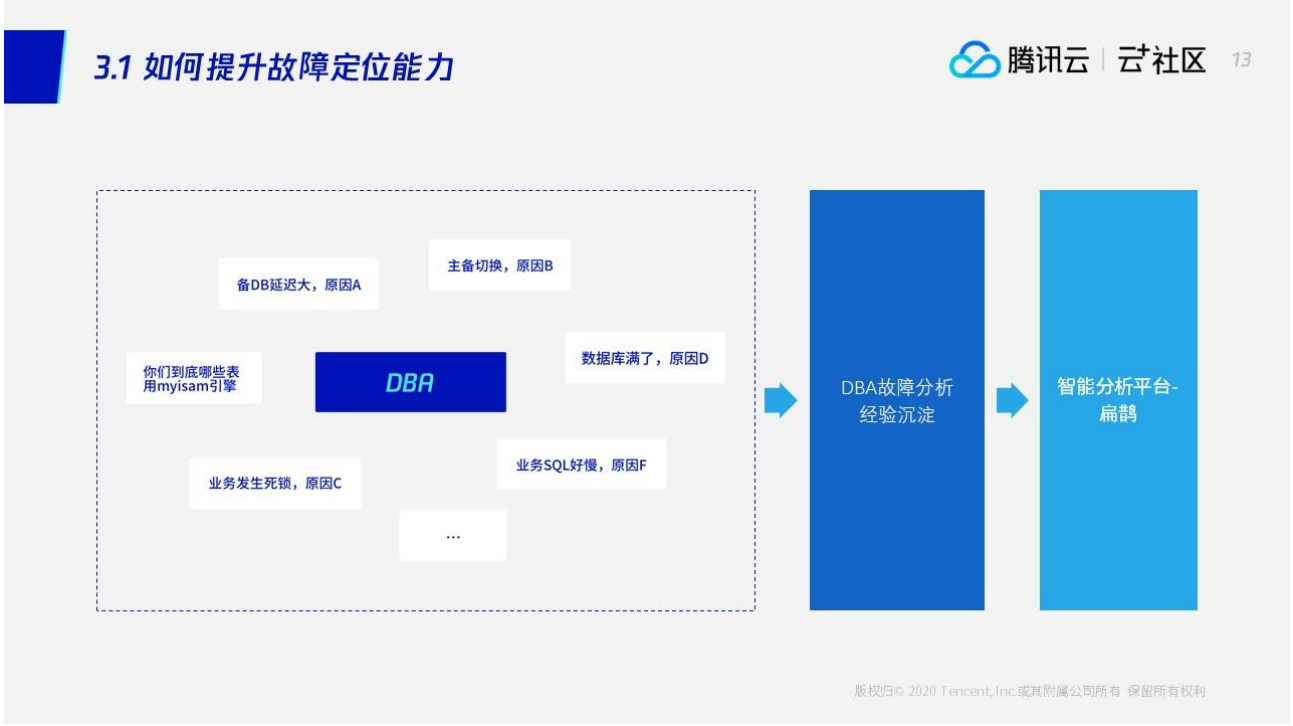
TDSQL 运营这么长时间各种经验的总结，和不断优化的结果，所以大家可以放心使用。

3 TDSQL 智能化故障分析平台

下面我们进入第三章节：TDSQL 故障分析自动化。

刚才我们分析到，DBA 日常遇到的第二类问题主要是发现问题的时候如何定位分析。

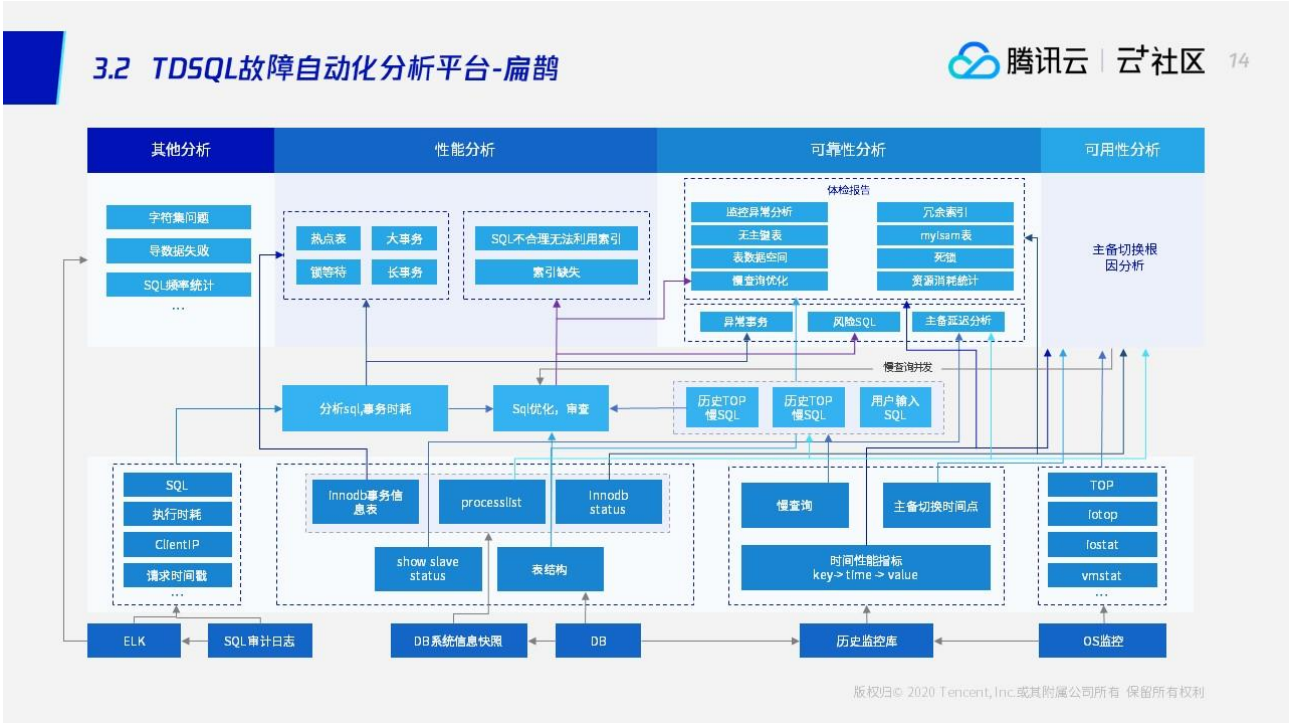
3.1 TDSQL “扁鹊”如何帮助 DBA 提升故障定位能力



DBA 在面对故障的时候往往非常烦燥，各种问题非常多。大家在定位问题的时候归根结底有几个难点：第一个是 DBA 的经验能力对问题定位影响非常大，很多优秀的 DBA 都是通过不断的故障积累经验才能成为优秀的 DBA。第二个，我们定位的时候往往要通过各种认证登录后台，查看各种指标综合分析，这样效率非常低。其实很多的问题都是重复发生、重复发现，但是我们要重复定位和解决。

所以我们希望通过"扁鹊"平台把 DBA 故障分析经验沉淀下来，沉淀到赤兔智能运营平台，让它来自动分析、发现这些问题，为数据库用户和 DBA 带来高效便捷的体验。如果有新的思路、新的问题出现包括新的原因，我们都会持续沉淀到这个平台来做循环，这个平台会逐渐变得非常强大。

3.2 TDSQL 故障自动化分析平台“扁鹊”



扁鹊主要有四个主要功能：可用性分析、可靠性分析、性能分析和其他分析，其他分析也是在不断强化。可用分析主要围绕主备切换场景展开；可靠性分析主要以较大范围场景的体检报告来分析数据库目前的问题，可以对 DB 状态进行了如指掌的分析。

性能分析针对的场景就是数据库运行变慢了。我们可以大概总结为这几类，比如热点表、大事务、锁等待、长事务等，下面一层可以分析 SQL 事务时耗，包括对 SQL 的检查优化等，来看 SQL 有没有问题。

下面这一层就依赖数据层，上面的模块是数据的采集方式，最下面是数据的存储方式，比如审计日志对 TDSQL 的数据都有采集，而 DB 的状态包括 DB 的快照、事务信息、隐藏状态等；

同步信息包括表结构信息等，例如表结构不合理要先摘出来看看哪不合理。

还有是负责监控 DB 的模块，包括赤兔前台能看到的各种指标都在里面，还可以看到慢查询、主备切换的流程等，包括切换成功与否、切换点，切换时间点等关键指标。

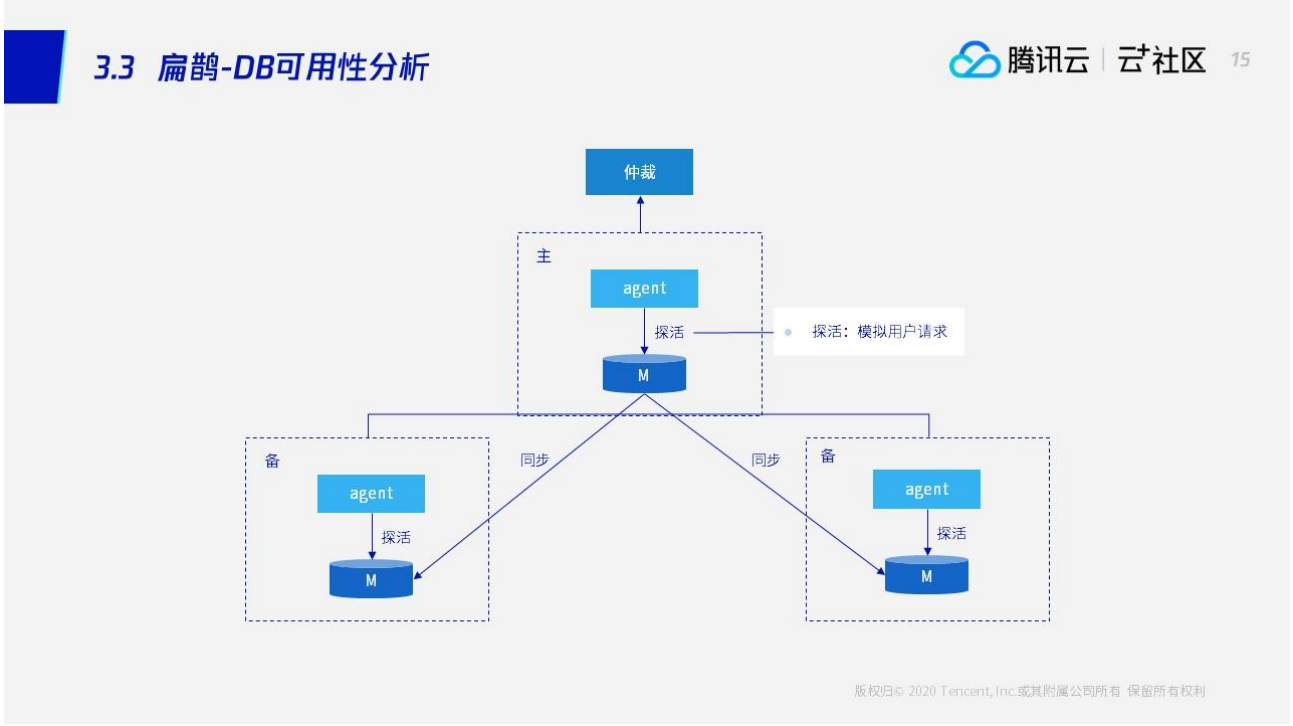
右下角就是操作系统状态，包括 IO 内存、CPU 等的状态。

这张图从上往下看，首先是可以分析可用性由哪些原因造成，然后有个逻辑分析层。最后是新增数据接入层。通过这个图大家可以直观了解到扁鹊工作方式以及内部逻辑。

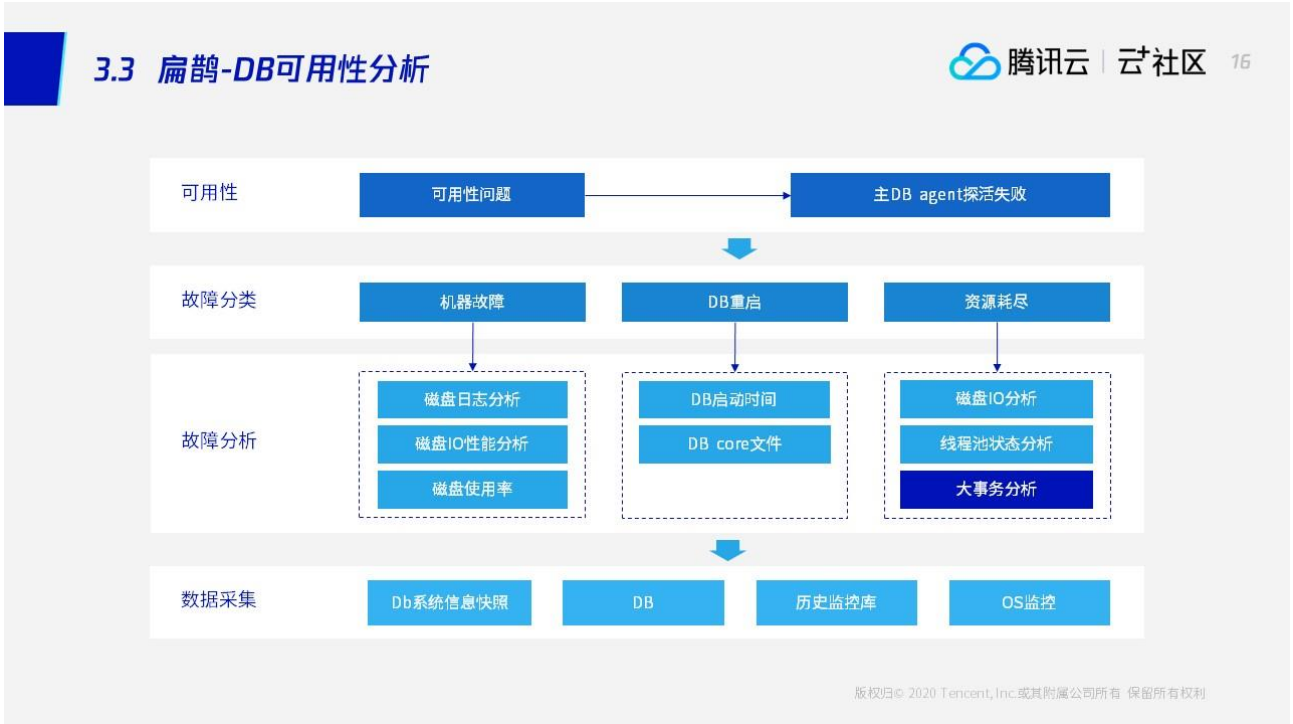
接下来针对扁鹊的三大功能举例看一下它怎么做故障分析。

3.3 扁鹊数据库智能分析平台之 DB 可用性分析

TDSQL 分布式数据库通常是一主两备的架构，TDSQL 的 Agent 会周期性地对 DB 做探活。探活是指模拟用户的请求，建立 TCP 连接后然后执行查询和写入，比如监控表的查询，模拟用户的请求看是否正常。TDSQL 的可用性在于探活异常，如果认为 DB 发生异常，就会自动发起切换流程。



这是一个自动化流程，但是切完之后我们要看一下为什么引发了这次切换。这个可归结为为什么切换时间点发生了探活失败。



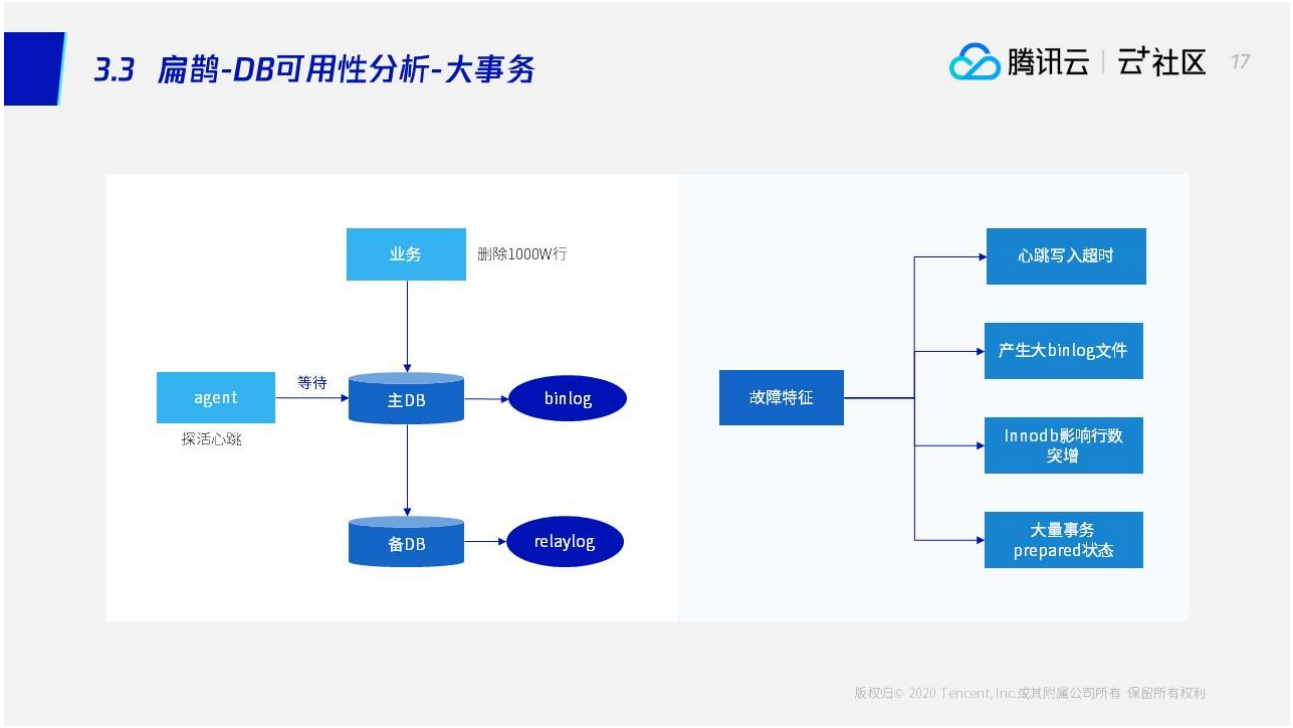
可用性问题归结为主 DB　Agent 探活失败，大致可以分为三类：磁盘故障、DB 重启和资源耗尽。我们分别看这三类故障的原因：

- 磁盘故障运行时间长会有故障，我们要分析日志信息来看磁盘在主备切换的时间点以及切换有没有发生异常。一可以通过分析 IO 性能来判断——磁盘在故障特别是 SSD 性能耗尽的时候会导致 IO 异常，IO 非常高但是读写性非常差，每秒都执行不了几次读写，而且服务响应时间非常长。
- DB 重启依托于实时上报 DB 启动时间。只要启动时间发生变化我们认为 DB 发生了重启，这个也可作为 DB 重启故障原因。
- 资源耗尽这一类故障分析中，磁盘 IO 分析和刚才的 IO 是有区别的，磁盘故障 IO 是因为磁盘故障，由正常请求引起，这个可能是因为做了大的更新查询；此外还包括线程池状态和大事务状态等场景分析。

3.3.1 DB 可用性分析：大事务问题

我们分析主 DB 的请求，刚才提到了有 Agent 负责探活心跳周期性，同时也会有业务的请求——假设这个地方一个大表删除了 1000W 行……这些问题 TDSQL 结合成一个组提交。提交后可想而知会产生很大的 binlog，据我们了解的可能会产生 5G、10G 甚至几十 G。因为一个事务必须在一个 binlog 里，所以非常大的 binlog。产生大的 binlog 又会产生哪些因素呢？整个流程下来会发现耗时非常长。而探活的时候会有时间频率限制，超过时间就会认为失败，探活失

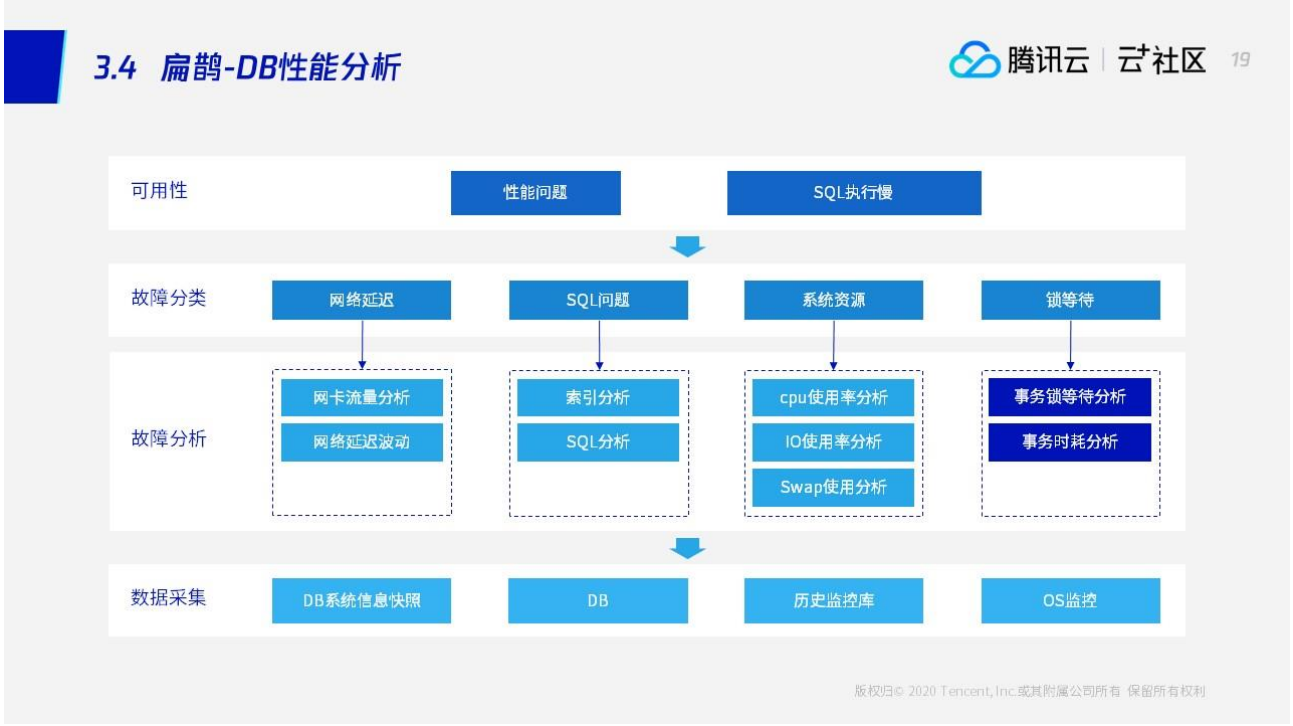
败提交了一分钟必然会发生主备切换，因为可能很多心跳已经上报仲裁主 DB 已经故障。



我们通过分析可以看这种故障类型有几个特征：心跳写入超时、产生大 binlog 文件、Innodb 影响行数突增，以及事务处理 prepared 状态。

通过提取出的这四个特征——四个特征都是符合的，我们可以认为是由大事务引起的，从而导致切换。TDSQL 自动运营平台针对主备切换的故障流程做分析，可以一键分析生成一个分析报告。

3.4 DB 性能分析



我们可以根据经验大概分成四类，网络问题、TDSQL 本身问题、资源性问题和锁。网络问题比较容易理解，网卡流量、网络波动性；SQL 问题包括索引分析、SQL 分析；系统资源比如 CPU、IO、Swap 活跃度和锁等待的问题。需要分析的内容也是依托于采集数据来完成操作。

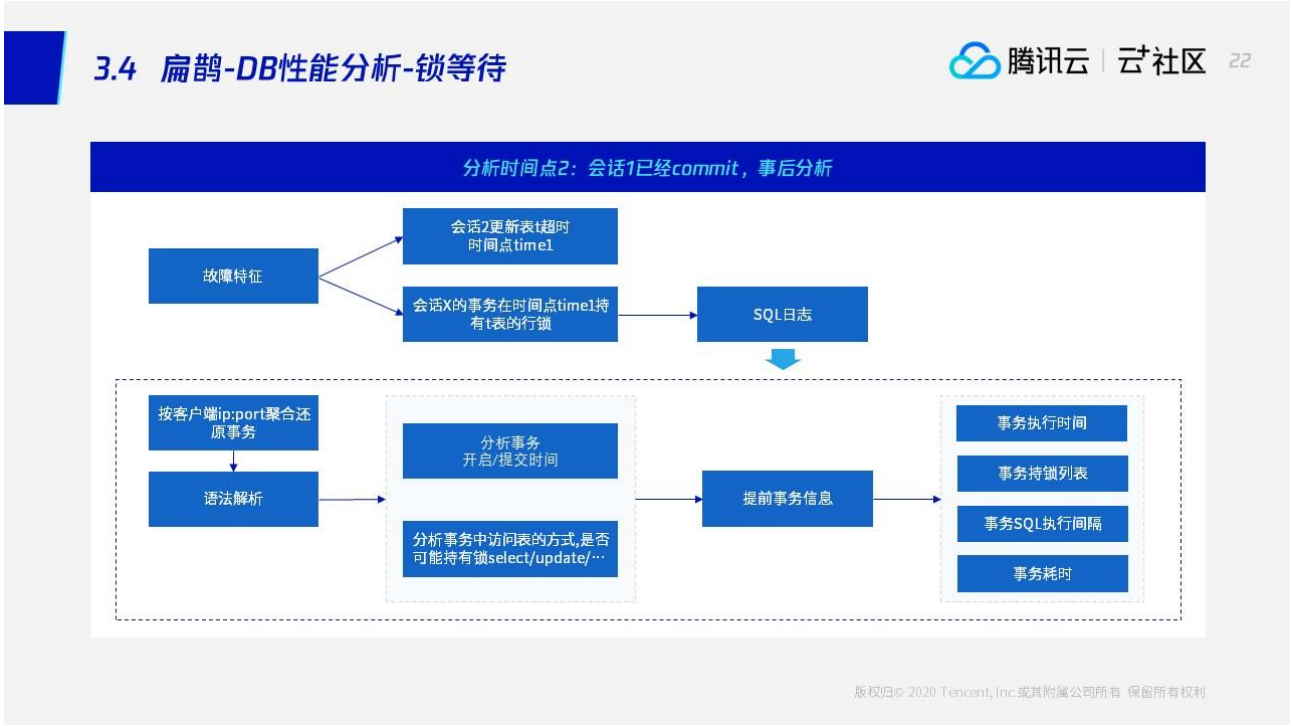
3.4.1 DB 性能智能分析：锁等待

接下来看锁等待引起的 DB 性能问题设计思路。

右边这个看似有两个对话，这是典型的锁冲突对话：首先是 begin 开启事务。在第一秒的时候会话 1 开始更新，在第二秒的时候会话 2 也要更新。又过了一段时间对话 1 完成了——锁可能停留在两个时间点，这个时间点极有可能出现的情况是 DB 处于锁的状态。这里简单举了两个会话，几千个同时在等待某人执行 SQL 的时候给锁住了，现场 DB 分析的话，可能会经历这样一个流程：

第二个时间点，会话已经提交了，会话 2 时间比较久，在提交的一瞬间 SQL 就执行成功了。会话 2 整个已经超时，时间点二业务场景下 1 个小时之前会话超时了，赶紧看一下当时是什么情况。时间点 1 非常简单，熟悉 DB 的比较了解这种方法，底下有表就是事务表、锁等待表，通过关系我们可以查出来会话 1 没有提交，把会话 2 给堵塞了，所以这种场景最容易分析。平常做把三个表的关系记录一下去查询，看看到底是哪个会话有问题然后把会话 1 杀掉，在业务看一下是不是有问题。

而如果在赤兔平台，这些可以一键完成，在赤兔上点"实时分析"可以看到现场案例是什么，我们有建议把会话杀掉然后可以恢复正常，当然这个之后要找业务看一下事务为什么这么长时间而不释放。



第二个场景，会话 1 已经提交了，事后分析没有故障现场怎么办？我们可以看一下这边的故障特征，这类的故障特征是会话 2 更新的表超时了，或者结余时间比较久。它的时间超时在 T1，或者很久在 T1 执行完了都有可能。

我们的目标是找出来会话 X，到底是哪个会话把会话 2 堵塞了，首先会话 X 在时间点是持有 T 表的行锁，只有它具备这个条件才有可能把会话 2 堵塞住。我们可以通过 SQL 日志分析怎么把会话找出来。刚才提到了 SQL 有所有信息，其中就是客户端 IP，由于 SQL 日志有很多请求

是交错在一起的，比如开启一条事务执行一条 SQL，又开始另外一条执行 SQL，是很多事务连接的请求交错在一起的，我们很难分析出来一个事务的关系。所以我们第一步要做的是先要根据客户端的 ip port 聚合还原事物，按照维度做聚合然后可以知道 ip port 所有的执行数据。我们会对 SQL 进行依法解析然后提交时间。

另外，我们还想提取事务中间可能有各种查询更新操作，这些 SQL 到底是持有哪个表的锁，它没有锁的话肯定对会话 2 没有影响。紧接着我们得出这样的结论：首先是事务的执行时间，什么时候开始、什么时候结束。事务的持锁列表有没有可能造成会话 2 的锁堵塞，还有 SQL 的间隔时间，这个也是帮助业务去看两个事务之间间隔这么长时间，中间到底发生了什么把会话 2 锁了，是否合理。

还有 SQL 的耗时，这里面也包括每个 SQL 的耗时，有个 SQL 执行时间非常长，确实把会话 2 锁了，我们要找出来看看为什么执行时间这么长。

通过这些信息表 T1 时间点可以得出来是由会话 X 对会话 2 造成的锁定，然后再看会话 X 为什么要执行得不合理，至少看一下业务是否正常。我们再看一个案例，在时间点包括锁来看是哪个会话引起了锁的等待，然后我们会看间隔时间包括信息，用来定位的会话引起了锁等待。

3.4.2 DB 可靠性智能分析

DB 可用性分析，是指对数据库的状态可以提前了解，包括：系统状态、表空间分布、索引、死锁诊断、锁等待诊断、慢查询分析、DB 状态检查等等。我们可以看到这样的案例：数据库评分非常低，做了分析之后看到它是哪些问题——CPU 空间过度？任务非常多？下面的状态信息都可以帮助大家来看 DB 到底有没有问题，是否可以提前发现问题并解决。

平时对重要 DB 觉得运营状态不太了解就可以做一次诊断。当然这个系统也可以做自动诊断，也支持每天针对重要 DB 每天出一个预报。这个是 DB 的可靠性分析介绍。

总结

本章节主要分享了几部分内容：

总结而言，TDSQL 自动化运营体系可以帮助大家把日常中非常烦琐、需要手动操作的事情进行标准化、自动化、智能化，一键完成，减轻大家的负担。因为我们做了标准化沉淀，很多需求不需要 DBA 自己做，基于 TDSQL 运营平台，业务也可以直接解决。日常的一些操作可以通过赤兔的权限管理放开给业务。

Q&A

Q：是否有多活机制?

A：我们有多活机制，并且有强同步复制机制，保证数据一致。。

Q：online ddl 是否保持两个数据一致?

A：online ddl 是基于工具进行改造。简单而言就是会实时把更新类操作写进新表，然后分批把原本的数据覆盖到新表里。数据覆盖完之后两个表的数据一致，触发器可以把业务请求的数据实时同步，直到切表流程结束。

Q：对于大事务如果刚开始执行没有注意到等过了十分钟才发现事务没有执行完，这个时候可以做哪些处理?

A：如果终止会话事务也会持续比较长的时间，如果一直等事务持续也是一个比较长的时间。这个时候我建议把它杀掉，如果不放心，刚才提到的有大事务极有可能触发主备切换。我们会强制做切换来保证高可用。

Q：死锁检测是通过定位吗?

A：这里不是死锁，是锁等待，是两个事务都无法执行，我们刚才的例子是可以提交，只是长时间未提交的事务。会话 2 一直在等，在某个时间点超时了，这个时间点 2 之前肯定是有会话把表锁住了，我们的目标是要找出来在会话 2 之前某个时间点的某个会话是否持有表的锁。我们根据表信息和时间点通过引擎日志，这个日志里记录了所有用户 SQL 执行信息，可以通过这个表的信息来分析锁超时的前后，主要是前有没有会话持有。当然这是一个筛选的过程，在那个时间点会有多个会话，这个会话就是做一个筛选，然后看会话是否合理。因为没有 DB 故障现场只能通过发生的事务信息来看。

第七章：整个部署过程最快仅需 9 分钟，TDSQL 全球灵活部署实践

TDSQL 交付的话题内容，包括 TDSQL 曾经面临的交付要求和挑战，以及我们开发沉淀的自动化交付方案，最后更重要的是这套质量保障体系后续可以如何继续在交付后的用户的全生产流程中为用户提供全方位质量保障。

1、TDSQL 交付要求和挑战：快速、灵活、安全

1.1 复杂产品组件交付

首先我们想讲的是 TDSQL 的交付挑战，我们也是以三个方面去展开，第一个我们遇到的挑战是我们 TDSQL 产品架构所带来的特点：一是产品化不断完善带来的特点——组件多，包括拥有数据库内核，任务分发、冷备中心、平台告警、性能诊断等；二是组件之间相互依赖关系比较复杂。

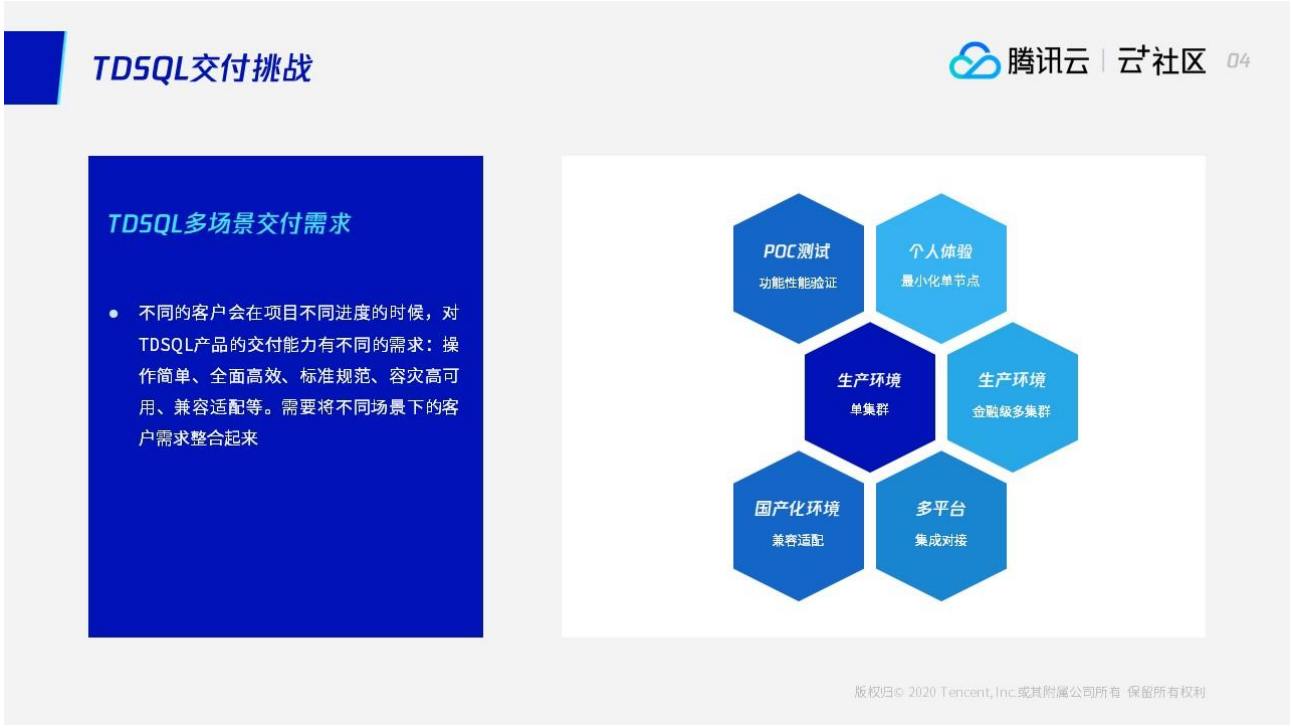
首先从层次上把这些组件进行划分：赤兔、监控采集、OSS、MetaCluster、扁鹊、onlineddl 等可以划分为一个角色，叫管理节点。对业务来说，实际访问数据库的过程是，先是负载均衡层，然后到 SQL 引擎层，而 SQL 引擎层会直接访问底层 DB，DB 上会部署 Agent。图中左侧列这些叫做管理节点；右侧列如冷备中心、消息队列、多源同步等，一般划分为数据节点。而日志分析平台其实是一个其他模块，可划分为其他的节点。

这些节点之间的依赖关系比较复杂。比如管理节点，其主要负责元数据管理，元数据包括比如以监控采集模块为核心的监控数据、以任务分发系统为核心的任务节点数据；第二是 DB 模块，DB 会和管理节点有一些交互——除了 DB 节点，还有其他的节点都会向管理节点发送监控信息；而管理节点也会下发任务，比如客户在前台进行的垂直扩容、水平扩容、主备切换等变更动作，也会到实际的 DB 进行交互；数据节点会向管理节点发送数据，和 DB 节点做一些交互……

所以其实各个组件之间的依赖关系比较复杂，这对于交付带来一定的难处。

1.2 多场景适应性交付

第二个挑战来自于 TDSQL 多个场景。



TDSQL 多个场景主要来源于使用 TDSQL 的对象是不同的，包括个人、企业、第三方平台。不同的对象使用 TDSQL 的需求和场景也不同。个人使用可能更想低门槛快速上手产品。企业使用最主要是 POC 测试和生产场景，关注的是整个产品的性能和功能，包括高可用性、容灾能力、国产化适配等。,

不同场景的需求如何高效满足？是要做多个分支去适配不同的场景，还是用一个分支去适配不同的场景？当然我们是用一个分支去适配不同的场景。

1.3 TDSQL 交付质量保障：安全、合规、多层级扫描

第三个挑战是，由于时间的推移，负责 TDSQL 交付的人产生了变化。早期 TDSQL 由产品研发团队、DBA 同学去现场给客户做交付。产品研发团队和 DBA 团队，大家都是一个团队，团队内由于长期的合作协同是形成了标准和质量可靠。而随着 TDSQL 产品化做大做强，用户规模不断扩大以后，交付人员会发生变化。不同的交付实施方，他们的操作和使用如果不够标准化，则容易带来隐患，体现在几个方面：

第一个是安全。比如说环境的安全，我们知道数据库场景是对内存、CPU、硬盘、IO 等能力都是要求比较高的场景，包括对 TCP 的内核参数优化等这些工作都是作为潜在风险来统一考虑。

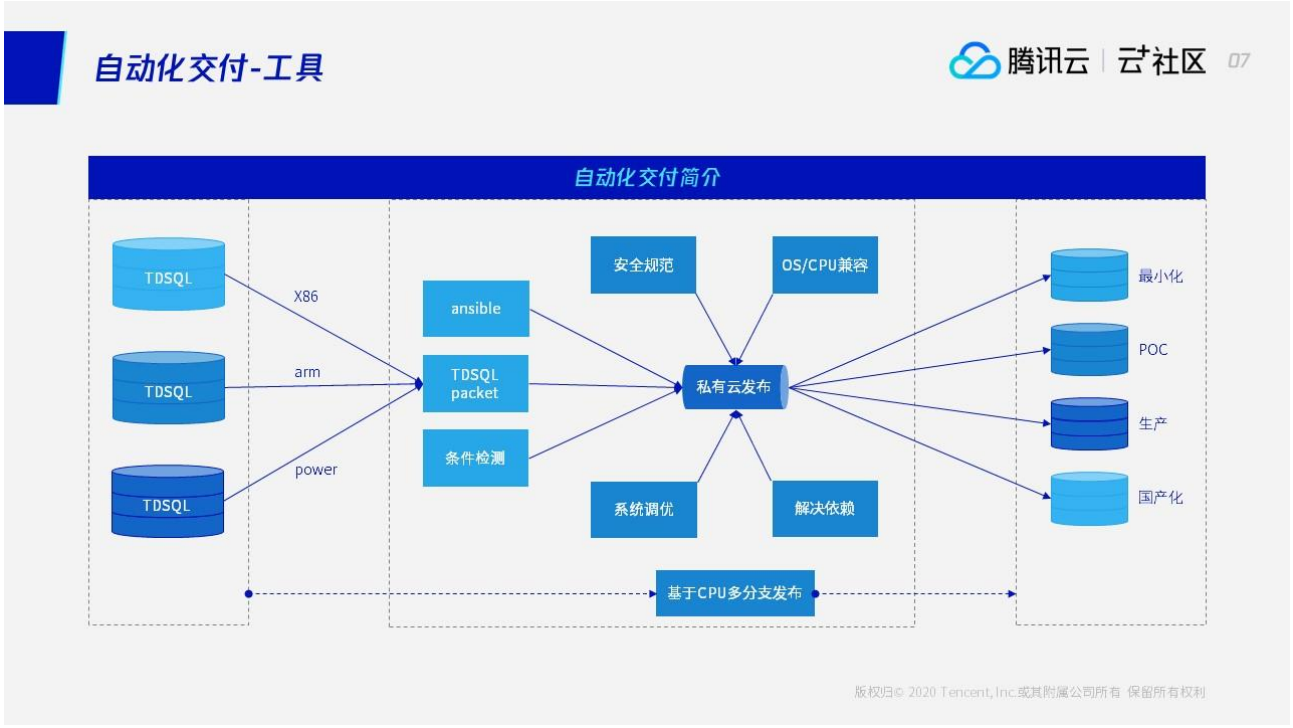
第二个是监控。对整个集群、进程、机器的监控，以及自动拉起，即机器级别故障之后，快速恢复的能力，这些都要作为完善的体系来考虑。其他比如定时任务，包括定时清理一些日志，清理一些历史数据，否则磁盘就会撑满，这在生产的环境上也是风险很大。最后是如何保障整个集群的高可用性、容灾能力；如何杜绝潜在旧版本带来的隐患，检测这些版本的漏洞等方面，都是交付质量体系中需要解决的问题。

其实 TDSQL 交付质量服务和保障就是围绕着上述的各方面问题，实现由不同的实施人、实施方去交付 TDSQL 产品，都能保证 TDSQL 的投产质量。这是我们在做的事情。

2、TDSQL 自动化交付方案：全球灵活部署，最快 9 分钟！

针对上述的挑战，TDSQL 沉淀出了一套 TDSQL 自动化交付方案。

2.1 自动化交付方案规划



这是 TDSQL 自动化交付方案架构图。

TDSQL 是基于一个分支来实现多场景、复杂关系下的自动化交付的，其实也可以说是基于三个分支去做的——TDSQL 内核包，当前有三个分支：基于 CPU 的多分支进行发布，当前支持 X86、arm、power。TDSQL 对客户的发布包中，一个包自动集成了不同 CPU 版本的 TDSQL packet——以 ansible 组件为基础，加上了条件检测、操作系统调优、环境依赖的解决、安全规范、兼容性问题，是 TDSQL 标准发布包，可针对于客户不同的场景和不同的环境做适配。

TDSQL 的组件分为四个角色，快速交付 TDSQL 集群，打个比方说就是把不同的鸡蛋放到不同的篮子里。鸡蛋是指这些组件；篮子就是我们准备的机器，可以是虚拟机，也可以是物理机。

- 首先个人体验的环境：个人体验环境更注重的是较低门槛，在这里我们只需要一台虚拟机

的配置就可以达到目的。然后可以把管理节点、DB 节点、数据节点和其他的节点都部署在这台机器上。当然在体验的环境下，数据节点和其他的节点这两个功能可以不进行部署。

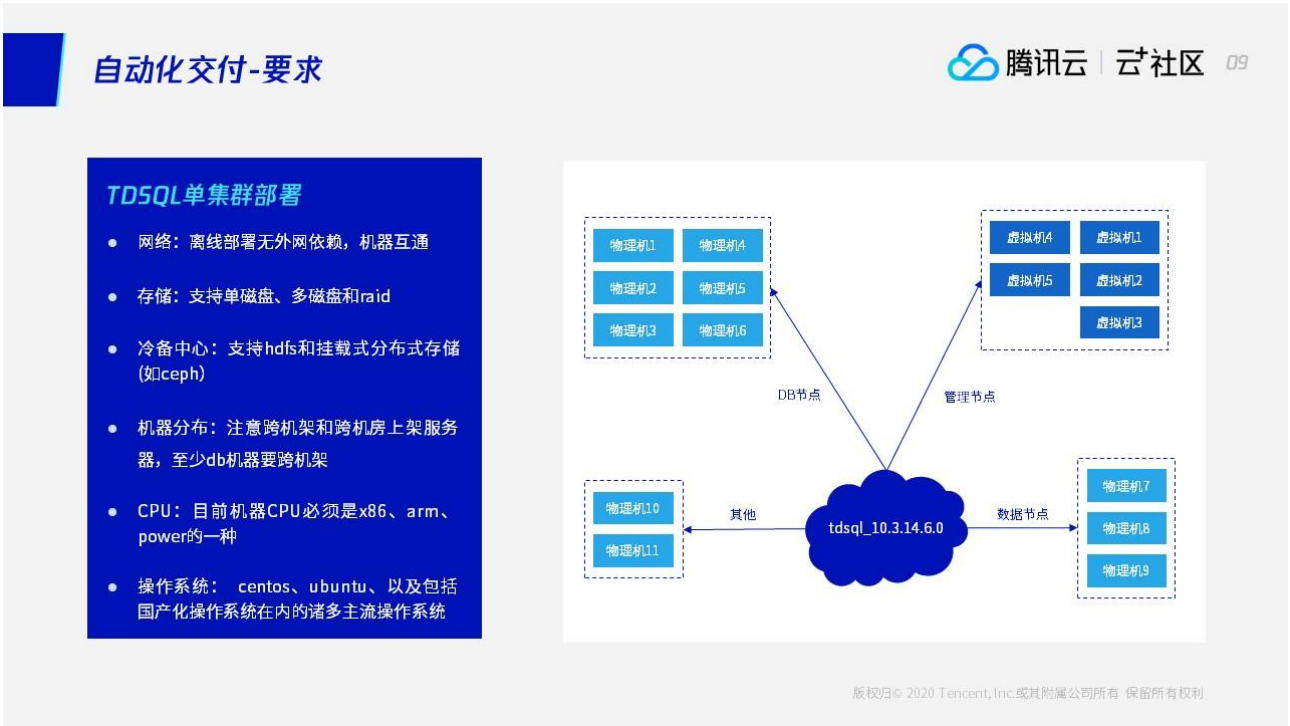
- 测试环境：该环境下注重的是性能、功能。首先从管理节点来看，管理节点提供的是元数据的管理和任务的分发功能，要求的是稳定性和容灾能力。在测试环境可以稍微弱化这个要求，比如可以准备一台或者三台虚拟机、配置 4C/8G 普通磁盘；在测试环境下要做 DB 节点的话，要考虑到 TDSQL 的性能问题，这里推荐使用物理机；进行性能测试的时候要求一定是 SSD 盘，否则性能数据没有任何参考性——这也是由数据库的场景决定的，因为 SSD 和普通的磁盘，一方面主要表现在随机读写能力上的差距会比较大；数据节点和其他的节点方面，如果有一些客户对测试的功能要求没有那么强，就可以不部署这些节点的功能，而如果想体验完整的 TDSQL 的功能，则需要准备这些机器，以体验完整的 TDSQL 的功能；如要部署数据节点，可以选择一台机器或者三台虚拟机，以及准备较大容量的磁盘做数据节点;其他的节点,比如负载均衡和日志分析平台,TDSQL 的负载均衡比较灵活，位于 SQL 引擎层上的上一层，这里推荐开源的 LVS，当然也有很多客户会使用 F5。最后，以上环境我们的推荐是部署两节点来实现容灾能力。总体而言，为了保证测试的性能，测试环境要求最多的是 DB 节点模块。

- 生产环境：生产环境中要求管理节点可以部署在三台或者五台虚拟机，但最好是跨三个机房，比如说"1+1+1"的模式或者"2+2+1"的模式，因为元数据集群是基于多数选举的机制来保障高可用，如果只有两个机房则会失去了本身容灾的意义，因此我们建议生产环境中部署三个机房；DB 节点生产环境更推荐的是 NVME 接口的 SSD，因为传统的 SSD 和 NVME 的 SSD 在接口性能上会有比较大的差距，而数量上推荐的是 3*N 台——事实上这个要去

评估生产环境 TDSQL 集群的数据量，TDSQL 是一个分布式数据库，数据量级可以根据用户机器数量实现水平拓展。

举个例子假设客户有 3T 数据，如果单台物理机是 1T、一个 set 内做的是一主两备三个节点，我们此时需要三个 set，三个 set 可以承担 3T 数据量，同时会有两个副本的冗余，DB 节点的这些数就需要 9 台这样的机器，这三个 set 会组成 group shard；数据节点的机器也是推荐物理机，同时在生产环境需要考虑容灾能力，因此推荐是三台机器以上。此外，需要一个高性能磁盘来保证回档和备份的效率；最后，访问链路上接入层是非常重要的一层，我们强烈推荐物理机来提高稳定性。

2.2 TDSQL 自动化交付特性与要求

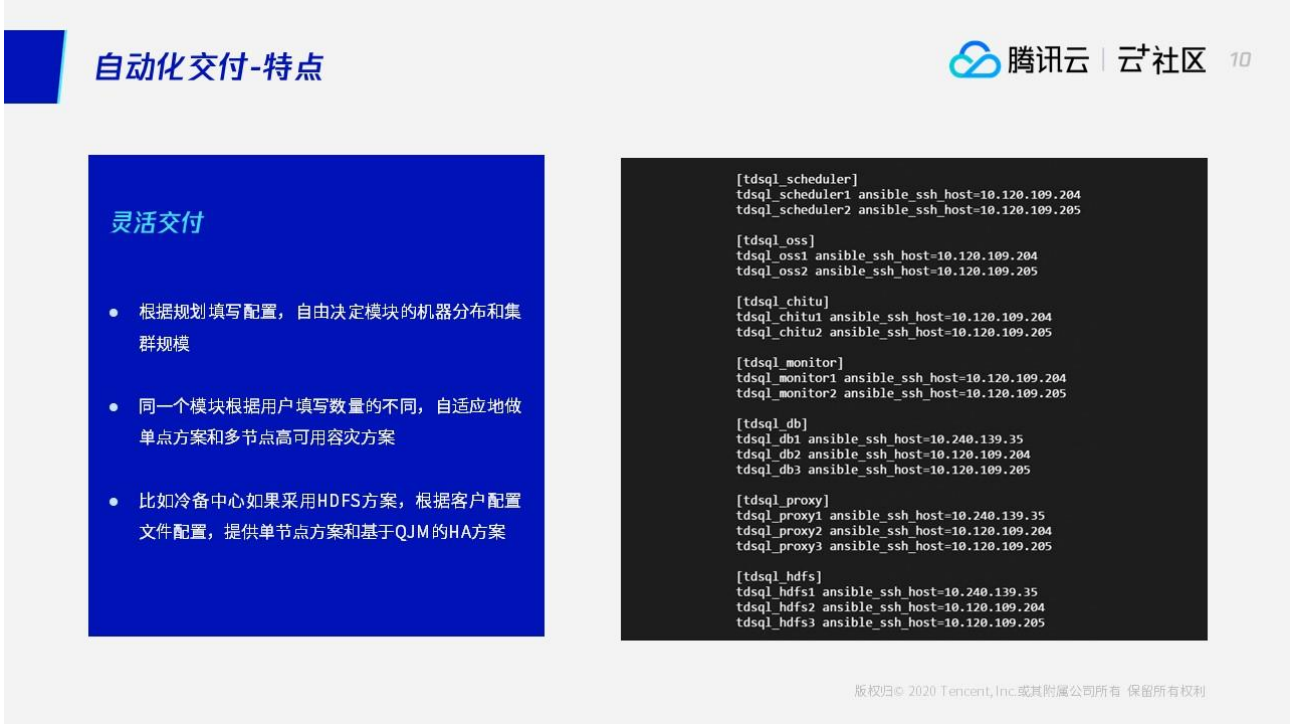


前文讲到了 TDSQL 不同的组件以及我们怎样去管理其中的层次逻辑。在 TDSQL 真正交付过程中，为了保证交付质量，结合金融级场景的安全合规、高可用容灾考虑，我们沉淀出一些基本要求和特性：

- 网络：离线部署无外网依赖，机器互通；
- 存储：支持单磁盘、多磁盘和 raid；
- 冷备中心：支持 hdfs 和挂载式分布式存储(如 ceph)；
- 机器分布：支持跨机架和跨机房上架服务器，支持多种机器分布模式下的高可用容灾；
- CPU：在国产化趋势下，目前机器 CPU 除了适配 x86，还包括 arm、power ，而且首要推荐以上其中一款；
- 操作系统：适配支持 centos、ubuntu、以及包括国产化操作系统在内的诸多主流操作系统，

上图右侧展示了简要分布关系，交付过程中我们只要理清楚如何把鸡蛋放到对应的篮子里即可实现自动化交付：我们先选出篮子，一组物理机就是一个篮子，随之把一组的组件 DB 节点放到这个篮子里。

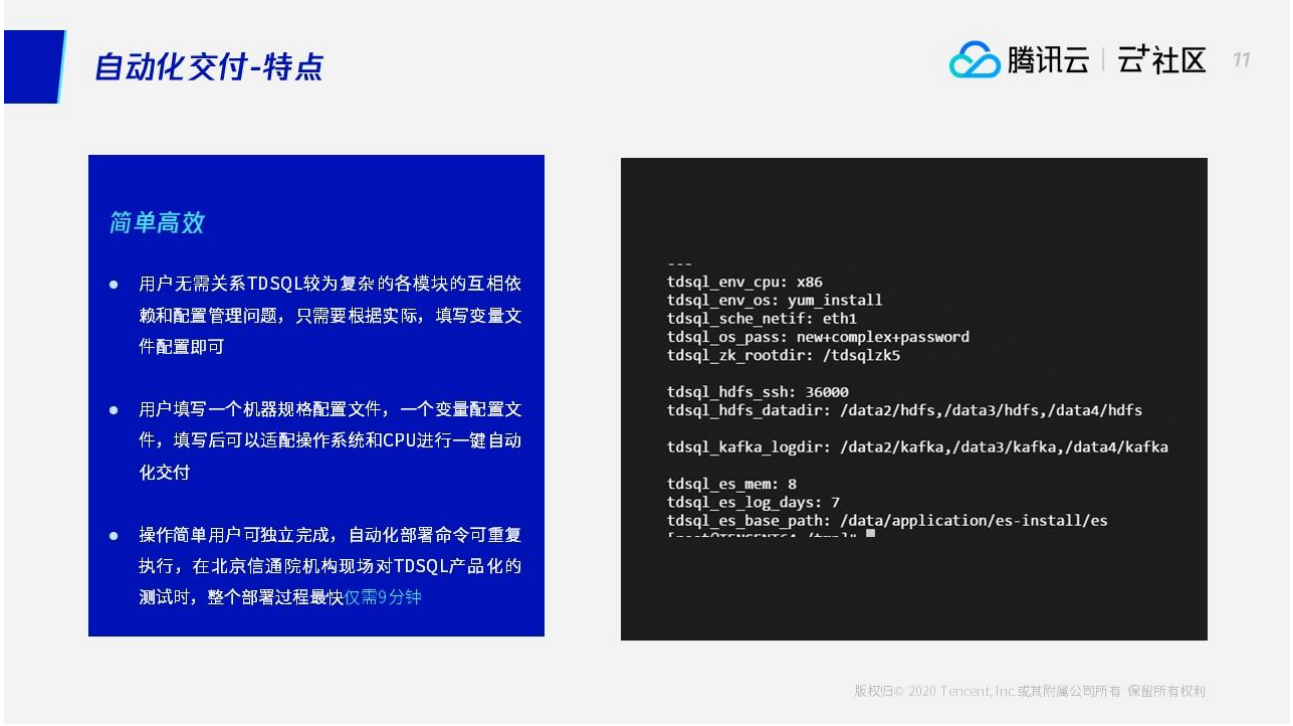
2.2.1 灵活交付



当然这其中有很多细节，客户要做的，是自由决定模块的机器分布和集群规模，TDSQL 可以通过模块之间的数量差异，自适应地做出单点方案和多节点高可用容灾方案。这个过程用户在操作上是无感知的。举个例子，比如 TDSQL 是支持 HDFS 作为冷备中心，如果 HDFS 选的是一个节点，系统会做一个 HDFS 的单点方案。如果是三节点的配置规划，它会自动感知到要做的是一个高可用容灾方案。HDFS 用的高可用容灾方案是基于 QJM 的方式。

2.2.2 简单高效：整个部署过程最快仅需 9 分钟

做完部署规划，第二件事情是解决各个组件之间的一些关系，包括兼容性等问题。举个例子，如果部署的 TDSQL 环境是基于 ARM 国产服务器的操作系统的国产化环境。我们如何通过一个交付物料包去适配不同的环境？其实秘密就在这个配置文件里：

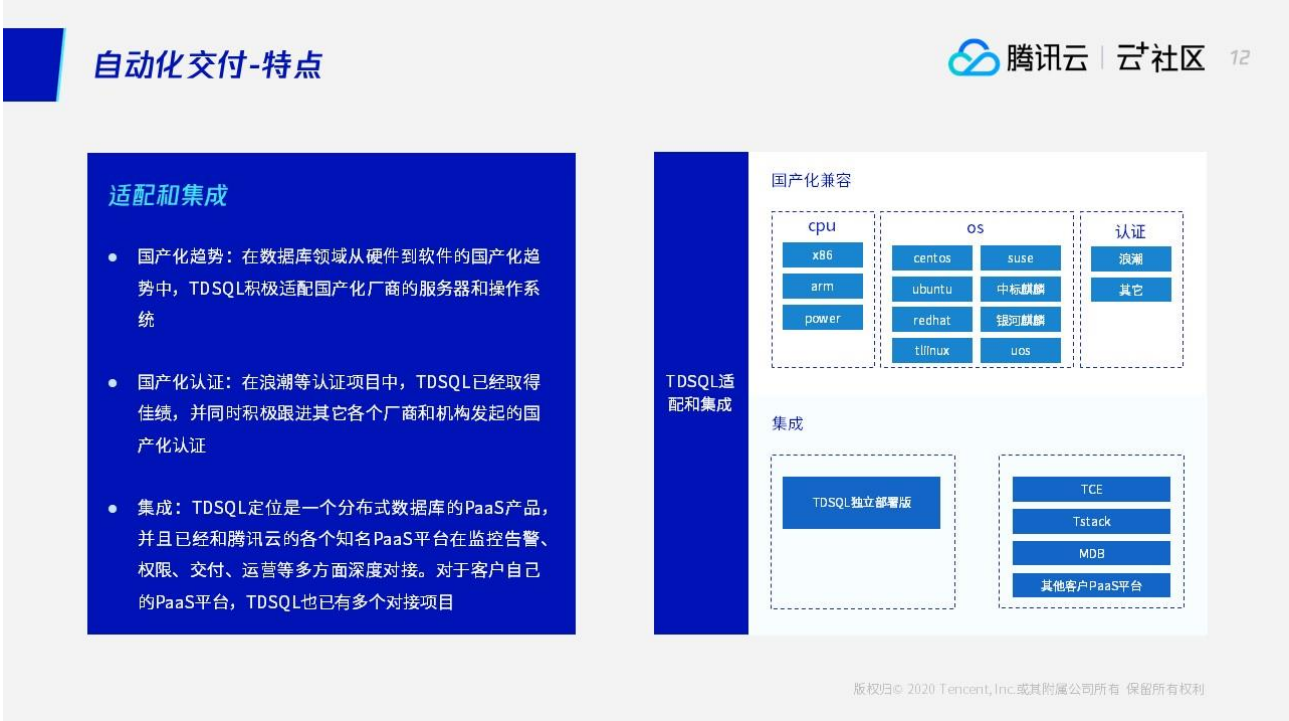


- 用户无需关注 TDSQL 较为复杂的各模块的互相依赖和配置管理问题，只需要根据实际，填写变量文件配置即可；
- 用户填写一个机器规格配置文件、一个变量配置文件，填写后可以适配操作系统和 CPU 实现一键自动化交付
- 操作简单用户可独立完成，自动化部署命令可重复执行，在北京信通院机构现场对 TDSQL 产品化的测试显示，整个部署过程最快仅需 9 分钟

2.2.3 适配与集成：国产化、全栈式

当前国产化已经成为一个趋势，在国产化的浪潮下，TDSQL 作为一个腾讯自研分布式数据库，我们也是义不容辞的担当了国产化的责任。此前，TDSQL 在国产化适配方面也做了很多工作，

从底层的服务器到存储器、操作系统、CPU、行业软件、数据库软件等，都在相关部门指导下进行了与各个厂商合作实现从下层到上层全方位的国产化适配。包括腾讯内部的操作系统 tlinux，以及中标麒麟、银河麒麟、UOS 等主流国产化操作系统，TDSQL 都完成了适配。



除了适配全系国产操作系统，TDSQL 同时已相继完成对全系国产芯片，全系列国产服务器等的兼容适配工作。而在完成适配工作的同时，腾讯也提供了对应的技术服务，帮助行业用户更好地迁移到国产基础技术生态当中。这些是我们国产化方面的工作。

技术服务生态方面，TDSQL 不仅可作为一个独立发布的产品，在 TDSQL 发展的历程中，也被其他很多平台厂商和合作伙伴接纳，包括腾讯内部的 TCE、Tstack、MDB 架构等。TCE 是腾讯云金融级平台，TDSQL 和 TCE 在部署方案、告警、用户权限等等各种维度和 TCE 进行了深度的集成，可为金融政务机构提供全方位的 PaaS 基础技术服务，在完成高性能的分布式架构转

型升级的同时保障金融级稳定高可用。除了内部的平台，TDSQL 许多合作伙伴的行业解决方案中也集成了 TDSQL，把 TDSQL 的能力输入到他们自己的平台。

2.2.4 安全保障：秒级监测

TDSQL 在发展中对交付场景做了许多优化：

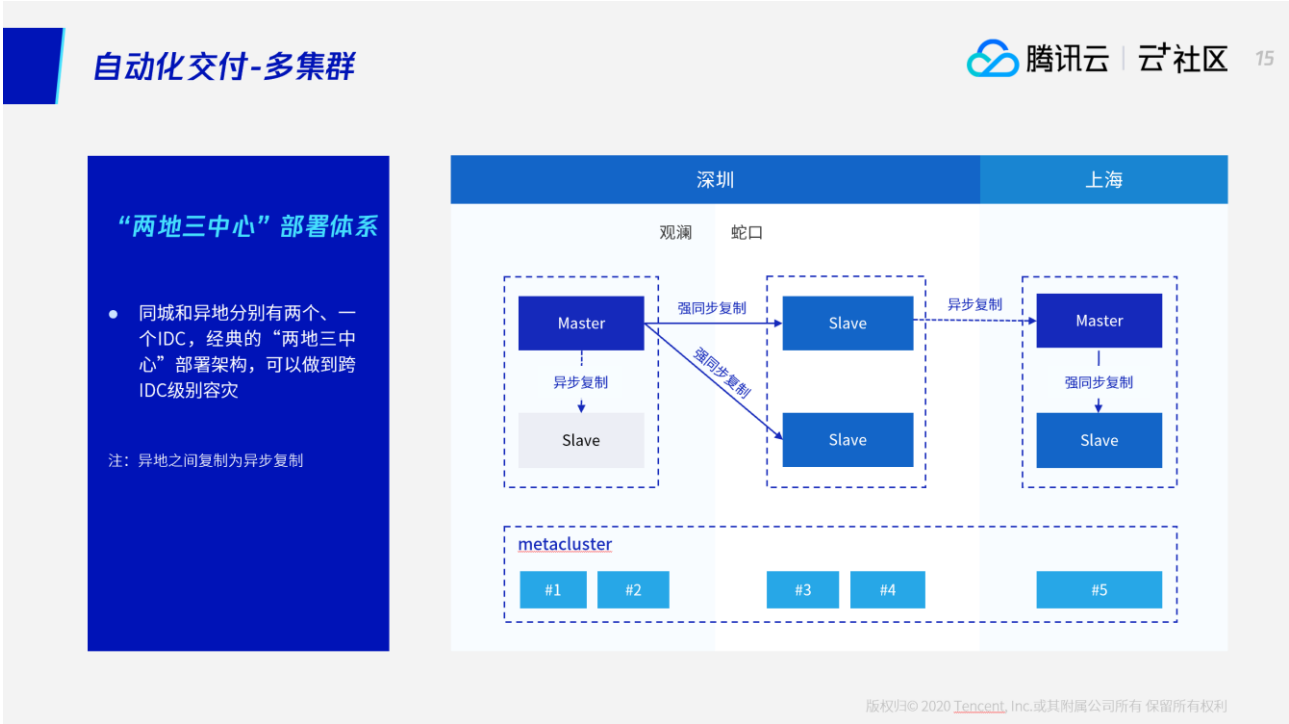
- 条件检测：首先会自动对规划的 TDSQL 集群下的所有机器做前置检测，包括机器时间同步、时区一致、端口占用、系统默认 sh、机器规格等做检；
- 环境优化：针对关系型数据库场景，对系统 50 处左右进行针对性调优，并解决一些基础的依赖；

- 机器秒级监控：大部分的监控平台都是基于分钟级的，对于金融级数据库这种敏感场景，分钟级的监控是不够的，所以我们针对这样的场景提供了秒级监控，包括针对机器的 IO、CPU、网络、内存等多个维度。

2.3 多集群下的自动化交付

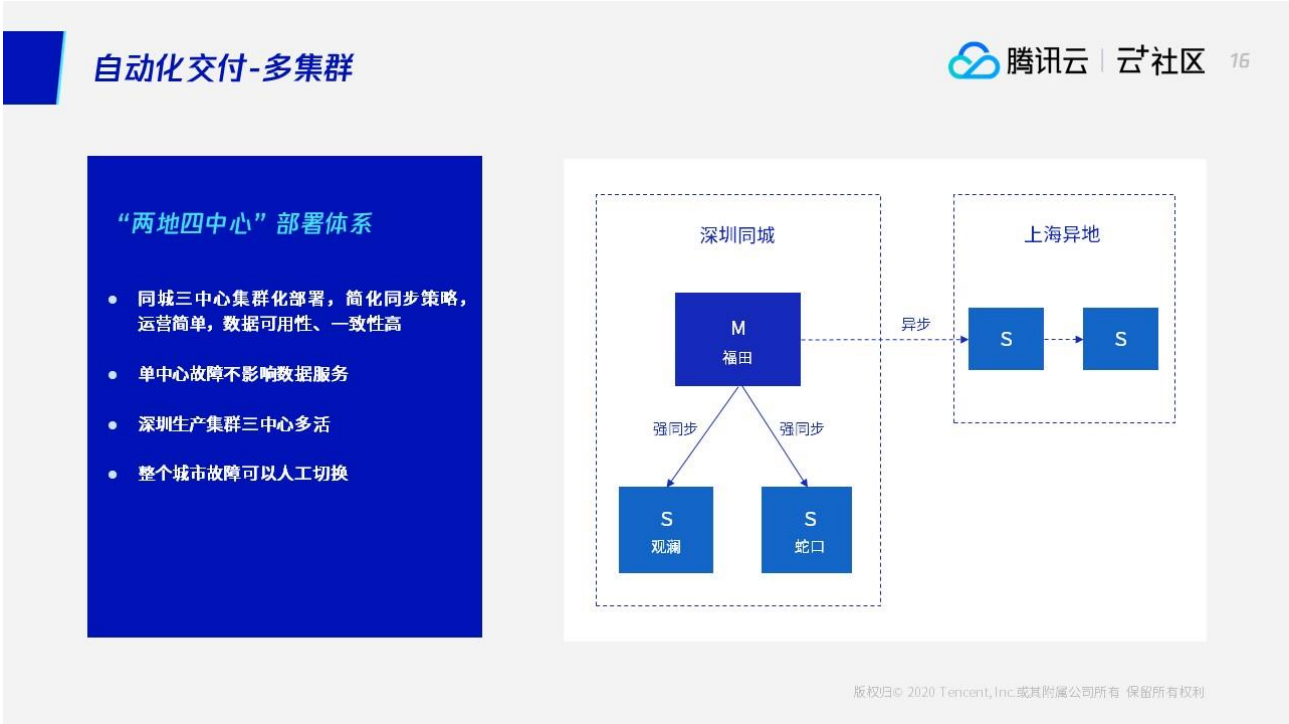
前文讲的是 TDSQL 在单集群下的交付场景和交付细节，接下来介绍在多集群下的交付具体是怎样进行的。

“两地三中心”部署体系



“两地三中心”架构顾名思义：在一个城市有 A、B 两个机房，另一个城市有 C 机房，在第一个城市中 TDSQL 数据库实例采用同 IDC 异步、跨 IDC 强同步的方式，我们需要在第一个城市将四个数据节点部署在二个机房，其中主节点和一个备节点在一个机房，另外两个备节点在另一个机房。并且在第一个城市和第二个城市的数据库实例间，采用的是异步复制，保障金融城市级高可用容灾。

“两地四中心”部署体系



“两地四中心”的架构，是一个自动化切换的强同步架构，对任何数据中心及故障都能 30 秒内切换，并且数据零丢失，性能也稳定可靠，对业务和用户来说是实现更高的可用性和更低的成本。

3、TDSQL 质量保障服务：全生产流程自动化巡检

TDSQL 的交付质量通过一个自动化巡检的方案保证。TDSQL 自动化巡检方案通过三个维度保障交付质量：

监控指标分析

第一个维度基于 TDSQL 现有的监控中心进行相关指标性的分析，包括当前时刻的指标分析和历史时刻的指标分析。当我们要在验证一个集群是否有问题的时候，往往除了要分析此时此刻的集群是否存在异常和告警、是否存在资源负载过重等情况，还需要分析历史性的问题，比如

说在历史过去七天中各个指标的曲线如何。为什么要分析过去历史七天的指标曲线？举个简单的场景案例，例如一个场景在每天下午三点到五点是业务高峰期，在业务高峰期期间可能有很多业务的慢查询，甚至有一些慢查询带来的性能的问题。系统如何监控在历史某个时刻出现的问题？那么我们发起自动化巡检方案的时候，比如是上午 8 点钟，适逢业务低峰期，此时是发现不了问题的，所以我们需要对历史指标进行分析。

方案中具体分析的指标包括检测前台连通性、实例的复制方式、主备切换方式等。监控主要分为两个方面：第一是监控指标的采集、上报、搜集，这是监控中心负责。第二是对监控数据进行分析，并对认为异常的分析进行告警。分析和告警过程中会遵循一定的策略——怎样的监控数据才是异常、有必要告警的？当前 TDSQL 维护了一套告警模板，也给客户提供了可配置的、定制化的选项，客户可以根据自己的实际情况进行告警策略的修改；同时提供基于实践经验积累的告警策略对比，以防用户做出不合理的修改，暴露告警策略的潜在风险。

在这个维度，TDSQL 多源同步等模块可以对数据同步情况进行监控，他们当前同步的稳定性、同步的性能如何，等其他就是各个模块的告警的监控指标。

集群环境扫描

第二个维度是对第一个维度的补充。第二个维度的分析是机器级的，不是通过采的监控数据，是直接访问服务器后台，对机器级的 IO、CPU、内存、磁盘、稳定性等进行检测。

除了机器级和进程级，我们还会进行实例级的定制化扫描，这个体现在实例体检模块——实例的体检就是 TDSQL 智能诊断分析平台“扁鹊”的接口，可以为实例提供从运营、开发、性能等各个指标的系统性分析。

集群级层面，我们会关注这个集群各个机器之间是否是同步、实例下元数据集群是否是有备份、备份是否是正常等。

自动化演练

在我们以各个维度去扫描当前集群没有问题的情况下，TDSQL 还会从结果出发，对整个集群做一次 P0 级别的自动化演练，演练的场景就是我们正常运营和管理的场景，包括购买实例、创建用户、用户授权、创建库表，在这个库表上做一些表结构的变更、水平扩容、垂直的扩容、重做备机、慢查询入库、备份和回档等。最后系统会对购买的实例进行删除，实现对 P0 级别的场景进行闭环的自动化演练。

总结来说，TDSQL 自动化巡检方案从指标级，到整个集群环境进行扫描，以及通过自动化演练这三个维度确保整个交付的集群安全、稳定、可靠、高可用。

除了技术上的保障方案，TDSQL 同时沉淀了大量产品化工作，帮助用户快速、方便地使用分布式数据库。

我们也会对客户信息进行定期维护，首先对客户定期发起集群的巡检，通过这个巡检可以保证客户当前以及历史一段时间内环境是没有问题的。巡检主要进行功能性和容灾性的演练，通过自动的定期巡检，管理系统如果扫描到有建议客户要升级的版本，则会自动推送到客户代表，由客户代表推动客户升级。

最后，在客户日常运营、日常变更中，可能运营面临的大部分问题是怎么扩容、升级、处理告警？TDSQL 对各个节点的扩容提供了自动化的扩容方案，可以一键扩容。同样升级也是提供了前台化一键操作的功能，既可以进行点对点升级，也可以进行整个集群的批量升级。TDSQL 的高可用性一方面在于自身的弹性架构和容灾能力，以及数据强一致性。

可用性方面 TDSQL 提供了自动化告警处理方案，可实现自动化告警分析，并对部分告警自动处理，减少现网运营的工作量。

以上我们以交付为核心介绍了 TDSQL 在历史过程中遇到的几个交付上的挑战，和针对这些交付挑战，我们提出的自动化交付方案，以及最后对整个 TDSQL 标准化交付的质量和客户服务提供了一系列机制和能力方面的提升。

Q&A

Q：TDSQL 支持数据库离线备份吗?

A： TDSQL 支持多种备份方式，可以基于物理式的备份，也可以基于逻辑备份。整个备份过程在备机上进行，不会影响正常的业务访问，也不会对业务访问的性能带来影响。

Q：TDSQL 的告警信息如何接入短信、语音、邮件告警平台?

A：TDSQL 的告警接入比较灵活，首先 TDSQL 的告警信息是文本的形式，可以发送到任何平台，当前客户已经适配过的告警接入方式有很多，比如说客户有 HTTP 接口的告警平台，也有一些其他接口的。根据客户想要的接口，TDSQL 可以对应地发一个包，包含了告警信息，发到你的告警接收平台就可以了。