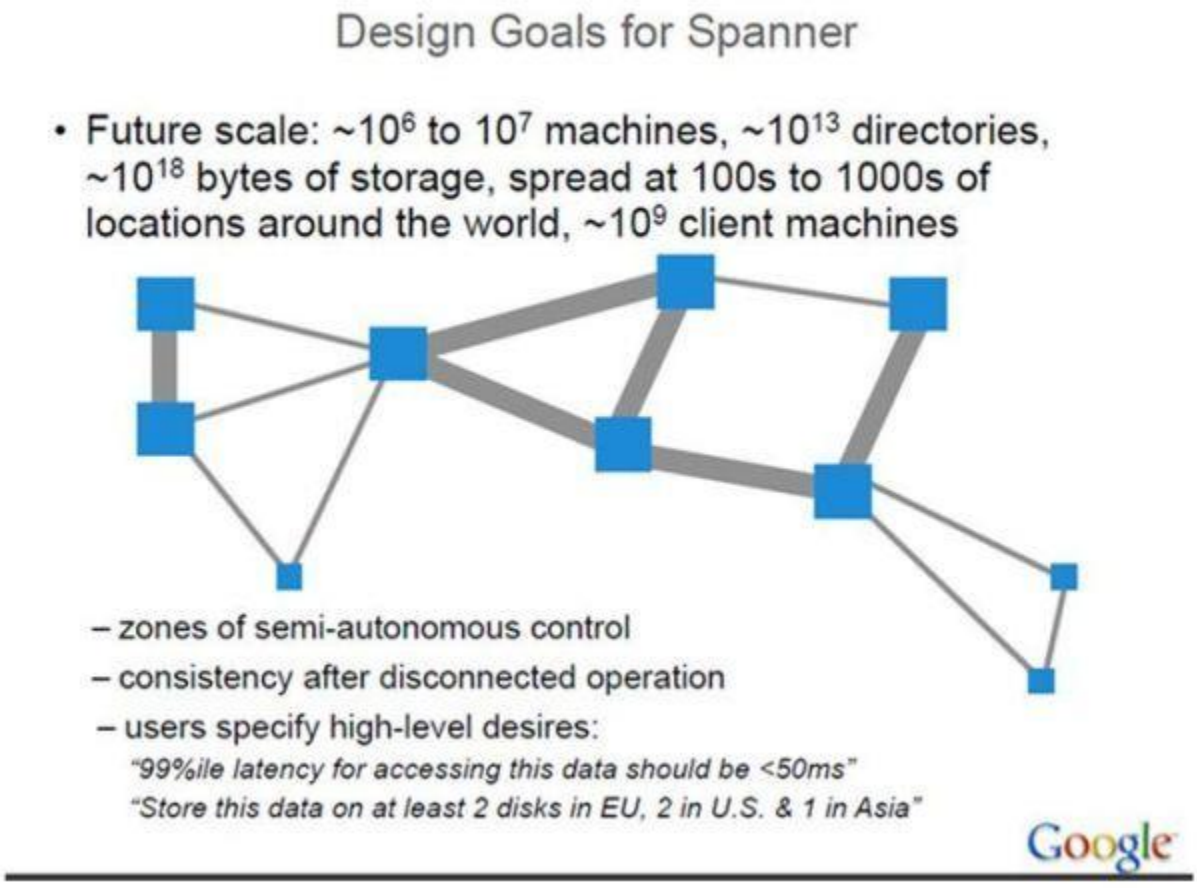
Google Spanner 原理- 全球级的分布式数据库

标签:大数据云计算

Google Spanner 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。

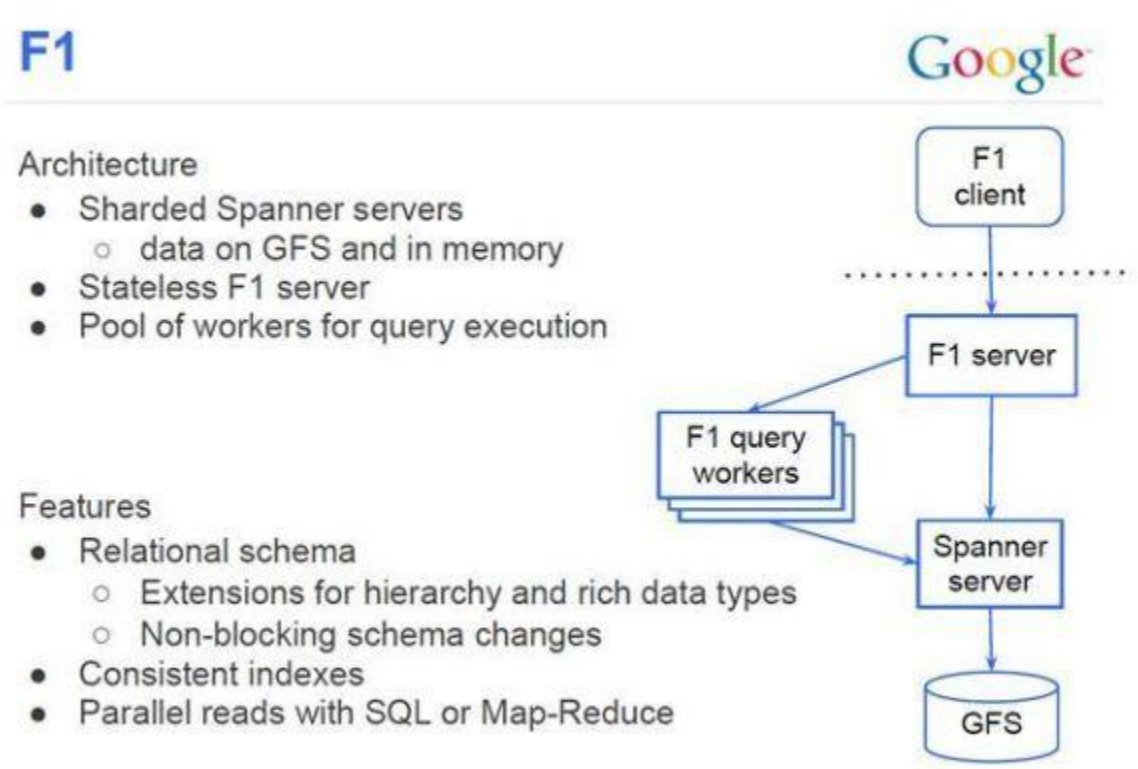
EMC 中国研究院实时紧盯业界动态，Google 最近发布的一篇论文《Spanner: Google's Globally-Distributed Database》, 笔者非常感兴趣，对 Spanner 进行了一些调研，并在这里

分享。由于 Spanner 并不是开源产品，笔者的知识主要来源于 Google 的公开资料，通过现有公开资料仅仅只能窥得 Spanner 的沧海一粟，Spanner 背后还依赖有大量 Google 的专有技术。

下文主要是 Spanner 的背景，设计和并发控制。

Spanner 背景

要搞清楚 Spanner 原理，先得了解 Spanner 在 Google 的定位。



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。

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。

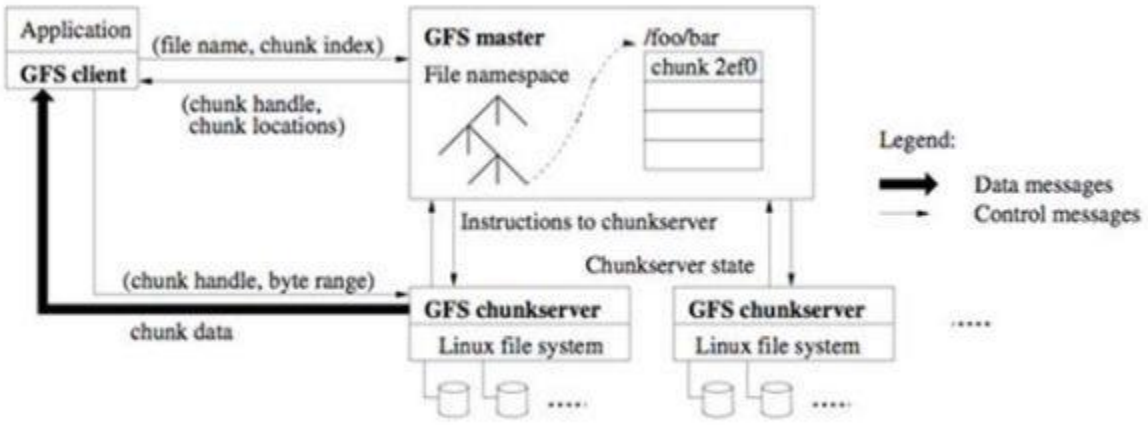


Figure 1: GFS Architecture

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。笔者预测，最早在 2015 年才会出现 Spanner 和 F1 的山寨开源产品。

Colossus 是第二代 GFS。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。具体细节笔者也猜不出。

与 BigTable， Megastore 对比

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。

Google Spanner 设计

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。

· 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)

· 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。

体系结构

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。

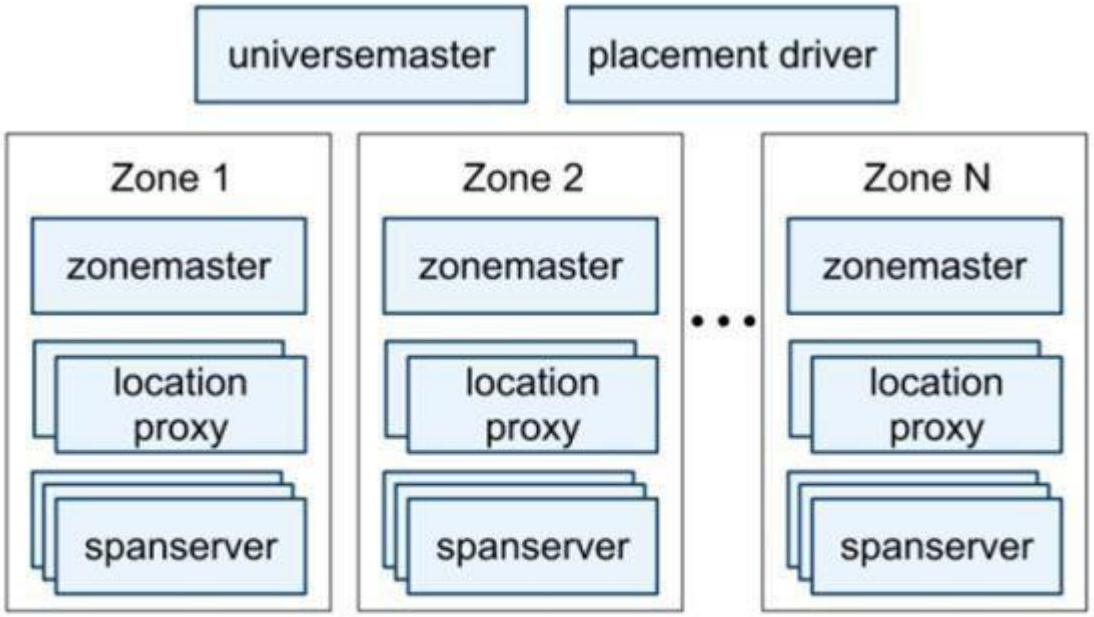


Figure 1: Spanner server organization.

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。那笔者就简要介绍一下。

· 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，笔者也只能介绍到这里。下面详细阐述 Spanserver 的设计。

Spanserver

本章详细介绍 Spanserver 的设计实现。Spanserver 的设计和 BigTable 非常的相似。参照下图

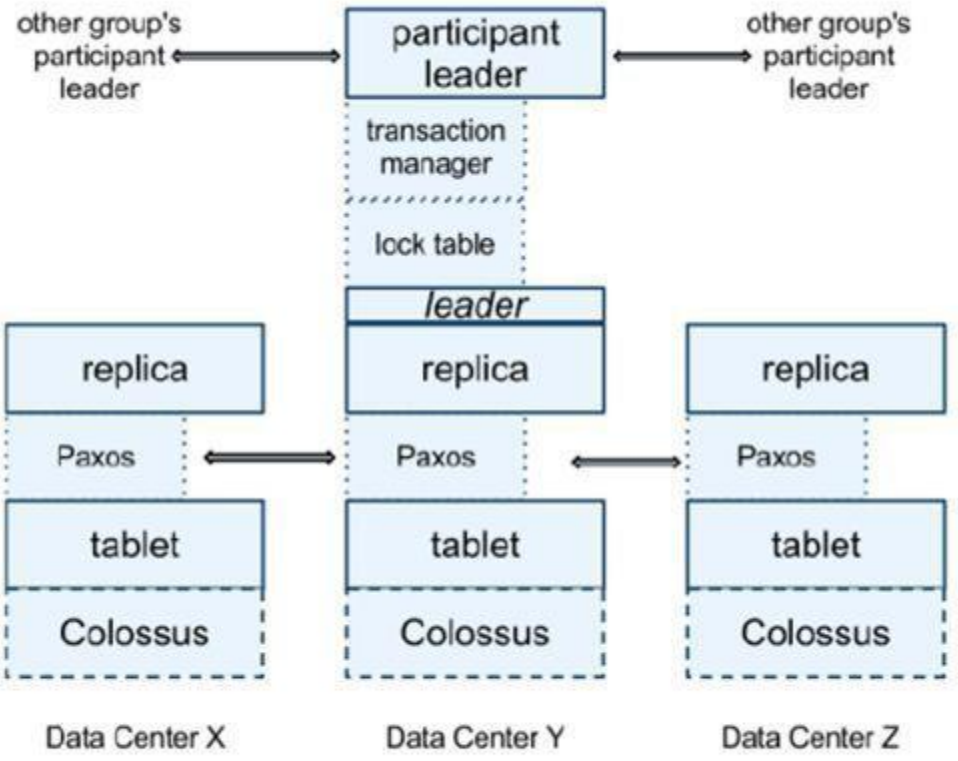


Figure 2: Spanserver software stack.

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。

```
CREATE TABLE Users {
  uid INT64 NOT NULL, email STRING
} PRIMARY KEY (uid), DIRECTORY;

CREATE TABLE Albums {
  uid INT64 NOT NULL, aid INT64 NOT NULL,
  name STRING
} PRIMARY KEY (uid, aid),
  INTERLEAVE IN PARENT Users ON DELETE CASCADE;
```

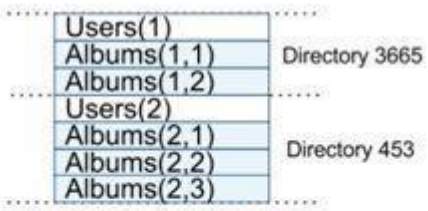


Figure 4: Example Spanner schema for photo metadata, and the interleaving implied by `INTERLEAVE IN`.

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

Method	Returns
TT.now()	*TTinterval*: [*earliest*, *latest*]
TT.after(*t*)	true if *t* has definitely passed
TT.before(*t*)	true if *t* has definitely not arrived

Table 1: TrueTime API. The argument *t* is of type *TTstamp*.

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。

Google Spanner 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。

Operation	Concurrency Control	Replica Required
Read-Write Transaction	pessimistic	leader
Read-Only Transaction	lock-free	leader for timestamp; any for read
Snapshot Read, client-provided timestamp	lock-free	any
Snapshot Read, client-provided bound	lock-free	any

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。

最后的话

本文介绍了 GoogleSpanner 的背景，设计和并发控制。希望不久的将来，会有开源产品出现。

关于作者

颜开，EMC 中国研究院研究员，关注大数据，云计算等领域

博客: http://weibo.com/yankaycom