Spanner 开启了 NewSQL 时代的序幕。

其主要有三点特色：

- 2PC + 共识组来避免 2PC 的无限超时阻塞
- GPS 原子钟同步技术以支持快速的**只读事务**
- 支持 ACID 的全球型 NewSQL 数据库

有些数据库可能不支持分布式事务，只能支持数据分布在同一台机器上的事务。

Spanner 中只读事务有数十亿，读写型事务占了几百万，所以 Spanner 对只读事务的性能非常感兴趣。Spanner 想要外部一致性，即一个事务提交后，后一个执行的事务需要看到前一个事务执行所作的任何修改。

Spanner 对于数据采用了复制和分片的策略，数据分布在不同的数据中心，每个数据中心的数据都是一样的，每个数据中心内会进行分片，比如对 a 开头的 key 进行分片。并且每个数据中心中还会有很多 Spanner client，这些 client 就是 web 服务器。

然后不同数据中心的同一个分片会组成一个 Paxos 组，一个 Paxos 组里会有一个 leader（其中一个数据中心的一个分片），其负责管理所有的分片。这样就有很多个 Paxos 组，这些 Paxos 实例都是彼此独立的。独立可以让我们对这些数据进行并行处理，以提高并行吞吐量。

因此对于大量的并发请求，其要付出巨大的代价将这些请求分散到数据分片所在的多个 Paxos 组中。

如果一个 client 要进行写操作，那么就需要将该写请求发送给对应分片所在 Paxos 组的 leader，Paxos 实例做的就是发送日志，然后 leader 将这些日志发送给 follower，然后 Paxos 组就会以完全相同的顺序执行这些日志记录的操作。

将数据的多个副本放在多个不同的数据中心有两个原因：

1. 容错。这样一个数据中心发生了故障，其他数据中心可能不会受到影响。但是 Paxos 组的 leader 需要和距离非常远的 leader 进行通信
2. 我们可以让 client 从靠近该 client 的数据中心中读取该数据

此外，Paxos 和 Raft 很像，其只需要过半服务器的支持，这样即便有些服务器很慢，Paxos 也能较快地处理并接收新的请求。

但这也会带来两个问题：

1. Paxos 只需获得过半服务器的支持，这意味着会有部分 replica 中的数据会落后，这样读取到的数据可能是过时的。由于 Spanner 要求外部一致性，即每次读操作都要看到最新的数据，因此需要用某种方式来处理这种情况。
2. 一个事务可能会涉及多个 Paxos 组，因此需要分布式事务来进行处理。

读写型事务

以银行转账为例。首先 client 需要连接到数据中心，其发起了这个事务，首先 client 会选择一个唯一的事务 id 来给它所要发生的所有消息打上标记，然后它会往所属数据分片的 lieder 发送第一个读请求，并对数据加锁，如果该数据已经被加锁了，那么只有持有该锁的事务提交并释放锁后，才能对这个 client 响应，然后该分片的 leader 将 x 的值发给 client。

client 在受到相应的数据后，就会对记录进行更新，然后将更新的值发送给对应分片的 leader。注意 client 会首先选择一个 Paxos 组来作为事务协调器来使用，它会将作为事务协调器的那个 Paxos 组的 id 发出去。然后 client 会发送一个关于 x 的写请求给 x 的 leader，请求中包含修改后的值以及事务协调器的 id。leader 受到后，其会发送一个 prepare 消息给 Paxos 组中的其他 follower，并将它写入 Paxos 的日志中，当他受到过半的回复后，这个 Paxos 组的 leader 就会发送一个 Yes 给事务协调器，说明可以执行这部分任务。

当事务协调器收到来自所有的 Yes 时，事务协调器就会去提交这个事务，然后向各组的 leader 发送 commit 消息，说你们可以将这个事务提交了。注意 Paxos 组的 leader 在收到 commit 消息后，同样要通过组内的 follower 达成共识，一旦被提交到了不同分片中的日志后，每个分片就可以去执行写操作，将这些数据写入，并释放这些数据的锁。

如果有两个事务冲突了，比如请求数据时，该数据正被别的事务占用，那么它只能等待别的事务释放锁。Spanner 通过对事务协调器进行复制，解决了事务协调器崩溃时持有锁造成的无限阻塞问题。

注意，不管事务有没有被提交，它都会被复制到 Paxos log 中，如果事务协调器选择提交事务，那么它们的日志中就会出现 commit 消息，此时如果崩溃了，其他服务器也可以接手事务协调器的工作，因为它会在自己的日志中看到这条 commit 消息，然后可以去通知别的 leader 进行提交。

但我们也能感觉到，这其中需要的跨数据中心的通信太多了，Spanner 有很多用途，它们的所有 replica 可能都是放的比较近的，以提升速度。

只读事务

只读事务的读操作和读写型事务中的读操作有很大不同。首先，它从本地 replica 中读取数据，但本地 replica 的数据可能不是最新的，后面会处理这个问题。第二个不同是，它没有使用锁，也没有使用两阶段提交，即不需要事务管理器进行管理，这样也可以避免跨数据中心地从别的 Paxos 组中的 leader 读取数据。

只读事务引入两个约束以保证正确性：

1. 它们想让所有事务的执行依然是有序的。即并发执行事务时，执行的结果也和某种确定的顺序一样。对于只读事务来说，其能看到在它执行只读事务前所有写操作的结果
2. 另一个约束就是获取外部一致性的能力

下面来看一个例子，有三个事务并发执行，T1、T2 是转账事务，T3 是只读事务。T1 执行时，结果为 x = 9，y = 11，T2 执行后，结果为 x = 8，y = 12，T3 是读取 x 和 y 的值的事务。如果 T3 在 T1 执行完前读到了 x 的值，在 T2 执行完后才得到了 y 的值，那么 T3 的结果就是 9、12。这个结果和我们预期的不符，这和任意顺序执行事务的结果都不一样，所以不满足一致性。

Spanner 解决这个问题的方式有点复杂。

第一个解决思路就是快照隔离。假设所有机器的时钟是同步的，我们对每个事务都会分配一个时间戳，对于读写型事务来说，它的时间戳为事务提交的时间，对于只读事务来说，它的时间戳为事务开始的时间。然后我们会根据时间戳来对事务的执行顺序进行安排。

当每个 replica 保存数据时，它会保存该数据的多个副本，它会记录每个数据的多次写入。当事务发送读请求时，它会让读请求携带一个时间戳，不管哪台服务器保存了该数据副本，它都会去寻找最大的小于只读事务时间戳的数据副本来获取最新的数据。因此上述执行时 T3 可以通过对比时间戳获取正确的值。

那为什么 T3 读到旧的 y 值是正确的呢？因为 T2 和 T3 是并发执行的，对于线性一致性和外部一致性来说，如果两个事务并发执行，那么执行顺序可以是任意的，即要么执行第一个再执行第二个，要么先执行第二个再执行第一个，因此这是 ok 的。

此外存储数据的多个副本是否会有耗尽空间的问题？

Spanner 有丢失较老数据副本的策略，如果数据太老了，其会将老的数据进行清除。

外部一致性强加的一条规则为：如果一个事务已经完成了，那么在它之后开始执行的事务必须看到前面事务写操作所做的修改。

下面需要处理读本地 replica 时，如果里面的数据版本不是最新的怎么办？

每个 replica 会去对比当前读请求的时间戳和 leader 最近发送的时间戳，如果 leader 发送的时间戳是比较老的，那么 replica 就会延迟返回数据，这样可能会造成一点延迟，如果时间戳大于等于读请求的时间戳，那么就可以返回相应的数据。论文中确保 leader 会严格按照时间戳增加的方式来发送日志记录。这也被称为安全时间：safe time

时钟同步

我们要确保不同服务器上的时钟所读取的是同一个值。因为我们无法如此精确地对时钟进行同步。正常来说获取时间是通过政府实验室将时间传播到 GPS 卫星上的，然后GPS 卫星再传播到有 GPS 卫星接收器的设备上。

对于读写型事务，由于其使用了锁和二阶段提交，它们实际上没有使用快照隔离。基于两阶段锁的机制，读写型事务依然是有序执行的。

由于时钟不同步，只读事务可能选择了一个很大的时间戳，导致在本地读取数据时，需要等待很久（等待 replica 收到时间戳大于该时间戳的日志为止），这里可能会有一个超时机制，对 reader 说过一段时间再来访问。如果由于发起只读事务的服务器时钟慢了，导致其读到的是旧的副本，但是在正确时间下，其前面已经对数据做了很多更新，这样就不能保证外部一致性了。

下面来看看时钟同步。每个数据中心中会有 GPS 接收器，GPS 接收器会和 time master 连接，为了避免故障导致不可用，数据中心里可能会有多个 time master 服务器。每个数据中心有数百台运行着 Spanner 的服务器，有的作为 server 使用，有的作为 client 使用。每台服务器会定期向本地 time master 服务器发送一个询问时间的请求，然后 time master 会告诉我们当前时间是多少。

但是在时间返回的过程中可能会有延迟，对方可能立马就响应了，但是传播时间慢了几纳秒。误差是一直存在的，而且是没法忽略的。并且在服务器发送查询时间的请求时，其也会走自己的本地时钟（这些时钟很糟糕），发送时间请求时本地时钟会发生一些偏移，因此不确定性非常多。

Spanner 采用的是 TrueTime 方案。当你询问时间的时候，你得到的是时间区间，里面有一个最早时间（earliest）和最晚时间（latest），当前时间是在这两个时间中的某一点（这是我们需要保证的东西）。

当一个事务询问系统时间的时候，就会得到这个区间。Spanner 通过两条规则来做到外部一致性：

1. Start rule

   该规则指的是一个事务选择的执行时间为询问时间得到的区间的最晚时间，这是一个还未发生的时间点。读写型事务和只读事务都是采用 latest 来作为时间戳。

2. commit wait

   该规则只用于读写型事务。他表示事务协调器去收集投票信息，并检查是否能够提交该事务，并为该事务选择一个时间戳。当选完时间戳后，其需要等待一段时间，直到到达它选择的时间戳为止，才会进行提交。判断是否到达的方式就是通过循环不断访问当前时间，直到返回的区间的 earliest 大于我们选择的时间戳，此时就一定能保证我们的时间戳已经到达了

这就是 Spanner 如何强制让它的事务保证外部一致性的方法了。

总的来说，有两个很重要的东西：

1. 快照隔离。快照隔离能够保证只读事务的有序进行，通过时间戳来看到之前所有读写型事务的修改。但是一般来说它无法保证外部一致性。因为时钟可能是不同步的。
2. 时钟同步。

Spanner 中的只读事务执行速度很快，不需要锁，也不需要两阶段提交，而且是从本地 replica 中读取数据。不过由于 safe time 和 commit wait 的原因，很多时候 Spanner 还是会遇到阻塞的情况。