# 背景

Spanner是一个可扩展的、全球分布式的数据库，是在谷歌公司设计、开发和部署的。在最高抽象层面，Spanner就是一个数据库，把数据分片存储在许多Paxos状态机上，这些机器位于遍布全球的数据中心内。复制技术可以用来服务于全球可用性和地理局部性。客户端会自动在副本之间进行失败恢复。随着数据的变化和服务器的变化，Spanner会自动把数据进行重新分片，从而有效应对负载变化和处理失败。Spanner被设计成可以扩展到几百万个机器节点，跨越成百上千个数据中心，具备几万亿数据库行的规模。

应用可以借助于Spanner来实现高可用性，通过在一个洲的内部和跨越不同的洲之间复制数据，保证即使面对大范围的自然灾害时数据依然可用。我们最初的客户是F1，一个谷歌广告后台的重新编程实现。F1使用了跨越美国的5个副本。绝大多数其他应用很可能会在属于同一个地理范围内的3-5个数据中心内放置数据副本，采用相对独立的失败模式。也就是说，许多应用都会首先选择低延迟，而不是高可用性，只要系统能够从1-2个数据中心失败中恢复过来。

Spanner的主要工作，就是管理跨越多个数据中心的数据副本，但是，在我们的分布式系统体系架构之上设计和实现重要的数据库特性方面，我们也花费了大量的时间。尽管有许多项目可以很好地使用BigTable，我们也不断收到来自客户的抱怨，客户反映BigTable无法应用到一些特定类型的应用上面，比如具备复杂可变的模式，或者对于在大范围内分布的多个副本数据具有较高的一致性要求。其他研究人员也提出了类似的抱怨。谷歌的许多应用已经选择使用Megastore，主要是因为它的半关系数据模型和对同步复制的支持，尽管Megastore具备较差的写操作吞吐量。由于上述多个方面的因素，Spanner已经从一个类似BigTable的单一版本的键值存储，演化成为一个具有时间属性的多版本的数据库。数据被存储到模式化的、半关系的表中，数据被版本化，每个版本都会自动以提交时间作为时间戳，旧版本的数据会更容易被垃圾回收。应用可以读取旧版本的数据。Spanner支持通用的事务，提供了基于SQL的查询语言。

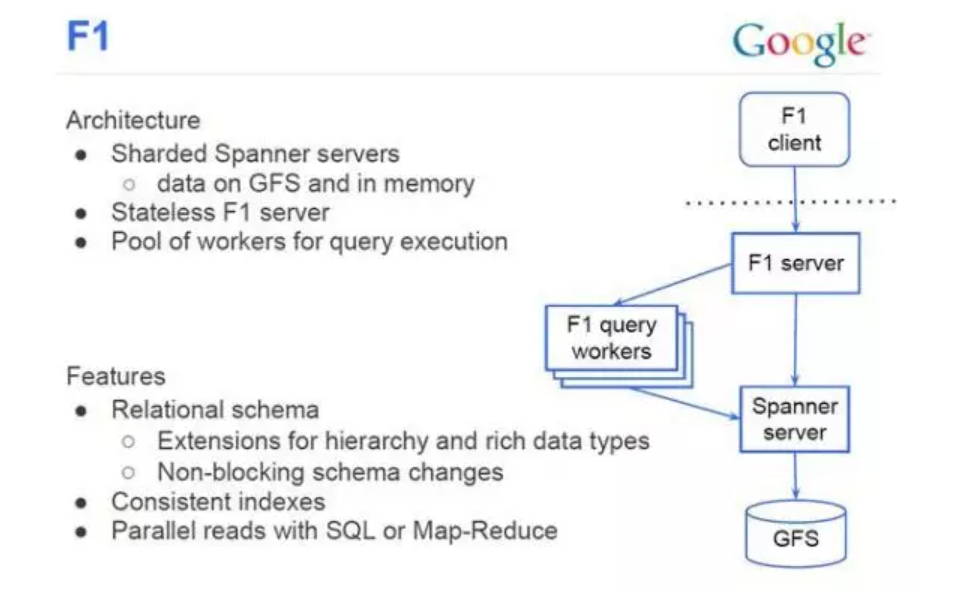
作为一个全球分布式数据库，Spanner提供了几个有趣的特性：

第一，在数据的副本配置方面，应用可以在一个很细的粒度上进行动态控制。应用可以详细规定，哪些数据中心包含哪些数据，数据距离用户有多远（控制用户读取数据的延迟），不同数据副本之间距离有多远（控制写操作的延迟），以及需要维护多少个副本（控制可用性和读操作性能）。数据也可以被动态和透明地在数据中心之间进行移动，从而平衡不同数据中心内资源的使用。

第二，Spanner有两个重要的特性，很难在一个分布式数据库上实现，即Spanner提供了读和写操作的外部一致性，以及在一个时间戳下面的跨越数据库的全球一致性的读操作。这些特性使得Spanner可以支持一致的备份、一致的MapReduce执行和原子模式变更，所有都是在全球范围内实现，即使存在正在处理中的事务也可以。

之所以可以支持这些特性，是因为Spanner可以为事务分配全球范围内有意义的提交时间戳，即使事务可能是分布式的。这些时间戳反映了事务序列化的顺序。除此以外，这些序列化的顺序满足了外部一致性的要求：如果一个事务T1在另一个事务T2开始之前就已经提交了，那么，T1的时间戳就要比T2的时间戳小。Spanner是第一个可以在全球范围内提供这种保证的系统。

实现这种特性的关键技术就是一个新的TrueTime API及其实现。这个API可以直接暴露时钟不确定性，Spanner时间戳的保证就是取决于这个API实现的界限。如果这个不确定性很大，Spanner就降低速度来等待这个大的不确定性结束。谷歌的簇管理器软件提供了一个TrueTime API的实现。这种实现可以保持较小的不确定性（通常小于10ms），主要是借助于现代时钟参考值（比如GPS和原子钟）。



从上图可以看到。Spanner位于F1和GFS之间，承上启下。所以先提一提F1和GFS。

## F1

和众多互联网公司一样，在早期Google大量使用了Mysql。Mysql是单机的，可以用Master-Slave来容错，分区来扩展。但是需要大量的手工运维工作，有很多的限制。因此Google开发了一个可容错可扩展的RDBMS—F1。和一般的分布式数据库不同，F1对应RDMS应有的功能，毫不妥协。起初F1是基于Mysql的，不过会逐渐迁移到Spanner。

F1有如下特点：

7×24高可用。哪怕某一个数据中心停止运转，仍然可用。

可以同时提供强一致性和弱一致。

可扩展

支持SQL

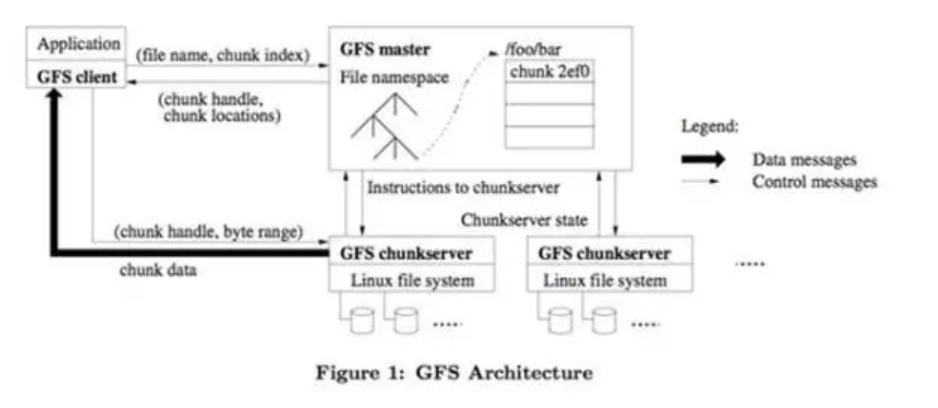
事务提交延迟50-100ms，读延迟5-10ms，高吞吐

众所周知Google BigTable是重要的NoSql产品，提供很好的扩展性，开源世界有HBase与之对应。为什么Google还需要F1，而不是都使用BigTable呢？因为**BigTable提供的最终一致性，一些需要事务级别的应用无法使用**。同时**BigTable还是NoSql，而大量的应用场景需要有关系模型**。就像现在大量的互联网企业都使用Mysql而不愿意使用HBase，因此Google才有这个可扩展数据库的F1。而Spanner就是F1的至关重要的底层存储技术。

初代GFS是为批处理设计的。对于大文件很友好，吞吐量很大，但是延迟较高。所以使用他的系统不得不对GFS做各种优化，才能获得良好的性能。那为什么Google没有考虑到这些问题，设计出更完美的GFS ?因为那个时候是2001年，Hadoop出生是在2007年。如果Hadoop是世界领先水平的话，GFS比世界领先水平还领先了6年。同样的Spanner出生大概是2009年，现在我们看到了论文，估计Spanner在Google已经很完善，同时Google内部已经有更先进的替代技术在酝酿了。

## Colossus（GFS II）

Colossus也是一个不得不提起的技术。他是第二代GFS，对应开源世界的新HDFS。GFS是著名的分布式文件系统。



Colossus是Google重要的基础设施，因为他可以满足主流应用对FS的要求。 **Colossus的重要改进有：**

优雅Master容错处理 (不再有2s的停止服务时间)

Chunk大小只有1MB (对小文件很友好)

Master可以存储更多的Metadata(当Chunk从64MB变为1MB后，Metadata会扩大64倍，但是Google也解决了)

Colossus可以自动分区Metadata。使用Reed-Solomon算法来复制，可以将原先的3份减小到1.5份，提高写的性能，降低延迟。客户端来复制数据。

## BigTable/Megastore

Spanner主要致力于跨数据中心的数据复制上，同时也能提供数据库功能。在Google类似的系统有BigTable和Megastore。和这两者相比，Spanner又有什么优势呢。

BigTable在Google得到了广泛的使用，但是他不能提供较为复杂的Schema，还有在跨数据中心环境下的强一致性。Megastore有类RDBMS的数据模型，同时也支持同步复制，但是他的吞吐量太差，不能适应应用要求。

Spanner不再是类似BigTable的版本化 key-value存储，而是一个“临时多版本”的数据库。何为“临时多版本”，数据是存储在一个版本化的关系表里面，存储的时间数据会根据其提交的时间打上时间戳，应用可以访问到较老的版本，另外老的版本也会被垃圾回收掉。

Google官方认为Spanner是下一代BigTable，也是Megastore的继任者。

# 概述

## 简介

Spanner是Google的全球级的分布式数据库(Globally-Distributed Database) 。Spanner的扩展性达到了令人咋舌的全球级，可以扩展到数百万的机器，数已百计的数据中心，上万亿的行。更给力的是，除了夸张的扩展性之外，他还能同时通过同步复制和多版本来满足外部一致性，可用性也是很好的。冲破CAP的枷锁，在三者之间完美平衡。

Spanner是个可扩展，多版本，全球分布式还支持同步复制的数据库。他是Google的第一个可以全球扩展并且支持外部一致的事务。Spanner能做到这些，离不开一个**用GPS和原子钟实现的时间API**。这个API能将数据中心之间的时间同步精确到10ms以内。因此有几个给力的功能：无锁读事务，原子schema修改，读历史数据无block。

## 功能

从高层看Spanner是通过Paxos状态机将分区好的数据分布在全球的。数据复制全球化的，用户可以指定数据复制的份数和存储的地点。Spanner可以在集群或者数据发生变化的时候将数据迁移到合适的地点，做负载均衡。用户可以指定将数据分布在多个数据中心，不过更多的数据中心将造成更多的延迟。用户需要在可靠性和延迟之间做权衡，一般来说复制1，2个数据中心足以保证可靠性。

作为一个全球化分布式系统，Spanner提供一些有趣的特性。

1、应用可以细粒度的指定数据分布的位置。精确的指定数据离用户有多远，可以有效的控制读延迟(读延迟取决于最近的拷贝)。指定数据拷贝之间有多远，可以控制写的延迟(写延迟取决于最远的拷贝)。还要数据的复制份数，可以控制数据的可靠性和读性能。(多写几份，可以抵御更大的事故)

2、Spanner还有两个一般分布式数据库不具备的特性：读写的外部一致性，基于时间戳的全局的读一致。这两个特性可以让Spanner支持一致的备份，一致的MapReduce，还有原子的Schema修改。

这写特性都得益有Spanner有一个全球时间同步机制，可以在数据提交的时候给出一个时间戳。因为时间是系列化的，所以才有外部一致性。这个很容易理解，如果有两个提交，一个在T1,一个在T2。那有更晚的时间戳那个提交是正确的。

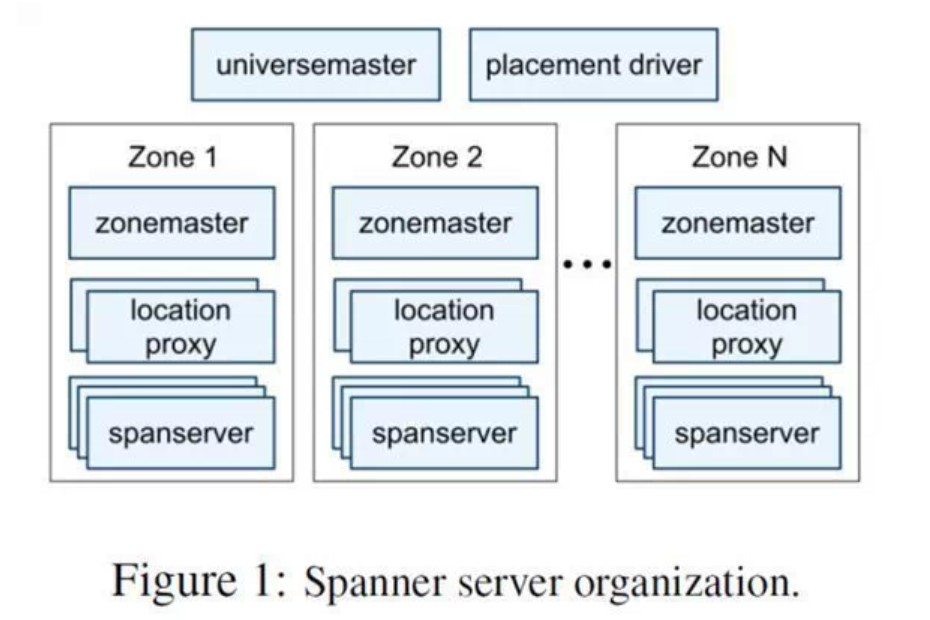
这个全球时间同步机制是用一个具有GPS和原子钟的TrueTime API提供了。这个TrueTime API能够将不同数据中心的时间偏差缩短在10ms内。这个API可以提供一个精确的时间，同时给出误差范围。Google已经有了一个TrueTime API的实现。笔者觉得这个TrueTimeAPI 非常有意义，如果能单独开源这部分的话，很多数据库如MongoDB都可以从中受益。

# 架构

Spanner由于是全球化的，所以有两个其他分布式数据库没有的概念。

**Universe。**一个Spanner部署实例称之为一个Universe。目前全世界有3个。一个开发，一个测试，一个线上。因为一个Universe就能覆盖全球，不需要多个。

**Zones。**每个Zone相当于一个数据中心，一个Zone内部物理上必须在一起。而一个数据中心可能有多个Zone。可以在运行时添加移除Zone。一个Zone可以理解为一个BigTable部署实例。



如图所示。一个Spanner有上面一些组件。实际的组件肯定不止这些，比如TrueTime API Server。如果仅仅知道这些知识，来构建Spanner是远远不够的。但Google都略去了。

**Universemaster:** 监控这个universe里zone级别的状态信息

**Placement driver**：提供跨区数据迁移时管理功能

**Zonemaster**：相当于BigTable的Master。管理Spanserver上的数据。

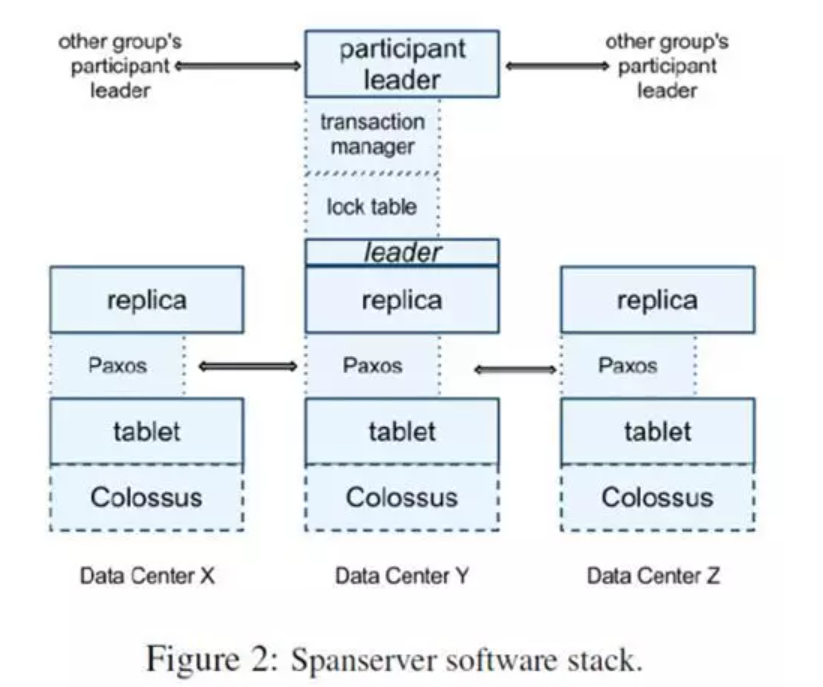
**Location proxy**：存储数据的Location信息。客户端要先访问他才知道数据在那个Spanserver上。

**Spanserver**：相当于BigTable的ThunkServer。用于存储数据。

可以看出来这里每个组件都很有料，但是Google的论文里只具体介绍了Spanserver的设计。

## Spanserver

Spanserver的设计和BigTable非常的相似。参照下图:



从下往上看。每个数据中心会运行一套Colossus (GFS II) 。每个机器有100-1000个tablet。Tablet概念上将相当于数据库一张表里的一些行，物理上是数据文件。打个比方，一张1000行的表，有10个tablet，第1-100行是一个tablet，第101-200是一个tablet。但和BigTable不同的是BigTable里面的tablet存储的是Key-Value都是string，Spanner存储的Key多了一个时间戳：

(Key: string, timestamp: int64) ->string。

因此spanner天生就支持多版本，tablet在文件系统中是一个B-tree-like的文件和一个write-ahead日志。

每个Tablet上会有一个Paxos状态机。Paxos是一个分布式一致性协议。Table的元数据和log都存储在上面。Paxos会选出一个replica做leader，这个leader的寿命默认是10s,10s后重选。Leader就相当于复制数据的master，其他replica的数据都是从他那里复制的。读请求可以走任意的replica，但是写请求只有去leader。这些replica统称为一个paxos group。

每个leader replica的spanserver上会实现一个lock table还管理并发。Lock table记录了两阶段提交需要的锁信息。但是不论是在Spanner还是在BigTable上，但遇到冲突的时候长时间事务会将性能很差。所以有一些操作，如事务读可以走lock table，其他的操作可以绕开lock table。

每个leader replica的spanserver上还有一个transaction manager。如果事务在一个paxos group里面，可以绕过transaction manager。但是一旦事务跨多个paxos group，就需要transaction manager来协调。其中一个Transactionmanager被选为leader，其他的是slave听他指挥。这样可以保证事务。

## Directories and Placement

之所以Spanner比BigTable有更强的扩展性，在于Spanner还有一层抽象的概念directory, directory是一些key-value的集合，一个directory里面的key有一样的前缀。更妥当的叫法是bucketing。Directory是应用控制数据位置的最小单元，可以通过谨慎的选择Key的前缀来控制。据此笔者可以猜出，在设计初期，Spanner是作为F1的存储系统而设立，甚至还设计有类似directory的层次结构，这样的层次有很多好处，但是实现太复杂被摒弃了。

Directory作为数据放置的最小单元，可以在paxos group里面移来移去。Spanner移动一个directory一般出于如下几个原因：

· 一个paxos group的负载太大，需要切分

· 将数据移动到access更近的地方

· 将经常同时访问的directory放到一个paxos group里面

Directory可以在不影响client的前提下，在后台移动。移动一个50MB的directory大概需要的几秒钟。

那么directory和tablet又是什么关系呢。可以理解为Directory是一个抽象的概念，管理数据的单元；而tablet是物理的东西，数据文件。由于一个Paxos group可能会有多个directory，所以spanner的tablet实现和BigTable的tablet实现有些不同。BigTable的tablet是单个顺序文件。Google有个项目，名为Level DB，是BigTable的底层，可以看到其实现细节。而Spanner的tablet可以理解是一些基于行的分区的容器。这样就可以将一些经常同时访问的directory放在一个tablet里面，而不用太在意顺序关系。

在paxos group之间移动directory是后台任务。这个操作还被用来移动replicas。移动操作设计的时候不是事务的，因为这样会造成大量的读写block。操作的时候是先将实际数据移动到指定位置，然后再用一个原子的操作更新元数据，完成整个移动过程。

Directory还是记录地理位置的最小单元。数据的地理位置是由应用决定的，配置的时候需要指定复制数目和类型，还有地理的位置。比如(上海，复制2份；南京复制1分) 。这样应用就可以根据用户指定终端用户实际情况决定的数据存储位置。比如中国队的数据在亚洲有3份拷贝, 日本队的数据全球都有拷贝。

前面对directory还是被简化过的，还有很多无法详述。

# 数据模型

Spanner的数据模型来自于Google内部的实践。在设计之初，Spanner就决心有以下的特性：

支持类似关系数据库的schema

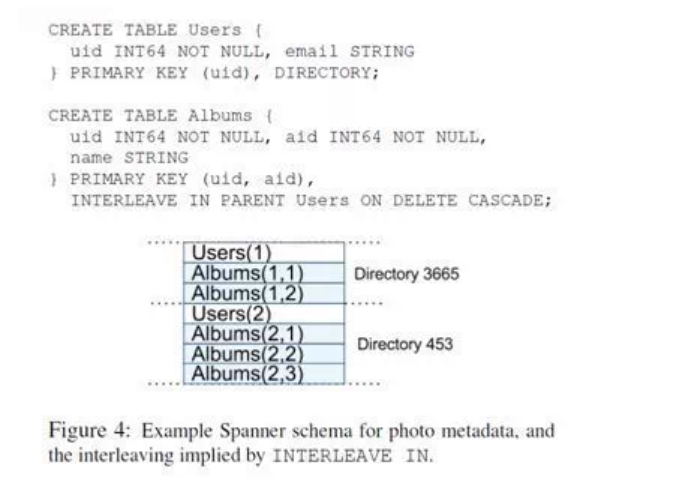
Query语句

支持广义上的事务

为何会这样决定呢？在Google内部还有一个Megastore，尽管要忍受性能不够的折磨，但是在Google有300多个应用在用它，因为Megastore支持一个类似关系数据库的schema，而且支持同步复制（BigTable只支持最终一致的复制）。使用Megastore的应用有大名鼎鼎的Gmail, Picasa, Calendar, Android Market和AppEngine。而必须对Query语句的支持，来自于广受欢迎的Dremel，笔者不久前写了篇文章来介绍他。 最后对事务的支持是比不可少了，BigTable在Google内部被抱怨的最多的就是其只能支持行事务，再大粒度的事务就无能为力了。Spanner的开发者认为，过度使用事务造成的性能下降的恶果，应该由应用的开发者承担。应用开发者在使用事务的时候，必须考虑到性能问题。而数据库必须提供事务机制，而不是因为性能问题，就干脆不提供事务支持。

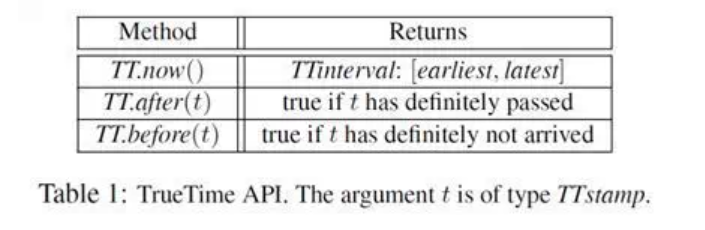
数据模型是建立在directory和key-value模型的抽象之上的。一个应用可以在一个universe中建立一个或多个database，在每个database中建立任意的table。Table看起来就像关系型数据库的表。有行，有列，还有版本。Query语句看起来是多了一些扩展的SQL语句。

Spanner的数据模型也不是纯正的关系模型，每一行都必须有一列或多列组件。看起来还是Key-value。主键组成Key,其他的列是Value。但这样的设计对应用也是很有裨益的，应用可以通过主键来定位到某一行。



上图是一个例子。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，最顶层的表是directory table。其他的表创建的时候，可以用interleave in parent来什么层次关系。这样的结构，在实现的时候，Spanner可以将嵌套的数据放在一起，这样在分区的时候性能会提升很多。否则Spanner无法获知最重要的表之间的关系。

# TrueTime



TrueTime API 是一个非常有创意的东西，可以同步全球的时间。上表就是TrueTime API。TT.now()可以获得一个绝对时间TTinterval，这个值和UnixTime是相同的，同时还能够得到一个误差e。TT.after(t)和TT.before(t)是基于TT.now()实现的。

那这个TrueTime API实现靠的是GFS和原子钟。之所以要用两种技术来处理，是因为导致这两个技术的失败的原因是不同的。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，原子钟仍然能保证在相当长的时间内，不会出现偏差。

实际部署的时候。每个数据中心需要部署一些Master机器，其他机器上需要有一个slave进程来从Master同步。有的Master用GPS，有的Master用原子钟。这些Master物理上分布的比较远，怕出现物理上的干扰。比如如果放在一个机架上，机架被人碰倒了，就全宕了。另外原子钟不是并很贵。Master自己还会不断比对，新的时间信息还会和Master自身时钟的比对，会排除掉偏差比较大的，并获得一个保守的结果。最终GPS master提供时间精确度很高，误差接近于0。

每个Slave后台进程会每个30秒从若干个Master更新自己的时钟。为了降低误差，使用Marzullo算法。每个slave还会计算出自己的误差。这里的误差包括的通信的延迟，机器的负载。如果不能访问Master，误差就会越走越大，知道重新可以访问。

# 并发控制

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

读写事务

只读事务

快照读，客户端提供时间戳

快照读，客户端提供时间范围

例如一个读写事务发生在时间t，那么在全世界任何一个地方，指定t快照读都可以读到写入的值。



上表是Spanner现在支持的事务。单独的写操作都被实现为读写事务 ； 单独的非快照被实现为只读事务。事务总有失败的时候，如果失败，对于这两种操作会自己重试，无需应用自己实现重试循环。

时间戳的设计大大提高了只读事务的性能。事务开始的时候，要声明这个事务里没有写操作，只读事务可不是一个简单的没有写操作的读写事务。它会用一个系统时间戳去读，所以对于同时的其他的写操作是没有Block的。而且只读事务可以在任意一台已经更新过的replica上面读。

对于快照读操作，可以读取以前的数据，需要客户端指定一个时间戳或者一个时间范围。Spanner会找到一个已经充分更新好的replica上读取。

还有一个有趣的特性的是，对于只读事务，如果执行到一半，该replica出现了错误。客户端没有必要在本地缓存刚刚读过的时间，因为是根据时间戳读取的。只要再用刚刚的时间戳读取，就可以获得一样的结果。

## 读写事务

正如BigTable一样，Spanner的事务是会将所有的写操作先缓存起来，在Commit的时候一次提交。这样的话，就读不出在同一个事务中写的数据了。不过这没有关系，因为Spanner的数据都是有版本的。

在读写事务中使用wound-wait算法来避免死锁。当客户端发起一个读写事务的时候，首先是读操作，他先找到相关数据的leader replica，然后加上读锁，读取最近的数据。在客户端事务存活的时候会不断的向leader发心跳，防止超时。当客户端完成了所有的读操作，并且缓存了所有的写操作，就开始了两阶段提交。客户端闲置一个coordinator group，并给每一个leader发送coordinator的id和缓存的写数据。

leader首先会上一个写锁，他要找一个比现有事务晚的时间戳。通过Paxos记录。每一个相关的都要给coordinator发送他自己准备的那个时间戳。

Coordinatorleader一开始也会上个写锁，当大家发送时间戳给他之后，他就选择一个提交时间戳。这个提交的时间戳，必须比刚刚的所有时间戳晚，而且还要比TT.now()+误差时间 还有晚。这个Coordinator将这个信息记录到Paxos。

在让replica写入数据生效之前，coordinator还有再等一会。需要等两倍时间误差。这段时间也刚好让Paxos来同步。因为等待之后，在任意机器上发起的下一个事务的开始时间，都比如不会比这个事务的结束时间早了。然后coordinator将提交时间戳发送给客户端还有其他的replica。他们记录日志，写入生效，释放锁。

## 只读事务

对于只读事务，Spanner首先要指定一个读事务时间戳。还需要了解在这个读操作中，需要访问的所有的读的Key。Spanner可以自动确定Key的范围。

如果Key的范围在一个Paxos group内。客户端可以发起一个只读请求给group leader。leader选一个时间戳，这个时间戳要比上一个事务的结束时间要大。然后读取相应的数据。这个事务可以满足外部一致性，读出的结果是最后一次写的结果，并且不会有不一致的数据。

如果Key的范围在多个Paxos group内，就相对复杂一些。其中一个比较复杂的例子是，可以遍历所有的group leaders，寻找最近的事务发生的时间，并读取。客户端只要时间戳在TT.now().latest之后就可以满足要求了。