

## 分布式系统-分布式事务 (P2)

OriginalAccela推箱子 Accela推箱子 3天前

(续前文……)

### 分布式系统-分布式事务 (P1)

分布式事务是存储系统和数据库最有用也是最有挑战的一方面。前文介绍了事务隔离级别，读写冲突，ACID，等等。下面，我们来看经典的Percolator和Spanner如何实现分布式事务。

### TLA+ (Temporal Logic of Actions)

首先介绍TLA+，后文分析Percolator事务协议时会涉及。

分布式事务微妙而复杂，如何清晰地表达算法，如何验证正确性？形式化语言TLA+非常好用。TLA+为分布式系统设计，可以表达复杂的并发多结点交互，用数学严格验证算法正确性。

详细地介绍可以看TLA+ 主页和TLA+ Book。—(直接介绍知识写腻了)—，本文换种方式，来思考TLA+为什么出色。

[\[TLA+ Homepage\]\(http://lamport.azurewebsites.net/tla/tla.html\)](http://lamport.azurewebsites.net/tla/tla.html)

### TLA+的强大背景

对开源软件等，血缘分析往往能暗示其前景。

TLA+是谁推出来的？Lesile Lamport，分布式系统理论的祖师爷，最著名的贡献有Vector Clock、Paxos、Consistency Snapshot等。

Lamport花了多少精力做TLA+？罄竹难书……（成语误用，但真好用）

[\[Lamport Publications\]\(http://lamport.azurewebsites.net/pubs/pubs.html\)](http://lamport.azurewebsites.net/pubs/pubs.html)

94. <a href="#">Critique of the Lake Arrowhead Three</a>	139. <a href="#">Paxos Made Simple</a>
95. <a href="#">The Reduction Theorem</a>	140. <a href="#">Specifying and Verifying Systems with TLA+</a>
96. <a href="#">Mechanical Verification of Concurrent Systems with TLA</a>	141. <a href="#">Arbiter-Free Synchronization</a>
97. <a href="#">Composing Specifications</a>	142. <a href="#">A Discussion With Leslie Lamport</a>
98. <a href="#">Verification of a Multiplier: 64 Bits and Beyond</a>	143. <a href="#">Lower Bounds for Asynchronous Consensus</a>
99. <a href="#">Verification and Specification of Concurrent Programs</a>	144. <a href="#">Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers</a>
100. <a href="#">Hybrid Systems in TLA+</a>	145. <a href="#">Checking Cache-Coherence Protocols with TLA+</a>
101. <a href="#">How to Write a Proof</a>	146. <a href="#">High-Level Specifications: Lessons from Industry</a>
102. <a href="#">The Temporal Logic of Actions</a>	147. <a href="#">The Future of Computing: Logic or Biology</a>
103. <a href="#">Decomposing Specifications of Concurrent Systems</a>	148. <a href="#">Consensus on Transaction Commit</a>
104. <a href="#">Open Systems in TLA</a>	149. <a href="#">On Hair Color in France</a>
105. <a href="#">TLZ (Abstract)</a>	150. <a href="#">Formal Specification of a Web Services Protocol</a>
106. <a href="#">An Old-Fashioned Recipe for Real Time</a>	151. <a href="#">Cheap Paxos</a>
107. <a href="#">Specifying and Verifying Fault-Tolerant Systems</a>	152. <a href="#">Implementing and Combining Specifications</a>
108. <a href="#">How to Write a Long Formula</a>	153. <a href="#">Lower Bounds for Asynchronous Consensus</a>
109. <a href="#">Introduction to TLA</a>	154. <a href="#">Generalized Consensus and Paxos</a>
110. <a href="#">Adding "Process Algebra" to TLA</a>	155. <a href="#">Real Time is Really Simple</a>
111. <a href="#">What Process Algebra Proofs Use Instead of Invariance</a>	156. <a href="#">How Fast Can Eventual Synchrony Lead to Consensus?</a>
112. <a href="#">Conjoining Specifications</a>	157. <a href="#">Real-Time Model Checking is Really Simple</a>
113. <a href="#">TLA in Pictures</a>	158. <a href="#">Fast Paxos</a>
114. <a href="#">The RPC-Memory Specification Problem: Problem Statement</a>	159. <a href="#">Measuring Celebrity</a>
115. <a href="#">A TLA Solution to the RPC-Memory Specification Problem</a>	160. <a href="#">Checking a Multithreaded Algorithm with +CAL</a>
116. <a href="#">How to Tell a Program from an Automobile</a>	161. <a href="#">The PlusCal Algorithm Language</a>
117. <a href="#">Refinement in State-Based Formalisms</a>	162. <a href="#">TLA+</a>
118. <a href="#">Marching to Many Distant Drummers</a>	163. <a href="#">Implementing Dataflow With Threads</a>
119. <a href="#">Processes are in the Eye of the Beholder</a>	164. <a href="#">Leslie Lamport: The Specification Language TLA+</a>
120. <a href="#">How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor</a>	165. <a href="#">Computation and State Machines</a>
121. <a href="#">Substitution: Syntactic versus Semantic</a>	166. <a href="#">A TLA+ Proof System</a>
122. <a href="#">The Part-Time Parliament</a>	167. <a href="#">The Mailbox Problem</a>
123. <a href="#">Reduction in TLA</a>	168. <a href="#">Teaching Concurrency</a>
124. <a href="#">Composition: A Way to Make Proofs Harder</a>	169. <a href="#">Vertical Paxos and Primary-Backup Replication</a>
125. <a href="#">Proving Possibility Properties</a>	170. <a href="#">Computer Science and State Machines</a>
126. <a href="#">A Lazy Caching Proof in TLA</a>	171. <a href="#">Reconfiguring a State Machine</a>
127. <a href="#">Specifying Concurrent Systems with TLA+</a>	172. <a href="#">Stoppable Paxos</a>
128. <a href="#">TLA+ Verification of Cache-Coherence Protocols</a>	173. <a href="#">Verifying Safety Properties With the TLA+ Proof System</a>
129. <a href="#">Should Your Specification Language Be Typed?</a>	174. <a href="#">Byzantizing Paxos by Refinement</a>
130. <a href="#">Model Checking TLA+ Specifications</a>	175. <a href="#">Leaderless Byzantine Paxos</a>
131. <a href="#">How (La)TeX changed the face of Mathematics</a>	176. <a href="#">Euclid Writes an Algorithm: A Fairytale</a>
132. <a href="#">Fairness and Hyperfairness</a>	177. <a href="#">How to Write a 21st Century Proof</a>
133. <a href="#">Archival References to Web Pages</a>	178. <a href="#">TLA+ Proofs</a>
134. <a href="#">Disk Paxos</a>	179. <a href="#">Why We Should Build Software Like We Build Houses</a>
135. <a href="#">Disk Paxos (Conference Version)</a>	180. <a href="#">Adaptive Register Allocation with a Linear Number of Registers</a>
136. <a href="#">When Does a Correct Mutual Exclusion Algorithm Guarantee Mutual Exclusion</a>	181. <a href="#">Coalescing: Syntactic Abstraction for Reasoning in</a>
137. <a href="#">Lower Bounds on Consensus</a>	182. <a href="#">Who Builds a House without Drawing Blueprints?</a>
138. <a href="#">The Wildfire Challenge Problem</a>	183. <a href="#">Auxiliary Variables in TLA+</a>

谁在使用TLA+? 首先是云计算领头羊AWS, 在S3、DynamoDB、EBS等分布式系统中应用TLA+, 找到多种微妙Bug, 并且投资回报出色 (人们最常担心的就是形式化验证成本过高)。TiDB大量使用TLA+, 贡献了出色的Github代码。阿里X-DB使用TLA+ 验证其X-Paxos算法。另外, 一些论文也以TLA+作为其算法的Formal Proof。

[How Amazon Web Services Uses Formal Methods]  
(<http://www.cslab.pepperdine.edu/warford/math221/How-Amazon-Web-Services-Uses-Formal-Methods.pdf>)

[TiDB Pingcap/TLA-Plus](<https://github.com/pingcap/tla-plus>)

[Alibaba X-DB & X-Paxos]  
(<https://mp.weixin.qq.com/s/BCBRewfxCg2i3bDqmHzoLg>)

[Paper: CASPaxos](<https://arxiv.org/pdf/1802.07000.pdf>)

**TLA+本身的特点**

与以往形式化语言复杂晦涩的印象相比，TLA+非常简单简洁。即使Percolator这样复杂的分布式协议，也只需不到500行代码，大部分还是注释。

TLA+代码不仅是严谨的数学，而且可执行，不用担心其正确性。执行验证有两套工具，TLC通过状态迭代，而TLAPS用数学推导。

这也使TLA+成为传播算法知识的优良途径。例如，读论文后，希望深入了解，希望看到算法的完整、严谨、可执行的实现，那就可以找TLA+。如果疑惑算法中某条规则的作用，那么大可以在TLA+代码中删除之再运行，看看哪里出错。Github上已有大量算法的TLA+代码，例如Paxos和Raft。

[\[Github DrTLAPlus\]\(https://github.com/tlaplus/DrTLAPlus\)](https://github.com/tlaplus/DrTLAPlus)

## TLA+语言的设计

在基本的数学逻辑上，TLA+语言的主要不同，也是神来之笔，在于引入了时空运算符（Temporal Operator，翻译成“时空”更拉风，写作“[]”），从而建立了时空逻辑（Temporal Logic）。从此数学逻辑有了在时间上的表达，正适用于分布式系统随时间演进的状态。并且，时空运算还满足分配律……

### 8.2. TEMPORAL TAUTOLOGIES

93

$$\Box(F \wedge G) \equiv (\Box F) \wedge (\Box G)$$

$F$  and  $G$  are both always true iff  $F$  is always true and  $G$  is always true.

Another way of saying this is that  $\Box$  distributes over  $\wedge$ .

$$\Diamond(F \vee G) \equiv (\Diamond F) \vee (\Diamond G)$$

$F$  or  $G$  is eventually true iff  $F$  is eventually true or  $G$  is eventually true.

Another way of saying this is that  $\Diamond$  distributes over  $\vee$ .

从TLA+的角度，对分布式系统的约束可以分为Safety和Liveness两大类。Safety指系统的某些属性不应被违反，相应的数学表达式恒真；Liveness指算法的进程能够持续推进，而不是卡在某处不动，它可以被表达成一个包含时空运算符的表达式。

对应到TLA+代码，通常开头是变量声明，然后是各个子函数的定义。接下来是Init和Next，分别表达系统初始态和状态推进规则。最后是定理部分，检查Safety和Liveness；它们写成表达式，会在TLA+执行时被验证。

此外，就运算符的写法而言，TLA+和C++差得较远，不知道TLA+最初是以什么语言为模板设计的；总之习惯就好。

## TLA+不能做的事

目前TLA+代码无法直接翻译成完整的项目工程代码，所以算法实现过程仍可能引入Bug。此外，通常只能把项目中的关键算法写为TLA+进行验证，而翻译全部逻辑则太过繁杂。

尽管有这些限制，但哪怕仅用TLA+验证关键算法，实践效果已非常出色。

## Percolator

现在回到分布式事务的代表作Percolator。Percolator建于BigTable上，批量计算Google网页搜索索引，核心是跨行跨表的分布式事务。TiDB作者非常良心地用TLA+写出了Percolator事务算法，甚至比原论文更清晰：

[[Github tla-plus/Percolator/Percolator.tla](https://github.com/pingcap/tla-plus/blob/master/Percolator/Percolator.tla)](https://github.com/pingcap/tla-plus/blob/master/Percolator/Percolator.tla)

与其直接介绍原算法，本文试图换个角度。假如我们需要设计一套分布式事务系统，通过逐个审视设计上的问题和解决方法，来理解Percolator的精妙。

[[Percolator Paper](https://storage.googleapis.com/pub-tools-public-publication-data/pdf/36726.pdf)](https://storage.googleapis.com/pub-tools-public-publication-data/pdf/36726.pdf)

## 首先，我们的积木有什么

在搭建Percolator楼房前，先看看我们有什么基础积木可用。

Percolator构建于BigTable上，这意味着，BigTable为我们解决了事务数据的持久化（Persistence）和复制（Replication）问题。

BigTable提供行级别的CAS（Compare-and-Set）操作，意味着我们可以用CAS作为原子，搭建跨行跨表的事务。此外，Percolator.tla中可以看到，其并不需要事务中常见的日志（Journal），这应该也是因为CAS自身已包含了它。

## 分布式事务的原子提交

整个事务的过程，大体是上锁、读写、提交（Commit）。最关键的一步，提交，需要原子地检查锁仍持有并写入提交。需要提交的数据可能分布在多个表、多个结点上；而可能夺走锁的事务也分布于各处。

问题来了：我们已有行内CAS操作，如何实现原子提交分布式事务呢？这就是Percolator的精妙。

- 第一点，提交跟随Primary Key。这里Primary Key不是数据库主键，而是Percolator新引入的概念。分布式事务最终需要写入的，是一组分布在各结点的Key（行的主键）。我们选出其中一个作为Primary Key，认定Primary提交则整个事务都提交，Primary未提交则整个事务都未提交。选择方法可以是Paxos或者客户端自己。这样，提交操作就划归到单结点单表了。接下来的问题是，Secondary Key（Primary之外的其它Key）的提交应该在何时完成？Percolator的方案是，在读操作中懒惰地替前面的事务完成它们Secondary Key的提交。
- 第二点，Staging式的写入和提交。下图是Percolator表在BigTable中的数据结构。写入数据在bal:data列中，写入的数据是对外不可见的，事务可以多次写入。而提交操作，则是在bal:write列中，写入一个指向bal:data中实际数据的指针；拥有指针，则数据可见。这个思路类似Git的Staging和Commit；写入的数据Staging在bal:data里，数据量可以很大，但是对外不可见；而提交则用很小的原子的指针，作为开关。
- 第三点，锁在行中。下图中可以看见，上锁与否bal:lock，和数据bal:data、指针bal:write被设计在同一行中。这样，一个行内CAS操作，就可以原子地检查上锁情况，同时进行提交了。此外，锁的数据还记录了Primary Key在哪里，方便从Secondary查询Primary是否上锁。与“第一点”中类似，上锁也是跟随Primary的。

<i>key</i>	<i>bal:data</i>	<i>bal:lock</i>	<i>bal:write</i>
Bob	8:	8:	8: <b>data @ 7</b>
	7: \$3	7:	7:
	6:	6:	6: data @ 5
	5: \$10	5:	5:
Joe	7: \$9	7: primary @ Bob.bal	7:
	6:	6:	6: data @ 5
	5: \$2	5:	5:

## 可以实现无锁化算法吗？

无锁化算法非常吸引人，通常它们需要CPU的CAS操作支持。但对于分布在多个结点的事务，并没有一个巨大的CPU来为它们提供CAS操作。因此“锁”常常仍然不是一个能避免的概念；不过我们可以锁得更“乐观”。Percolator的CAS则来得力于BigTable。

另一方面，则是锁的阻塞行为的实现困难。阻塞需要唤醒，唤醒需要消息通知，而在分布式系统的不可靠消息传输环境下，唤醒消息成了麻烦事；或者靠轮询来检查唤醒，则牺牲了效率和延迟。而Percolator虽然上锁，但并不阻塞；后来者事务会抢夺前者的锁，前者发现锁被抢后会退出或重试。

## 锁的冲突和阻塞

读碰上写怎么办？Percolator的读操作会首先清除掉之前事务的锁，即抢夺之而非阻塞；这样会导致之前事务的失败或重试，后面会看到它有助于事务隔离。实际实现时，可以更“温和”，例如先等待一小段时间，或者检查之前事务的Liveness。

写碰上写怎么办？Percolator会为所有需要写入的Key上锁。如果写锁冲突，则两个事务之一需要退出或重试。

## 时间戳和时序 (Time Ordering)

事务算法离不开时间排序，我们需要知道哪个事务更晚提交，事务生命期是否重叠。Percolator使用中心服务器 (Timestamp Oracle) 提供数据中心内统一的时间戳，这个问题就被简化了。而Spanner因为有跨数据中心事务的需求，引入了TrueTime，后文将讲解。

## 事务隔离

Percolator如何实现其声称的Snapshot Isolation事务隔离级别呢？

其实读了前文，凭直觉我们也可以知道，事务可分成读阶段和写阶段：既上读锁也上写锁，则Serializable；不上读锁但上写锁，则Snapshot Isolation；而遇到锁时，选择阻塞等待则“悲观”；不等待而去抢夺，或者不上锁而最后CAS检查，则“乐观”。

理解Percolator，重点是看它的读写操作在哪里上锁，又在哪里检查锁。首先，提交Primary时要求Primary已上锁，而上锁操作要求没有比自己事务更晚的写入；这样就避免了写-写冲突，配合BigTable的时间戳版本机制，达到了Snapshot Isolation的要求。

读-写冲突在Percolator中如何呢？假设，事务Tx1先读取数据；在Tx1提交前，事务Tx2提交并修改了Tx1所读的数据；那么就发生了读-写冲突。这是不可能发生的：因为在Tx1读取数据时，如果Tx2已上锁，则Tx1会清除掉Tx2的锁，导致Tx2提交失败；Tx1提交前上锁时，会检查是否有更晚的写入，如果Tx2已提交，则Tx1上锁失败。这样看来，Percolator可以达到Serializable要求；但实际实现也许可以配置省掉一些锁，提高性能，而只达到Snapshot Isolation。

## Percolator的事务流程

砖石完毕，最后我们组装Percolator的分布式事务实现。

下图以Percolator.tla为基础，把流程拼接起来，加以注释；从上到下即为事务流程。

TLA+语言的Percolator.tla（前文有链接），不超过500行，已经是清晰完整可执行的算法，可以详细阅读。

```

ClientOp(c) ==
  \ Start(c)           // 事务开始, 获取起始时间戳
  \ Get(c)
    cleanupStaleLock  // 如果之前事务未提交, 则清除其锁; 如果已移交, 则帮助其提交Secondary Keys
    readKey
  \ Prewrite(c)
    lock
      canLockKey
        /\ key_lock[k] = {} // no any lock for the key.
        /\ writes = {}     // no any newer write.
      lockKey           // 先后为Primary和Secondary上锁
      commit_ts = next_ts' // 获取commit时间戳
  \ Commit(c)
    commitPrimary      // 提交只需Primary
    hasLockEQ(primary, start_ts)
    key_write' = .. Append // 写入bal:write指针, 提交可见
    key_lock' = .. // 释放锁
  \ Abort(c)

```



## Spanner

Spanner是谷歌的全球分布式数据库，支持强一致ACID事务。相比Percolator用于单数据中心或邻近地理区域，Spanner需要解决跨数据中心跨地区的分布式事务。遥远的地理距离带来了高网络延迟、时间同步困难等问题。

与前文相似，相比介绍知识，本文角度思考构建这样分布式事务会碰到的问题，和Spanner所采用的解决方法。

[Spanner Paper](<http://research.google.com/archive/spanner-osdi2012.pdf>)

### 数据持久化 (Persistence) 和数据复制 (Replication)

分布式数据库首先作为存储系统，需要解决数据如何可靠存储的问题。

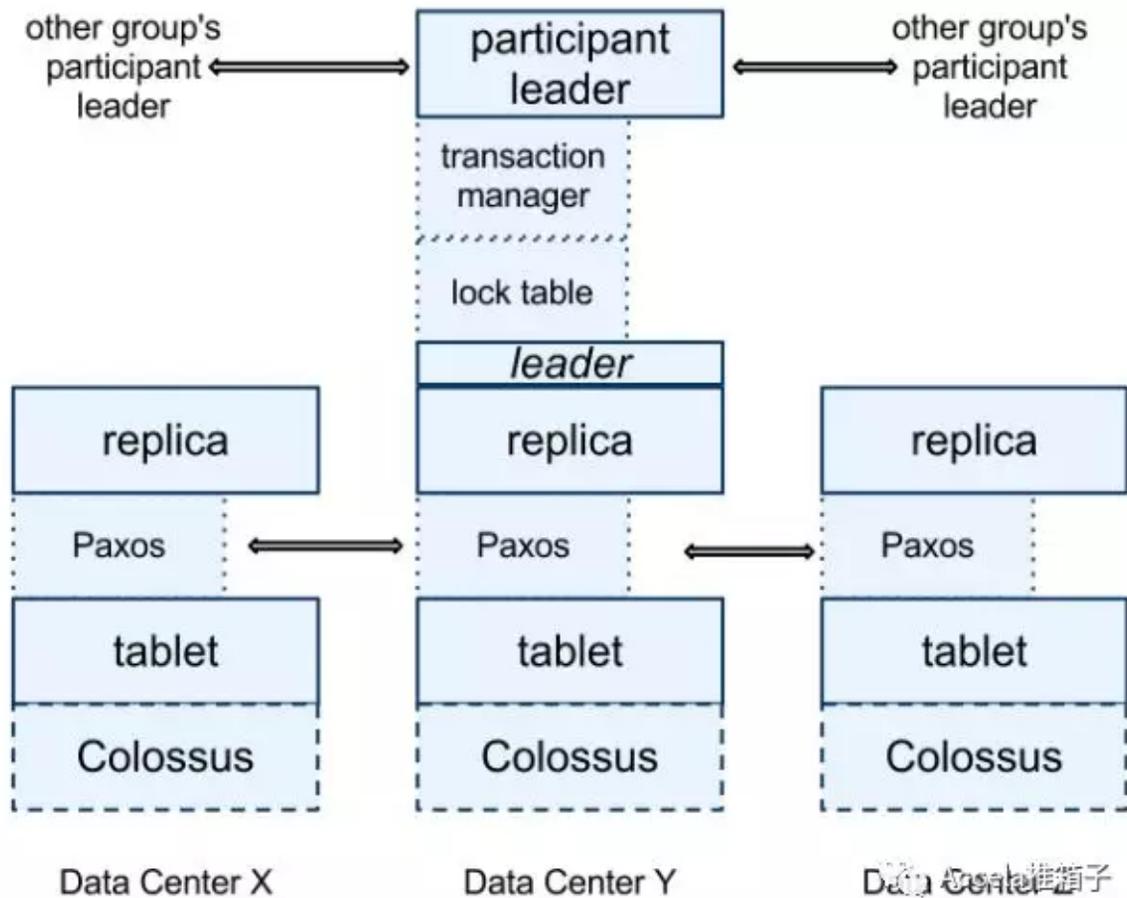
Spanner并不像Percolator构建于BigTable上，而是在底层使用Colossus（Google Filesystem），并自己管理数据复制。

数据复制的基本方法是3-Way Replication，复制3个副本（Replica）；而3副本间一致性的裁决通常需要Paxos协议。我们可以使用中心式的Paxos元数据管理服务，例如Ceph的Monitor组件。

或者更炫酷地，每一组3副本都组成一个Paxos Group，用Paxos作为数据复制的协议，例如Google Megastore。集群中会有成千上万个独立Paxos Group运行。这种方法避开了

中心式元数据管理的Paxos吞吐量上限。

Spanner采用的是后者。Paxos Group中会选出一个副本作为Leader。



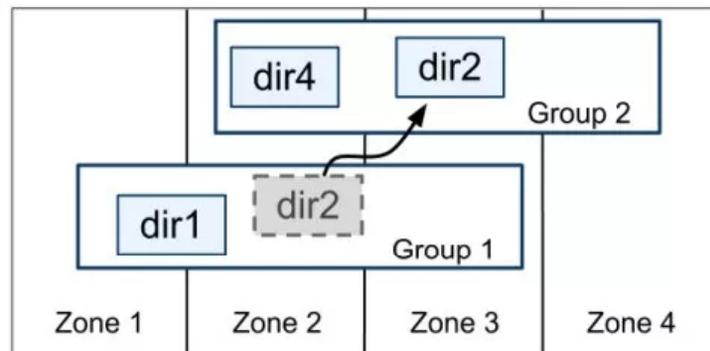
## 数据的组织结构

除了副本作为数据复制的Replication Unit，数据库中还有其它数据单元需要考虑。例如

- 数据分配和迁移的单元， Placement Unit。我们可以把Placement Unit放到访问热度、存储空间平衡（Balanced）的服务器上，可以迁移Placement Unit到离客户端更近的位置。
- 表格连续的区间查询（Range Query）的单元， Table Partition。SQL常查询一组连续的Key区间。将Key连续存储，性能明显优于简单地哈希（Hash）打散。一小片连续的Key区间，成为Partition。

- 分布式事务的Locality。跨结点的分布式事务毕竟昂贵，我们希望合理放置数据，使尽量多的事务只需单结点的数据，或包含尽量少的结点。由此也可见支持业务定制Placement算法的重要性。
- 业务需求也会影响数据组织。如Conway's law，软件架构反映部门组织架构。

Spanner中，则引入了Directory作为Placement Unit，体积小可快速迁移。1 Replica包含n个Directory，1个Directory包含多个Table Partition。如果Directory过大，则会把它分裂为更小的Fragments。



说起Data Placement，如何完全使用哈希方法，可以省去大量“什么数据放在哪个服务器”的元数据。这个方法在Ceph的CRUSH算法中发扬光大。但也有弊端，除集群扩容时易引起大规模数据迁移外，它难以做精细的Placement平衡，因而集群容量可能无法完全利用。另一方面，哈希方法主要按照数据量来平衡，但同体积的数据访问热度差异巨大，则难以纳入到哈希方法中。更多内容见下文

[汪涉洋- Ceph运维告诉你分布式存储的那些“坑”](<https://mp.weixin.qq.com/s/u1-Vj9mllemzBLkgFvWC5w>)

## 跨地理区域的时间同步

Spanner的TrueTime大名鼎鼎。

时间戳是分布式事务的关键，同一数据中心内可以使用中心时间戳服务器，或者依赖低延迟通信的时间同步算法。但对于跨地理区域的多数据中心，快速同步地生成时间戳就成了难题。

软件的难题总是可以看向硬件。TrueTime采用了原子钟（Atomic Clock）和GPS天线两种特殊硬件，作为时间源（Time Reference）获取全球同步的精确时间。采用两种硬件，可以互补其不同的故障模式（Failure Mode）；毕竟时间戳是分布式事务可用性（Availability）的根本。数据中心内只需放置数台时间Master服务器，并不需要为所有服务器配备特殊硬件，成本并不高。

硬件层之上，可以通过软件进一步优化时间服务。Spanner的时间Master服务器间会互相校验失准，排除故障服务器。Spanner客户端查询本地和邻近数据中心的多个时间Master服务器，对比排除故障服务器。Spanner之外，也有类似NTP时间同步的算法，通过服务器互相通信校正时间漂移（Time Drift）。

### 事务隔离级别和一致性模型

Spanner采用Serializable隔离级别和External Consistency。后者应该是Spanner新引入的概念，其想要解决的问题在于，Serializable仅仅要求事务读写在序列化图（Serialization Graph）中不形成环，而对事务的Commit Timestamp没有要求；后提交的事务可以拥有更新的时间戳，导致Snapshot Read误把旧数据当新数据。因此，引入External Consistency的概念，如论文中所述，其定义是：如果事务TX2比TX1后提交，那么TX2的Commit Timestamp必须大于TX1。从另一方面看，External Consistency实际上是试图把Linearizability的概念，混入事务隔离级别概念中。

下文有对包括Linearizability的一致性模型和事务隔离级别的完整讲解

[唐刘- 一致性模型- TiDB 的后花园](<https://zhuanlan.zhihu.com/p/47445841>)

下文中有专对Spanner的External Consistency的解读

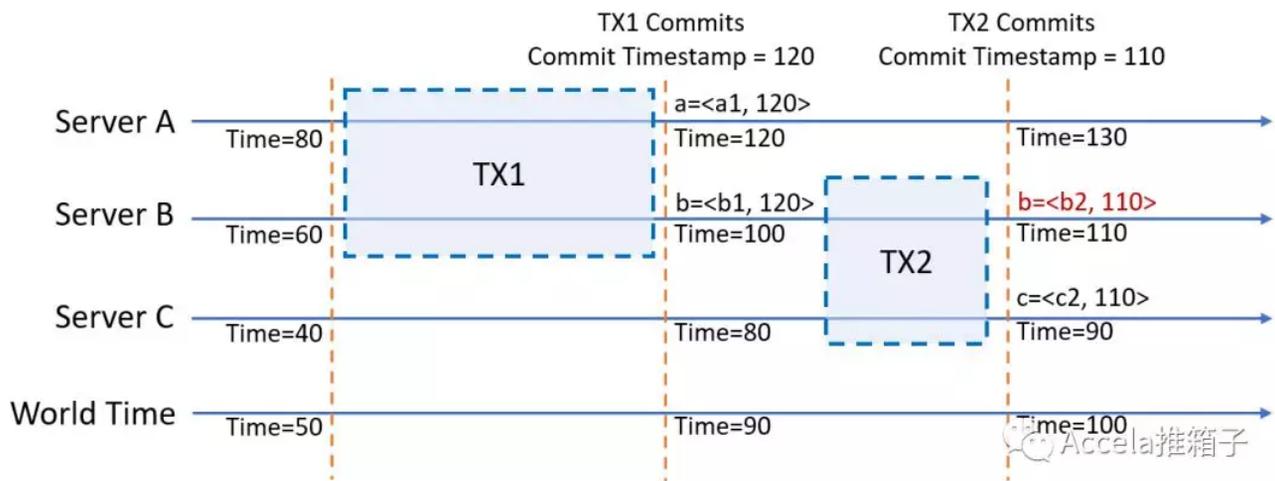
[阿莱克西斯 - 如何理解数据库的内部一致性和外部一致性 - 知乎] (<https://www.zhihu.com/question/56073588/answer/519284998>)

### 关于Commit Wait

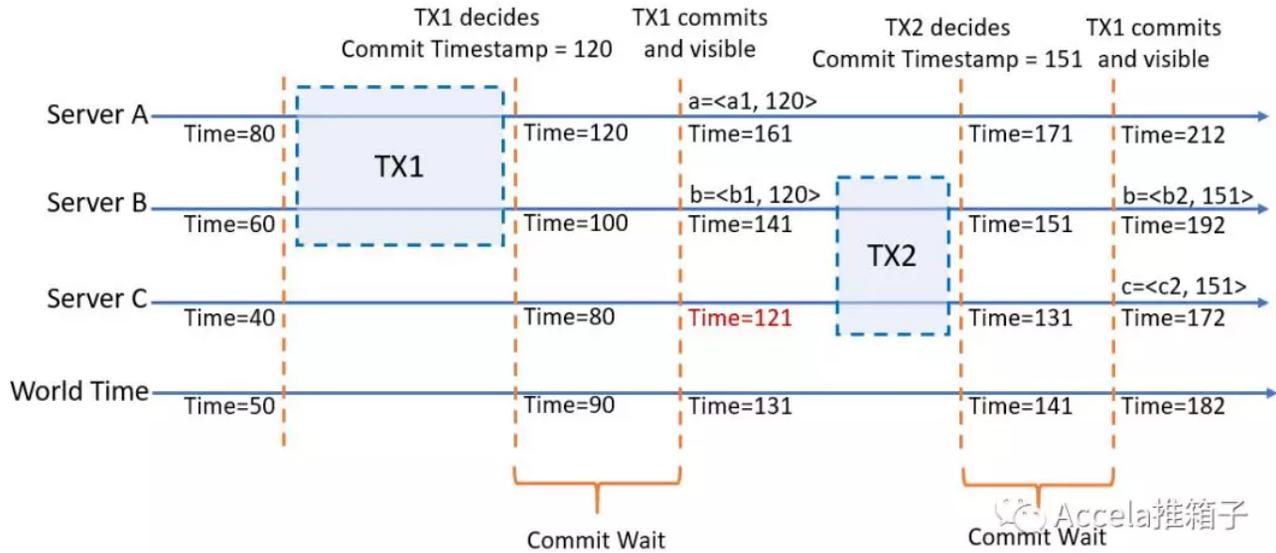
Commit Wait是Spanner的标志性设计。

引入Commit Wait是为了实现External Consistency；以保证后提交的事务，一定有更大的Commit Timestamp。系统首先分配Commit Timestamps，Commit Wait需要等待到TrueTime.after(s)为true，即越过时间不确定期。这样看来，TrueTime时间定位越精确，Commit Wait代价越小，这也是引入需要特殊硬件的TrueTime的原因。

Commit Wait如何保证先后事务的Commit Timestamp顺序呢？假想没有Commit Wait，分布式事务可以在多个互相重叠的结点组间执行，在特定的执行顺序和时间戳下，会出现后提交的事务，因为看不到之前事务的时间戳，反而得到更小的Commit Timestamp的情形，从而违反了External Consistency。如下图所示， $b = \langle b2, 110 \rangle$ 后提交，但时间戳110反而小于先提交的 $b = \langle b1, 120 \rangle$ 的120。



下图是增加了Commit Wait之后的事务执行过程。TX1 Commit Wait后，才会释放锁，TX2才能开始。此时所有结点的Timestamp都大于TX1的Commit Timestamp了，因而无论TX2执行速度多快，都无法违反External Consistency。当然，一切的隐含条件是，时间戳必须单调递增。



下文有对Spanner、TrueTime、Commit Wait更加详细的讲解

[阿莱克西斯 - 简单解释 Spanner 的 TrueTime 在分布式事务中的作用] (<https://zhuanlan.zhihu.com/p/44254954>)

谈起TrueTime和Commit Wait, 可以进一步改进。事务一致性有两条路线: TrueTime试图逼近精确的现实时间, 代价是Commit Wait; Vector Clock则抛弃现实时间, 追逐因果关联, 以Causal Consistency解决问题。下面论文结合两条路线, 将Spanner中的Commit Wait机制进一步优化。例如, 上面时序图中, 因为TX1和TX2重叠于Server B, Vector Clock信息会沿着Server B从TX1传递至TX2; 从而即使没有Commit Wait, TX2也不会给出小于TX1的Commit Timestamp。

[Beyond TrueTime: Using AugmentedTime for Improving Spanner] (<https://cse.buffalo.edu/~demirbas/publications/augmentedTime.pdf>)

## 锁和阻塞

在实现分布式事务前, 需要先解决锁的实现问题。从Spanner论文中, 可以看到其使用读锁和写锁, 通过2PC Locking (两阶段上锁), 或对应地, 称作2PC Commit (两阶段提交), 实现Serializable隔离级别的事务。中规中矩。

与Percolator的读会积极清除前事务的锁不同, Spanner的读写事务中, 读可能阻塞等待。其使用Wound-Wait方法, 即老事务可以抢夺新事务的锁使后者重启, 而新事务需阻

塞等待老事务的锁。Wound-Wait是一种带抢断（Preemptive）的死锁避免方法，与之相呼应的是Wait-Die。Spanner采用前者，可能是为了适配业务中长运行时间的事务，例如报表。

锁的信息记录在哪里？Spanner中，Paxos Group的Leader结点上，运行有Lock Table，用于存储2PC Locking中的锁。锁的信息能够通过Paxos Group可靠地持久化。

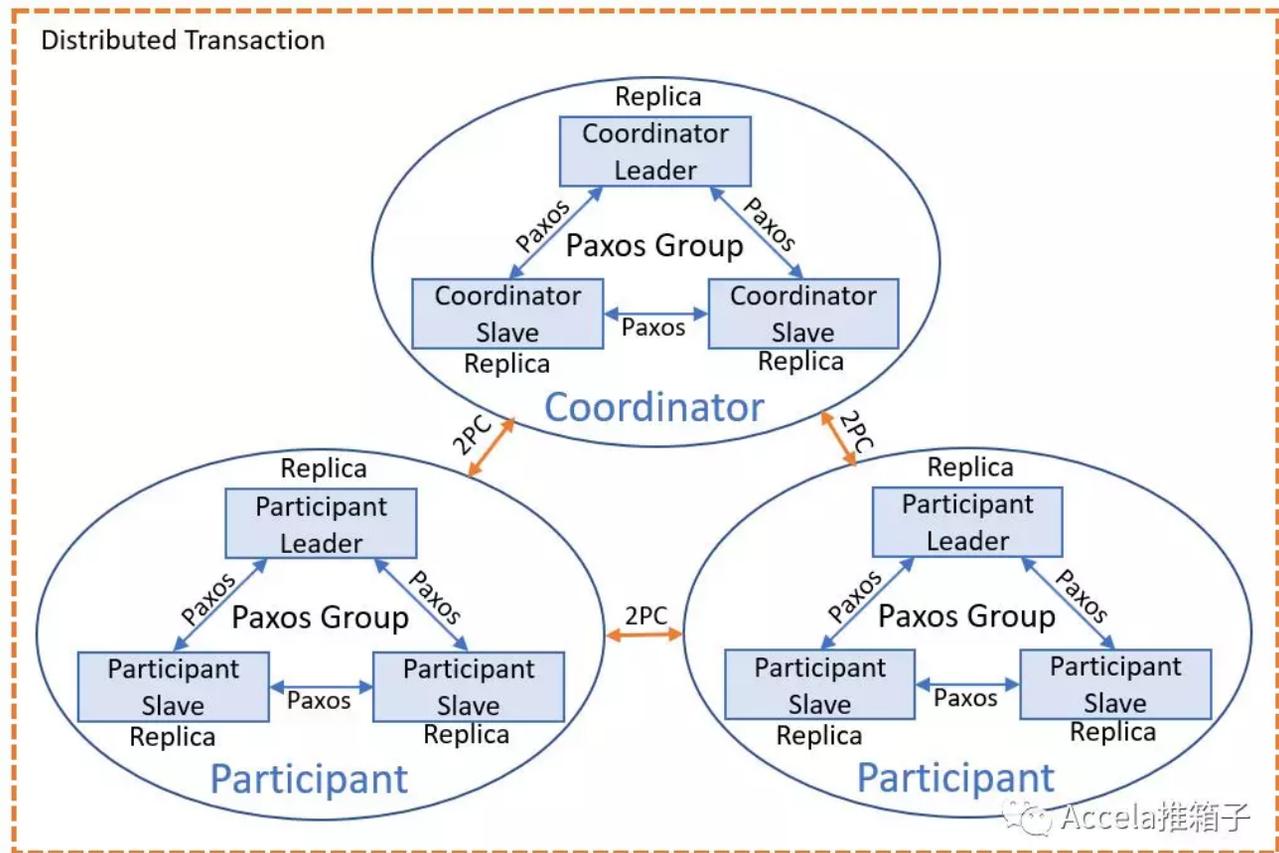
关于锁的阻塞，等待锁的一方，如何在锁释放时得到通知？Spanner论文中似乎没有提及。可能的实现是，等待方轮询远端的Lock Table，或者，当锁释放时Lock Table方发送通知。

## 分布式事务的实现

解决了上面的问题，接下来我们可以直奔主题。Spanner如何实现Serializable和External Consistency的分布式事务呢？

对于读写事务，用的是中规中矩的2PC Locking，依次上读锁和写锁。但是，上写锁的时间被延迟到提交之前；先计算写数据再上写锁，这就有了“乐观”（Optimistic）的并发控制（Concurrency Control）。

读写事务可能涉及多个Paxos Group，它们被称为Participant。与Percolator类似，Spanner会选出其中一个作为Coordinator，引领整个2PC过程；其Leader被称为Coordinator Leader。下图有更完整的结构



下面把Spanner读写事务的流程详细列出来

1. Client通过Would-Wait获取读锁，锁记录在Paxos Group的Leader结点的Lock Table中。
2. Client读取数据。进行计算，写数据缓存在Client中。
3. Client开始2PC过程。首先，Client选择Coordinator。
4. Client将Coordinator身份和缓存的写数据发送至所有Participant Leader。
5. 非Coordinator的Participant Leader进行2PC Prepare。获取写锁，选择Prepare Timestamp，将Prepare Record通过Paxos持久化，最后通知Coordinator。
6. Coordinator Leader获取写锁，选择Commit Timestamp，将Commit Record通过Paxos持久化。
7. Coordinator Leader等待Commit Wait。为利用等待时间，并行进行Paxos通信。



[Cloud Spanner: Life of Reads & Writes]  
(<https://cloud.google.com/spanner/docs/whitepapers/life-of-reads-and-writes?hl=zh-cn>)

知乎专栏“分布式和存储的那些事”有文章对Spanner进行更深入的解读

[hellocode - Spanner十问](<https://zhuanlan.zhihu.com/p/47870235>)

## 2PC Locking的可用性问题

2PC事务最常被质疑的问题在于可用性；如果一个Participant没有响应，会导致整个事务卡住。

Spanner 如何解决这个问题呢？首先 Participant 是 Paxos Group，响应高可用（Availability），数据高可靠（Reliability）。其次，Coordinator和Participant之间可以互相使用租约和心跳，确保2PC过程的Liveness（TLA+部分提到的概念）。

从前文到此，我们已经解决了分布式事务设计中所遇到的大部分问题。

## 下一步

我们已经看过了Percolator、Spanner的分布式事务设计。对于分布式事务，还有更多的巧妙设计和不同策略。下篇文章中，我们将在抽象层次上更上一层楼，观察各种设计策略和不同取舍所形成的技术光谱（Spectrum），它将带我们到分布式事务的本质。

（注：本文为个人观点总结，作者工作于微软）