# 概述

有很多思路可以实现关系数据库的可扩展性。例如，在应用层划分数据，将不同的数据分片划分到不同的关系数据库上，如MySQL Sharding；或者在关系型数据库内部支持数据自动分片，如Microsoft SQL Azure；或者干脆从存储引擎开始重写一个全新的分布式数据库，如Google Spanner以及Alibaba OceanBase。

# 分类

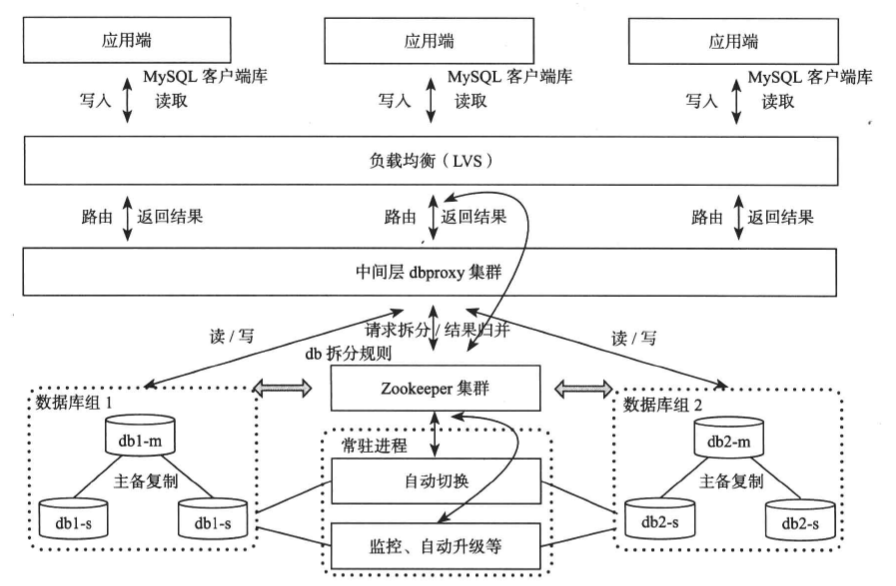
## 数据库中间层

为了扩展关系数据库，最简单也是最为常见的做法就是应用层按照规则将数据拆分为多个分片，分布到多个数据库节点，并引入一个中间层来对应用屏蔽后端的数据库拆分细节。

### 架构

以MySQL Sharding架构为例，分为几个部分：中间层dbproxy集群、数据库组、元数据服务器、常驻进程。

其架构如下：



1. MySQL客户端库

应用程序通过MySQL原生的客户端与系统交互，支持JDBC，原有的单机访问数据库程序可以无缝迁移。

1. 中间层dbproxy

中间层解析客户端SQL请求并转发到后端的数据库。具体来讲，它解析MySQL协议，执行SQL路由，SQL过滤，读写分离，结果归并，排序以及分组等等。中间层由多个无状态的dbproxy进程组成，不存在单点故障的情况。另外，可以在客户端与中间层之间引入LVS（Linux Virtual Server）对客户端请求进行负载均衡。需要注意的是，引入LVS后，客户端请求需要额外增加一层通信开销，因此，常见的作答是直接在客户端配置中间层服务器列表，由客户端处理请求负载均衡以及中间层服务器故障等情况。

1. 数据库组dbagent

每个dbgroup由N台数据库机器组成，其中一台为主机（Master），另外N-1台为备机（Slave）。主机负责所有的写事务及强一致性读事务，并将操作以binlog的形式复制到备机，备机可以支持有一定延迟的读事务。

1. 元数据服务器

元数据服务器主要负责维护dbgroup拆分规则并用于dbgroup选主。Dbproxy通过元数据服务器获取拆分规则从而确定SQL语句的执行计划。元数据服务器本身也需要多个副本实现HA，一种长江的方式就是采用Zookeeper实现。

1. 常驻进程agent

部署在每台服务器上的常驻进程，用于实现监控，单点切换，安装，卸载程序等。Dbgroup中的数据库需要进行主备切换，软件升级等，这些控制逻辑需要与数据库读写事务处理逻辑隔离开来。

假设数据库按照用户哈希分区，同一个用户的数据分布在一个数据库组上。如果SQL请求只涉及同一个用户（这对于大多数应用都是成立的），那么，中间层将请求转发给相应的数据库组，等待返回结果并将结果返回给客户端；如果SQL请求涉及多个用户，那么中间层需要转发给多个数据库组，等待返回结果并将结果执行合并、分组、排序等操作后返回客户端。由于中间层的协议与MySQL兼容，客户端完全感受不到与访问单台MySQL机器之间的差别。

### 扩容

MySQL Sharding集群一般按照用户id进行哈希分区，这里面存在两个问题：

1. 集群的容量不够怎么办？
2. 单个用户的数据量太大怎么办？

对于第1个问题，MySQL Sharding集群往往会采用双倍扩容的方案，即从2台服务器扩到4台，接着再扩到8台......，以此类推。

假设原来有2个dbgroup，第一个dbgroup的主机为A0，备机为A1，第二个dbgroup的主机为B0，备机为B1。按照用户id哈希取模，结果为奇数的用户分布在第一个dbgroup，结果为偶数的用户分布在第二个dbgroup。常见的一种扩容方式如下：

1. 等待A0和B0的数据同步到其备机服务器，即A1和B1。
2. 停止写服务，等待主备完全同步后解除A0与A1、B0与B1的主备关系。
3. 修改中间层的映射规则，将哈希值模4等于1的用户数据映射到A1，哈希值模4等于3的用户数据映射到B1。
4. 开启写服务，用户id哈希值模4等于0、1、2、3的数据将分别写入到A0、A1、B0、B1。这就相当于有一半的数据分别从A0、B0迁移到A1、B1。
5. 分别给A0、A1、B0、B1增加一台备机。

最终，集群由2个dbgroup变为4个dbgroup。可以看到，扩容过程需要停一会服务，另外，扩容进行过程中如果再次发生服务器故障，将使扩容变得非常复杂，很难做到完全自动化。

对于第2个问题，可以在应用层定期统计大用户，并且将这些用户的数据按照数据量拆分到多个dbgroup。当然，定期维护这些信息对于应用层是一个很大的代价。

### 讨论

引入数据库中间层将后端分库分表对应用透明化在大型互联网公司内部很常见。这种作答实现简单，对应用友好，但是也面临一些问题：

1. 数据库复制：MySQL主备之间只支持异步复制，而且主库压力较大时可能产生很大的延迟，因此，主备切换可能会丢失最后一部分更新事务，这时往往需要人工介入。
2. 扩容问题：如果系统压力过大需要增加新的机器，这个过程涉及数据重新划分，整个过程比较复杂且容易出错。
3. 动态数据迁移问题：如果某个数据库压力过大，需要将其中部分数据迁移出去，迁移过程需要总控节点整体协调，以及数据库节点的配合。这个过程很难做到自动化。

针对上述问题，我们自研的MySQL Sharding解决方案如下：

1. 主从复制采用一主多从，只要有一个备机返回同步成功则返回客户端主从复制完成，其余备机异步去做同步。
2. 针对单表或者少量数据的迁移，采用重分布，即对表的记录通过增加一个隐藏列bucket\_id，进行更细粒度的数据划分，最后通过篮子号对数据重新分布。
3. 通过导入导出工具对表数据文件进行拆分，并且通过多线程分别执行，然后再合并文件。

## Google Spanner

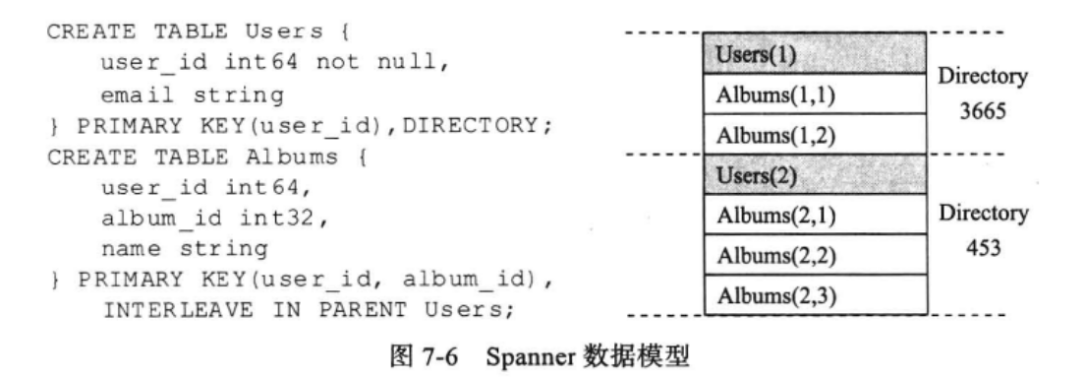
Google Spanner是Google的全球级分布式数据库（Globally-Distributed Database）。Spanner的扩展性达到了全球级，可以扩展到数百个数据中心，数百万台机器，上万亿行记录。更为重要的是，除了夸张的可扩展性之外，它还能通过同步复制和多版本控制来满足外部一致性，支持跨数据中心事务。

无论从学术研究还是工程实践的角度看，Spanner都是一个划时代的分布式存储系统。Spanner的成功说明了这一点，分布式技术能够和数据库技术有机地结合起来，通过分布式技术实现高可扩展性，并呈现给使用者类似关系数据库的数据模型。

### 数据模型

Spanner的数据模型与Megastore系统比较类似。

对于一个典型的相册应用，需要存储其用户和相册，可以用上面的两个SQL来创建表。Spanner的表是层次化的，最底层的表是目录表（Directory table），其他表创建时，可以用INTERVAL IN PARENT来表示层次关系。Spanner中的目录相当于Megastore中的实体组，一个用户的信息（user\_id，email）以及这个用户下的所有相册信息构成一个目录。实际存储时，Spanner会将同一个目录的数据存放到一起，只要目录不太大，同一个目录的每个副本都会分配到同一台机器。因此，针对同一个目录的读写事务大部分情况下都不会涉及跨机操作。



### 架构

Spanner构建在Google下一代分布式文件系统Colossus之上。Colossus是GFS的延续，相比GFS，Colossus的主要改进点在于实时性，并且支持海量小文件。

由于Spanner是全球性的，因此它有两个其他分布式存储系统没有的概念：

1. Universe。一个Spanner部署实例成为一个Universe。目前全世界有3个，一个开发、一个测试、一个线上。Universe支持多数据中心部署，且多个业务可以共用一个Universe。
2. Zone。每个Zone属于一个数据中心（Region），而一个数据中心可以用多个Zone。一般来说，Zone内部的网络通信代价比较低，而Zone与Zone之间通信代价很高。

注：

region：可以简单理解为地理上的分区，比如亚洲地区，或者华北地区，再或者北京等等，没有具体大小的限制。根据项目具体的情况，可以自行合理划分region。

zone：可以简单理解为region内的具体机房，比如说region划分为北京，然后北京有两个机房，就可以在此region之下划分出zone1,zone2两个zone。

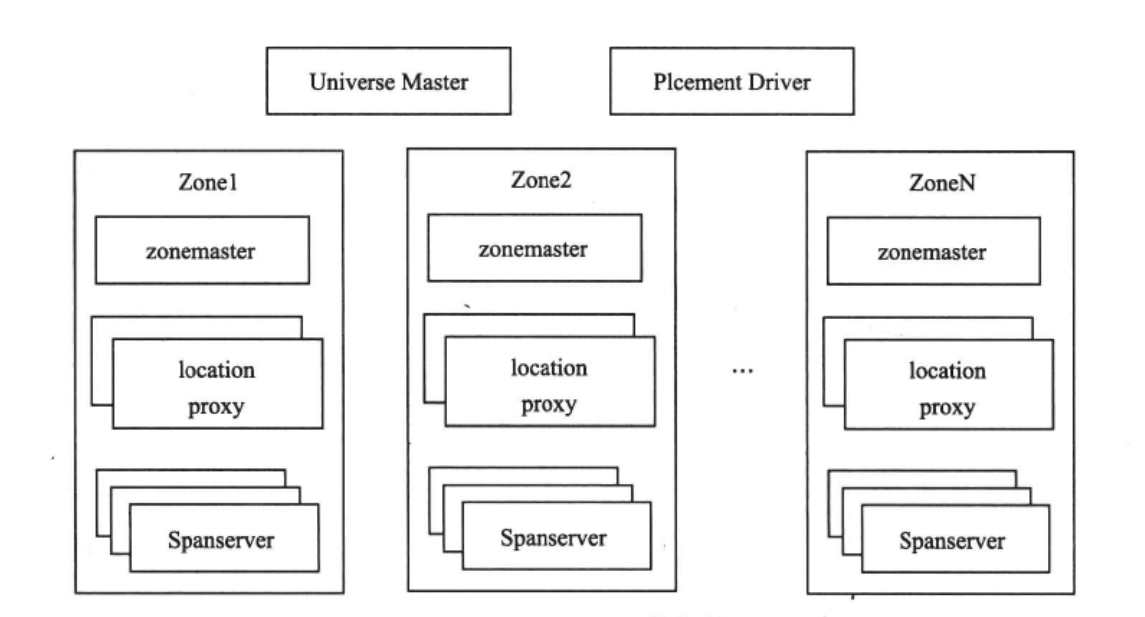
比如，某大型银行，有华北分区和华南分区两个Region，华北分区又有北京机房和河北机房两个Zone，华南分区有武汉机房一个Zone。

Spanner系统包含如下组件：

1. Universe Master：监控这个Universe里Zone级别的状态信息。
2. Placement Driver：提供跨Zone数据迁移功能。
3. Location Proxy：提供获取数据的位置信息服务。客户端需要通过它才能够知道数据由哪台Spanserver服务。
4. Spaserver，提供存储服务，功能上相当于Bigtable系统中的Table Server。

每个Spanserver会服务多个子表，而每个子表又包含多个目录。客户端Spanner发送读写请求时，首先查找目录所在的Spanserver，接着从Spanserver读写数据。

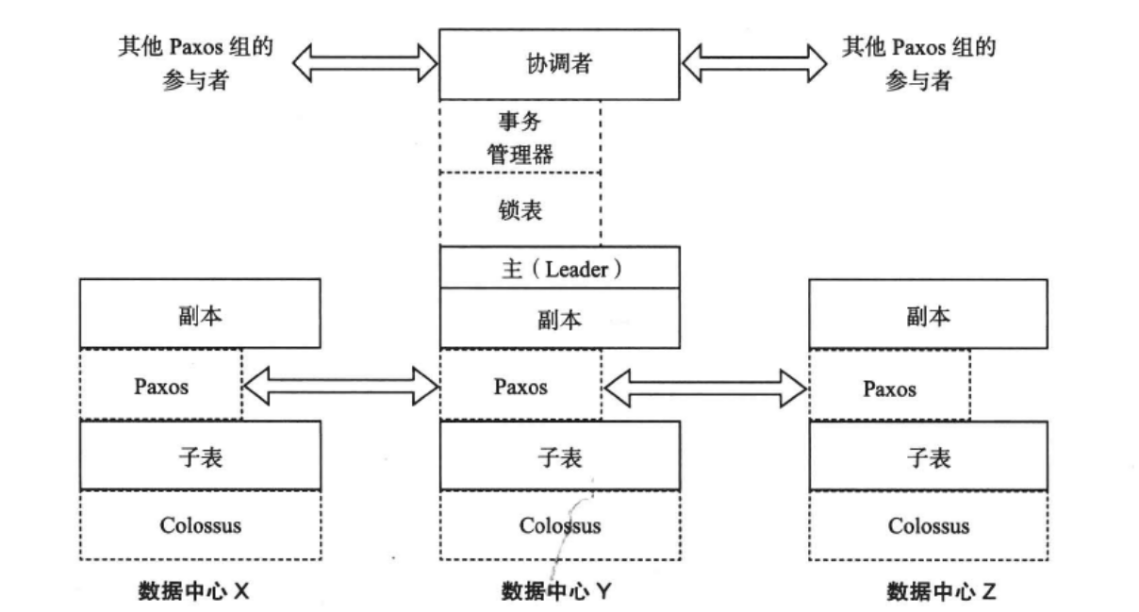
Spanner整体架构：



这里有个问题：如何存储目录与Spanserver之间的映射关系？假设每个用户对应一个目录，全球总共有50亿用户，那么，映射关系的数据规模为几十亿到几百亿，单台服务器无法存放。Spanner沦为中没有明确说明，猜测这里的做法和Bigtable类似，即将映射关系这样的元数据信息当成元数据表格，和普通用户表格采取相同的存储方式。

### 复制与一致性

Spanner多集群复制：



如图所示，每个数据中心运行着一套Colossus，每个机器有100~1000个子表，每个子表会在多个数据中心部署多个副本。为了同步系统中的操作日志，每个子表上会运行一个paxos状态机。Paxos协议会选出一个副本作为主副本，这个主副本的寿命默认是10秒。正常情况下，这个主副本会在快要到期的时候将自己再次选为主副本；如果出现异常，例如主副本所在的Spanserver宕机，其他副本会在10秒后通过paxos协议选举为新的主副本。

通过Paxos协议，实现了跨数据中心的多个副本之间的一致性。另外，每个主副本所在的Spanserver还会实现一个锁表用于并发控制，读写事务操作某个子表上的目录时需要通过锁表避免多个事务之间互相干扰。

除了锁表，每个主副本上还有一个事务管理器。如果事务在一个Paxos组里面，可以绕过事务管理器。但是一旦事务跨多个Paxos组，就需要事务管理器来协调。

注：这里的锁表并不是全部表数据上锁，而是这个目录（或者说分区）上的数据上锁。

锁表实现每个Paxos组内的单机事务，事务管理器实现跨多个Paxos组的分布式事务。为了实现分布式事务，需要实现两阶段提交协议。有一个Paxos组的主副本会成为两阶段提交协议中的协调者，其他Paxos组的主副本作为参与者。

### TrueTime

为了实现并发控制，数据库需要给每个事物分配全局唯一的事务id。然而，在分布式系统中，很难生成全局唯一id。一种方式是采用Google Percolator（Google Caffeine的底层存储系统）中的做法，即专门部署一套Oracle数据库用于生成全局唯一id。虽然Oracle逻辑上是一个单点，但是实现的功能单一，因而能够做得很高效。Spanner选择了另外一种做法，即全球时钟同步机制TrueTime。

TrueTime是一个提供本地时间的接口，但是与Linux的gettimeofday接口不一样的是，它除了可以返回一个时间戳t，还会给出一个误差e。例如，返回的时间戳是20点23分30秒100毫秒，而误差是5毫秒，那么真实的时间在20点23分30秒95毫秒到105毫秒之间。真实的系统e平均下来只有4毫秒。

TrueTime API实现的基础是GPS和原子钟。之所以要用两种技术来处理，是因为导致这两种技术失效的原因是不同的。GPS会有一个天线，电波干扰会导致其失灵。原子钟很稳定。当GPS失灵的时候，原子钟任然能保证在相当长的时间内，不会出现偏差。

每个数据中心需要部署一些主时钟服务器（Master），其他机器上部署一个从时钟进程（Slave）来从主时钟服务器同步时钟信息。有的主时钟服务器采用GPS，有的主时钟服务器采用原子钟。每个从时钟进程每隔30秒会从若干个主时钟服务器同步时钟信息。主时钟服务器自己还会将最新的实践信息和本地时钟对比，排除掉偏差较大的结果。

### 并发控制

Spanner采用TrueTime来控制并发，实现外部一致性，支持以下几种事务：

1. 读写事务
2. 只读事务
3. 快照读，客户端提供时间戳
4. 快照读，客户端提供实践范围
5. 不考虑TrueTime

首先，不考虑TrueTime的影响，也就是说，假设TrueTime API获得的时间是精确的。如果事务读写的数据只属于同一个Paxo组，那么，每个读写事务的执行步骤如下：

1. 获取系统的当前时间戳；
2. 执行读写事务，并将第1步取得的时间戳作为事务的提交版本。

每个只读事务的执行步骤如下：

1. 获取系统的当前时间戳，作为读事务的版本；
2. 执行读取操作，返回客户端所有提交版本小于读事务版本的事务操作结果。

快照读和只读事务的区别在于：快照读将指定事务的版本，而不是取系统的当前时间戳。

如果事务读写的数据涉及多个Paxos组，那么对于读写事务，需要执行一次两阶段提交协议，执行步骤如下：

1. prepare：客户端将数据发往多个Paxos组的主副本，同时，协调者主副本发起prepare协议，请求其他的参与者主副本锁住需要操作的数据。
2. Commit：协调者主副本发起commit协议，要求每个参与者主副本执行提交操作并解除prepare阶段锁定的数据。协调者主副本可以将它的当前时间戳作为该事务的提交版本，并发送给每个参与者主副本。

只读事务读取每个Paxos组中提交版本小于读事务版本的事务操作结果。需要注意的是，只读事务需要保证不会读到不完整的事务。假设有一个读写事务修改了两个Paxos组：Paxos组A和Paxos组B，Paxos组A上的修改已经提交，Paxos组B上的修改还未提交。那么，只读事务会发现Paxos组B处于两阶段提交协议中的prepare阶段，需要等待一会，知道paxos组B上的修改生效后才能读到正确的数据。

1. 考虑TrueTime

如果考虑TrueTime，并发控制就会变得复杂。这里的核心思想在于，只要事务T1的提交操作早于事务T2的开始操作，即使考虑TrueTime API的误差因素（-e~+e之间，e值平均为4ms），Spanner也能保证事务T1的提交版本小于事务T2的提交版本。Spanner使用了一种称为延迟提交（Commit Wait）的手段，即如果事务T1的提交版本为时间戳tcommit，那么，事务T1会在tcommitt+e之后才能提交。另外，如果事务T2开始的绝对时间为tabs，那么事务T2的提交版本至少为2e。从这一点也可以看出，Spanner实现功能完备的全球数据库是付出一定代价的，设计架构时不能盲目崇拜。

### 数据迁移

### 讨论

## Microsoft SQL Azure

## OceanBase

### 背景

### 设计思路

OceanBase的目标是支持数百TB的数据量以及数十万TPS、数百万QPS的访问量，无论是数据量还是访问量，即使采用非常昂贵的小型机甚至是大型机，单台关系数据库系统都无法承受。

一种常见的做法是根据业务特点对数据库进行水平拆分，通常的作答是根据某个业务字段（通常取用户编号user\_id）哈希后取模，根据取模的结果将数据分布到不同的数据库服务器上，客户端请求通过数据库中间层路由到不同的分区。这种方式目前还存在一定的弊端，如下所示：

1. 数据和负载增加后增加机器的操作比较复杂，往往需要人工介入；
2. 有些范围查找需要访问不同的分区，例如，按照user\_id分区，查询收藏了一个商品的所有用户需要访问所有分区；
3. 目前广泛使用的关系数据库存储引擎都是针对机械硬盘的特点设计的，不能够完全发挥新硬件（SSD）的能力。

另外一种做法是参考分布式表格系统的做法，例如Google Bigtable系统，将大表划分为几万、几十万甚至几百万个子表，子表之间按照主键有序，如果某台服务器发生故障，它上面服务的数据能够在很短的时间内自动迁移到集群中所有的其他服务器。这种方式解决了可扩展性问题，少量突发的服务器故障或者增加服务器对使用者基本是透明的，能够轻松应对促销或者热点事件等突发流量增长。另外，由于子表是按照主键有序分布的，很好地解决了范围查询的问题。

万事有利必有一弊，分布式表格系统虽然解决了可扩展性的问题，但是往往是无法支持事务的，例如Bigtable只支持单行事务，针对同一个user\_id下的多条记录的操作都无法保证原子性。而OceanBase希望能够支持跨行跨表事务，这样使用起来会比较方面。

最直接的做法就是在Bigtable开源实现（如HBASE或Hypertable）的基础上引入两阶段提交（Two-phase Commit）协议支持分布式事务，这种思路在Google的Percolator系统中得到了体现。然而，Percolator系统中事务的平均响应时间达到2~5秒，只能应用在类似网页建库这样的半线上业务中。另外，Bigtable的开源实现也不够成熟，单条服务器能够支持的数据量有限，单个请求的最大响应时间很难得到保证，机器故障等异常处理机制也有狠多比较严重的问题。总体来说，这种做法的工作量和难度超过了项目组的承受能力，因此，我们需要根据业务特点做出一些定制。

### 系统架构

### 架构剖析

### 分布式存储引擎

### 数据功能