

分布式数据库, Spanner I

TrueTime

Method	Returns
TT.now()	TTinterval: [earliest, latest]
TT.after(t)	true if t has definitely passed
TT.before(t)	true if t has definitely no arrived

Table: TrueTime API. The argument t is of type TTstamp.

注意的地方:

- The `TT.now()` method returns a *TTinterval* that is **guaranteed** to contain the absolute time during which `TT.now()` was invoked.

分布式数据库, Spanner II

Spanner 特性:

- Read-Write Transaction (读写事务)
- Read-Only Transaction (只读事务)

Paxos Leader Leases

Spanner 通过 timed leased (租约) 实现长时间的 Leader, 对比 Raft 协议的 timeout

- upon receiving a quorum of lease votes the leader knows it has a lease. (租约获得)
- A replica extends its lease votes implicitly on a successful write. (默认成功写入, 大部分支持, 就续约)
- the leader requests lease-vote extensions if they are near expiration. (leader 再次续写租约)

分布式数据库, Spanner III

Paxos Leader Leases 续

- **lease interval (租约区间)**
 - starting when it discovers it has a quorum of lease votes.
 - ending when it no longer has a quorum of lease votes.

Paxos Leader Leases 续

disjointness invariant(不相交不变式): **for each Paxos group, each Paxos leader's lease interval is disjoint(不相交) from every other leader's .(对于每个 Paxos group 中, 每个 leader's 租约时间是不相交的, 提供服务的时间是不能相交的。)**

The spanner implement permits a Paxos leader to abdicate by releasing its slaves from their lease votes.

- **Spanner constrains when abdication is permissible.**
- Define s_{max} to be the maximum timestamp used by a leader.
- Before abdicating, a leader must **wait** until $TT.after(s_{max})$ is true.

分布式数据库, Spanner IV

Assigning Timestamps to RW Transactions

Transactional reads and writes use two-phase locking. As a result, they can be assigned timestamps at any time when all locks have been acquired, but before any locks have been released.

Spanner depends on the following monotonicity invariant:

- within each Paxos group, Spanner assigns timestamps to Paxos writes in monotonically increasing order, even across leaders.
 - A single leader replica can trivially assign timestamps in monotonically increasing order.
 - This invariant is enforced(强制) across leaders by making use of the disjointness invariant
 - a leader must only assign timestamps within the interval of its leader lease.(必须在 leader 租约内赋值) Note that whenever a timestamp s is assigned, s_{max} is advanced to s (s_{max} 时间上比 s 晚) to preserve disjointness.

分布式数据库, Spanner V

Assigning Timestamps to RW Transactions 续

- 对于每个 Paxos 组中, 即使 leader 被重新选举了, spanner 赋予的时间戳都是单调增长的。
 - 对于每个单独的 leader, 可以在其租约时间内, 赋予单调增长的时间戳。
 - 因为, 每个租约的时间, 都是不相交的。(比如, 新的 leader 被选举出来了。)



Figure: leader 租约示意图, leader1 红色表示, leader2 绿色表示

分布式数据库, Spanner VI

Assigning Timestamps to RW Transactions 续

Spanner also enforces the following external consistency(外部一致性) invariant:

- if the start of a transaction T_2 occurs after the commit of a transaction T_1 , then the commit timestamp of T_2 must be greater than the commit timestamp of T_1 .
- start events for a transaction T_i , e_i^{start} .
- commit events for a transaction T_i , e_i^{commit} .
- the commit timestamp of a transaction T_i , s_i .

不变式为: $t_{abs}(e_1^{commit}) < t_{abs}(e_2^{start}) \Rightarrow s_1 < s_2$

分布式数据库, Spanner VII

Assigning Timestamps to RW Transactions 续

The protocol for executing transactions and assigning timestamps obeys two rules:

- Define the arrival event of the commit request at the coordinator leader for a write/read T_i to be e_i^{server} . (对于一个到达中心协调的事件, 定义为 e_i^{server} 。)
- **Start:**
 - The coordinator leader for a write/read T_i assigns a commit timestamp s_i no less than the value of $TT.now().lastest$, ($[earliest, lastest]$), computed after e_i^{server} .
- **Commit Wait:**
 - The coordinator leader ensures that clients cannot see **any data committed by T_i until $TT.after(s_i)$ is true.** (s_i 赋予的 commit timestamp, 延迟写入, 直到 $TT.after(s_i)$ 满足, 当前不确定时间严格大于 s_i , 后续产生的时间戳必大于 s_i 。). Commit wait ensures that s_i is less than the absolute commit time of T_i , or $s_i < t_{abs}(e_i^{commit})$.

分布式数据库, Spanner VIII

Assigning Timestamps to RW Transactions 续

证明:

$$\begin{array}{llll} s_1 & < & t_{abs}(e_1^{commit}) & \text{(commit wait)} \\ t_{abs}(e_1^{commit}) & < & t_{abs}(e_2^{start}) & \text{(assumption)} \\ t_{abs}(e_2^{start}) & \leq & t_{abs}(e_2^{server}) & \text{(causality)} \\ t_{abs}(e_2^{server}) & \leq & s_2 & \text{(start)} \\ s_1 & < & s_2 & \text{(transitivity)} \end{array}$$

分布式数据库, Spanner IX

Serving Reads at a Timestamp

The monotonicity invariant allows Spanner to correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read.

- Every replica tracks a value called safe time t_{safe} which is maximum timestamp at which a replica is up-to-date.
 - A replica can satisfy a read at a timestamp t if $t \leq t_{safe}$.
- Define $t_{safe} = \min\{t_{safe}^{Paxos}, t_{safe}^{TM}\}$, where each Paxos
 - state machine has a safe time t_{safe}^{Paxos} .
 - each transaction manager has a safe time t_{safe}^{TM} .
- t_{safe}^{Paxos} is simpler:
 - It is the **timestamp of the highest-applied Paxos write.**
 - Because timestamps increase monotonically (timestamps 总是单调增长的)
 - writes are applied in order
 - will no longer occur at or below t_{safe}^{Paxos} with respect to Paxos. (对应一个的 Paxos)

分布式数据库, Spanner X

Serving Reads at a Timestamp 续

t_{safe}^{TM}

- t_{safe}^{TM} is ∞ if there are zero prepared transactions (没有二阶段协议的 prepared 事务)
- if there are prepared transactions(有 prepared 事务), the state affected by those transactions is indeterminate(不可确定的).
 1. a participant replica(参与的复制体) does not know yet whether such transactions will commit.(不知道该事务是否最终会被提交)。
 2. the commit protocol ensures that every participant knows a lower bound(下边界) on a prepared transaction's timestamp.
 3. Every **participant leader** (for a group g) for a transaction T_i assigns a prepare timestamp $s_{i,g}^{prepare}$ (i 事务, g 组) to its prepare record.
 4. The **coordinator leader** ensures that the transaction's **commit timestamp** $s_i \geq s_{i,g}^{prepare}$ over all participant groups g .
 5. Therefore , for every replica in a group g , (对于 g 中的所有的复制体), over all transactions T_i prepared at g , (所有的事务在 g 中准备的,)
 - $t_{safe}^{TM} = \min_i (s_{i,g}^{prepared} - 1)$ over all transactions prepared at g .

分布式数据库, Spanner XI

总结: 每个复制体 (replica)

$$t_{safe} = \min\{t_{safe}^{Paxos}, t_{safe}^{TM}\}$$

- 如果, 当前 (a group g) 没有二阶段协议的 prepared transactions, 那么, $t_{safe} = t_{safe}^{Paxos}$ (Paxos 最高时间戳, 最高必然是 commit 时间戳, Commit Wait)
- 如果, 当前 (a group g) 有许多的 prepared transactions, 选择所有 prepared transactions T_i 中, $t_{safe}^{TM} = \min_i(s_{i,g}^{prepared}) - 1$ 。

分布式数据库, Spanner XII

Read-Write Transactions (读写事务)

流程:

1. The client issues reads to the leader replica of the appropriate group, which acquires read locks and then reads the most recent data.
2. While a client transaction remains open, it sends keepalive message to prevent participant leaders from timing out its transaction.
3. When a client has completed all reads and buffered all writes, it begins **two-phase commit**.
4. The client chooses a **coordinator group** and sends a commit message to each participant's leader **with the identify of the coordinator and any buffered writes**.

分布式数据库, Spanner XIII

Read-Write Transactions (读写事务) 续

5. A **non-coordinator-participant** leader (二阶段协议中, 非协调者)
 - first, acquires write locks.
 - second, chooses a prepare timestamp that must be larger than any timestamps it has assigned to previous transactions (满足单调性), and logs a prepare record through Paxos. Each participant then notify the coordinator of its prepare timestamp.
6. A **coordinator** leader (二阶段协议中, 协调者)
 - first acquires write locks, but skips the prepare phase.(跳过 prepare 阶段)
 - It chooses a timestamp for the entire transaction after hearing from all other participant leaders.
 - the **commit timestamp s** must be greater or equal to all prepare timestamps.
 - the **commit timestamp s** must be greater than $TT.now().latest$ at the time the coordinator received its commit message,
 - the **commit timestamp s** greater than any timestamps the leader has assigned to previous transactions.
 - The coordinator leader then logs a commit record through Paxos.

分布式数据库, Spanner XIV

Read-Write Transactions (读写事务) 续

7. Before allowing any coordinator replica to apply the commit record, the coordinator leader waits until $TT.after(s)$, so as to obey the **commit-wait rule**.
 - Because the coordinator leader chose s based on $TT.now().lastest$, and now waits until that **timestamp is guaranteed to be int past**.
8. After commit wait, the coordinator sends the commit timestamp to the client and **all other participant leaders**.
9. Each participant leader logs the transaction's outcome through Paxos.
10. All participants apply at the same timestamp and then release locks.

分布式数据库, Spanner XV

Read-Only Transactions(只读事务)(不跨数据库)

- If the scope's values are served for a single Paxos group, (不跨数据库)
 - then the client issues the read-only transaction to that group's leader.
 - the leader assigns s_{read} and executes the read.
- Define $LastTS()$ to the **timestamp of the last committed write** at a Paxos group.
 - If there are no prepared transactions(没有 prepared 事务), the assignment $s_{read} = LastTS()$ trivially satisfies external consistency.
the transaction will see the result of the last write, and therefore be ordered after it.

分布式数据库, Spanner XVI

Read-Only Transactions(只读事务)(跨数据库)

- If the scope's values are served by multiple Paxos groups, there are several options.
 - The most complicated option is do a round of communication **with all of the groups's leaders** to negotiate(谈判) s_{read} based on $LastTS()$.
 - Spanner 使用了一个简单的方法, The client avoids a negotiation round, and just has its reads execute at $s_{read} = TT.now().lastest$ (which may wait for safe time to advance). All reads in the transactions can be sent to replicas that are sufficiently up-to-date.

分布式数据库, Spanner XVII

FAQ: How Spanner ensures that if $r/w\ T_1$ finishes before $r/o\ T_2$ starts, $TS_1 < TS_2$.

Two rules:

- **Start rule:**

- $xaction\ TS = TT.now().latest$
 - for r/o , at start time.
 - for r/w , when commit begins.

- **Commit wait, for r/w xaction:**

- Before commit, delay until $TS < TS.now().earliest$. Guarantees that TS has passed.

分布式数据库, Spanner XVIII

Start rule: $\text{xaction TS} = \text{TT.now().latest}$

Commit wait: Before commit, delay until $\text{TS} < \text{TS.now().earliest}$, Guarantees that TS has passed.

例子: T1 commits(真实提交的时间), then T2 starts, T2 must see T1's writes. I.e. we need $\text{TS1} < \text{TS2}$.

```
r/w T0 @ 0: Wx1 C
              |1-----10| |11-----20|
r/w T1 @ 10:      Wx2 P          C
                  |10-----12|
r/o T2 @ 12:                  Rx?
```

1. C guaranteed to occur after its TS (10) due to commit wait.
2. Rx occurs after C by assumption(假设成立), and thus after time 10.
3. T2 choose TT.now().latest , which is after current time,(因果性),which is after 10.
4. So $\text{TS2} > \text{TS1}$.

分布式数据库, Spanner XIX

FAQ: Why this provides **external consistency**:

- Commit wait means r/w TS(给实际写入的 timestamp) is guaranteed to be in the past.
- r/o TS = TT.now().lastest is guaranteed to be \geq correct time.
- thus \geq TS of any previous committed transaction (due to its commit wait).

More generally:

- Snapshot Isolation gives you serializable r/o transactions.
 - Timestamps set an order.
 - Snapshot versions (and safe time (副本维持的安全读的 timestamp)) implement consistent reads at a timestamp.
 - Xaction sees all writes from lower-TS xactions, none from higher.
 - Any number will do for TS if you don't care about external consistency.
- Synchronized timestamps yield external consistency.
 - Even among transactions at different data centers.
 - Even though reading from local replicas that might lag.

分布式数据库, Spanner XX

Correctness constraints on r/o transactions:

- **Serializable(串行化)**
 - Same results as if transactions executed one-by-one. Even though they may actually execute concurrently.
 - an r/o xaction must essentially(本质上) fit between r/w xactions.
 - See all writes from prior transactions(前面的事务), nothing from subsequent.(后续)
- **Externally consistent(外部一致性)**
 - **If T1 completes before T2 starts, T2 must see T1's writes.(T1 在 T2 开始之前提交了, T2 必须看到 T1 的写入)**
 - "Before" refers to real (wall-clock) (真实的时钟) time.
 - Similar to linearizable.
 - Rules out reading stale data.(排除阅读旧数据)

分布式数据库, Spanner XXI

FAQ: **Why not have r/o transactions just read the latest committed values?**

Suppose we have two bank transfers, and a transaction that reads both.

T1: Wx Wy C

T2: Wx Wy C

T3: Rx Ry

The result won't match **any serial order!**

- Not T1,T2,T3,
- Not T1,T3,T2.

We want T3 to see both of T2's writes, or none.

We want T3's reads to **all** occur at the **same** point relative to T1/T2.

分布式数据库, Spanner XXII

FAQ: 如何解决上面的问题呢? 引入 Snapshot Isolation (快照隔离)

Idea:

- Synchronize all computer's clocks(to real wall-clock time (真实的墙上时钟)).
- Assign every transaction a time-stamp.
 - r/w: commit time. (提交时间戳)
 - r/o: start time. (开始时间戳)
- Execute as if one-at-a-time (好像每个时间点都在执行) in time-stamp order.(按照时间戳执行)
 - Even if actual reads occur in different order.
- Each replica stores **multiple time-stamped versions of each record.**
 - All of a r/w transactions's writes get the same time-stamp.
- An r/o transactions's reads seen version as of xaction's time-stamp.
 - The record version with the highest time-stamp less than the xaction's.

分布式数据库, Spanner XXIII

Snapshot Isolation 例子:

			x@10=9			x@20=8
			y@10=11			y@20=12
T1 @ 10:	Wx	Wy	C			
T2 @ 20:				Wx	Wy	C
T3 @ 15:			Rx			Ry

- "@ 10" indicates the time-stamp.
- Now T3's reads will both be served from the @10 versions.
 - T3 won't see T2's write even though T3's read of y occurs after T2.
- Now the results are serializable: $T1 \rightarrow T3 \rightarrow T2$
 - The **serial order** is the same as **time-stamp order**.

分布式数据库, Spanner XXIV

Why OK for T3 to read the *old* value of y even though there's a new value ?

- T2 and T3 are concurrent, so external consistency allows **either order**.
- Remember: r/o transactions need to read values as of their timestamp, and *not* see later writes.

分布式数据库, Spanner XXV

Problem: what if T3 read x from a replica that hasn't seen T1's write?
Because the replica wasn't in the Paxos majority?

Solution: replica "safe time".

- Paxos leaders send writes in timestamp order.
- Before serving a read at time 20, replica must see Paxos write for time > 20 . (**Start rule** & **Commit wait rule**) So it knows it has seen all writes < 20 .
- Must also delay it prepared but uncommitted transactions
- Thus: r/o transactions can read from local replica – usually fast.

分布式数据库, Spanner XXVI

Problem: What if clocks are not perfectly synchronized? (时钟不是很好的同步时?)

What goes wrong if clocks aren't synchronized exactly ?

- No problem for r/w transactions, which use locks.
- If an r/o transaction's TS is too **large**:
 - Its TS will be higher than replica safe times, and **reads will block**.
Correct but slow – delay increased by amount of clock error.
- If an r/o transaction's TS is too **small**:
 - It will miss writes that committed before the r/o transaction started.
Since its slow TS will cause it to use old versions of record. (读到了以前的数据) This violates external consistency.

分布式数据库, Spanner XXVII

Example of problem if r/o xaction's TS is too small:

```
r/w T0 @ 0: WX1 C
r/w T1 @ 10:           WX2 C
r/w T2 @ 5:           Rx?
(C for commit)
```

- This would case T2 to read the version of x at time 0, which was 1.
- But T2 started after T1 committed (in real time),
 - so external consistency requires that T2 see x=2.
- So there must be a solution to the possibility of incorrect clocks!