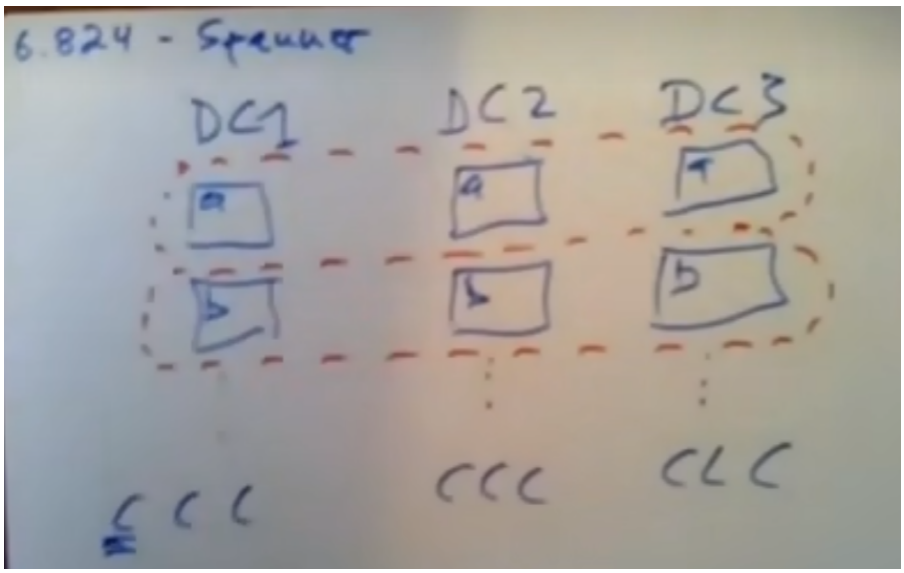
LECTURE 13 SPANNER

1. 前置知识

- 外部一致性：A事务提交之后，在A事务提交之后的B事务能够看到A事务造成的任何修改。可以认为External Consistency = Linearizability + Serializability
    - External Consistency 比 Serializability 更严格，知乎收藏
    - 注意事务ACID的一致性和分布式系统的一致性不是一回事！事务一致性指的是事务没有违反相关的约束条件
- CAP理论：不可能存在分布式系统同时满足：Consistency、Availability 和Partition Tolerance，最多只能满足两者组合
- Spanner主要应用于Geographic replication（地理上相隔较远的复制），考虑的是整个数据中心崩溃的问题。对于Spanner而言，性能的瓶颈主要在于网络延时。

2. 基本结构



spanner 可以被分成两层，外层为Lecture 12 中提到的2PC ，内层为Paxos算法。在上图中：

- 红色圈圈为一个Paxos group，在上面运行复制状态机。Paxos group的各个结点都位于不同的数据中心，原因包括：
    - 用于容灾，避免地震、所在城市停电等因素对整个数据中心造成的影响
    - 允许client就近访问数据 。但，Paxos和Raft都允许慢结点的存在，因此client就近访问数据可能会导致client访问慢节点，从而访问stale的数据。（注意，原生Raft只允许leader

处理所有的请求，但是可以改装Raft使得所有的结点均能处理读请求）

- 一个paxos group作为2PC算法的一个结点
- 当一个Paxos Group被选作Coordinator时，该Paxos Group的Leader称为Coordinator Leader，Slave称为 Coordinator Slave。同时，Coordinator相关的工作都由Coordinator Leader完成，Coordinator Slave仅仅是提供容错容灾的复制状态机而已
- **锁维护**：spanner中仅由Paxos Group的Leader来维护锁（以及事务管理器），论文中结构图如下：

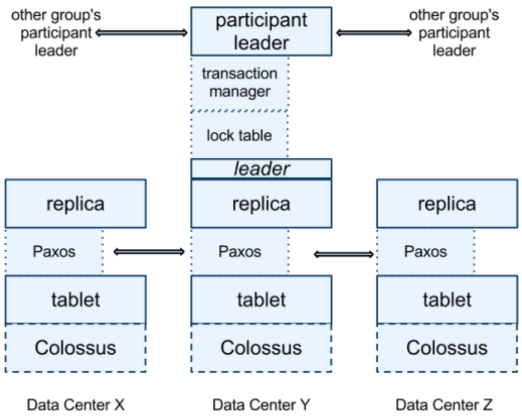
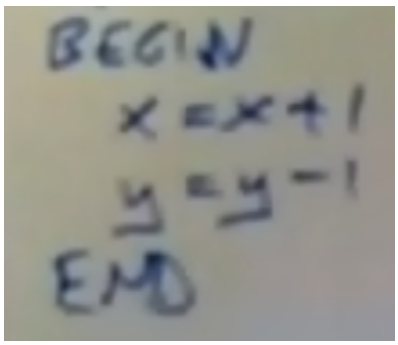


Figure 2: Spanserver software stack.

3. 事务执行

3.1. R/W Xaction

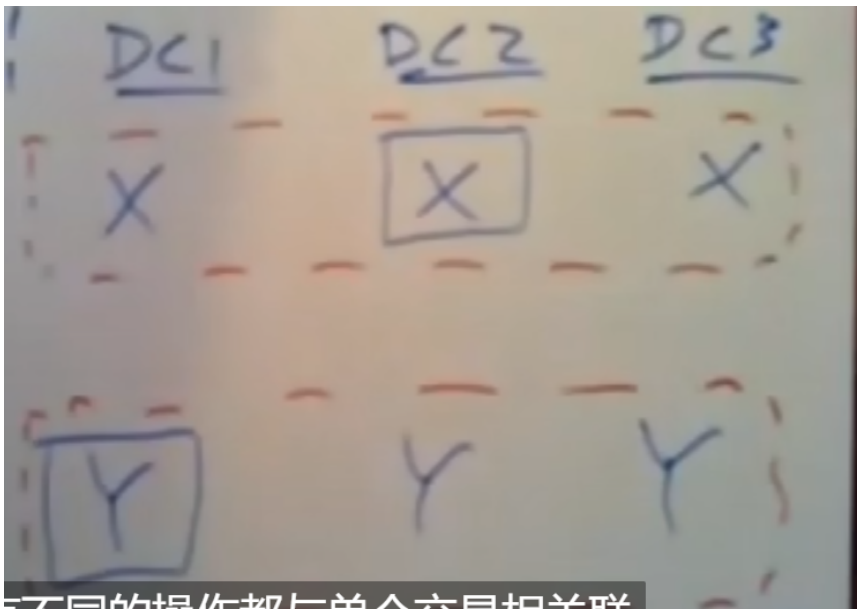
- 事务如下：



尽管事务编码如上图所示，但事务在进行执行时需要重新构建顺序。在上图事务中，必须先读取x, y，而后再写入x, y，而非先读x，写x，再读y，写y

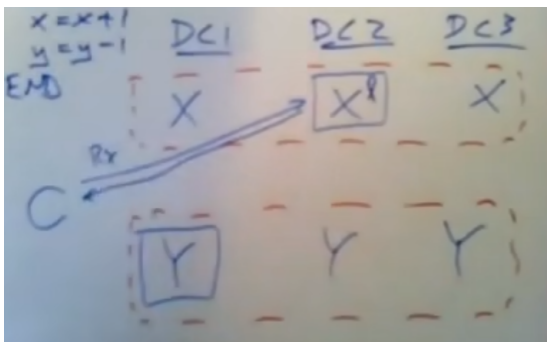
- 执行流程：
  - 假设每个paxos group的leader为蓝框所示：

为方便起见，分别吧x, y在DC1，DC2，DC3中 所在的机器（是结点/机器，而不是数据中心！！！）称为 $x_1, x_2, x_3, y_1, y_2, y_3$



1. client读取x：client先向x所在的Paxos Group的Leader发送读请求。DC2向client返回数据，并且Leader为x上锁

   当无法获得锁时，则client阻塞



2. client读取y：同上，读操作完成

3. client选择一个Paxos Group作为coordinator，假设是y所在的Paxos Group作为coordinator，假设coordinator leader

4. 2PC 的Phase 1：

   下面的操作不需要顺序

   - client向x所在Paxos Group的Leader（位于DC2，为x_2）发送W_x请求（即**prepare** 请求），而后x_2作为Paxos Leader，给x_1和x_2的机器发送**prepared** 消息（用于恢复），在收到了大多数机器的ack消息之后，向coordinator（y_1）发送prepared message（里面会包含一个由该Participant Leader 选择的时间戳）

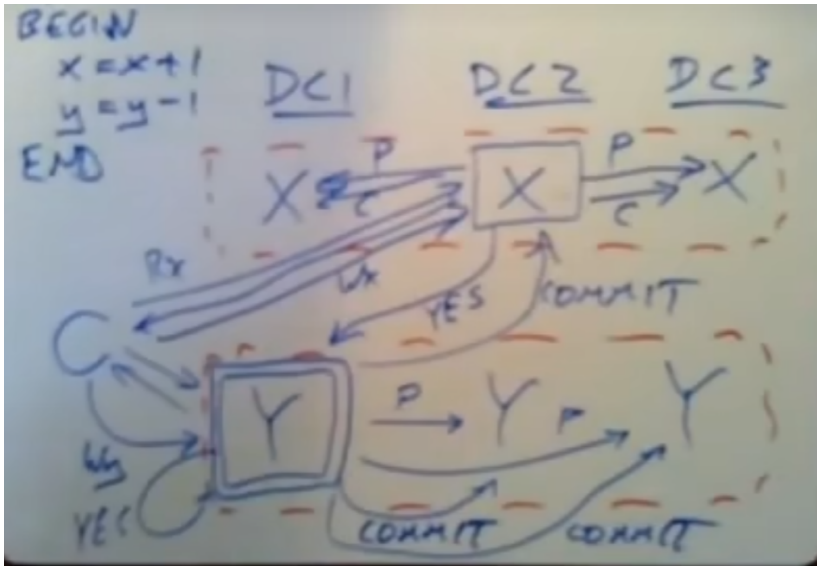
- client向y所在Paxos Group的Leader （位于DC1，y_1） 发送W_y请求（Prepare），而后同上。但在收到了大多数机器的ack消息之后，作为Paxos Leader的y_1要向作为Coordinator的$y1$发送prepared message（注意，这个行为应该是逻辑上如此，论文中说**Coordinator Leader会跳过Phase 1!**）

5. 2PC 的Phase2：

   注意下面的操作必须有严格顺序

   1. Coordinator Leader（y_1）向Coordinator Slave（y_2, y_3）发送commit 消息，将其进行复制用于恢复

   2. Coordinator Leader向其他 Paxos Leader发送commit消息。Paxos Leader收到commit消息之后，将commit消息 发送给其他Slave，进行复制用于恢复。而后Paxos Leader执行写操作，并且释放锁。



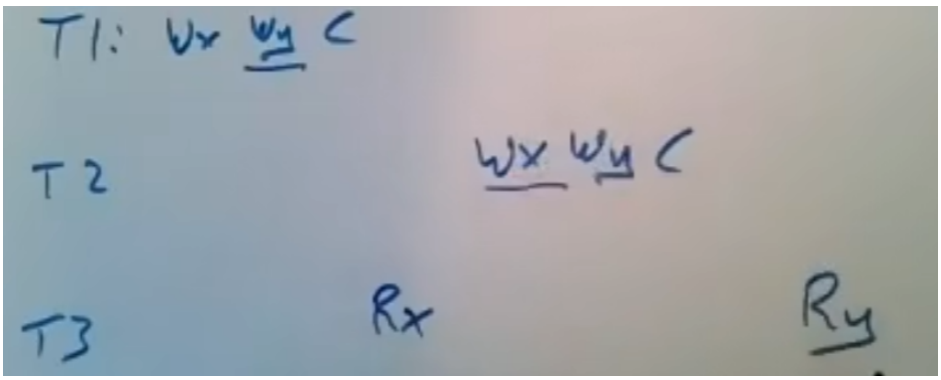
- 在普通的两段提交协议中，当coordinator故障时，则与该事务相关的所有数据都不能够被读写，会极大程度上影响其他的事务，必须等待coordinator恢复。在spanner中，通过Paxos复制了**事务管理器（Xaction Manager）**，任何一个Coordinator Slave可以在Coordinator Leader 故障时接管事务

3.2. R/O Xaction 只读事务

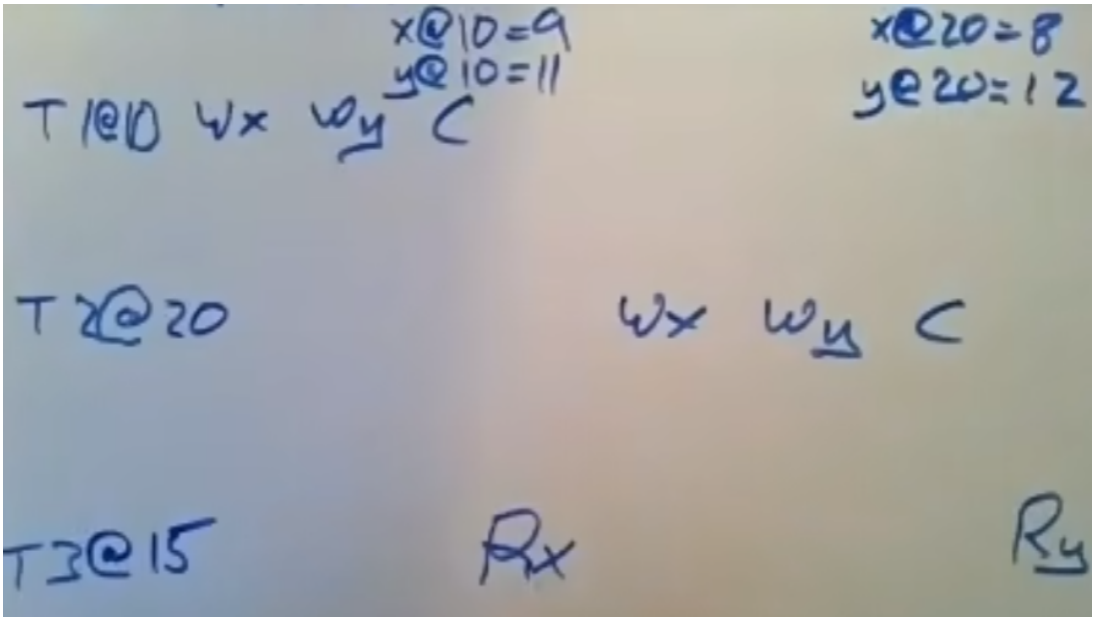
- 必须要显式声明
- 可以从最近的Paxos Replica 读取数据，但存在读取stale数据的问题2
- 不使用锁机制、Xaction Manager 和2PC协议，使用**快照隔离（Snapshot Isolation）** 技术：

- **时间戳（timestamp）**：需要分R/O Xaction 和 R/W Xaction讨论，所有的事务都有时间戳
  - R/W Xaction：在事务提交时为每个数据打上时间戳
  - R/O Xaction： 在事务开始时，选择一个时间戳。如果一个R/O Xaction 仅访问了同一个Paxos Group的数据，则只需要将时间戳设定为**该Paxos Group Leader 的最后写入数据的timestamp即可**；若跨多个Paxos Group，则设为$TT.Now().lastest$
- 向 **最近** 的Paxos Replica发送读取请求，比较该Replica **已提交** 日志中最后一条的时间戳：
  - 如果事务的时间戳大于Replica 的时间戳，则等待（说明该Replica 较慢或者出过故障，还没有跟上majority）
  - 如果事务的时间戳小于等于 Replica的时间戳，则返回最新的数据（指时间戳小于事务时间戳的最新数据）

- 例子：



上图并发事务，如果不使用快照隔离，则无法保证外部一致性。（不存在串行化序列，更不可能是外部一致的）



上图使用了快照隔离，则读取y时，由于$y = 12$这一数据的时间戳为20，比T_3开始的时间戳大，因此只会选择时间戳为10的数据$y = 11$。之所以读到过时的y是正确的，是因为$T2, T3$本质上是并发事务，因此只要结果与它们的任意串行序列一致则都是正确的

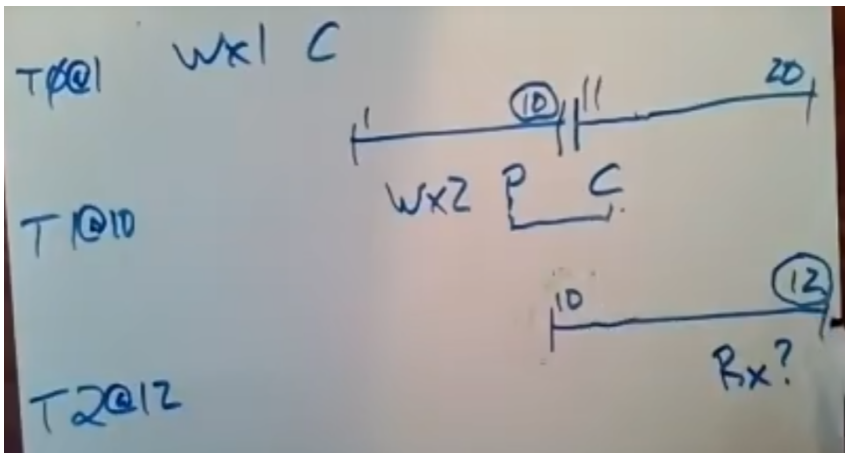
3.3. TrueTime

- 基本结构：每个数据中心有一个GPS Receive Master（显然需要通过共识算法备份），各个Server会定期向GPS Receive Master校准当前时间。
- API：
  - TT.Now()，返回一个范围$[earliest, lastest]$，由于真实时间的传输处理等过程一定存在耗时，因此GPS Receive Master不能提供确切的答案，但正确的时间一定在这个范围之内
  - TT.after()
  - TT.Before()
- **单调不变量（monotonicity invariant）**：Spanner给提交的R/W事务打上的时间戳一定是随着时间单调递增的
- 为保证外部一致性而必须遵守的规则
  - Start：
    - R/O事务在开始时，选取的时间戳为TT.Now().lastest
    - R/W事务在提交时，选取的时间戳为TT.Now().lastest

- Commit Wait：R/W事务必须保证当前时间已经过了选取的时间戳之后才提交。也即：

  假设事务选取的时间戳为t，则必须满足：$t < TT.Now.earliest$ 才提交，否则就一直等待。在实际的R/W中，由于需要使用2PC协议，因此在Phase 2开始之前，Coordinator Leader就已经选取了提交的时间戳，而后等满足上述条件之后向各个Participant Leader发送Commit Message 用于提交。

- 例子：



  T2 在P点调用了TT.Now()，准备提交。此时TT.Now()返回了$[1, 10]$，因此T2 选择10作为提交时间。而后T2循环调用$TT.Now()$。当$TT.Now().earliest > 10$时（也即$earliest = 11$时）开始提交。

  随后T2读取，T2是R/O事务。T2调用TT.Now()后选择12作为本只读事务开始的时间戳。而后T2读取，则能够读取到T1修改的数据

4. Schema Change Xaction 模式变化事务

- 实现：

  1. 选取一个未来的时间戳t

  2. 对于所有的读写操作，若其时间戳大于t则必须阻塞，反之则可以进行

- 论文中没有指出如何选取时间戳t?

5. 一些设计细节

- 存储空间问题：Spanner存储了一个数据的多个版本，因此可能会导致数据过多。然而Spanner不需要记住太过古老的版本。若能保证大多数事务在某个时间t之内均能完成，则Spanner只需要保存t秒之内的数据即可

6.

7. 相关论文

- Snapshot Isolation：6
- R/O Xaction： 8
- 防止读操作产生死锁/ wound wait