分布式系统-分布式事务（P3完）

Accela推箱子 Accela推箱子 今天

（续前文……）
分布式系统-分布式事务（P1）
分布式系统-分布式事务（P2）

前文讲述了分布式事务的原理，如事务隔离级别，Percolator、Spanner的不同事务方案，以及形式化语言TLA+等。这篇文章旨在从抽象层次上，更进一步思考构建分布式事务的过程。

正确性-事务隔离级别

除去所有额外的并发控制，事务可以看作是对一组数据集的一连串读和写。多个事务交错执行，有些执行顺序是我们认可的，有些执行顺序是我们认为错误的。

如何区分对与错？这样就建立了P1文章中的事务隔离级别模型。当把模型进一步细化，观察各种事务读写的异常执行顺序，我们又可以建立读写冲突等概念。

接下来的问题是，如何实际控制事务按我们预期的顺序执行？

事务并发控制-目标

首先，可以预想的是多个事务并行执行时，使顺序完全依照理想模型。但实现往往有性能和复杂度的考虑，而业务需求也未必要求完全。

于是，首先，"劣化"的模型可以引入，可以命名为各种隔离级别。其次，算法可以为了简单高效，拒绝掉比模型所要求的更多或更少的事务执行顺序；这就引出了P1文章中，Serializable和SSI的区分。

事务并发控制-方法

在代码执行时，我们要求事务的读写依照预期顺序执行，这可以看作调度问题。第一种方法，与P2文章中的Percolator和Spanner不同，最简单直接的是**中心调度**。

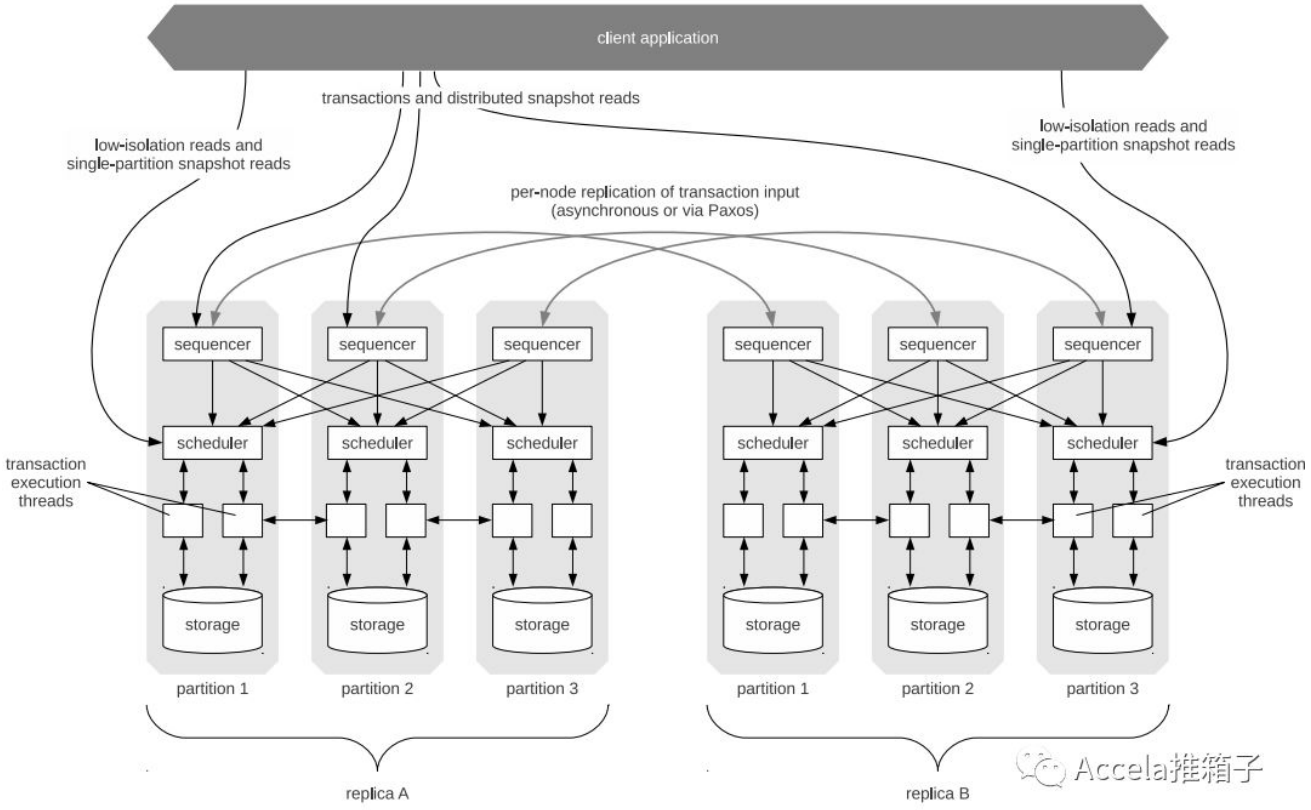
如下面Critique SI论文，文章提出中心Status Oracle服务器，知晓全部事务信息，也对所有事务排序。单节点内的调度，可以为所欲为。

[A Critique of Snapshot Isolation](https://dl.acm.org/citation.cfm?id=2168853)

经过合理的计算，即使是拥有大量数据的分布式系统，事务信息也可以足够小到存入单台服务器。如果Status Oracle故障，退出所有运行中事务，无状态地快速重启，也是可接受的。

另一篇论文是Calvin Transaction。事务调度器是分布式的，但每个调度器（文中Scheduler）知晓所有自己参与的事务的状态（由Sequencer送进来），作出完全一样的调度（Deterministic）。本质上也是中心式调度（或曰共享状态），多个调度器间还能互相避免单点。当然，前提是全部事务信息足够放进调度器。

[Calvin: Fast Distributed Transactions for PartitionedDatabase Systems](http://cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf)



中心调度器还可以这样做，为每个事务的读写操作分配一个（逻辑的）时间戳，从而预订它们的先后执行顺序。这里也能看到时间戳的重要性渐渐浮出水面。不过在如今普及的多版本系统中，时间戳已经是离不开的了。

事务并发控制-方法-锁

与中心调度相反，我们也可以希望事务各自独立沟通，协调出符合预期的执行顺序。这样我们就来到了"锁"的世界。

本质上，锁是在为我们交换信息。所以能看见Serializable Snapshot Isolation论文中，用锁去记录和传递冲突结构的T.inConflict和T.outConflict；在Percolator中，锁只是记录在行中的一列，被相关事务查询；Spanner中也有类似的Lock Table。

（当然，和通常意义的锁一样，系统还需要能够一致性地读取"锁"的状态，以及原子性地"检查-上锁"的方法。）

接下来的问题是，当遇到锁冲突时，相关的两个事务应该做什么？可以选择阻塞等待，这是传统的"悲观"方式。可以偏爱更年轻的事务，夺取老事务的锁使其失败，如Percolator.tla中的读操作。也可以偏爱老事务，让年轻事务退出。或者更复杂的，用Wound-Wait、Wait-Die等方法。或者先计算，直到最后才加锁，从而"乐观"。

然后，在事务的什么地方加锁呢？传统的Serializable，Strict-2PC，选择既加读锁也加写锁，这样排除掉了读-写冲突、写-写冲突，也带来更高的性能代价。而Snapshot Isolation标称性能快，在于读可以不加锁，而写加锁排除写-写冲突，不过这样带来了Write Skew问题。

有特殊用途的事务

从上文可以看到，分布式事务设计没有范本。分析清楚各方需要后，可以任意添配料，组合不同策略，装配出合适自己场景的方案。当然，鉴于实现难度，更流行的方法还是工业复用，从而需要建立标准，从而有了流行于世的各种事务的"知识"。

根据具体场景裁剪，是个常用的系统优化思路。比如，分布式文件系统是需要分布式事务的一大场景，而需求则跟数据库事务完全不同。

例如HopFS分布式文件系统的论文，事务用于处理文件系统元数据，主要对象是文件和目录的Inode。如何根据用户使用合理规划Partition，使事务总需要尽量少的服务器参与，是一大重点；与之相关的还有如何使Cache策略更有效。POSIX接口为事务带来较大难度的是move、delete等大规模数据操作，论文中为它们做了详尽的优化。

[HopsFS: Scaling Hierarchical File System Metadata UsingNewSQL Databases] (https://www.usenix.org/system/files/conference/fast17/fast17-niazi.pdf)

当然，如果与分布式事务缠斗腻了，也可以"弃疗"。可以弱化POSIX接口规范以方便实现分布式，如HDFS；如果只需要简单的目录结果，存储图片视频等，可以选择不需要事务的对象存储；而需要强一致的则可以使用数据库。

那么，有更加Think-out-of-box跳出局限的思路吗？

最终一致性事务

可以参考下文的"最终一致性事务和补偿"。这可能才是众多互联网公司采用的事务方案。

[Eventual consistency transaction + compensations] (https://developer.jboss.org/wiki/CompensatingTransactionsWhenACIDIsTooMuch)

试想一个订票网站，用户订票必须事务地成功出票且付费，或者完全地失败且不扣钱。订票网站会由诸多子系统组成，除了订票、付费外，还有积分等。许多系统可能需要对接外部，例如获取票务信息，从而有不可忽略的延迟。

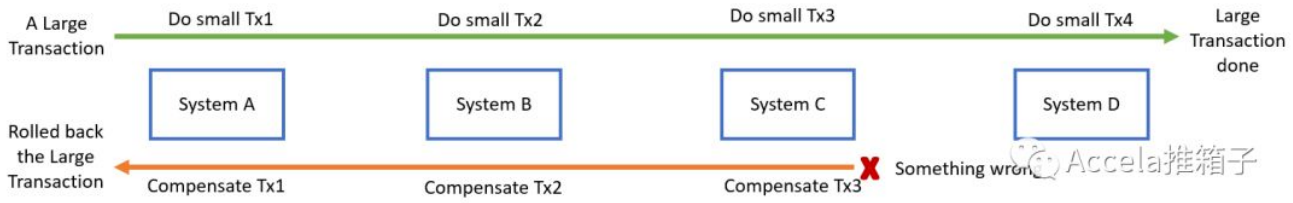
如何为这样的一个大系统实现ACID的分布式事务呢？当然是不实现。更实际的解决方案是最终一致性事务

1）每个子系统支持自己内部的ACID事务（它们可以自由地使用数据库）

2）全局大事务由各个子系统的ACID小事务组成。

3）大事务执行时，逐个执行其ACID小事务，直到完成。

4）事务何时完成是未知的，但事务的传播方向是已知的、既定的。这就成了最终一致性。



问题在于，如果执行序列中某一步遇到故障无法继续前进，应该如何像ACID事务一样完整地Rollback回滚？解决方案是引入补偿事务（Compensation）

1）补偿事务是和上文小事务对应的，可以被执行以完全Rollback一个小事务。

2）当大事务Rollback时，按相反顺序逐个执行补偿事务，直到回到原点。

当然，最终一致性事务的中枢是事务信息的传播；它常常由消息队列来承担。其中就可以引入消息执行幂等性，消息传播至少一次，消息发送和小事务绑进一个Transaction，等等优化了。

（看到这里，也可以理解很多订票订旅馆应用，需要点击确认后等待一段时间告知是否预订成功。）

总结

将分布式事务层层分解，复现如斯的现代魔法，重寻历久的符号、新创的技法，人类最闪耀的思维在时空中雕刻。

假以时日，魔法被瓶装出售，奇迹被封装复用。这是最好的时代，也是最疯狂的时代。

（全文完。注：本文为个人观点总结，作者工作于微软）