分布式系统-分布式事务（P1）

OriginalAccela推箱子 Accela推箱子 9月24日

分布式事务理论古老，实现不易。谷歌发布的Percolator、Spanner大型数据系统，让分布式事务有了经典实现。如今开源分布式数据库TiDB、CockroachDB，更让其有了平民版。TLA+形式化语言（和TiDB开发组所写的Percolator.tla），让复杂的分布式事务协议，有了透明严谨的范本。

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

云服务数据库常需考虑分布式事务，分布式文件系统的元数据管理也需事务支持，互联网业订单库存等管理也常需分布式事务及其变种。

事务模型、读写依赖、Serializable

~~（直接介绍知识写腻子）~~换种方式，假如分布式事务是个新课题，如何从零到有建立概念研究它？

首先做一个标杆，把最理想的边界定出来

- **Serializable（可序列化）**：在分布式系统中，多个事务中的多个读写操作是混合在一起执行的。Serializable即，最终的执行结果，和这些事务由单线程一个个执行一样。它是最高的事务隔离级别（Isolation Level）。

这样就划归到已知的单机事务了。这也意味着，同时发起多个事务，满足Serializable的最终调度结果可以有多种。

当然，“调度”二字预示分布式系统得以某种方式作协调。协调需要共享状态，那么一种路径，用“锁“来制造共享协调；另一种路径，用中心协调服务器（Transaction Manager）；后文可见对应。如果单机，Lock-free算法、CAS操作也可用上。

接下来，给事务一个简化的描述

- **事务（Ttransaction、Tx）**：在事务期间，可以读取数据和写入数据。通常先读取数据，作出判断和计算，最后写入（即使不是，也可以被调度重新排列如此）。

此时有问题：如果事务多次读取同一Key，能否看到同一个值？更贴近Serializable的答案应该是“能”。于是我们有了

- **Repeatable Read（可重复读）**：事务隔离级别之一，事务内多次读取同一Key返回相同的值。此外，这个隔离界别还要求不能读到未提交（uncommitted）的值。通常的实现方法是，把事务读过的值放到事务缓存中以备再用。

问了同一Key的重复，再问多个Key之间的“重复”。Key1、Key2的值在不断变化中；如果事务读了Key1，再读了Key2，它们返回的是Key1、Key2在同一时刻的值吗？更贴近Serializable的答案是“是”。这就引出了

- **Snapshot Isolation（快照隔离）**：比Repeatable Read更高的事务隔离级别。事务开始时仿佛给数据库打了某一时刻的快照。读取Key1、Key2时，返回的都是那个快照时刻的值。此外，Snapshot Isolation还禁止写-写冲突（write-write conflict、ww-conflict），后文讲。实现方法上，自然少不了时间戳（Timestamp）、多版本之类。

话说最早提出Snapshot Isolation的是一篇微软主创的论文：

[Critique ANSI isolation levels](https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf)

前文讨论完事务的简化描述，下面自然是试图描述事务间的关系。想象事务Tx1和Tx2，读写×读写，有四种组合。首先，读-读有什么关系，没什么关系，读读可交换。

写-写有什么关系？

- **写-写依赖（ww-dependencies）/写-写冲突**：Tx1和Tx2都写入同一个Key，比如Tx2写入的版本覆盖了Tx1的。结果上看，呈现出Tx2执行在Tx1后。“依赖”之名，即来自于这里呈现出结果上的先后顺序。如果Tx1和Tx2的生命周期（Transaction span）互相重叠，那它们就造成了写-写冲突；Snapshot Isolation禁止这种冲突。

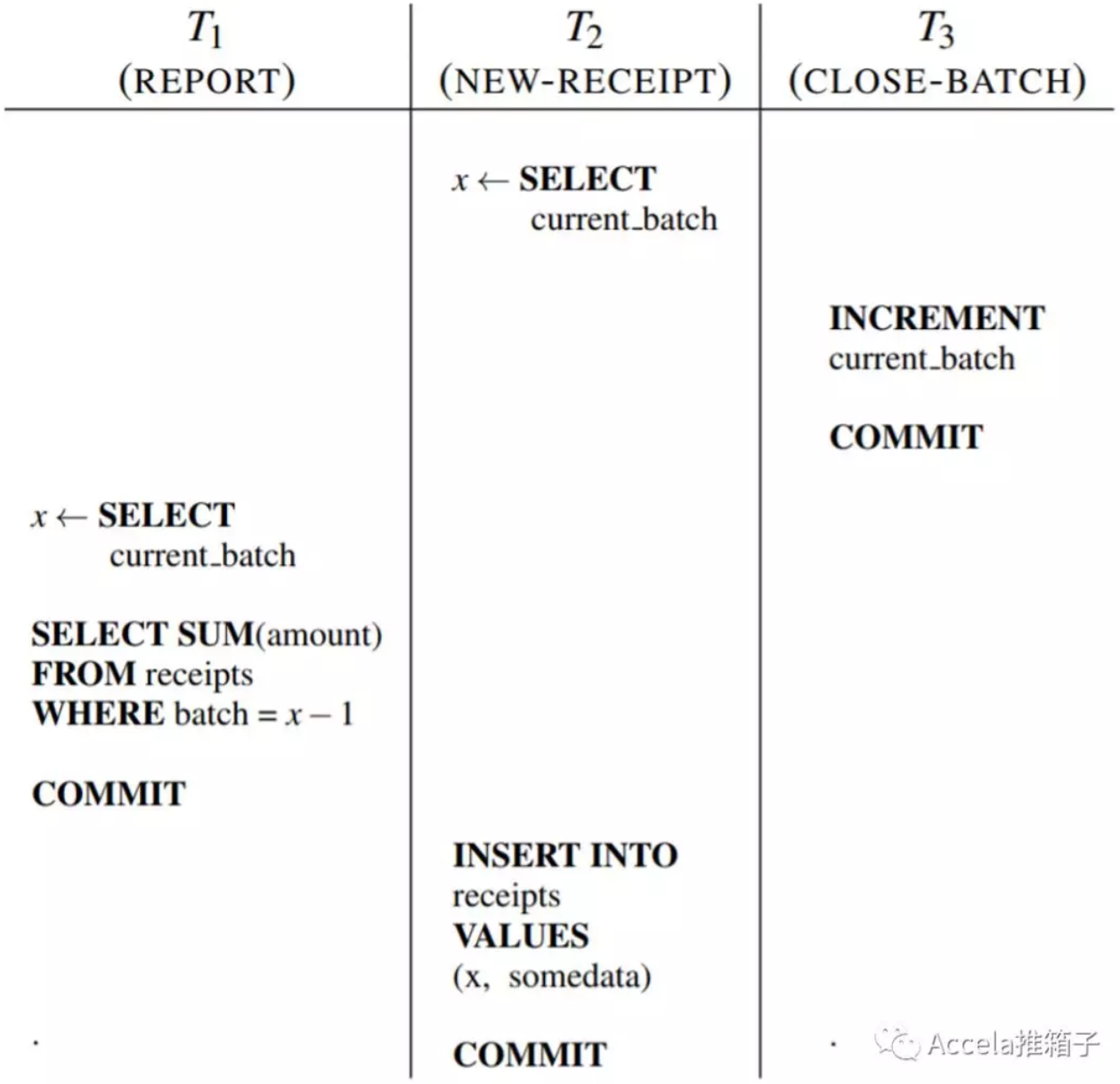
然后，读-写有什么关系？有两种，读-写和写-读

- **写-读依赖（wr-dependency）**：Tx1向Key写入某版本的数据，Tx2读取了Key的这个版本的数据。结果上看，呈现出Tx2执行于Tx1之后。之所以说“结果”、“呈现”，是因为事务读写实际执行时，可能是任何顺序的。

- **读-写反依赖（rw-antidependencies）/读-写冲突（rw-conflict）**：Tx1向Key写了某版本数据，但Tx2读的是之前版本的值。结果上看，呈现出Tx1执行于Tx2之后，因为Tx2没有看到Tx1对Key值的更新。这里隐含了Tx1和Tx2的生命周期是互相重叠的。这种依赖也被称为读-写冲突。（“反（anti）”是论文中给出的名称，应该是为了表明依赖箭头方向和其它几个不同）

现在我们有了事务间关系的描述，也有了对事务间读写冲突的解读。PostgreSQL SSI论文对其有详细的介绍。下面的例子也来自该论文中。

[Serializable Snapshot Isolation in PostgreSQL](https://www.drkp.net/papers/ssi-vldb12.pdf)

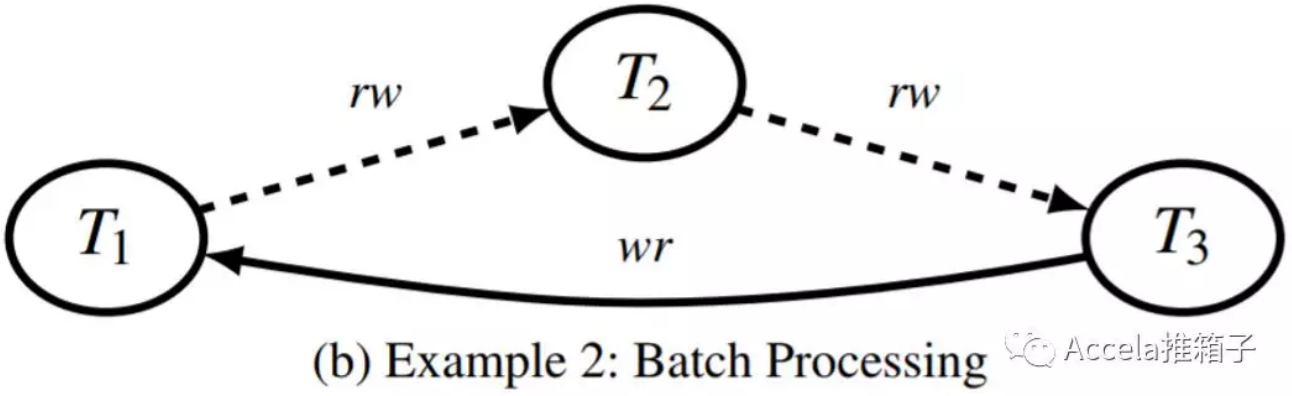


例子中，三个事务T1、T2、T3的生命周期互相重叠，从上到下可见其读写操作的交错顺序。T1、T2、T3之间的依赖关系在下图中绘出；与依赖关系一致，有向边反映的是事务执行结果所呈现的先后顺序。

1）T2到T3的箭头：T3中途修改了T2读出的current.batch，呈现出T3执行于T2之后。这是读-写反依赖。

2）T3到T1的箭头：T3所写current_batch值被T1读取，呈现出T1执行于T3之后。这是写-读依赖。

3）T1到T2的箭头：T1所读receipts，被T2在之后写入修改，呈现出T2执行于T1之后。这是读-写反依赖。



(b) Example 2: Batch Processing

可以看见，上图中的依赖关系中出现了环。这就是说，从不同角度，可以解读出互相矛盾的T1、T2、T3执行顺序。即，上面的事务调度结果，不满足Serializable。这里也引出了事务序列化图的概念

- **事务序列化图（Serialization Graph）**：事务作节点，依赖关系作有向边，绘成图；边反映的，即是依赖关系，也是结果所呈现的事务先后执行顺序。事务调度是Serializable的，等价于，事务序列化图无环；换句话说，没有环，那么事务呈现出一个一个按某种顺序执行的结果。

再看事务隔离级别

有了事务的模型，和研究事务关系的工具，我们可以进一步看各个事务隔离级别之间的关系。第一个问题是，Snapshot Isolation满足Serializable吗？

- **Snapshot Isolation不满足Serializable；其违反Serializable的特例被称作Write Skew**。Snapshot Isolation要求1）从Snapshot读2）禁止写-写冲突。“Write Skew”得名，大概是因为它已被反复研究。从下面的Write Skew例子，可以看到Snaphost Isolation是如何违反Serializable的。

Write Skew的例子来自Wikipedia。

大体构造是，Tx1和Tx2在同一组数据上读写，读区间重叠，写到这组数据的不同地方（所以没有写-写冲突）。Tx1、Tx2的读写操作交错执行，假如Tx2首先完成写入，那么Tx1曾读的数据就被改写。也就是说，Tx1用作判断和计算的基础的数据，已经过时了；当Tx1写入时，其所持有的计算结果已经是错误的了。最终，Tx1把错误数据写入，卒。

[Wiki: Snapshot Isolation](https://en.wikipedia.org/wiki/Snapshot_isolation)

> In a *write skew* anomaly, two transactions (T1 and T2) concurrently read an overlapping data set (e.g. values V1 and V2), concurrently make disjoint updates (e.g. T1 updates V1, T2 updates V2), and finally concurrently commit, neither having seen the update performed by the other. Were the system serializable, such an anomaly would be impossible, as either T1 or T2 would have to occur "first", and be visible to the other. In contrast, snapshot isolation permits write skew anomalies.
>
> As a concrete example, imagine V1 and V2 are two balances held by a single person, Phil. The bank will allow either V1 or V2 to run a deficit, provided the total held in both is never negative (i.e. V1 + V2 ≥ 0). Both balances are currently $100. Phil initiates two transactions concurrently, T1 withdrawing $200 from V1, and T2 withdrawing $200 from V2.
>
> If the database guaranteed serializable transactions, the simplest way of coding T1 is to deduct $200 from V1, and then verify that V1 + V2 ≥ 0 still holds, aborting if not. T2 similarly deducts $200 from V2 and then verifies V1 + V2 ≥ 0. Since the transactions must serialize, either T1 happens first, leaving V1 = -$100, V2 = $100, and preventing T2 from succeeding (since V1 + (V2 - $200) is now -$200), or T2 happens first and similarly prevents T1 from committing.
>
> If the database is under snapshot isolation(MVCC), however, T1 and T2 operate on private snapshots of the database: each deducts $200 from an account, and then verifies that the new total is zero, using the other account value that exists in the snapshot was taken. Since neither *update* conflicts, both commit successfully, leaving V1 = V2 = -$100, and V1 + V2 = -$200.

那么，如何构造满足Serializable的事务读写执行顺序呢？也即问如何调度（Schedule）。

首先，最为暴力也最为简单的方式：禁止一切有依赖的事务被同时执行，这样就不用担心事务序列化图里有环了，因为没有边了：

- **Serializable实现方法，禁止读-写冲突，禁止写-写冲突**：如果两个事务的生命周期重叠，并且它们所读、写的数据集有重叠，那么其中一个必须等待或者终止（abort）。实现方式上，常常是对读、写数据全部加锁；这也是数据库实现Serializable的经典方法。

从这里，和Snapshot Isolation进行比较可以发现，Snapshot Isolation只需禁止写-写冲突，于是

- **Snapshot Isolation优点-只需写锁**：因为只需要禁止写-写冲突，我们只需写锁，不需读锁。于是有了Snapshot Isolation更好的读性能；这是Snapshot Isolation性能

优势的本质。“锁”也可以是其它的同步控制方法，总之，省掉了对读约束带来的开销。

接下来的问题，Serializable有更好的实现方法吗？

基本的分析方法是，比较Serializable所允许的全部事务序列化图，与我们的方法所允许的全部事务序列化图。

1）如果后者比前者多，说明我们将错误的调度当成了Serializable。

2）如果后者比前者少，说明我们拒绝掉了多余的调度，造成了多余的事务阻塞或终止；那么就存在改进空间。

对于更好的Serializable实现方法，Critique SI论文给出了进一步的分析

[A Critique of Snapshot Isolation](https://drive.google.com/file/d/0B9GCVTp_FHJIMjJ2U2t6aGpHLTFUVHFnMTRUbnBwc2pLa1RN/edit)

- **禁止写-写冲突，不是Serializable的必要条件**。论文中有证明。现在可以发现，写-写冲突虽然作为Snapshot Isolation的核心，然而它不是一个好的选择。

那么读-写冲突呢？

- **禁止读-写冲突，是Serializable的充分条件**。论文中有证明。换句话说，如果事务Tx1从读到写之间，没有被另一事务Tx2中途修改曾读的数据，那么Tx1和Tx2就是Serializable了；这符合直觉。实现方法上，常可见事务在提交前检查，自己所读数据上有没有时间戳更新的写（No any newer writes）。

那么禁止读-写冲突，就可以实现Serializable。我们还可以再进一步问，禁止读-写冲突，是Serializable的必要条件吗？

答案是否；还有更加精细的Serializable实现方法——Serializable Snapshot Isolation；它由下面论文提出。而前面提到的PostgreSQL SSI论文有其在PostgreSQL中的实现，对理论的讲解较易懂。

[Serializable Isolation for Snapshot Databases](https://courses.cs.washington.edu/courses/cse444/08au/544M/READING-LIST/fekete-sigmod2008.pdf)

- **SerializableSnapshot Isolation（SSI）的关键定理**：如果事务序列化图中有环，那么必定存在至少两个连续的读-写反依赖边。例如前文图“Example2: Batch Processing”。这种结构在论文中被称作“危险结构（Dangerous structure）”，它比禁止所有读-写冲突更加精确，用来检查Serializable。

Theorem 1 (Fekete et al. [10]). *Every cycle in the serialization history graph contains a sequence of edges* $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ *where each edge is a rw-antidependency. Furthermore,* T_3 *must be the first transaction in the cycle to commit.*

关于实现，SSI论文中构造了一种新的锁SIREAD，本质上用于薄记依赖事务的信息，配合WRITE锁来检测违反Serializable的事务，将它们阻塞或终止。

那么，Serializable Snapshot Isolation是Serializable的必要条件吗？我们可以继续问下去……

实际上，如何更高效精细的实现Serializable仍是研究中的课题。而如何在分布式系统中有效地实现它们，也是快速发展、很有挑战的领域。

关于ACID

前文从零到有搭建的事务概念，主要从隔离级别入手。有了它，我们就有了解释事务ACID概念的核心

- **A-Atomic-原子性**：一个事务，要么全部执行，要么全部不执行。事务的全部执行和全部回退，可由Journaling或Logging实现；经典算法有Write-ahead Logging和ARIES。事务执行中途，临时结果不应对其它事务可见；这和隔离性息息相关，通常由事务自己维护的读写缓存、快照实现。

- **C-Consistency-一致性**：隔离性想要接近的Serializable，事务并发执行结果如同一个个单线程执行，其本意也是在维护事务的一致性了。一致性体现在，那些单线程一

个个执行事务时所能维持的约束，并发执行时不应被打破。

- **I-Isolation-隔离性**：即前文所讲。它和其它ACID属性的内涵是互相重叠的。
- **D-Durability-持久性**：事务提交时应保证Journal、Log、数据等的落盘。数据可以通过副本，Paxos复制，备份，存储编码等方法加强可靠性。

可以看见，ACID概念大部分围绕着事务的隔离性。

下一步

前文搭建了理解分析事务的框架，这些理论在单机和分布式中是通用的。这些都是前菜。

后面系列中，我们将通过Percolator、Spanner，展示分布式事务实现的精妙。我们将看到TLA+形式化语言的强大作用。我们将看到不同策略取舍所带来的不同分布式事务设计。

（未完待续…… 注：本文为个人观点总结，作者工作于微软）