参考：<http://blog.csdn.net/y87329396/article/details/52612485>

# 怎样打造一个分布式数据库

转载 2016年09月21日 19:57:00

* 7341

在技术方面，我自己热衷于 Open Source，写了很多 Open Source 的东西，擅长的是 Infrastructure 领域。Infrastructure 领域现在范围很广，比如说很典型的分布式 Scheduler、Mesos、Kubernetes，另外它和 Microservices 所结合的东西也特别多。Infrastructure 领域还有比如 Database 有分 AP（分析型）和 TP（事务型），比如说很典型的大家知道的 Spark、Greenplum、Apache Phoenix 等等，这些都属于在 AP 的，它们也会去尝试支持有限的 TP。另外，还有一个比较有意思的就是 Kudu——Cloudera Open Source 的那个项目，它的目标很有意思：我不做最强的 AP 系统，也不做最强的 TP 系统，我选择一个相对折中的方案。从文化哲学上看，它比较符合中国的中庸思想。

另外，我先后创建了 Codis、TiDB。去年12月份创建了 TiKV 这个 project，TiKV 在所有的 rust 项目里目前排名前三。

首先我们聊聊 Database 的历史，在已经有这么多种数据库的背景下我们为什么要创建另外一个数据库；以及说一下现在方案遇到的困境，说一下 Google Spanner 和 F1、TiKV 和 TiDB，说一下架构的事情，在这里我们会重点聊一下 TiKV。因为我们产品的很多特性是 TiKV 提供的，比如说跨数据中心的复制、Transaction、auto-scale。

接下来聊一下为什么 TiKV 用 Raft 能实现所有这些重要的特性，以及 scale、MVCC 和事务模型。东西非常多，我今天不太可能把里面的技术细节都描述得特别细，因为几乎每一个话题都可以找到一篇或者是多篇论文，所以详细的技术问题大家可以单独来找我聊。

后面再说一下我们现在遇到的窘境，就是大家常规遇到的分布式方案有哪些问题，比如 MySQL Sharding。我们创建了无数 MySQL Proxy，比如官方的 MySQL proxy、Youtube 的 Vitess、淘宝的 Cobar、TDDL以及基于 Cobar 的 MyCAT、金山的 Kingshard、360 的 Atlas、京东的 JProxy，我在豌豆荚也写了一个。可以说，随便一个大公司都会造一个 MySQL Sharding 的方案。

为什么我们要创建另外一个数据库？

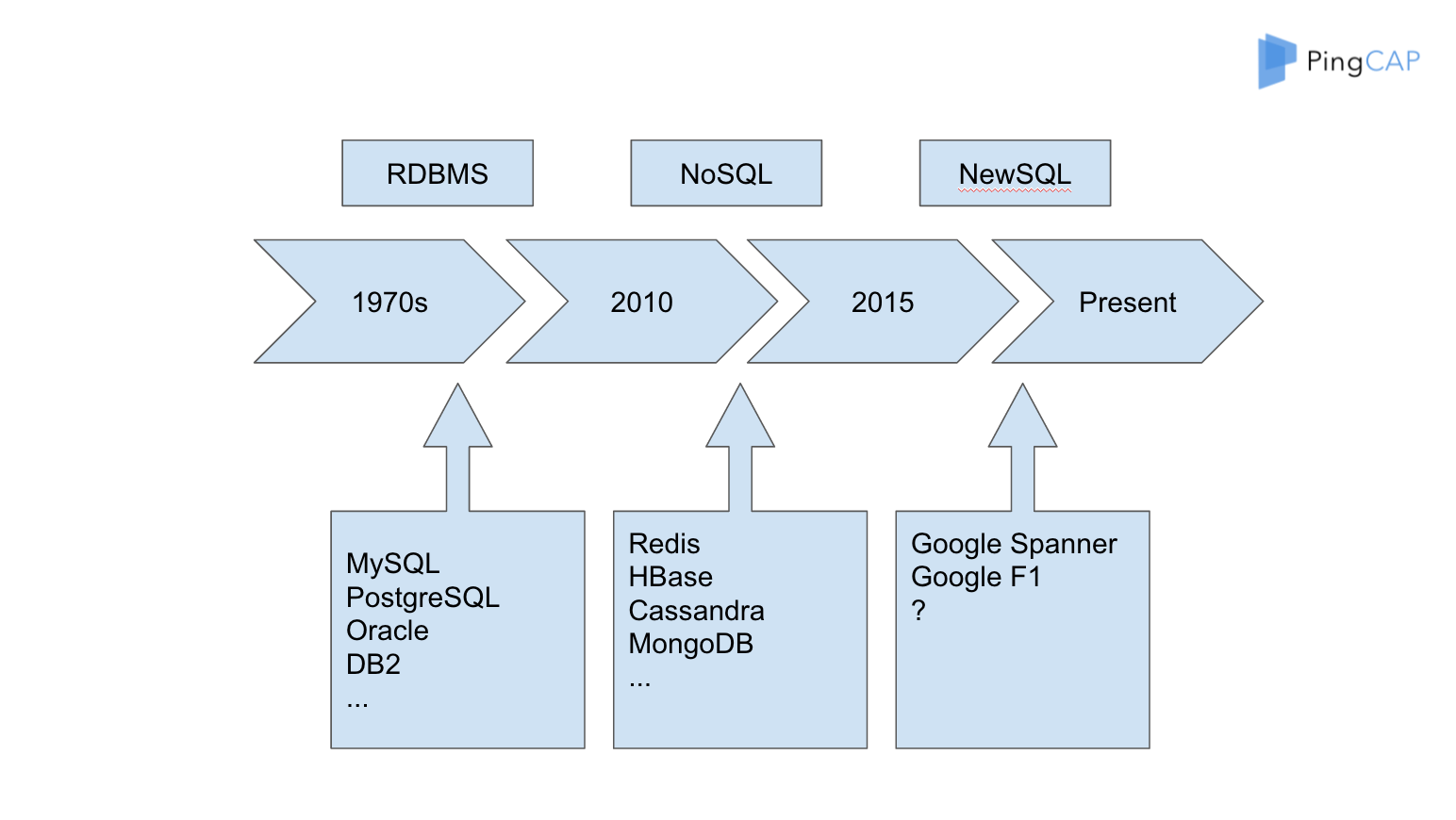
昨天晚上我还跟一个同学聊到，基于 MySQL 的方案它的天花板在哪里，它的天花板特别明显。有一个思路是能不能通过 MySQL 的 server 把 InnoDB 变成一个分布式数据库，听起来这个方案很完美，但是很快就会遇到天花板。因为 MySQL 生成的执行计划是个单机的，它认为整个计划的 cost 也是单机的，我读取一行和读取下一行之间的开销是很小的，比如迭代 next row 可以立刻拿到下一行。实际上在一个分布式系统里面，这是不一定的。

另外，你把数据都拿回来计算这个太慢了，很多时候我们需要把我们的 expression 或者计算过程等等运算推下去，向上返回一个最终的计算结果，这个一定要用分布式的 plan，前面控制执行计划的节点，它必须要理解下面是分布式的东西，才能生成最好的 plan，这样才能实现最高的执行效率。

比如说你做一个 sum，你是一条条拿回来加，还是让一堆机器一起算，最后给我一个结果。 例如我有 100 亿条数据分布在 10 台机器上，并行在这 10台机器我可能只拿到 10 个结果，如果把所有的数据每一条都拿回来，这就太慢了，完全丧失了分布式的价值。聊到 MySQL 想实现分布式，另外一个实现分布式的方案就是 Proxy。但是 Proxy 本身的天花板在那里，就是它不支持分布式的 transaction，它不支持跨节点的 join，它无法理解复杂的 plan，一个复杂的 plan 打到 Proxy 上面，Proxy 就傻了，我到底应该往哪一个节点上转发呢，如果我涉及到 subquery sql 怎么办？所以这个天花板是瞬间会到，在传统模型下面的修改，很快会达不到我们的要求。

另外一个很重要的是，MySQL 支持的复制方式是半同步或者是异步，但是半同步可以降级成异步，也就是说任何时候数据出了问题你不敢切换，因为有可能是异步复制，有一部分数据还没有同步过来，这时候切换数据就不一致了。前一阵子出现过某公司突然不能支付了这种事件，今年有很多这种类似的 case，所以微博上大家都在说“说好的异地多活呢？”……

为什么传统的方案在这上面解决起来特别的困难，天花板马上到了，基本上不可能解决这个问题。另外是多数据中心的复制和数据中心的容灾，MySQL 在这上面是做不好的。



在前面三十年基本上是关系数据库的时代，那个时代创建了很多伟大的公司，比如说 IBM、Oracle、微软也有自己的数据库，早期还有一个公司叫 Sybase，有一部分特别老的程序员同学在当年的教程里面还可以找到这些东西，但是现在基本上看不到了。

另外是 NoSQL。NoSQL 也是一度非常火，像 Cassandra、MongoDB 等等，这些都属于在互联网快速发展的时候创建这些能够 scale 的方案，但 Redis scale 出来比较晚，所以很多时候大家把 Redis 当成一个 Cache，现在慢慢大家把它当成存储不那么重要的数据的数据库。因为它有了 scale 支持以后，大家会把更多的数据放在里面。

然后到了 2015，严格来讲是到 2014 年到 2015 年之间，Raft 论文发表以后，真正的 NewSQL 的理论基础终于完成了。我觉得 NewSQL 这个理论基础，最重要的划时代的几篇论文，一个是谷歌的 Spanner，是在 2013 年初发布的；再就是 Raft 是在 2014 年上半年发布的。这几篇相当于打下了分布式数据库 NewSQL 的理论基础，这个模型是非常重要的，如果没有模型在上面是堆不起来东西的。说到现在，大家可能对于模型还是可以理解的，但是对于它的实现难度很难想象。

前面我大概提到了我们为什么需要另外一个数据库，说到 Scalability 数据的伸缩，然后我们讲到需要 SQL，比如你给我一个纯粹的 key-velue 系统的 API，比如我要查找年龄在 10 岁到 20 岁之间的 email 要满足一个什么要求的。如果只有 KV 的 API 这是会写死人的，要写很多代码，但是实际上用 SQL 写一句话就可以了，而且 SQL 的优化器对整个数据的分布是知道的，它可以很快理解你这个 SQL，然后会得到一个最优的 plan，他得到这个最优的 plan 基本上等价于一个真正理解 KV 每一步操作的人写出来的程序。通常情况下，SQL 的优化器是为了更加了解或者做出更好的选择。

另外一个就是 ACID 的事务，这是传统数据库必须要提供的基础。以前你不提供 ACID 就不能叫数据库，但是近些年大家写一个内存的 map 也可以叫自己是数据库。大家写一个 append-only 文件，我们也可以叫只读数据库，数据库的概念比以前极大的泛化了。

另外就是高可用和自动恢复，他们的概念是什么呢？有些人会有一些误解，因为今天还有朋友在现场问到，出了故障，比如说一个机房挂掉以后我应该怎么做切换，怎么操作。这个实际上相当于还是上一代的概念，还需要人去干预，这种不算是高可用。

未来的高可用一定是系统出了问题马上可以自动恢复，马上可以变成可用。比如说一个机房挂掉了，十秒钟不能支付，十秒钟之后系统自动恢复了变得可以支付，即使这个数据中心再也不起来我整个系统仍然是可以支付的。Auto-Failover 的重要性就在这里（me：fast-fail）。大家不希望在睡觉的时候被一个报警给拉起来，我相信大家以后具备这样一个能力，5 分钟以内的报警不用理会，挂掉一个机房，又挂掉一个机房，这种连续报警才会理。我们内部开玩笑说，希望大家都能睡个好觉，很重要的事情就是这个。

说完应用层的事情，现在还有很多业务，在应用层自己去分片，比如说我按照 user ID在代码里面分片，还有一部分是更高级一点我会用到一致性哈希。问题在于它的复杂度，到一定程度之后我自动的分库，自动的分表，我觉得下一代数据库是不需要理解这些东西的，不需要了解什么叫做分库，不需要了解什么叫做分表，因为系统是全部自动搞定的。同时复杂度，如果一个应用不支持事务，那么在应用层去做，通常的做法是引入一个外部队列，引入大量的程序机制和状态转换，A 状态的时候允许转换到 B 状态，B 状态允许转换到 C 状态（me:happe-before）。

举一个简单的例子，比如说在京东上买东西，先下订单，支付状态之后这个商品才能出库，如果不是支付状态一定不能出库，每一步都有严格的流程。

//===============20171030

Google Spanner / F1

说一下 Google 的 Spanner 和 F1，这是我非常喜欢的论文，也是我最近几年看过很多遍的论文。 Google Spanner 已经强大到什么程度呢？Google Spanner 是全球分布的数据库，在国内目前普遍做法叫做同城两地三中心，它们的差别是什么呢？以 Google 的数据来讲，谷歌比较高的级别是他们有 7 个副本，通常是美国保存 3 个副本，再在另外 2 个国家可以保存 2 个副本，这样的好处是万一美国两个数据中心出了问题，那整个系统还能继续可用，这个概念就是比如美国 3 个副本全挂了，整个数据都还在，这个数据安全级别比很多国家的安全级别还要高，这是 Google 目前做到的，这是全球分布的好处。

现在国内主流的做法是两地三中心，但现在基本上都不能自动切换。大家可以看到很多号称实现了两地三中心或者异地多活，但是一出现问题都说不好意思这段时间我不能提供服务了。大家无数次的见到这种 case， 我就不列举了。

Spanner 现在也提供一部分 SQL 特性。在以前，大部分 SQL 特性是在 F1 里面提供的，现在 Spanner 也在逐步丰富它的功能，Google 是全球第一个做到这个规模或者是做到这个级别的数据库。事务支持里面 Google 有点黑科技（其实也没有那么黑），就是它有**GPS 时钟和原子钟**。大家知道在分布式系统里面，比如说数千台机器，两个事务启动先后顺序，这个顺序怎么界定(事务外部一致性)。这个时候 Google 内部使用了 GPS 时钟和原子钟，正常情况下它会使用一个GPS 时钟的一个集群，就是说我拿的一个时间戳，并不是从一个 GPS 上来拿的时间戳，因为大家知道所有的硬件都会有误差。如果这时候我从一个上拿到的 GPS 本身有点问题，那么你拿到的这个时钟是不精确的。而 Google 它实际上是在一批 GPS 时钟上去拿了能够满足 majority 的精度，再用时间的算法，得到一个比较精确的时间。大家知道 GPS 也不太安全，因为它是美国军方的，对于 Google 来讲要实现比国家安全级别更高的数据库，而 GPS 是可能受到干扰的，因为 GPS 信号是可以调整的，这在军事用途上面很典型的，大家知道导弹的制导需要依赖 GPS，如果调整了 GPS 精度，那么导弹精度就废了。所以他们还**用原子钟去校正 GPS**，如果 GPS 突然跳跃了，原子钟上是可以检测到 GPS 跳跃的，这部分相对有一点黑科技，但是从原理上来讲还是比较简单，比较好理解的。

最开始它 Spanner 最大的用户就是 Google 的 Adwords，这是 Google 最赚钱的业务，Google 就是靠广告生存的，我们一直觉得 Google 是科技公司，但是他的钱是从广告那来的，所以一定程度来讲 Google 是一个广告公司。Google 内部的方向先有了 Big table ，然后有了 MegaStore ，MegaStore 的下一代是 Spanner ，F1 是在 Spanner 上面构建的。

TiDB and TiKV

TiKV 和 TiDB 基本上对应 Google Spanner 和 Google F1，用 Open Source 方式重建。目前这两个项目都开放在 GitHub 上面，两个项目都比较火爆，TiDB 是更早一点开源的， 目前 TiDB 在 GitHub 上 有 4300 多个 Star，每天都在增长。

另外，对于现在的社会来讲，我们觉得 Infrastructure 领域闭源的东西是没有任何生存机会的。没有任何一家公司，愿意把自己的身家性命压在一个闭源的项目上。举一个很典型的例子，在美国有一个数据库叫 FoundationDB，去年被苹果收购了。FoundationDB 之前和用户签的合约都是一年的合约。比如说，我给你服务周期是一年，现在我被另外一个公司收购了，我今年服务到期之后，我是满足合约的。但是其他公司再也不能找它服务了，因为它现在不叫 FoundationDB 了，它叫 Apple了，你不能找 Apple 给你提供一个 Enterprise service。

TiDB 和 TiKV 为什么是两个项目，因为它和 Google 的内部架构对比差不多是这样的：TiKV 对应的是 Spanner，TiDB 对应的是 F1 。F1 里面更强调上层的分布式的 SQL 层到底怎么做，分布式的 Plan 应该怎么做，分布式的 Plan 应该怎么去做优化。同时 TiDB 有一点做的比较好的是，它兼容了 MySQL 协议，当你出现了一个新型的数据库的时候，用户使用它是有成本的。大家都知道作为开发很讨厌的一个事情就是，我要每个语言都写一个 Driver，比如说你要支持 C++、你要支持 Java、你要支持 Go 等等，这个太累了，而且用户还得改他的程序，所以我们选择了一个更加好的东西兼容 MySQL 协议，让用户可以不用改。一会我会用一个视频来演示一下，为什么一行代码不改就可以用，用户就能体会到 TiDB 带来的所有的好处。

这个图实际上是整个协议栈或者是整个软件栈的实现。大家可以看到整个系统是高度分层的，从最底下开始是 RocksDB ，然后再上面用 Raft 构建一层可以被复制的 RocksDB ，在这一层的时候它还没有 Transaction，但是整个系统现在的状态是所有写入的数据一定要保证它复制到了足够多的副本。也就是说只要我写进来的数据一定有足够多的副本去 cover 它，这样才比较安全，在一个比较安全的 Key-value store 上面， 再去构建它的多版本，再去构建它的分布式事务，然后在分布式事务构建完成之后，就可以轻松的加上 SQL 层，再轻松的加上MySQL 协议的支持。然后，这两天我比较好奇，自己写了 MongoDB 协议的支持，然后我们可以用 MongoDB 的客户端来玩，就是说协议这一层是高度可插拔的。TiDB 上可以在上面构建一个 MongoDB 的协议，相当于这个是构建一个 SQL 的协议，可以构建一个 NoSQL 的协议。这一点主要是用来验证 TiKV 在模型上面的支持能力。

这是整个 TiKV 的架构图，从这个看来，整个集群里面有很多 Node，比如这里画了四个 Node ，分别对应了四个机器。每一个 Node 上可以有多个 Store，每个 Store 里面又会有很多小的 Region，就是说一小片数据，就是一个 Region 。从全局来看所有的数据被划分成很多小片，每个小片默认配置是 64M，它已经足够小，可以很轻松的从一个节点移到另外一个节点，Region 1 有三个副本，它分别在 Node1、Node 2 和 Node4 上面， 类似的Region 2，Region 3 也是有三个副本。每个 Region 的所有副本组成一个 Raft Group，整个系统可以看到很多这样的 Raft groups。

Raft 细节我不展开了，大家有兴趣可以找我私聊或者看一下相应的资料。

因为整个系统里面我们可以看到上一张图里面有很多 Raft group 给我们，不同 Raft group 之间的通讯都是有开销的。所以我们有一个类似于 MySQL 的 group commit 机制 ，你发消息的时候实际上可以 share 同一个 connection ， 然后 pipeline + batch 发送，很大程度上可以省掉大量 syscall 的开销。

另外，其实在一定程度上后面我们在支持压缩的时候，也有非常大的帮助，就是可以减少数据的传输。对于整个系统而言，可能有数百万的 Region，它的大小可以调整，比如说 64M、128M、256M，这个实际上依赖于整个系统里面当前的状况。

比如说我们曾经在有一个用户的机房里面做过测试，这个测试有一个香港机房和新加坡的机房。结果我们在做复制的时候，新加坡的机房大于 256M 就复制不过去，因为机房很不稳定，必须要保证数据切的足够小，这样才能复制过去。

如果一个 Region 太大以后我们会自动做 SPLIT，这是非常好玩的过程，有点像细胞的分裂。

然后 TiKV 的 Raft 实现，是从 etcd 里面 port 过来的，为什么要从 etcd 里面 port 过来呢？首先 TiKV 的 Raft 实现是用 Rust 写的。作为第一个做到生产级别的 Raft 实现，所以我们从 etcd 里面把它用 Go 语言写的 port 到这边。

这个是 Raft 官网上面列出来的 TiKV在里面的状态，大家可以看到 TiKV 把所有 Raft 的 feature 都实现了。 比如说 Leader Election、Membership Changes，这个是非常重要的，整个系统的 scale 过程高度依赖 Membership Changes，后面我用一个图来讲这个过程。后面这个是 Log Compaction，这个用户不太关心。

这是很典型的细胞分裂的图，实际上 Region 的分裂过程和这个是类似的。

我们看一下扩容是怎么做的。

比如说以现在的系统假设，我们刚开始说只有三个节点，有 Region1 分别是在 1 、2、4，我用虚线连接起来代表它是一个 Raft group ，大家可以看到整个系统里面有三个 Raft group ，在每一个 Node 上面数据的分布是比较均匀的，在这个假设每一个 Region 是 64M ，相当于只有一个 Node 上面负载比其他的稍微大一点点。

一个在线视频默认我们都是推荐 3 个副本或者 5 个副本的配置。Raft 本身有一个特点，如果一个 leader down 掉之后，其它的节点会选一个新的 leader ，那么这个新的 leader 会把它还没有 commit 但已经 reply 过去的 log 做一个 commit ，然后会再做 apply ，这个有点偏 Raft 协议，细节我不讲了。

复制数据的小的 Region，它实际上是跨多个数据中心做的复制。这里面最重要的一点是永远不丢失数据，无论如何我保证我的复制一定是复制到 majority ，任何时候我只要对外提供服务，允许外面写入数据一定要复制到 majority 。很重要的一点就是恢复的过程一定要是自动化的，我前面已经强调过，如果不能自动化恢复，那么中间的宕机时间或者对外不可服务的时间，便不是由整个系统决定的，这是相对回到了几十年前的状态。

MVCC

MVCC 我稍微仔细讲一下这一块。MVCC 的好处，它很好支持 Lock-free 的 snapshot read ，一会儿我有一个图会展示 MVCC 是怎么做的。isolation level 就不讲了， MySQL 里面的级别是可以调的，我们的 TiKV 有 SI，还有 SI+lock，默认是支持 SI 的这种隔离级别，然后你写一个 select for update 语句，这个会自动的调整到 SI 加上 lock 这个隔离级别。这个隔离级别基本上和 SSI 是一致的。还有一个就是 GC 的问题，如果你的系统里面的数据产生了很多版本，你需要把这个比较老的数据给 GC 掉，比如说正常情况下我们是不删除数据的， 你写入一行，然后再写入一行，不断去 update 同一行的时候，每一次 update 会产生新的版本，新的版本就会在系统里存在，所以我们需要一个 GC 的模块把比较老的数据给 GC 掉，实际上这个 GC 不是 Go 里面的GC，不是 Java 的 GC，而是数据的 GC。

这是一个数据版本，大家可以看到我们的数据分成两块，一个是 meta，一个是 data。meta 相对于描述我的数据当前有多少个版本。大家可以看到绿色的部分，比如说我们的 meta key 是 A ，keyA 有三个版本，是 A1 、A2、A3，我们把 key 自己和 version 拼到一起。那我们用 A1、A2、A3 分别描述 A 的三个版本，那么就是 version 1/2/3。meta 里面描述，就是我的整个 key 相对应哪个版本，我想找到那个版本。比如说我现在要读取 key A 的版本10，但显然现在版本 10 是没有的，那么小于版本 10 最大的版本是 3，所以这时我就能读取到 3，这是它的隔离级别决定的。关于 data，我刚才已经讲过了。

分布式事务模型

接下来是分布式事务模型，其实是基于 Google Percolator，这是 Google 在 2006 发表的一篇论文，是 Google 在做内部增量处理的时候发现了这个方法，本质上还是二阶段提交的。这使用的是一个乐观锁，比如说我提供一个 transaction ，我去改一个东西，改的时候是发布在本地的，并没有马上 commit 到数据存储那一端，这个模型就是说，我修改的东西我马上去 Lock 住，这个基本就是一个悲观锁。但如果到最后一刻我才提交出去，那么锁住的这一小段的时间，这个时候实现的是乐观锁。乐观锁的好处就是当你冲突很小的时候可以得到非常好的性能，因为冲突特别小，所以我本地修改通常都是有效的，所以我不需要去 Lock ，不需要去 roll back 。本质上分布式事务就是 2PC (两阶段提交) 或者是 2+x PC，基本上没有 1PC，除非你在别人的级别上做弱化。比如说我允许你读到当前最新的版本，也允许你读到前面的版本，书里面把这个叫做幻读。如果你调到这个程度是比较容易做 1PC 的，这个实际上还是依赖用户设定的隔离级别的，如果用户需要更高的隔离级别，这个 1PC就不太好做了。

这是一个路由，正常来讲，大家可能会好奇一个 SQL 语句怎么最后会落到存储层，然后能很好的运行，最后怎么能映射到 KV 上面，又怎么能路由到正确的节点，因为整个系统可能有上千个节点，你怎么能正确路由到那一个的节点。我们在 TiDB 有一个 TiKV driver ， 另外 TiKV 对外使用的是 Google Protocol Buffer 来作为通讯的编码格式。

Placement Driver

来说一下 Placement Driver 。Placement Driver 是什么呢？整个系统里面有一个节点，它会时刻知道现在整个系统的状态。比如说每个机器的负载，每个机器的容量，是否有新加的机器，新加机器的容量到底是怎么样的，是不是可以把一部分数据挪过去，是不是也是一样下线， 如果一个节点在十分钟之内无法被其他节点探测到，我认为它已经挂了，不管它实际上是不是真的挂了，但是我也认为它挂了。因为这个时候是有风险的，如果这个机器万一真的挂了，意味着你现在机器的副本数只有两个，有一部分数据的副本数只有两个。那么现在你必须马上要在系统里面重新选一台机器出来，它上面有足够的空间，让我现在只有两个副本的数据重新再做一份新的复制，系统始终维持在三个副本。整个系统里面如果机器挂掉了，副本数少了，这个时候应该会被自动发现，马上补充新的副本，这样会维持整个系统的副本数。这是很重要的 ，为了避免数据丢失，必须维持足够的副本数，因为副本数每少一个，你的风险就会再增加。这就是 Placement Driver 做的事情。

同时，Placement Driver 还会根据性能负载，不断去 move 这个 data 。比如说你这边负载已经很高了，一个磁盘假设有 100G，现在已经用了 80G，另外一个机器上也是 100G，但是他只用了 20G，所以这上面还可以有几十 G 的数据，比如 40G 的数据，你可以 move 过去，这样可以保证系统有很好的负载，不会出现一个磁盘巨忙无比，数据已经多的装不下了，另外一个上面还没有东西，这是 Placement Driver 要做的东西。

Raft 协议还提供一个很高级的特性叫 leader transfer。leader transfer 就是说在我不移动数据的时候，我把我的 leadership 给你，相当于从这个角度来讲，我把流量分给你，因为我是 leader，所以数据会到我这来，但我现在把 leader给你，我让你来当 leader，原来打给我的请求会被打给你，这样我的负载就降下来。这就可以很好的动态调整整个系统的负载，同时又不搬移数据。不搬移数据的好处就是，不会形成一个抖动。

MySQL Sharding

MySQL Sharding 我前面已经提到了它的各种天花板，MySQL Sharding 的方案很典型的就是解决基本问题以后，业务稍微复杂一点，你在 sharding 这一层根本搞不定。它永远需要一个 sharding key，你必须要告诉我的 proxy，我的数据要到哪里找，对用户来说是极不友好的，比如我现在是一个单机的，现在我要切入到一个分布式的环境，这时我必须要改我的代码，我必须要知道我这个 key ，我的 row 应该往哪里 Sharding。如果是用 ORM ，这个基本上就没法做这个事情了。有很多 ORM 它本身假设我后面只有一个 MySQL。但 TiDB 就可以很好的支持，因为我所有的角色都是对的，我不需要关注 Sharding 、分库、分表这类的事情。

这里面有一个很重要的问题没有提，我怎么做 DDL。如果这个表非常大的话，比如说我们有一百亿吧，横跨了四台机器，这个时候你要给它做一个新的 Index，就是我要添加一个新的索引，这个时候你必须要不影响任何现有的业务，实际上这是多阶段提交的算法，这个是 Google 和 F1 一起发出来那篇论文。

简单来讲是这样的，先把状态标记成 delete only ，delete only 是什么意思呢？因为在分布式系统里面，所有的系统对于 schema 的视野不是一致的，比如说我现在改了一个值，有一部分人发现这个值被改了，但是还有一部分人还没有开始访问这个，所以根本不知道它被改了。然后在一个分布系统里，你也不可能实时通知到所有人在同一时刻发现它改变了。比如说从有索引到没有索引，你不能一步切过去，因为有的人认为它有索引，所以他给它建了一个索引，但是另外一个机器他认为它没有索引，所以他就把数据给删了，索引就留在里面了。这样遇到一个问题，我通过索引找的时候告诉我有， 实际数据却没有了，这个时候一致性出了问题。比如说我 count 一个 email 等于多少的，我通过 email 建了一个索引，我认为它是在，但是 UID 再转过去的时候可能已经不存在了。

比如说我先标记成 delete only，我删除它的时候不管它现在有没有索引，我都会尝试删除索引，所以我的数据是干净的。如果我删除掉的话，我不管结果是什么样的，我尝试去删一下，可能这个索引还没 build 出来，但是我仍然删除，如果数据没有了，索引一定没有了，所以这可以很好的保持它的一致性。后面再类似于前面，先标记成 write only 这种方式，连续再迭代这个状态，就可以迭代到一个最终可以对外公开的状态。比如说当我迭代到一定程度的时候，我可以从后台 build index ，比如说我一百亿，正在操作的 index 会马上 build，但是还有很多没有 build index ，这个时候后台不断的跑 map-reduce 去 build index ，直到整个都 build 完成之后，再对外 public ，就是说我这个索引已经可用了，你可以直接拿索引来找，这个是非常经典的。在这个 Online，Asynchronous Schema Change in F1 paper之前，大家都不知道这事该怎么做。

Proxy Sharding 的方案不支持分布式事务，更不用说跨数据中心的一致性事务了。 TiKV 很好的支持 transaction，刚才提到的 Raft 除了增加副本之外，还有 leader transfer，这是一个传统的方案都无法提供的特性。以及它带来的好处，当我瞬间平衡整个系统负载的时候，对外是透明的， 做 leader transfer 的时候并不需要移动数据，只是个简单的 leader transfer 消息。

然后说一下如果大家想参与我们项目的话是怎样的过程，因为整个系统是完全开源的，如果大家想参与其中任何一部分都可以，比如说我想参与到分布式 KV，可以直接贡献到 TiKV。TiKV 需要写 Rust，如果大家对这块特别有激情可以体验写 Rust 的感觉 。

TiDB 是用 Go 写的，Go 在中国的群众基础是非常多的，目前也有很多人在贡献。整个 TiDB 和TiKV 是高度协作的项目，因为 TiDB 目前还用到了 etcd ，我们在和 CoreOS 在密切的合作，也特别感谢 CoreOS 帮我们做了很多的支持，我们也为 CoreOS 的 etcd 提了一些 patch。同时，TiKV 使用 RocksDB ，所以我们也为 RocksDB 提了一些 patch 和 test，我们也非常感谢 Facebook RocksDB team 对我们项目的支持。

另外一个是 PD，就是我们前面提的 Placement Driver，它负责监控整个系统。这部分的算法比较好玩，大家如果有兴趣的话，可以去自己控制整个集群的调度，它和 Kubernetes 或者是Mesos 的调度算法是不一样的，因为它调度的维度实际上比那个要更多。比如说磁盘的容量，你的 leader 的数量，你的网络当前的使用情况，你的 IO 的负载和 CPU 的负载都可以放进去。同时你还可以让它调度不要跨一个机房里面建多个副本。

感谢[魏星](http://www.infoq.com/cn/author/%E9%AD%8F%E6%98%9F)对本文的审校。

给InfoQ中文站投稿或者参与内容翻译工作，请邮件至[editors@cn.infoq.com](mailto:editors@cn.infoq.com)。也欢迎大家通过新浪微博（[@InfoQ](http://www.weibo.com/infoqchina)，[@丁晓昀](http://weibo.com/u/1451714913)），微信（微信号：[InfoQChina](http://weixin.sogou.com/gzh?openid=oIWsFt0HnZ93MfLi3pW2ggVJFRxY)）关注我们。

**怎样打造一个分布式数据库**

转载 2016年09月21日 19:57:00

* 7341

在技术方面，我自己热衷于 Open Source，写了很多 Open Source 的东西，擅长的是 Infrastructure 领域。Infrastructure 领域现在范围很广，比如说很典型的分布式 Scheduler、Mesos、Kubernetes，另外它和 Microservices 所结合的东西也特别多。Infrastructure 领域还有比如 Database 有分 AP（分析型）和 TP（事务型），比如说很典型的大家知道的 Spark、Greenplum、Apache Phoenix 等等，这些都属于在 AP 的，它们也会去尝试支持有限的 TP。另外，还有一个比较有意思的就是 Kudu——Cloudera Open Source 的那个项目，它的目标很有意思：我不做最强的 AP 系统，也不做最强的 TP 系统，我选择一个相对折中的方案。从文化哲学上看，它比较符合中国的中庸思想。

另外，我先后创建了 Codis、TiDB。去年12月份创建了 TiKV 这个 project，TiKV 在所有的 rust 项目里目前排名前三。

首先我们聊聊 Database 的历史，在已经有这么多种数据库的背景下我们为什么要创建另外一个数据库；以及说一下现在方案遇到的困境，说一下 Google Spanner 和 F1、TiKV 和 TiDB，说一下架构的事情，在这里我们会重点聊一下 TiKV。因为我们产品的很多特性是 TiKV 提供的，比如说跨数据中心的复制、Transaction、auto-scale。

接下来聊一下为什么 TiKV 用 Raft 能实现所有这些重要的特性，以及 scale、MVCC 和事务模型。东西非常多，我今天不太可能把里面的技术细节都描述得特别细，因为几乎每一个话题都可以找到一篇或者是多篇论文，所以详细的技术问题大家可以单独来找我聊。

后面再说一下我们现在遇到的窘境，就是大家常规遇到的分布式方案有哪些问题，比如 MySQL Sharding。我们创建了无数 MySQL Proxy，比如官方的 MySQL proxy、Youtube 的 Vitess、淘宝的 Cobar、TDDL以及基于 Cobar 的 MyCAT、金山的 Kingshard、360 的 Atlas、京东的 JProxy，我在豌豆荚也写了一个。可以说，随便一个大公司都会造一个 MySQL Sharding 的方案。

为什么我们要创建另外一个数据库？

昨天晚上我还跟一个同学聊到，基于 MySQL 的方案它的天花板在哪里，它的天花板特别明显。有一个思路是能不能通过 MySQL 的 server 把 InnoDB 变成一个分布式数据库，听起来这个方案很完美，但是很快就会遇到天花板。因为 MySQL 生成的执行计划是个单机的，它认为整个计划的 cost 也是单机的，我读取一行和读取下一行之间的开销是很小的，比如迭代 next row 可以立刻拿到下一行。实际上在一个分布式系统里面，这是不一定的。

另外，你把数据都拿回来计算这个太慢了，很多时候我们需要把我们的 expression 或者计算过程等等运算推下去，向上返回一个最终的计算结果，这个一定要用分布式的 plan，前面控制执行计划的节点，它必须要理解下面是分布式的东西，才能生成最好的 plan，这样才能实现最高的执行效率。

比如说你做一个 sum，你是一条条拿回来加，还是让一堆机器一起算，最后给我一个结果。 例如我有 100 亿条数据分布在 10 台机器上，并行在这 10台机器我可能只拿到 10 个结果，如果把所有的数据每一条都拿回来，这就太慢了，完全丧失了分布式的价值。聊到 MySQL 想实现分布式，另外一个实现分布式的方案就是 Proxy。但是 Proxy 本身的天花板在那里，就是它不支持分布式的 transaction，它不支持跨节点的 join，它无法理解复杂的 plan，一个复杂的 plan 打到 Proxy 上面，Proxy 就傻了，我到底应该往哪一个节点上转发呢，如果我涉及到 subquery sql 怎么办？所以这个天花板是瞬间会到，在传统模型下面的修改，很快会达不到我们的要求。

另外一个很重要的是，MySQL 支持的复制方式是半同步或者是异步，但是半同步可以降级成异步，也就是说任何时候数据出了问题你不敢切换，因为有可能是异步复制，有一部分数据还没有同步过来，这时候切换数据就不一致了。前一阵子出现过某公司突然不能支付了这种事件，今年有很多这种类似的 case，所以微博上大家都在说“说好的异地多活呢？”……

为什么传统的方案在这上面解决起来特别的困难，天花板马上到了，基本上不可能解决这个问题。另外是多数据中心的复制和数据中心的容灾，MySQL 在这上面是做不好的。

在前面三十年基本上是关系数据库的时代，那个时代创建了很多伟大的公司，比如说 IBM、Oracle、微软也有自己的数据库，早期还有一个公司叫 Sybase，有一部分特别老的程序员同学在当年的教程里面还可以找到这些东西，但是现在基本上看不到了。

另外是 NoSQL。NoSQL 也是一度非常火，像 Cassandra、MongoDB 等等，这些都属于在互联网快速发展的时候创建这些能够 scale 的方案，但 Redis scale 出来比较晚，所以很多时候大家把 Redis 当成一个 Cache，现在慢慢大家把它当成存储不那么重要的数据的数据库。因为它有了 scale 支持以后，大家会把更多的数据放在里面。

然后到了 2015，严格来讲是到 2014 年到 2015 年之间，Raft 论文发表以后，真正的 NewSQL 的理论基础终于完成了。我觉得 NewSQL 这个理论基础，最重要的划时代的几篇论文，一个是谷歌的 Spanner，是在 2013 年初发布的；再就是 Raft 是在 2014 年上半年发布的。这几篇相当于打下了分布式数据库 NewSQL 的理论基础，这个模型是非常重要的，如果没有模型在上面是堆不起来东西的。说到现在，大家可能对于模型还是可以理解的，但是对于它的实现难度很难想象。

前面我大概提到了我们为什么需要另外一个数据库，说到 Scalability 数据的伸缩，然后我们讲到需要 SQL，比如你给我一个纯粹的 key-velue 系统的 API，比如我要查找年龄在 10 岁到 20 岁之间的 email 要满足一个什么要求的。如果只有 KV 的 API 这是会写死人的，要写很多代码，但是实际上用 SQL 写一句话就可以了，而且 SQL 的优化器对整个数据的分布是知道的，它可以很快理解你这个 SQL，然后会得到一个最优的 plan，他得到这个最优的 plan 基本上等价于一个真正理解 KV 每一步操作的人写出来的程序。通常情况下，SQL 的优化器是为了更加了解或者做出更好的选择。

另外一个就是 ACID 的事务，这是传统数据库必须要提供的基础。以前你不提供 ACID 就不能叫数据库，但是近些年大家写一个内存的 map 也可以叫自己是数据库。大家写一个 append-only 文件，我们也可以叫只读数据库，数据库的概念比以前极大的泛化了。

另外就是高可用和自动恢复，他们的概念是什么呢？有些人会有一些误解，因为今天还有朋友在现场问到，出了故障，比如说一个机房挂掉以后我应该怎么做切换，怎么操作。这个实际上相当于还是上一代的概念，还需要人去干预，这种不算是高可用。

未来的高可用一定是系统出了问题马上可以自动恢复，马上可以变成可用。比如说一个机房挂掉了，十秒钟不能支付，十秒钟之后系统自动恢复了变得可以支付，即使这个数据中心再也不起来我整个系统仍然是可以支付的。Auto-Failover 的重要性就在这里。大家不希望在睡觉的时候被一个报警给拉起来，我相信大家以后具备这样一个能力，5 分钟以内的报警不用理会，挂掉一个机房，又挂掉一个机房，这种连续报警才会理。我们内部开玩笑说，希望大家都能睡个好觉，很重要的事情就是这个。

说完应用层的事情，现在很有很多业务，在应用层自己去分片，比如说我按照 user ID在代码里面分片，还有一部分是更高级一点我会用到一致性哈希。问题在于它的复杂度，到一定程度之后我自动的分库，自动的分表，我觉得下一代数据库是不需要理解这些东西的，不需要了解什么叫做分库，不需要了解什么叫做分表，因为系统是全部自动搞定的。同时复杂度，如果一个应用不支持事务，那么在应用层去做，通常的做法是引入一个外部队列，引入大量的程序机制和状态转换，A 状态的时候允许转换到 B 状态，B 状态允许转换到 C 状态。

举一个简单的例子，比如说在京东上买东西，先下订单，支付状态之后这个商品才能出库，如果不是支付状态一定不能出库，每一步都有严格的流程。

Google Spanner / F1

说一下 Google 的 Spanner 和 F1，这是我非常喜欢的论文，也是我最近几年看过很多遍的论文。 Google Spanner 已经强大到什么程度呢？Google Spanner 是全球分布的数据库，在国内目前普遍做法叫做同城两地三中心，它们的差别是什么呢？以 Google 的数据来讲，谷歌比较高的级别是他们有 7 个副本，通常是美国保存 3 个副本，再在另外 2 个国家可以保存 2 个副本，这样的好处是万一美国两个数据中心出了问题，那整个系统还能继续可用，这个概念就是比如美国 3 个副本全挂了，整个数据都还在，这个数据安全级别比很多国家的安全级别还要高，这是 Google 目前做到的，这是全球分布的好处。

现在国内主流的做法是两地三中心，但现在基本上都不能自动切换。大家可以看到很多号称实现了两地三中心或者异地多活，但是一出现问题都说不好意思这段时间我不能提供服务了。大家无数次的见到这种 case， 我就不列举了。

Spanner 现在也提供一部分 SQL 特性。在以前，大部分 SQL 特性是在 F1 里面提供的，现在 Spanner 也在逐步丰富它的功能，Google 是全球第一个做到这个规模或者是做到这个级别的数据库。事务支持里面 Google 有点黑科技（其实也没有那么黑），就是它有**GPS 时钟和原子钟**。大家知道在分布式系统里面，比如说数千台机器，两个事务启动先后顺序，这个顺序怎么界定(事务外部一致性)。这个时候 Google 内部使用了 GPS 时钟和原子钟，正常情况下它会使用一个GPS 时钟的一个集群，就是说我拿的一个时间戳，并不是从一个 GPS 上来拿的时间戳，因为大家知道所有的硬件都会有误差。如果这时候我从一个上拿到的 GPS 本身有点问题，那么你拿到的这个时钟是不精确的。而 Google 它实际上是在一批 GPS 时钟上去拿了能够满足 majority 的精度，再用时间的算法，得到一个比较精确的时间。大家知道 GPS 也不太安全，因为它是美国军方的，对于 Google 来讲要实现比国家安全级别更高的数据库，而 GPS 是可能受到干扰的，因为 GPS 信号是可以调整的，这在军事用途上面很典型的，大家知道导弹的制导需要依赖 GPS，如果调整了 GPS 精度，那么导弹精度就废了。所以他们还**用原子钟去校正 GPS**，如果 GPS 突然跳跃了，原子钟上是可以检测到 GPS 跳跃的，这部分相对有一点黑科技，但是从原理上来讲还是比较简单，比较好理解的。

最开始它 Spanner 最大的用户就是 Google 的 Adwords，这是 Google 最赚钱的业务，Google 就是靠广告生存的，我们一直觉得 Google 是科技公司，但是他的钱是从广告那来的，所以一定程度来讲 Google 是一个广告公司。Google 内部的方向先有了 Big table ，然后有了 MegaStore ，MegaStore 的下一代是 Spanner ，F1 是在 Spanner 上面构建的。

TiDB and TiKV

TiKV 和 TiDB 基本上对应 Google Spanner 和 Google F1，用 Open Source 方式重建。目前这两个项目都开放在 GitHub 上面，两个项目都比较火爆，TiDB 是更早一点开源的， 目前 TiDB 在 GitHub 上 有 4300 多个 Star，每天都在增长。

另外，对于现在的社会来讲，我们觉得 Infrastructure 领域闭源的东西是没有任何生存机会的。没有任何一家公司，愿意把自己的身家性命压在一个闭源的项目上。举一个很典型的例子，在美国有一个数据库叫 FoundationDB，去年被苹果收购了。FoundationDB 之前和用户签的合约都是一年的合约。比如说，我给你服务周期是一年，现在我被另外一个公司收购了，我今年服务到期之后，我是满足合约的。但是其他公司再也不能找它服务了，因为它现在不叫 FoundationDB 了，它叫 Apple了，你不能找 Apple 给你提供一个 Enterprise service。

TiDB 和 TiKV 为什么是两个项目，因为它和 Google 的内部架构对比差不多是这样的：TiKV 对应的是 Spanner，TiDB 对应的是 F1 。F1 里面更强调上层的分布式的 SQL 层到底怎么做，分布式的 Plan 应该怎么做，分布式的 Plan 应该怎么去做优化。同时 TiDB 有一点做的比较好的是，它兼容了 MySQL 协议，当你出现了一个新型的数据库的时候，用户使用它是有成本的。大家都知道作为开发很讨厌的一个事情就是，我要每个语言都写一个 Driver，比如说你要支持 C++、你要支持 Java、你要支持 Go 等等，这个太累了，而且用户还得改他的程序，所以我们选择了一个更加好的东西兼容 MySQL 协议，让用户可以不用改。一会我会用一个视频来演示一下，为什么一行代码不改就可以用，用户就能体会到 TiDB 带来的所有的好处。

这个图实际上是整个协议栈或者是整个软件栈的实现。大家可以看到整个系统是高度分层的，从最底下开始是 RocksDB ，然后再上面用 Raft 构建一层可以被复制的 RocksDB ，在这一层的时候它还没有 Transaction，但是整个系统现在的状态是所有写入的数据一定要保证它复制到了足够多的副本。也就是说只要我写进来的数据一定有足够多的副本去 cover 它，这样才比较安全，在一个比较安全的 Key-value store 上面， 再去构建它的多版本，再去构建它的分布式事务，然后在分布式事务构建完成之后，就可以轻松的加上 SQL 层，再轻松的加上MySQL 协议的支持。然后，这两天我比较好奇，自己写了 MongoDB 协议的支持，然后我们可以用 MongoDB 的客户端来玩，就是说协议这一层是高度可插拔的。TiDB 上可以在上面构建一个 MongoDB 的协议，相当于这个是构建一个 SQL 的协议，可以构建一个 NoSQL 的协议。这一点主要是用来验证 TiKV 在模型上面的支持能力。

这是整个 TiKV 的架构图，从这个看来，整个集群里面有很多 Node，比如这里画了四个 Node ，分别对应了四个机器。每一个 Node 上可以有多个 Store，每个 Store 里面又会有很多小的 Region，就是说一小片数据，就是一个 Region 。从全局来看所有的数据被划分成很多小片，每个小片默认配置是 64M，它已经足够小，可以很轻松的从一个节点移到另外一个节点，Region 1 有三个副本，它分别在 Node1、Node 2 和 Node4 上面， 类似的Region 2，Region 3 也是有三个副本。每个 Region 的所有副本组成一个 Raft Group，整个系统可以看到很多这样的 Raft groups。

Raft 细节我不展开了，大家有兴趣可以找我私聊或者看一下相应的资料。

因为整个系统里面我们可以看到上一张图里面有很多 Raft group 给我们，不同 Raft group 之间的通讯都是有开销的。所以我们有一个类似于 MySQL 的 group commit 机制 ，你发消息的时候实际上可以 share 同一个 connection ， 然后 pipeline + batch 发送，很大程度上可以省掉大量 syscall 的开销。

另外，其实在一定程度上后面我们在支持压缩的时候，也有非常大的帮助，就是可以减少数据的传输。对于整个系统而言，可能有数百万的 Region，它的大小可以调整，比如说 64M、128M、256M，这个实际上依赖于整个系统里面当前的状况。

比如说我们曾经在有一个用户的机房里面做过测试，这个测试有一个香港机房和新加坡的机房。结果我们在做复制的时候，新加坡的机房大于 256M 就复制不过去，因为机房很不稳定，必须要保证数据切的足够小，这样才能复制过去。

如果一个 Region 太大以后我们会自动做 SPLIT，这是非常好玩的过程，有点像细胞的分裂。

然后 TiKV 的 Raft 实现，是从 etcd 里面 port 过来的，为什么要从 etcd 里面 port 过来呢？首先 TiKV 的 Raft 实现是用 Rust 写的。作为第一个做到生产级别的 Raft 实现，所以我们从 etcd 里面把它用 Go 语言写的 port 到这边。

这个是 Raft 官网上面列出来的 TiKV在里面的状态，大家可以看到 TiKV 把所有 Raft 的 feature 都实现了。 比如说 Leader Election、Membership Changes，这个是非常重要的，整个系统的 scale 过程高度依赖 Membership Changes，后面我用一个图来讲这个过程。后面这个是 Log Compaction，这个用户不太关心。

这是很典型的细胞分裂的图，实际上 Region 的分裂过程和这个是类似的。

我们看一下扩容是怎么做的。

比如说以现在的系统假设，我们刚开始说只有三个节点，有 Region1 分别是在 1 、2、4，我用虚线连接起来代表它是一个 Raft group ，大家可以看到整个系统里面有三个 Raft group ，在每一个 Node 上面数据的分布是比较均匀的，在这个假设每一个 Region 是 64M ，相当于只有一个 Node 上面负载比其他的稍微大一点点。

一个在线视频默认我们都是推荐 3 个副本或者 5 个副本的配置。Raft 本身有一个特点，如果一个 leader down 掉之后，其它的节点会选一个新的 leader ，那么这个新的 leader 会把它还没有 commit 但已经 reply 过去的 log 做一个 commit ，然后会再做 apply ，这个有点偏 Raft 协议，细节我不讲了。

复制数据的小的 Region，它实际上是跨多个数据中心做的复制。这里面最重要的一点是永远不丢失数据，无论如何我保证我的复制一定是复制到 majority ，任何时候我只要对外提供服务，允许外面写入数据一定要复制到 majority 。很重要的一点就是恢复的过程一定要是自动化的，我前面已经强调过，如果不能自动化恢复，那么中间的宕机时间或者对外不可服务的时间，便不是由整个系统决定的，这是相对回到了几十年前的状态。

MVCC

MVCC 我稍微仔细讲一下这一块。MVCC 的好处，它很好支持 Lock-free 的 snapshot read ，一会儿我有一个图会展示 MVCC 是怎么做的。isolation level 就不讲了， MySQL 里面的级别是可以调的，我们的 TiKV 有 SI，还有 SI+lock，默认是支持 SI 的这种隔离级别，然后你写一个 select for update 语句，这个会自动的调整到 SI 加上 lock 这个隔离级别。这个隔离级别基本上和 SSI 是一致的。还有一个就是 GC 的问题，如果你的系统里面的数据产生了很多版本，你需要把这个比较老的数据给 GC 掉，比如说正常情况下我们是不删除数据的， 你写入一行，然后再写入一行，不断去 update 同一行的时候，每一次 update 会产生新的版本，新的版本就会在系统里存在，所以我们需要一个 GC 的模块把比较老的数据给 GC 掉，实际上这个 GC 不是 Go 里面的GC，不是 Java 的 GC，而是数据的 GC。

这是一个数据版本，大家可以看到我们的数据分成两块，一个是 meta，一个是 data。meta 相对于描述我的数据当前有多少个版本。大家可以看到绿色的部分，比如说我们的 meta key 是 A ，keyA 有三个版本，是 A1 、A2、A3，我们把 key 自己和 version 拼到一起。那我们用 A1、A2、A3 分别描述 A 的三个版本，那么就是 version 1/2/3。meta 里面描述，就是我的整个 key 相对应哪个版本，我想找到那个版本。比如说我现在要读取 key A 的版本10，但显然现在版本 10 是没有的，那么小于版本 10 最大的版本是 3，所以这时我就能读取到 3，这是它的隔离级别决定的。关于 data，我刚才已经讲过了。

分布式事务模型

接下来是分布式事务模型，其实是基于 Google Percolator，这是 Google 在 2006 发表的一篇论文，是 Google 在做内部增量处理的时候发现了这个方法，本质上还是二阶段提交的。这使用的是一个乐观锁，比如说我提供一个 transaction ，我去改一个东西，改的时候是发布在本地的，并没有马上 commit 到数据存储那一端，这个模型就是说，我修改的东西我马上去 Lock 住，这个基本就是一个悲观锁。但如果到最后一刻我才提交出去，那么锁住的这一小段的时间，这个时候实现的是乐观锁。乐观锁的好处就是当你冲突很小的时候可以得到非常好的性能，因为冲突特别小，所以我本地修改通常都是有效的，所以我不需要去 Lock ，不需要去 roll back 。本质上分布式事务就是 2PC (两阶段提交) 或者是 2+x PC，基本上没有 1PC，除非你在别人的级别上做弱化。比如说我允许你读到当前最新的版本，也允许你读到前面的版本，书里面把这个叫做幻读。如果你调到这个程度是比较容易做 1PC 的，这个实际上还是依赖用户设定的隔离级别的，如果用户需要更高的隔离级别，这个 1PC就不太好做了。

这是一个路由，正常来讲，大家可能会好奇一个 SQL 语句怎么最后会落到存储层，然后能很好的运行，最后怎么能映射到 KV 上面，又怎么能路由到正确的节点，因为整个系统可能有上千个节点，你怎么能正确路由到那一个的节点。我们在 TiDB 有一个 TiKV driver ， 另外 TiKV 对外使用的是 Google Protocol Buffer 来作为通讯的编码格式。

Placement Driver

来说一下 Placement Driver 。Placement Driver 是什么呢？整个系统里面有一个节点，它会时刻知道现在整个系统的状态。比如说每个机器的负载，每个机器的容量，是否有新加的机器，新加机器的容量到底是怎么样的，是不是可以把一部分数据挪过去，是不是也是一样下线， 如果一个节点在十分钟之内无法被其他节点探测到，我认为它已经挂了，不管它实际上是不是真的挂了，但是我也认为它挂了。因为这个时候是有风险的，如果这个机器万一真的挂了，意味着你现在机器的副本数只有两个，有一部分数据的副本数只有两个。那么现在你必须马上要在系统里面重新选一台机器出来，它上面有足够的空间，让我现在只有两个副本的数据重新再做一份新的复制，系统始终维持在三个副本。整个系统里面如果机器挂掉了，副本数少了，这个时候应该会被自动发现，马上补充新的副本，这样会维持整个系统的副本数。这是很重要的 ，为了避免数据丢失，必须维持足够的副本数，因为副本数每少一个，你的风险就会再增加。这就是 Placement Driver 做的事情。

同时，Placement Driver 还会根据性能负载，不断去 move 这个 data 。比如说你这边负载已经很高了，一个磁盘假设有 100G，现在已经用了 80G，另外一个机器上也是 100G，但是他只用了 20G，所以这上面还可以有几十 G 的数据，比如 40G 的数据，你可以 move 过去，这样可以保证系统有很好的负载，不会出现一个磁盘巨忙无比，数据已经多的装不下了，另外一个上面还没有东西，这是 Placement Driver 要做的东西。

Raft 协议还提供一个很高级的特性叫 leader transfer。leader transfer 就是说在我不移动数据的时候，我把我的 leadership 给你，相当于从这个角度来讲，我把流量分给你，因为我是 leader，所以数据会到我这来，但我现在把 leader给你，我让你来当 leader，原来打给我的请求会被打给你，这样我的负载就降下来。这就可以很好的动态调整整个系统的负载，同时又不搬移数据。不搬移数据的好处就是，不会形成一个抖动。

MySQL Sharding

MySQL Sharding 我前面已经提到了它的各种天花板，MySQL Sharding 的方案很典型的就是解决基本问题以后，业务稍微复杂一点，你在 sharding 这一层根本搞不定。它永远需要一个 sharding key，你必须要告诉我的 proxy，我的数据要到哪里找，对用户来说是极不友好的，比如我现在是一个单机的，现在我要切入到一个分布式的环境，这时我必须要改我的代码，我必须要知道我这个 key ，我的 row 应该往哪里 Sharding。如果是用 ORM ，这个基本上就没法做这个事情了。有很多 ORM 它本身假设我后面只有一个 MySQL。但 TiDB 就可以很好的支持，因为我所有的角色都是对的，我不需要关注 Sharding 、分库、分表这类的事情。

这里面有一个很重要的问题没有提，我怎么做 DDL。如果这个表非常大的话，比如说我们有一百亿吧，横跨了四台机器，这个时候你要给它做一个新的 Index，就是我要添加一个新的索引，这个时候你必须要不影响任何现有的业务，实际上这是多阶段提交的算法，这个是 Google 和 F1 一起发出来那篇论文。

简单来讲是这样的，先把状态标记成 delete only ，delete only 是什么意思呢？因为在分布式系统里面，所有的系统对于 schema 的视野不是一致的，比如说我现在改了一个值，有一部分人发现这个值被改了，但是还有一部分人还没有开始访问这个，所以根本不知道它被改了。然后在一个分布系统里，你也不可能实时通知到所有人在同一时刻发现它改变了。比如说从有索引到没有索引，你不能一步切过去，因为有的人认为它有索引，所以他给它建了一个索引，但是另外一个机器他认为它没有索引，所以他就把数据给删了，索引就留在里面了。这样遇到一个问题，我通过索引找的时候告诉我有， 实际数据却没有了，这个时候一致性出了问题。比如说我 count 一个 email 等于多少的，我通过 email 建了一个索引，我认为它是在，但是 UID 再转过去的时候可能已经不存在了。

比如说我先标记成 delete only，我删除它的时候不管它现在有没有索引，我都会尝试删除索引，所以我的数据是干净的。如果我删除掉的话，我不管结果是什么样的，我尝试去删一下，可能这个索引还没 build 出来，但是我仍然删除，如果数据没有了，索引一定没有了，所以这可以很好的保持它的一致性。后面再类似于前面，先标记成 write only 这种方式，连续再迭代这个状态，就可以迭代到一个最终可以对外公开的状态。比如说当我迭代到一定程度的时候，我可以从后台 build index ，比如说我一百亿，正在操作的 index 会马上 build，但是还有很多没有 build index ，这个时候后台不断的跑 map-reduce 去 build index ，直到整个都 build 完成之后，再对外 public ，就是说我这个索引已经可用了，你可以直接拿索引来找，这个是非常经典的。在这个 Online，Asynchronous Schema Change in F1 paper之前，大家都不知道这事该怎么做。

Proxy Sharding 的方案不支持分布式事务，更不用说跨数据中心的一致性事务了。 TiKV 很好的支持 transaction，刚才提到的 Raft 除了增加副本之外，还有 leader transfer，这是一个传统的方案都无法提供的特性。以及它带来的好处，当我瞬间平衡整个系统负载的时候，对外是透明的， 做 leader transfer 的时候并不需要移动数据，只是个简单的 leader transfer 消息。

然后说一下如果大家想参与我们项目的话是怎样的过程，因为整个系统是完全开源的，如果大家想参与其中任何一部分都可以，比如说我想参与到分布式 KV，可以直接贡献到 TiKV。TiKV 需要写 Rust，如果大家对这块特别有激情可以体验写 Rust 的感觉 。

TiDB 是用 Go 写的，Go 在中国的群众基础是非常多的，目前也有很多人在贡献。整个 TiDB 和TiKV 是高度协作的项目，因为 TiDB 目前还用到了 etcd ，我们在和 CoreOS 在密切的合作，也特别感谢 CoreOS 帮我们做了很多的支持，我们也为 CoreOS 的 etcd 提了一些 patch。同时，TiKV 使用 RocksDB ，所以我们也为 RocksDB 提了一些 patch 和 test，我们也非常感谢 Facebook RocksDB team 对我们项目的支持。

另外一个是 PD，就是我们前面提的 Placement Driver，它负责监控整个系统。这部分的算法比较好玩，大家如果有兴趣的话，可以去自己控制整个集群的调度，它和 Kubernetes 或者是Mesos 的调度算法是不一样的，因为它调度的维度实际上比那个要更多。比如说磁盘的容量，你的 leader 的数量，你的网络当前的使用情况，你的 IO 的负载和 CPU 的负载都可以放进去。同时你还可以让它调度不要跨一个机房里面建多个副本。

感谢[魏星](http://www.infoq.com/cn/author/%E9%AD%8F%E6%98%9F)对本文的审校。

给InfoQ中文站投稿或者参与内容翻译工作，请邮件至[editors@cn.infoq.com](mailto:editors@cn.infoq.com)。也欢迎大家通过新浪微博（[@InfoQ](http://www.weibo.com/infoqchina)，[@丁晓昀](http://weibo.com/u/1451714913)），微信（微信号：[InfoQChina](http://weixin.sogou.com/gzh?openid=oIWsFt0HnZ93MfLi3pW2ggVJFRxY)）关注我们。