数据库事务的概念和实现

2018-08-09
liht@fenbi.com

大纲

- 基本概念 ACID BASE CAP
- 隔离级别
- 可序列化
- Redis 事务处理
- MySQL 事务处理

上节回顾

- 1
- 2
- 3
- 4

本节预习

- 脏读
- 可重复读
- 幻读
- 更新丢失

事务

Some authors have claimed that general two-phase commit is too expensive to support, because of the performance or availability problems that is brings. We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.

基本概念

- 事务是简化错误处理、实现可靠性的首选机制
- 简化应用编程模型
- 安全保证（safety guarantees），通过使用事务，应用程序可以自由地忽略某些潜在的错误情况和并发问题，因为数据库会替应用处理好这些。
- 并不是所有的应用都需要事务

基本概念-ACID

- 原子性（Atomicity）
- 一致性（Consistency）
- 隔离性（Isolation）
- 持久性（Durability）

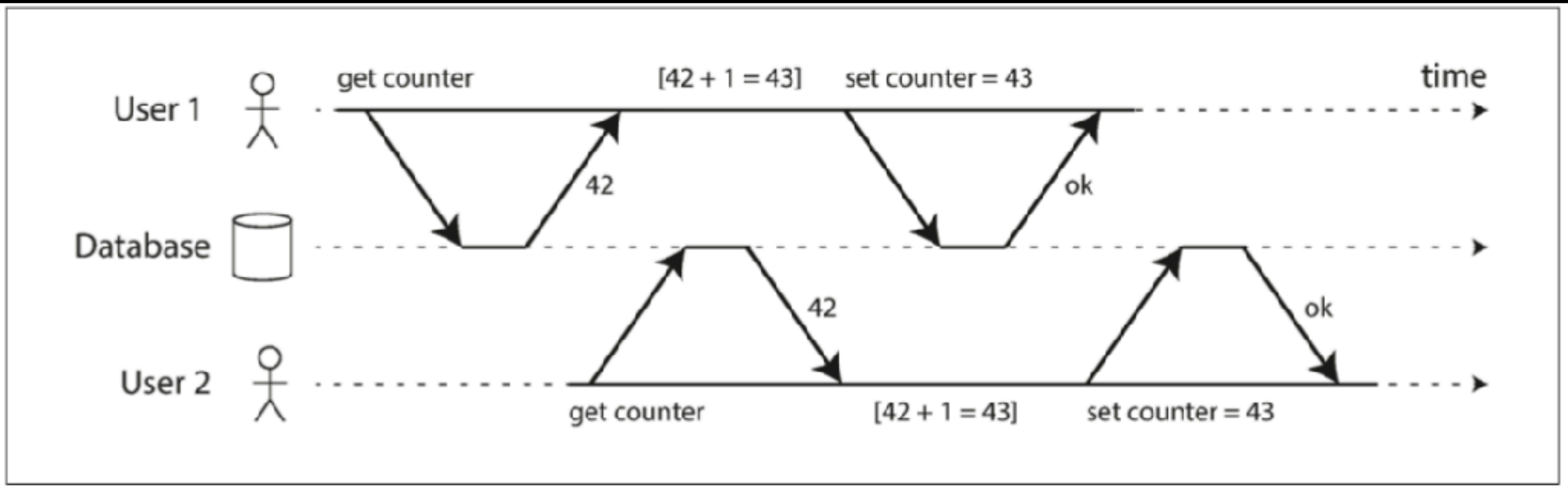
基本概念-ACID

- 原子性（Atomicity）
  - 能够在错误时中止事务，丢弃该事务进行的所有写入变更的能力。
  - 可中止性（abortability）
    - 宁为玉碎，不为瓦全（all-or-nothing）
- 一致性（Consistency）
  - 对数据的一组特定陈述必须始终成立，不变量（invariants）
  - 一致性（在ACID意义上）是应用程序的属性
  - 应用可能依赖数据库的原子性和隔离属性来实现一致性，但这并不仅取决于数据库。因此，字母C不属于ACID？？？

基本概念-ACID

- 隔离性（Isolation）
  - 竞争条件（race conditions）
  - 同时执行的事务是相互隔离的：它们不能相互影响
  - 可序列化（Serializability）
- 持久性（Durability）
  - 持久性 是一个承诺，即一旦事务成功完成，即使发生硬件故障或数据库崩溃，写入的任何数据也不会丢失。
  - 在单节点数据库中，持久性通常意味着数据已被写入非易失性存储设备，如硬盘或 SSD。
  - 在带复制的数据库中，持久性可能意味着数据已成功复制到一些节点。

隔离性的例子


User 1
get counter
[42 + 1 = 43]
set counter = 43
time
42
ok
Database
42
ok
User 2
get counter
[42 + 1 = 43]
set counter = 43

基本概念-BASE

- 佛系的 ACID
- 基本可用性（Basically Available）
    - 分布式系统在出现不可预知故障的时候，允许损失部分可用性
- 软状态（Soft State）
    - 允许系统中的数据存在中间状态
- 最终一致性（Eventual consistency）
- 关于一致性
    - 强一致性
    - 弱一致性
    - 最终一致性

基本概念 - CAP

- Consistency（一致性）
- Availability（可用性）
- Partition tolerance（分区容错性）
- 三选二

基本概念 - 单对象和多对象操作

- 单对象写入
  - 对单节点上的单个对象（例如键值对）上提供原子性和隔离性
    - 自增
    - CAS
- 多对象事务的需求
  - 同时修改多个对象（行，文档，记录）。通常需要多对象事务（multi-object transaction） 来保持多块数据同步。

讨论 - 基本概念

- 持久性意味着数据永远不会丢吗?

- 事务的一个关键特性是，如果发生错误，它可以中止并安全地重试。重试有没有问题?

小结 - ACID基本概念

- ACID
- BASE
- CAP
- 单对象和多对象操作

隔离级别

P1（"Dirty read"）*P2*（"Non-repeatable read"）*P3*（"Phantom"）

03 Table 8 — SQL-transaction isolation levels and the three phenomena

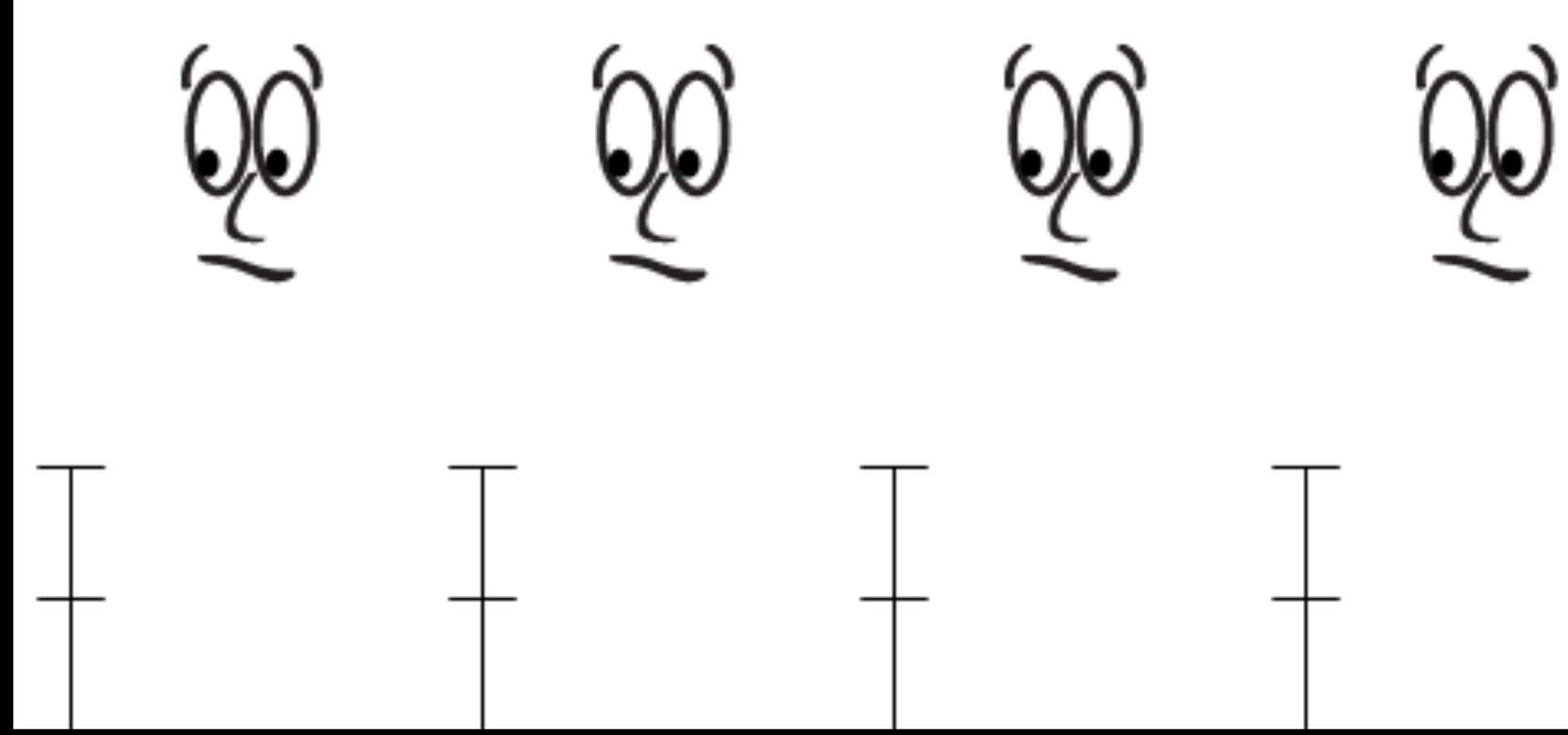
Level	*P1*	*P2*	*P3*
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

READ UNCOMMITTED, NOLOCK

READ COMMITTED

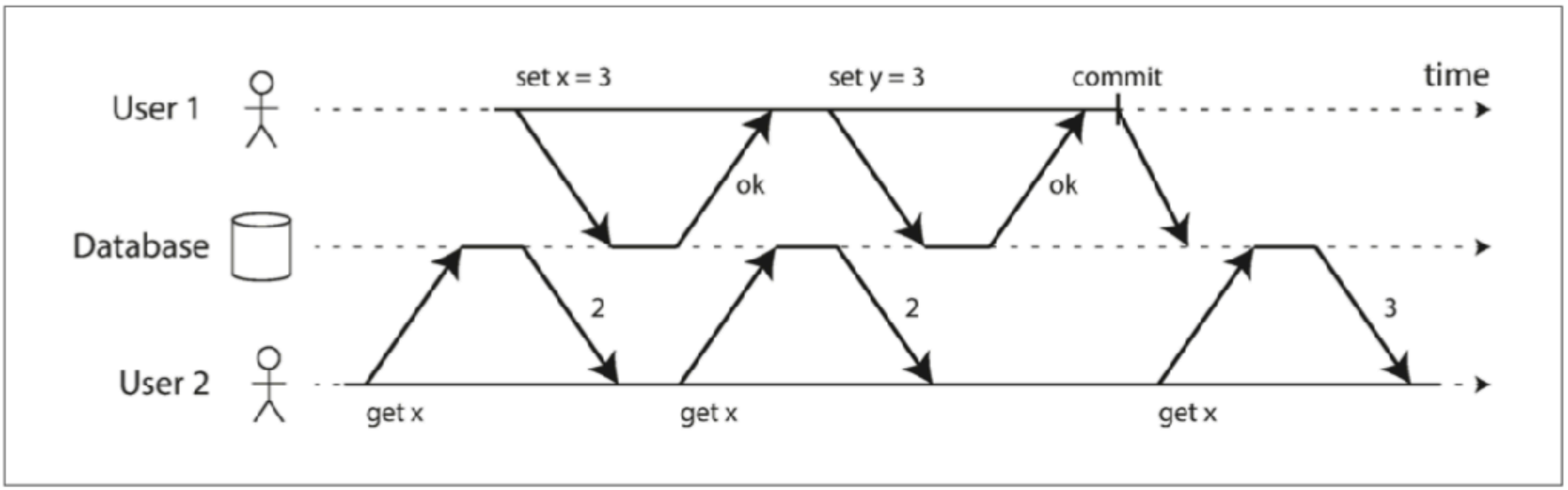
REPEATABLE READ

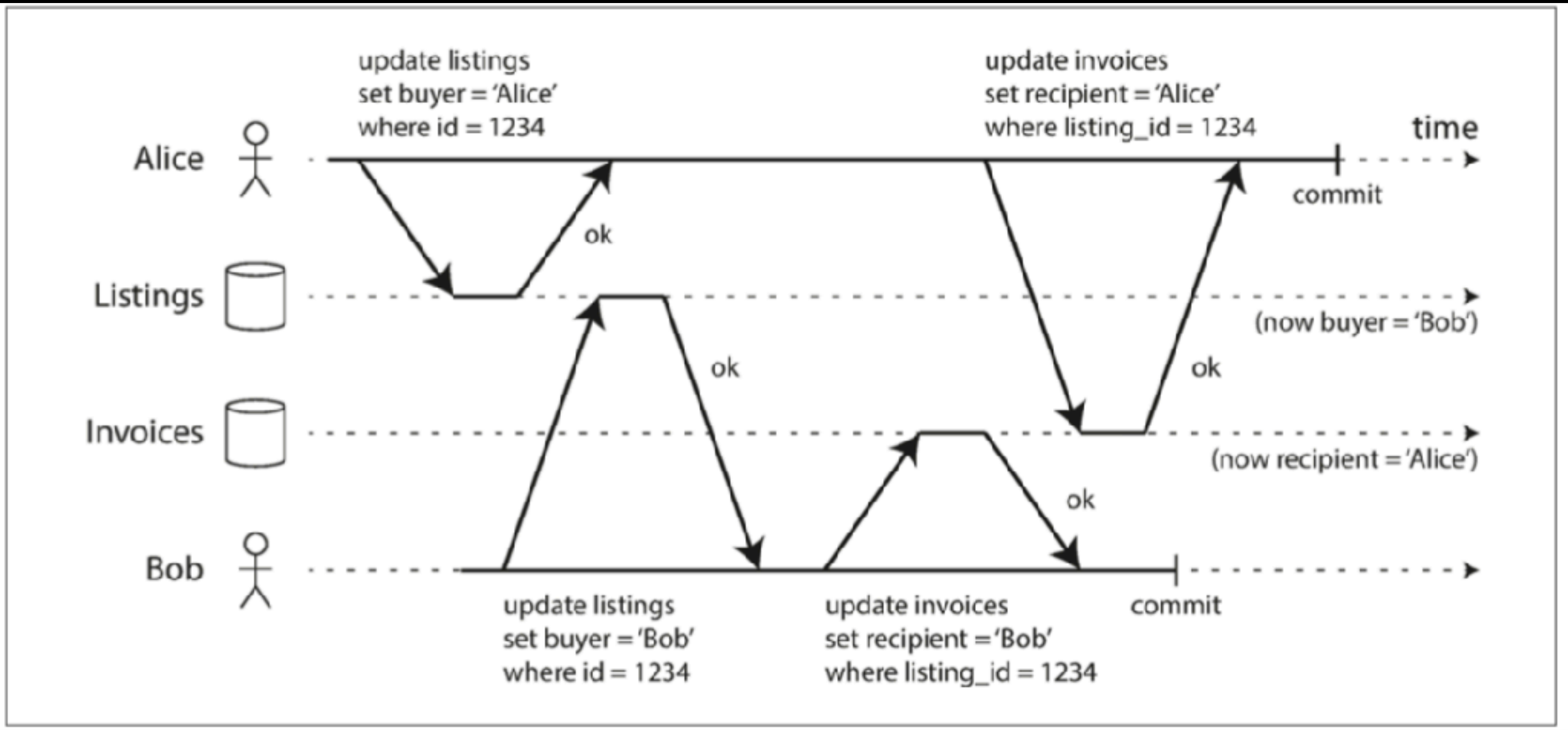
SERIALIZABLE, HOLDLOCK



弱隔离级别 - RC

- 从数据库读时，只能看到已提交的数据（没有脏读（dirty reads））
- 写入数据库时，只会覆盖已经写入的数据（没有脏写（dirty writes））

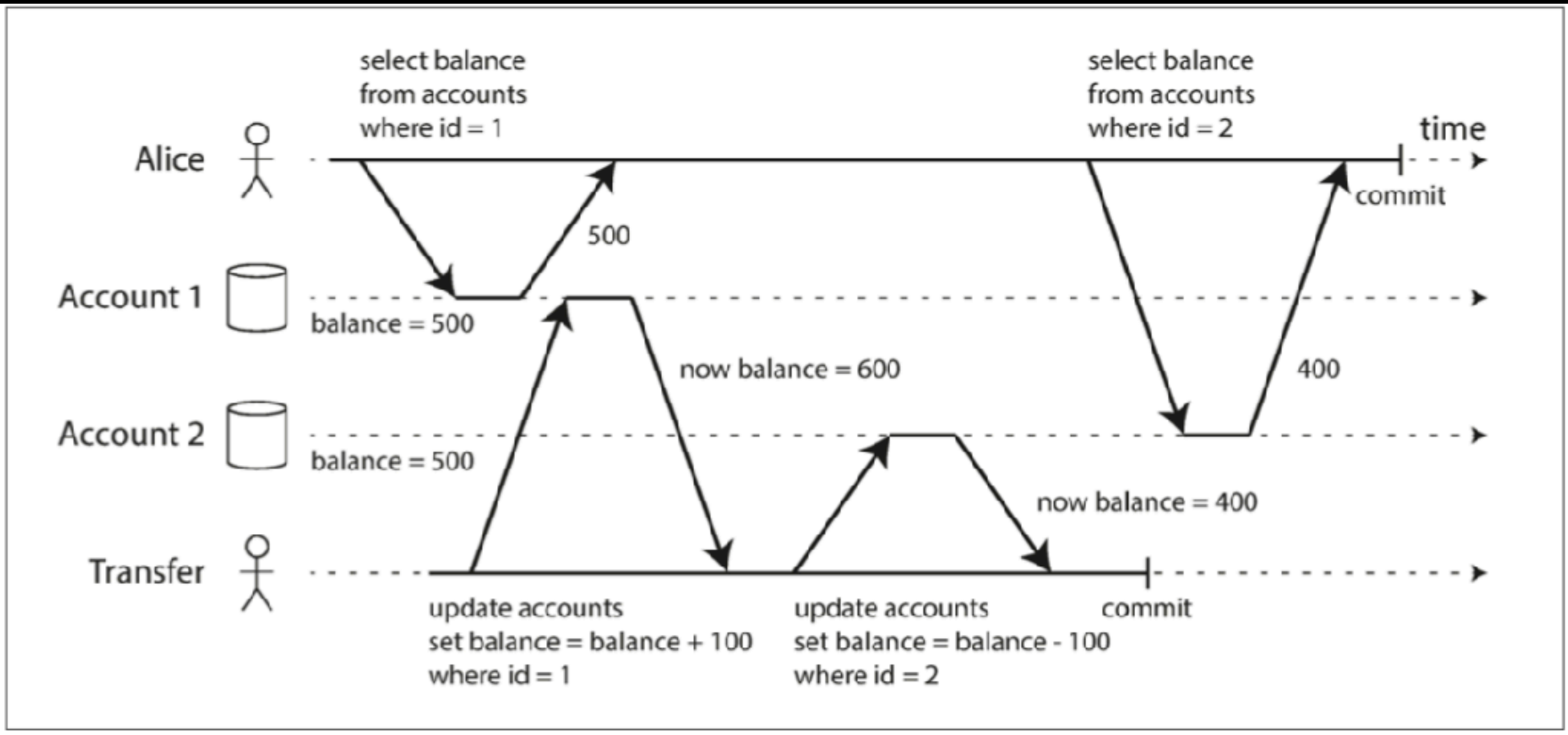

User 1
set x = 3
set y = 3
commit
time
ok
ok
Database
2
2
3
User 2
get x
get x
get x


update listings
set buyer = 'Alice'
where id = 1234
update invoices
set recipient = 'Alice'
where listing_id = 1234
time
Alice
ok
commit
Listings
(now buyer = 'Bob')
ok
ok
Invoices
(now recipient = 'Alice')
ok
Bob
update listings
set buyer = 'Bob'
where id = 1234
update invoices
set recipient = 'Bob'
where listing_id = 1234
commit

实现读已提交

- 防止脏写
  - 数据库通过使用行锁（row-level lock）
- 防止脏读
  - 使用相同的锁
  - 对于写入的每个对象，数据库都会记住旧的已提交值，和由当前持有写入锁的事务设置的新值。
  - 保留一个对象的两个版本就足够了：提交的版本和被覆盖但尚未提交的版本 （快照隔离）

快照隔离和可重复读

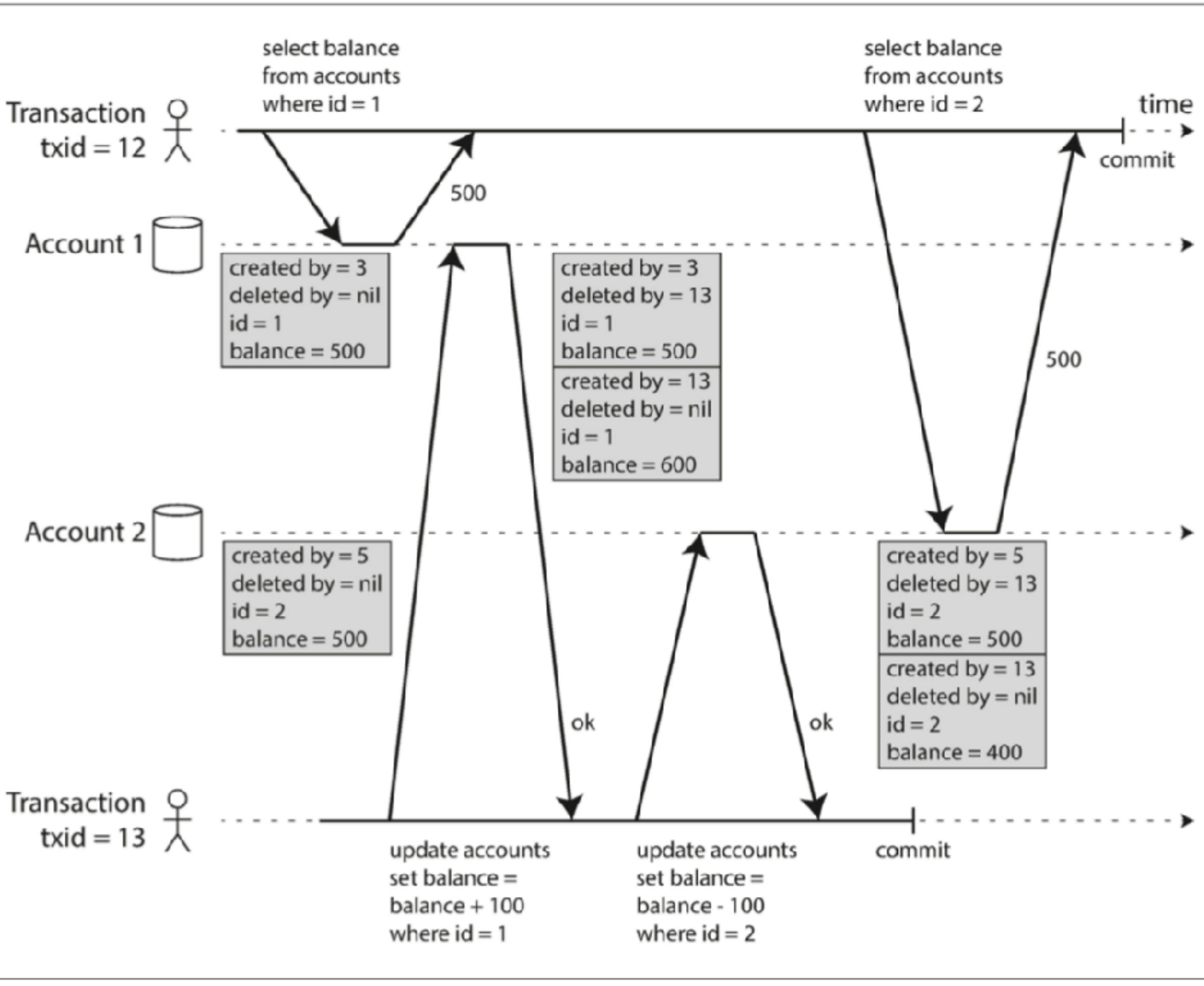

select balance
from accounts
where id = 1
select balance
from accounts
where id = 2
time
Alice
commit
500
Account 1
balance = 500
now balance = 600
400
Account 2
balance = 500
now balance = 400
Transfer
update accounts
set balance = balance + 100
where id = 1
update accounts
set balance = balance - 100
where id = 2
commit

不可重复读

- 不可重复读（nonrepeatable read）或读取偏差（read skew）
- 不可容忍的情况
  - 备份
  - 分析查询和完整性检查
- 常见解决方案
  - 快照隔离（snapshot isolation）

实现快照隔离

- 关键原则：读不阻塞写，写不阻塞读
  - 使用写锁来防止脏写
  - 读取不需要任何锁定
- 多版本并发控制（MVCC, multi-version concurrentcy control）
  - 读已提交为每个查询使用单独的快照
  - 快照隔离对整个事务使用相同的快照


Transaction
txid = 12
select balance
from accounts
where id = 1
500
select balance
from accounts
where id = 2
time
commit
Account 1
created by = 3
deleted by = nil
id = 1
balance = 500
created by = 3
deleted by = 13
id = 1
balance = 500
created by = 13
deleted by = nil
id = 1
balance = 600
500
Account 2
created by = 5
deleted by = nil
id = 2
balance = 500
created by = 5
deleted by = 13
id = 2
balance = 500
created by = 13
deleted by = nil
id = 2
balance = 400
ok
ok
Transaction
txid = 13
update accounts
set balance =
balance + 100
where id = 1
update accounts
set balance =
balance - 100
where id = 2
commit

MVCC

- 实现
  - 事务ID
  - created_by
  - deleted_by
- 可见性
  - 读事务开始时，创建该对象的事务已经提交。
  - 对象未被标记为删除，或如果被标记为删除，请求删除的事务在读事务开始时尚未提交。

防止丢失更新

- 并发写入
  - 后面写入狠揍（clobber） 了前面的写入
- read-modify-write 顺序执行
  - 原子写
    - 通常怎么实现?
  - 显示锁定
- 自动检测丢失的更新
  - 比较并设置（CAS）

写入偏差与幻读

- Write Skew and Phantoms
- 并发写入的race condition

Alice:

begin transaction

currently_on_call = (
  select count(*) **from** doctors
  where on_call = true
  and shift_id = 1234
)

Now currently_on_call = 2

if (currently_on_call >= 2) {
  update doctors
  set on_call = false
  where name = 'Alice'
  and shift_id = 1234
}

commit transaction

name	on_call
Alice	true
Bob	true
Carol	false

Bob:

begin transaction

currently_on_call = (
  select count(*) **from** doctors
  where on_call = true
  and shift_id = 1234
)

Now currently_on_call = 2

if (currently_on_call >= 2) {
  update doctors
  set on_call = false
  where name = 'Bob'
  and shift_id = 1234
}

commit transaction

Alice false

Bob false

name	on_call
Alice	false
Bob	false
Carol	false

写入偏差与幻读

- 如果两个事务读取相同的对象，然后更新其中一些对象（不同的事务可能更新不同的对象），则可能发生写入偏差。
- 在多个事务更新同一个对象的特殊情况下，就会发生脏写或丢失更新
- 幻读(Phantoms)
    - 一个事务中的写入改变另一个事务的搜索查询的结果
- 解决方法
    - Materializing conflicts （物化冲突）
    - Serializable （序列化）

讨论 - 弱隔离级别

- 概念
  - 脏读、脏写、读偏差(read skew)、更新丢失、写偏差(write skew)、幻读
- RC
  - 如何防止脏写
  - 如何防止脏读
- SI & RR
  - 不可重复读问题一定不能接受吗?
  - RR 怎么实现
  - MVCC 怎么实现
- 防止 Lost Update
  - 有哪些方法

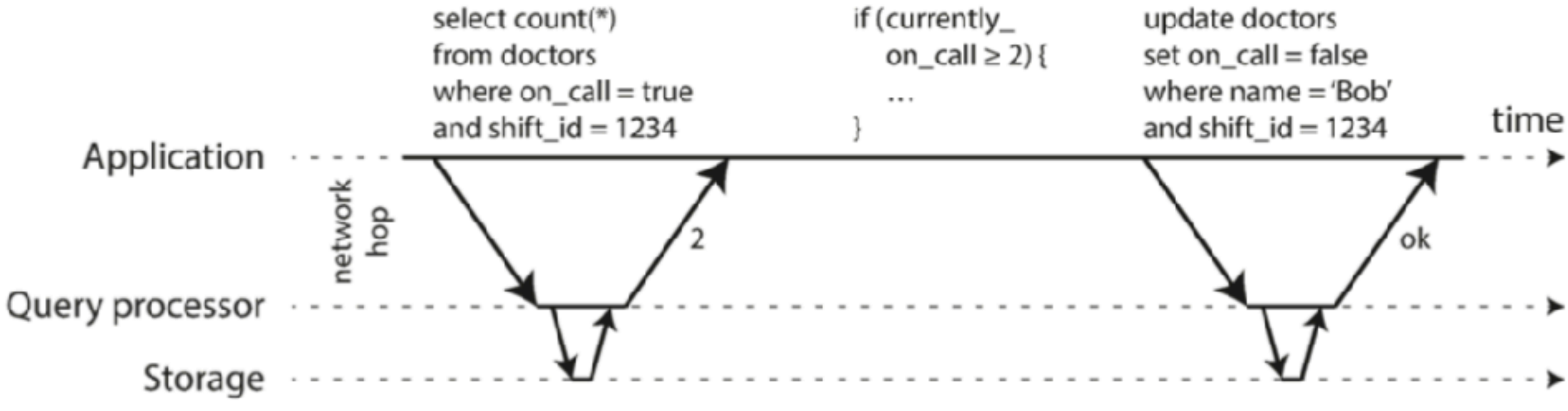
小结 - 弱隔离级别

- 读已提交（Read Committed）
- 快照隔离和可重复读（Snapshot Isolation and Repeatable Read）
- 防止丢失更新（Preventing Lost Updates）
- 写入偏差与幻读（Write Skew and Phantoms）

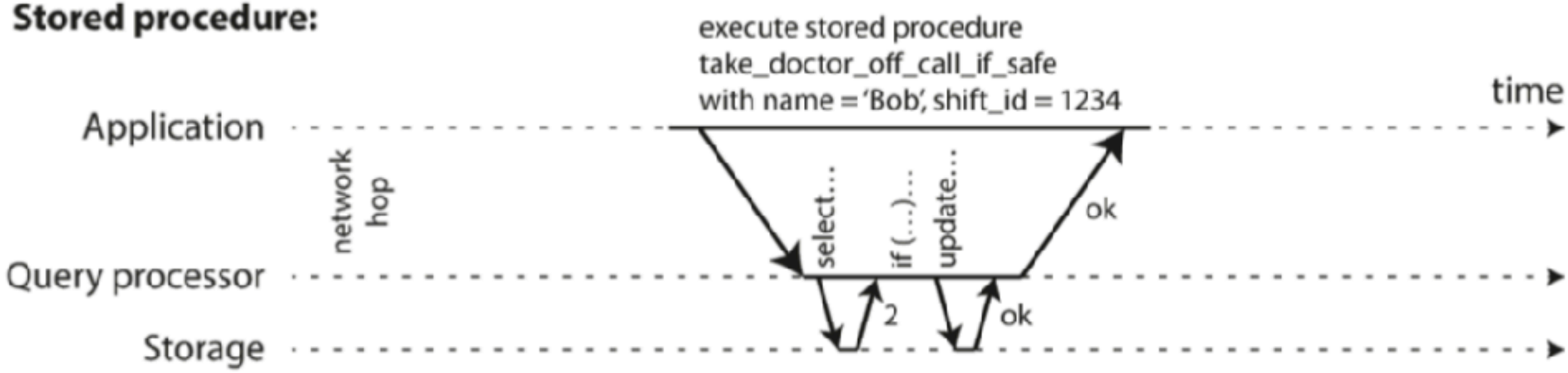
可序列化

- 真的串行执行
- 两阶段锁定（2PL）
- 序列化快照隔离（SSI Serializable Snapshot Isolation）

Interactive transaction:



Stored procedure:



真的串行

- 实现方式
  - 在单个线程上按顺序一次只执行一个事务
  - 在存储过程中封装事务
- 局限
  - 每个事务都必须小而快，只要有一个缓慢的事务，就会拖慢所有事务处理。
  - 写入吞吐量必须低到能在单个CPU核上处理

2PL

- SS2PL
  - 对象只要有写入（修改或删除），就需要独占访问（exclusive access） 权限
  - 读写相互阻塞
- 锁模式
  - 共享模式（shared mode）
  - 独占模式（exclusive mode）
- 两阶段
  - 第一阶段（当事务正在执行时）获取锁
  - 第二阶段（在事务结束时）释放所有的锁

2PL的实现

- 若事务要读取对象，则须先以共享模式获取锁。
- 若事务要写入一个对象，它必须首先以独占模式获取该锁。
- 如果事务先读取再写入对象，则它可能会将其共享锁升级为独占锁。
- 事务获得锁之后，必须继续持有锁直到事务结束（提交或中止）。

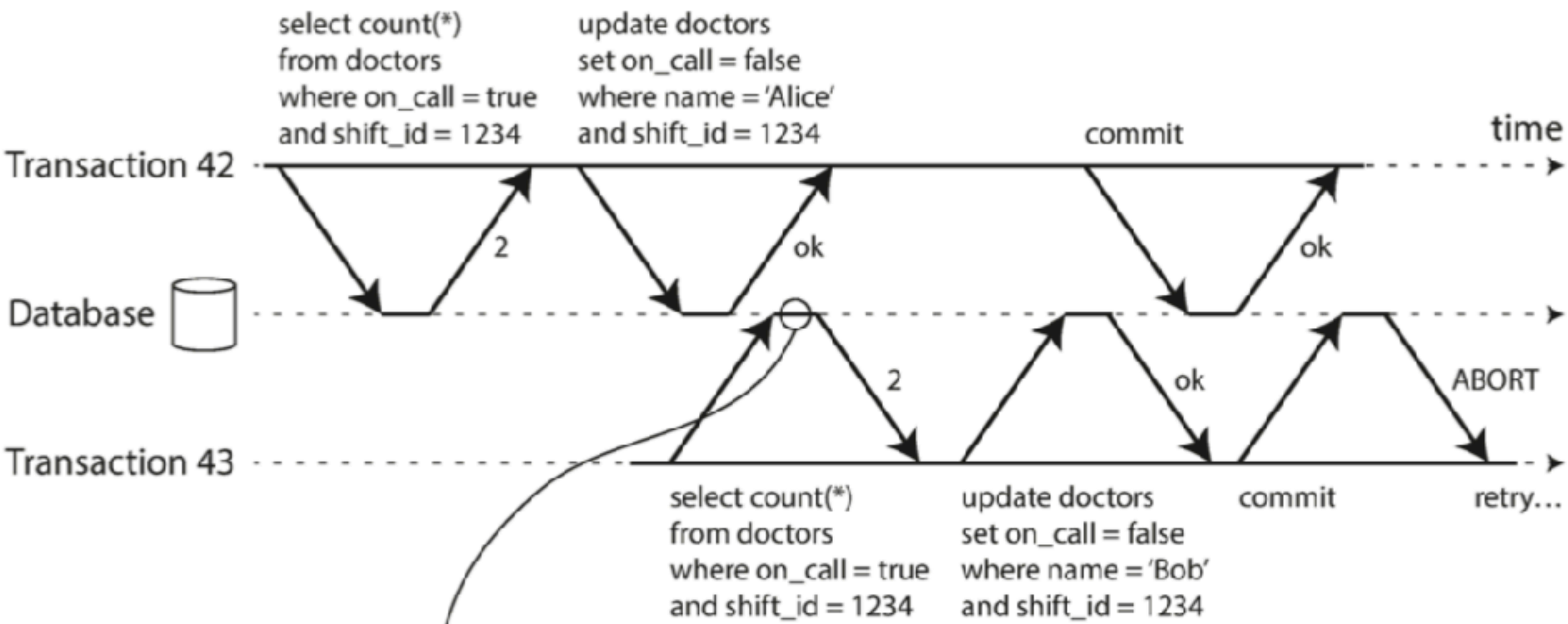
- 谓词锁 Predicate locks
  - 不属于特定的对象（例如，表中的一行），它属于所有符合某些搜索条件的对象
- 索引范围锁
  - 间隙锁（next-key locking）
  - 通过使谓词匹配到一个更大的集合来简化谓词锁是安全的

序列化快照隔离

- 悲观与乐观的并发控制
  - 两阶段锁
  - 序列化快照隔离
- 序列化快照隔离
  - 快照隔离 + 序列化冲突检测

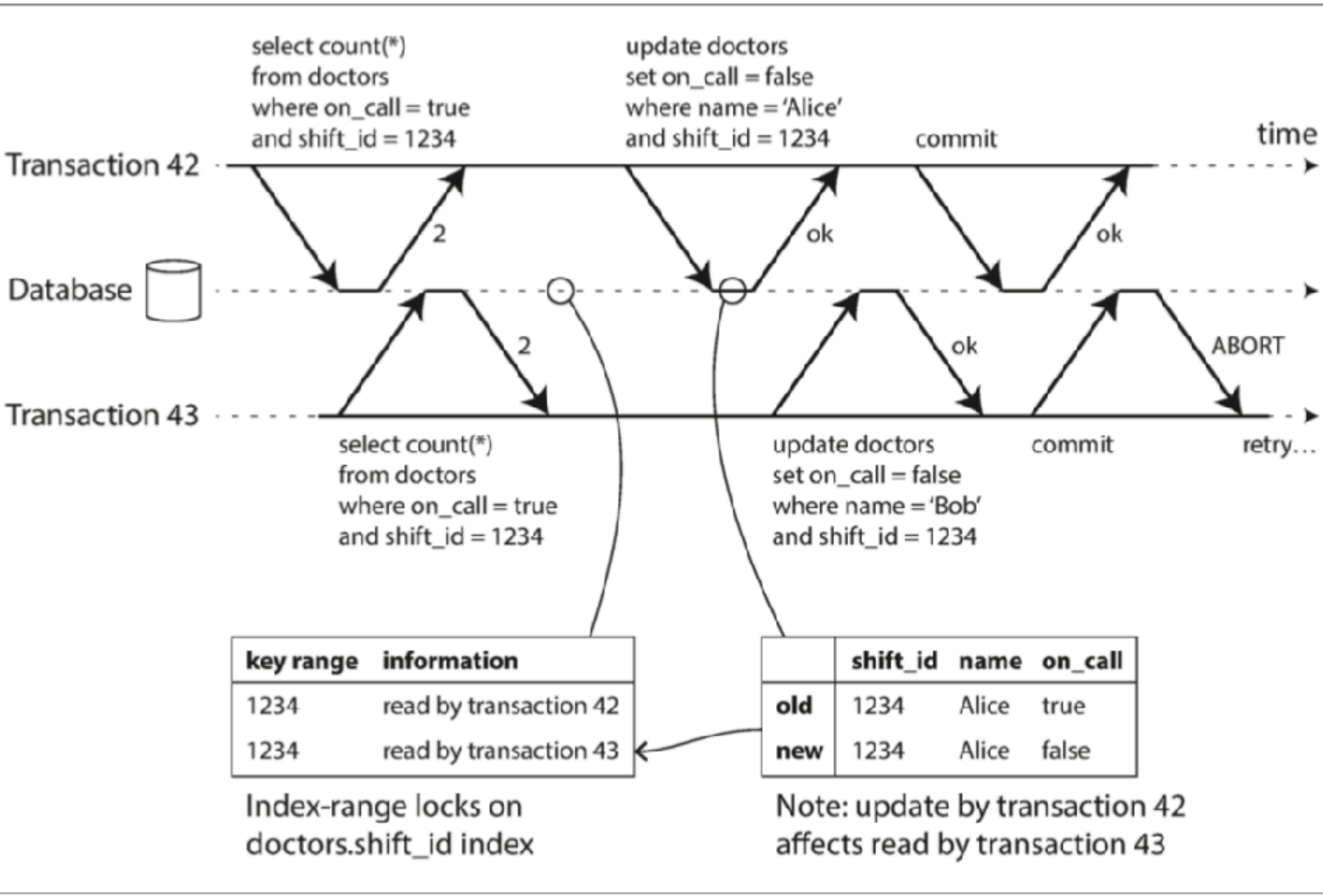
序列化快照隔离实现

- Decisions based on an outdated premise 基于过时前提的决策
  - 检测对旧MVCC对象版本的读取（读之前存在未提交的写入）
  - 检测影响先前读取的写入（读之后发生写入）



shift_id	name	on_call	created_by	deleted_by
1234	Alice	true	1	42
1234	Alice	false	42	—
1234	Bob	true	1	—
1234	Carol	false	1	—

Transaction 42 hasn't committed yet, so transaction 43 sees Alice as still being on call. However, the transaction manager notes that this value is no longer up-to-date.


select count(*)
from doctors
where on_call = true
and shift_id = 1234
update doctors
set on_call = false
where name = 'Alice'
and shift_id = 1234
commit
time
Transaction 42
2
ok
ok
Database
2
ok
ABORT
Transaction 43
select count(*)
from doctors
where on_call = true
and shift_id = 1234
update doctors
set on_call = false
where name = 'Bob'
and shift_id = 1234
commit
retry…
key range
information
1234
read by transaction 42
1234
read by transaction 43
shift_id
name
on_call
old
1234
Alice
true
new
1234
Alice
false
Index-range locks on
doctors.shift_id index
Note: update by transaction 42
affects read by transaction 43

讨论 - 可序列化

- 可序列化 == 串行？
- 2PL的实现?
- 谓词锁、间隙锁的概念
- 序列化快照隔离的基本思想

小结 - 可序列化

- 真的串行执行
- 两阶段锁定（2PL）
- 序列化快照隔离（SSI Serializable Snapshot Isolation）

本章小结

- 基本概念 ACID BASE CAP
- 隔离级别
- 弱隔离级别
    - RC
    - RR
    - 防止丢失更新
    - 写入偏差和幻读
- 可序列化
    - 串行
    - 2PL
    - SSI

Redis 事务处理

Redis 事务处理

- 事务的实现
  - 事务开始
  - 命令入队
  - 事务执行
- 事务相关命令
  - MULTI，告诉 Redis 服务器开启一个事务。注意，只是开启，而不是执行
  - EXEC，告诉 Redis 开始执行事务
  - DISCARD，告诉 Redis 取消事务
  - WATCH，监视某一个键值对，它的作用是在事务执行之前如果监视的键值被修改，事务会被取消。

- Redis 命令队列
  - 每个客户端有自己的事务状态
  - FIFO
- 键值的监视 WATCH，乐观锁
  - CAS
- Redis 事务的执行和取消
  - REDIS_DIRTY_CAS 更改的时候会设置此标记
  - REDIS_DIRTY_EXEC 命令入队时出现错误，此标记会导致 EXEC 命令执行失败

演示

Redis - ACID

- 原子性：打包操作，要么都执行，要么都不执行。Redis 支持原子性吗?

- 一致性：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这一点，Redis 事务能够保证。

- 隔离性：Redis 不存在多个事务的问题，因为 Redis 是单进程单线程的工作模式。

- 持久性：在事务完成以后，该事务对数据库所作的更改便持久地保存在数据库之中，并且是完全的。

Redis - Lua脚本

- 执行 Lua 脚本
  - 存储过程的一种实现方式
    - 是否是原子的?
    - 怎么保证相同脚本可以在不同的机器上产生相同的结果
    - 怎么做持久化
    - 不要让 Lua 脚本影响整个 Redis 服务器的性能

小结 - Redis事务处理

- 将多个命令打包，一次性有序执行
- 命令入队，FIFO
- 事务执行过程中不会中断
- 使用WATCH命令监视数据库键，乐观锁
- REDIS_DIRTY_CAS=1 服务器拒绝执行
- Redis中的ACID
- Redis中的存储过程 - Lua脚本

MySQL 事务处理

InnoDb 事务处理

- 隔离级别
- MVCC
- redo undo purge
- 锁

演示-隔离级别

MVCC

- 空间换时间
- 每行两个额外的隐藏值
  - createdVersion
  - expiredVersion
- 版本号：随新事务增长
- snapshot read：快照读
- locking read：当前读

MVCC 实现 RR

- SELECT时，读取创建版本号<=当前事务版本号，删除版本号为空或>当前事务版本号。
- INSERT时，保存当前事务版本号为行的创建版本号
- DELETE时，保存当前事务版本号为行的删除版本号
- UPDATE时，插入一条新纪录，保存当前事务版本号为行创建版本号，同时保存当前事务版本号到原来删除的行

快照读与当前读

- 快照读：就是select
  - select * from table ....;
- 当前读：特殊的读操作，插入/更新/删除操作，属于当前读，处理的都是当前的数据，需要加锁。
  - select * from table where ? lock in share mode;
  - select * from table where ? for update;
  - insert;
  - update ;
  - delete;

redo undo purge

- Redo
  - 重做日志，物理日志
  - 保证事务的持久性
  - Redo log buffer, redo log file, Force Log at Commit
- Undo
  - 保证事务的原子性、隔离性
  - 回滚
  - MVCC 非锁定读 / 快照读
- Purge
  - 标记删除
  - 后台清理
- Binlog
  - 逻辑日志，主从同步

InnoDb 事务处理 - 锁

有哪几种锁-锁类型

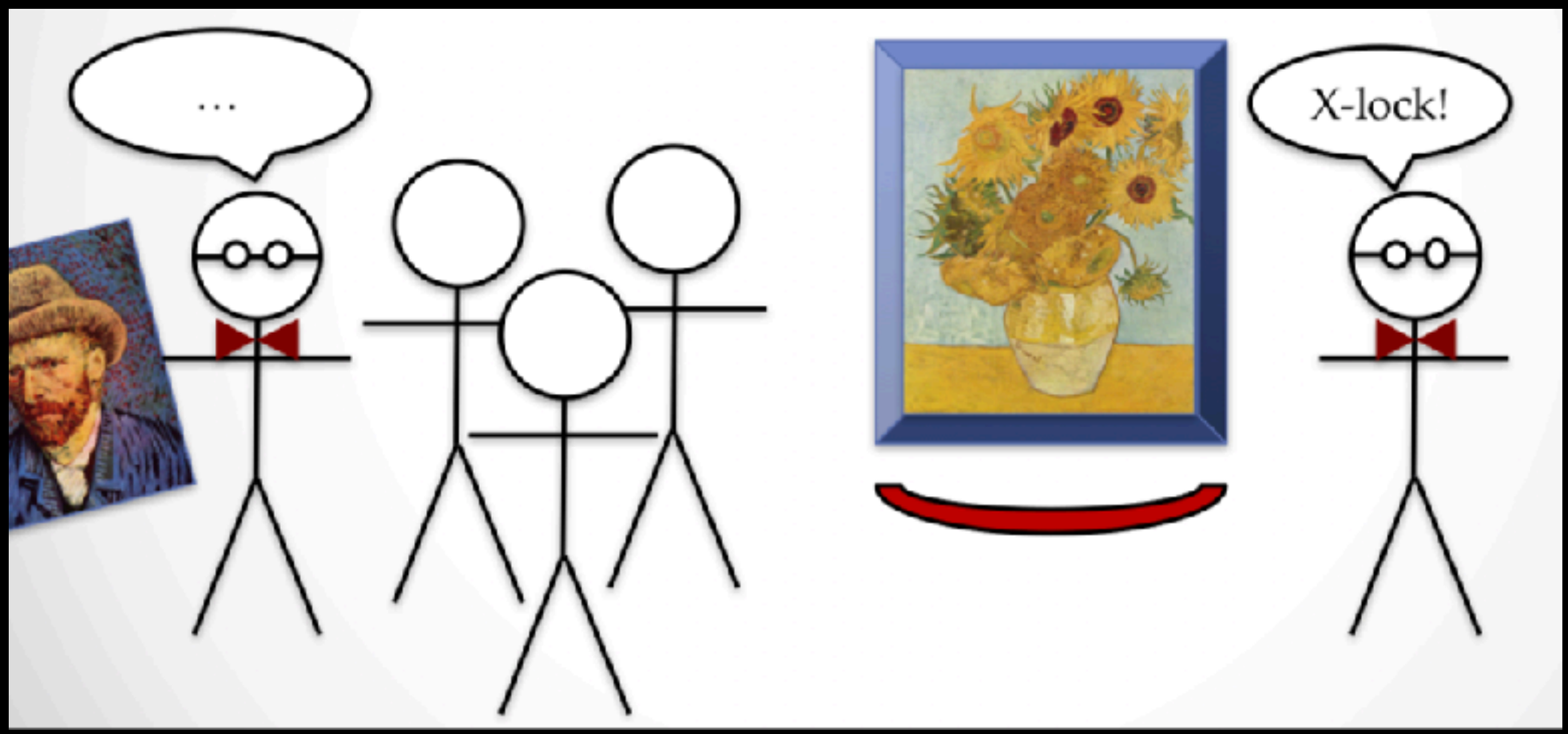
```
/* Basic lock modes */
enum lock_mode {
    LOCK_IS = 0,    /* intention shared */
    LOCK_IX,    /* intention exclusive */
    LOCK_S,     /* shared */
    LOCK_X,     /* exclusive */
    LOCK_AUTO_INC,  /* locks the auto-inc counter of a table
            in an exclusive mode */
    LOCK_NONE,  /* this is used elsewhere to note consistent read */
    LOCK_NUM = LOCK_NONE, /* number of lock modes */
    LOCK_NONE_UNSET = 255
};
```

有哪几种锁-锁冲突

LOCK COMPATIBILITY MATRIX

	IS	IX	S	X	AI
IS	+	+	+	-	+
IX	+	+	-	-	+
S	+	-	+	-	-
X	-	-	-	-	-
AI	+	+	-	-	-

+ 代表兼容， -代表不兼容


...
X-lock!

锁

- 一致性非锁定读
- 一致性锁定读
- 自增长与锁
- 外键与锁

锁的算法

- Record lock：锁索引
- Gap lock：锁间隙
- Next-key lock： record + gap

查看锁请求

- 1. show full processlist命令
    - 观察state和info列
- 2. show engine innodb status\G 命令
    - 查看 TRANSACTIONS 部分和 LATEST DETECTED DEADLOCK 两个部分
- 3. information_schema下的三张表（通过这三张表可以更新监控当前事物并且分析存在的锁问题）
    - innodb_trx （ 打印innodb内核中的当前活跃（ACTIVE）事务）
    - innodb_locks （ 打印当前状态产生的innodb锁 仅在有锁等待时打印）
    - innodb_lock_waits （打印当前状态产生的innodb锁等待 仅在有锁等待时打印）

演示-锁

讨论-MySQL 事务处理

- MySQL为什么使用RR作为默认的隔离级别?
- MySQL的RR 完美的解决了幻读问题吗？如果解决了幻读，还需要SERIALIZABLE?
- MVCC 怎么实现 RC 和 RR ？
- 自增键的加锁过程?
- 外键值的插入和更新的加锁过程?

小结-MySQL 事务处理

- 隔离级别
- redo undo purge
- MVCC
- 锁和死锁

谢谢

下集预告

- 分布式系统的基本问题