Formalization of (Conflict-)Serializability and Strict Two-Phase Locking

Dmitriy Traytel

February 11, 2025

Abstract

Concurrency control is an essential component of any transactional database management system, which is responsible for providing isolation (the "I" in ACID) to transactions. Formally, concurrency control aims to achieve serializability: a way to rearrange the actions of concurrently executing transactions that eliminates concurrency while leaves the database modifications unchanged. In this small entry, we define serializability, a syntactic over-approximation called conflict-serializability, and characterize schedules generated by the frequently used concurrency control mechanism of strict two-phase locking (S2PL). We also prove two inclusions: S2PL implies conflict-serializability, which in turn implies serializability. The formalization is based on standard material from an advanced database systems course [1, Chapter 17].

1 Transactions

We work with a rather abstract model of transactions comprised of read/write actions.

Read/written values are natural numbers.

type-alias *val* = *nat*

Transactions *'xid* may read from/write two addresses *'addr*.

datatype (*'xid*, *'addr*) *action* = *isRead*: *Read* (*xid-of*: ‹*'xid*›) (*addr-of*: *'addr*)
| *isWrite*: *Write* (*xid-of*: ‹*'xid*›) (*addr-of*: *'addr*)

A schedule is a sequence of actions.

type-synonym (*'xid*, *'addr*) *schedule* = ‹(*'xid*, *'addr*) *action list*›

A database, which is being modified by the read/write actions, maps addresses to values.

type-synonym *'addr db* = ‹*'addr* ⇒ *val*›

Each transaction has a local state, which is represented as the list of previously read values (and the addresses they have been read from).

type-synonym *'addr xstate* = ‹(*'addr* × *val*) *list*›

The values written by a transaction are given by a higher-order parameter and may depend on the previously read values.

context fixes *write-logic* :: ‹*'xid* ⇒ *'addr xstate* ⇒ *'addr* ⇒ *val*› **begin**

Read values are recorded in the transaction's local state; writes modify the database.

fun *action-effect* :: ‹(*'xid*, *'addr*) *action* ⇒ (*'xid* ⇒ *'addr xstate*) × *'addr db* ⇒ (*'xid* ⇒ *'addr xstate*) × *'addr db*› **where**
‹*action-effect* (*Read xid addr*) (*xst*, *db*) = (*xst*(*xid* := (*addr*, *db addr*) # *xst xid*), *db*)›
| ‹*action-effect* (*Write xid addr*) (*xst*, *db*) = (*xst*, *db*(*addr* := *write-logic xid* (*xst xid*) *addr*))›

We are interested in how a schedule modifies the database (local state changes are discarded at the end).

definition *schedule-effect* :: ‹(*'xid*, *'addr*) *schedule* ⇒ *'addr db* ⇒*'addr db*› **where**
‹*schedule-effect s db* = *snd* (*fold action-effect s* (λ-. [], *db*))›

end

Actions that belong to the same transaction.

definition *eq-xid* **where**
‹*eq-xid a b* = (*xid-of a* = *xid-of b*)›

2 Serial and Serializable Schedules

declare *length-dropWhile-le*[*termination-simp*]

A serial schedule does not interleave actions of different transactions.

fun *serial* :: ‹(*'xid*, *'addr*) *schedule* ⇒ *bool*› **where**
‹*serial* [] = *True*›
| ‹*serial* (*a* # *as*) = (*let bs* = *dropWhile* (λ*b*. *eq-xid a b*) *as*
in serial bs ∧ *xid-of a* ∉ *xid-of* ‘ *set bs*)›

A schedule s can be rearranged into schedule t by a permutation π, which preserves the relative order of actions related by *eq*.

definition *permutes-upto* **where**
‹*permutes-upto eq* π *s t* =
(*bij-betw* π {..<*length s*} {..<*length t*} ∧
(∀ *i*<*length s*. *s* ! *i* = *t* ! π *i*) ∧
(∀ *i*<*length s*. ∀ *j*<*length s*. *i* < *j* ∧ *eq* (*s* ! *i*) (*s* ! *j*) ⟶ π *i* < π *j*))›

lemma *permutes-upto-Nil*[*simp*]: ‹*permutes-upto R* π [] []›
⟨*proof*⟩

Two schedules are equivalent if one can be rearranged into another without rearranging the actions of each transaction and in addition they have the same effect on any database for any fixed write logic.

abbreviation *equivalent* :: ‹$('xid, 'addr)$ *schedule* $\Rightarrow$ $('xid, 'addr)$ *schedule* $\Rightarrow$ *bool*› **where**
‹*equivalent s t* $\equiv$ ($\exists\,\pi$. *permutes-upto eq-xid* π *s t* $\wedge$ ($\forall$ *write-logic db*. *schedule-effect write-logic s db* = *schedule-effect write-logic t db*))›

A schedule is serializable if it is equivalent to some serial schedule. Serializable schedules thus provide isolation: even though actions of different transactions may be interleaved, the effect from the point of view of each transaction is as if the transaction was the only one executing in the system (as is the case in serial schedules).

definition *serializable* :: ‹$('xid, 'addr)$ *schedule* $\Rightarrow$ *bool*› **where**
‹*serializable s* = ($\exists\,t$. *serial t* $\wedge$ *equivalent s t*)›

3 Conflict Serializable Schedules

Two actions of different transactions are conflicting if they access the same address and at least one of them is a write.

definition *conflict* **where**
‹*conflict a b* = (*xid-of a* $\neq$ *xid-of b* $\wedge$ *addr-of a* = *addr-of b* $\wedge$ (*isWrite a* $\vee$ *isWrite b*))›

Two schedules are conflict-equivalent if one can be rearranged into another without rearranging conflicting actions or actions of one transaction. Note that unlike equivalence, the conflict-equivalence notion is purely syntactic, i.e., not talking about databases and action/schedule effects.

abbreviation *conflict-equivalent* :: ‹$('xid, 'addr)$ *schedule* $\Rightarrow$ $('xid, 'addr)$ *schedule* $\Rightarrow$ *bool*› **where**
‹*conflict-equivalent s t* $\equiv$ ($\exists\,\pi$. *permutes-upto* (*sup eq-xid conflict*) π *s t*)›

A schedule is conflict-serializable if it is conflict equivalent to some serial schedule.

definition *conflict-serializable* :: ‹$('xid, 'addr)$ *schedule* $\Rightarrow$ *bool*› **where**
‹*conflict-serializable s* = ($\exists\,t$. *serial t* $\wedge$ *conflict-equivalent s t*)›

4 Conflict-Serializability Implies Serializability

In the following, we prove that the syntactic notion implies the semantic one. The key obsevation is that swapping non-conflicting actions of different transactions preserves the overall effect on the database.

lemma *swap-actions*: ‹$\neg$ *conflict a b* $\Longrightarrow$ $\neg$ *eq-xid a b* $\Longrightarrow$
action-effect wl a (*action-effect wl b st*) = *action-effect wl b* (*action-effect wl a st*)›
⟨*proof*⟩

lemma *swap-many-actions*: ‹$\forall$ *i* < *length p*. $\neg$ *conflict a* (*p* ! *i*) $\wedge$ $\neg$ *eq-xid a* (*p* ! *i*) $\Longrightarrow$
action-effect wl a (*fold* (*action-effect wl*) *p st*) = *fold* (*action-effect wl*) *p* (*action-effect wl a st*)›
⟨*proof*⟩

lemma *fold-action-effect-eq*:
assumes ‹*t* = *p* @ *a* # *u*›
shows
‹*fold* (*action-effect wl*) *s* (*action-effect wl a st*) =
fold (*action-effect wl*) (*p* @ *u*) (*action-effect wl a st*) $\Longrightarrow$
$\forall$ *i* < *length p*. $\neg$ *conflict a* (*p* ! *i*) $\wedge$ $\neg$ *eq-xid a* (*p* ! *i*) $\Longrightarrow$
fold (*action-effect wl*) (*a* # *s*) *st* = *fold* (*action-effect wl*) *t st*›
⟨*proof*⟩

definition *shift* **where**
‹*shift* π = (λ*i*. *if i* < π *0 then i else i* − *1*) *o* π *o Suc*›

lemma *bij-betw-remove*: ‹*bij-betw f A B* $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *bij-betw f* (*A* − {*x*}) (*B* − {*f x*})›
⟨*proof*⟩

lemma *permutes-upto-shift*:
assumes ‹*permutes-upto eq* π (*a* # *s*) *t*›
shows ‹*permutes-upto eq* (*shift* π) *s* (*take* (π *0*) *t* @ *drop* (*Suc* (π *0*)) *t*)›
⟨*proof*⟩

lemma *permutes-upto-prefix-upto*:
assumes ‹*permutes-upto eq* π (*t* ! π *0* # *s*) *t*› ‹*i* < π *0*›
shows ‹$\neg$ *eq* (*t* ! π *0*) (*t* ! *i*)›
⟨*proof*⟩

lemma *conflict-equivalent-imp-equivalent*:
assumes ‹*conflict-equivalent s t*›
shows ‹*equivalent s t*›
⟨*proof*⟩

theorem *conflict-serializable-imp-serializable*: ‹*conflict-serializable s* $\Longrightarrow$ *serializable s*›
⟨*proof*⟩

5 Schedules Generated by Strict Two-Phase Locking (S2PL).

To enforce conflict-serializability database management systems use locks. Locks come in two kinds: shared locks for reads and exclusive locks for writes. An address can be accessed in a reading fashion by multiple transactions, each holding a shared lock. If one transaction however holds an exclusive locks to write to an address, then no other transaction can hold a lock (neither shared nor exclusive) for the same address.

datatype $'addr\ lock = S\ (addr\text{-}of\colon 'addr) \mid X\ (addr\text{-}of\colon 'addr)$

fun *lock-for* **where**
$\langle lock\text{-}for\ (Read\ \text{-}\ addr) = S\ addr\rangle$
$\mid \langle lock\text{-}for\ (Write\ \text{-}\ addr) = X\ addr\rangle$

definition *valid-locks* **where**
$\langle valid\text{-}locks\ locks = (\forall\, addr\ xid1\ xid2.\ X\ addr \in locks\ xid1 \longrightarrow$
$X\ addr \in locks\ xid2 \lor S\ addr \in locks\ xid2 \longrightarrow xid1 = xid2)\rangle$

A frequently used lock strategy is strict two phase locking (S2PL) in which transactions attempt to acquire locks gradually (whenever they want to execute an action that needs a particular lock) and release them all at once at the end of each transaction.

The following predicate checks whether a schedule could have been generated using the S2PL strategy. To this end, the predicate checks for each action, whether the corresponding lock could have been acquired by the transaction executing the action. We also allow lock upgrades (from shared to exclusive), i.e., one transaction can hold both a shared and and exclusive lock

As in our model there is no explicit transaction end marker (commit), we treat each transaction as finished immediately when it has executed its last action in the given schedule. This is the moment, when the transaction's locks are released.

fun $s2pl :: \langle('xid \Rightarrow 'addr\ lock\ set) \Rightarrow ('xid, 'addr)\ schedule \Rightarrow bool\rangle$ **where**
$\langle s2pl\ locks\ [] = True\rangle$
$\mid \langle s2pl\ locks\ (a\ \#\ s) =$
$(let\ xid = xid\text{-}of\ a;\ addr = action.addr\text{-}of\ a$
$in\ if\ \exists\, xid'.\ xid' \neq xid \land (X\ addr \in locks\ xid' \lor isWrite\ a \land S\ addr \in locks\ xid')$
$then\ False$
$else\ s2pl\ (locks(xid := if\ xid \notin xid\text{-}of\ `\ set\ s\ then\ \{\}\ else\ locks\ xid \cup \{lock\text{-}for\ a\}))\ s)\rangle$

We prove in the following that S2PL schedules are conflict-serializable (and thus also serializable). The proof proceeds by induction on the number of transactions in a schedule. To construct the conflict-equivalent serial schedule we always move the actions of the transaction that finished first in our S2PL schedule to the front. To do so we show that these actions are not conflicting with any preceding actions (due to the acquired/held locks).

lemma *conflict-equivalent-trans*:
‹*conflict-equivalent s t* $\Longrightarrow$ *conflict-equivalent t u* $\Longrightarrow$ *conflict-equivalent s u*›
⟨*proof*⟩

lemma *conflict-equivalent-append*: ‹*conflict-equivalent s t* $\Longrightarrow$ *conflict-equivalent* (*u* @ *s*) (*u* @ *t*)›
⟨*proof*⟩

lemma *conflict-equivalent-Cons*: ‹*conflict-equivalent s t* $\Longrightarrow$ *conflict-equivalent* (*a* # *s*) (*a* # *t*)›
⟨*proof*⟩

lemma *conflict-equivalent-rearrange*:
assumes ‹$\bigwedge$*i j*. *xid-of* (*s* ! *i*) = *xid* $\Longrightarrow$ *j* < *i* $\Longrightarrow$ *i* < *length s* $\Longrightarrow$ ¬ *conflict* (*s* ! *j*) (*s* ! *i*)›
shows ‹*conflict-equivalent s* (*filter* ((=) *xid* ∘ *xid-of*) *s* @ *filter* (*Not* ∘ (=) *xid* ∘ *xid-of*) *s*)›
(**is** ‹*conflict-equivalent s* (*?filter s*)›)
⟨*proof*⟩

lemma *serial-append*:
‹*serial s* $\Longrightarrow$ *serial t* $\Longrightarrow$ *xid-of* ‘ *set s* ∩ *xid-of* ‘ *set t* = {} $\Longrightarrow$ *serial* (*s* @ *t*)›
⟨*proof*⟩

lemma *serial-same-xid*: ‹∀ *x* ∈ *set s*. *xid-of x* = *xid* $\Longrightarrow$ *serial s*›
⟨*proof*⟩

lemma *conflict-equivalent-same-set*: ‹*conflict-equivalent s t* $\Longrightarrow$ *set s* = *set t*›
⟨*proof*⟩

lemma *s2pl-filter*:
‹*s2pl locks s* $\Longrightarrow$ *s2pl* (*locks*(*xid* := {})) (*filter* (*Not o* (=) *xid o xid-of*) *s*)›
⟨*proof*⟩

lemma *valid-locks-grab*[*simp*]: ‹*valid-locks locks* $\Longrightarrow$
¬ (∃ *xid'*. *xid'* ≠ *xid-of a* ∧
(*X* (*action.addr-of a*) ∈ *locks xid'* ∨ *isWrite a* ∧ *S* (*action.addr-of a*) ∈ *locks xid'*)) $\Longrightarrow$
valid-locks (*locks*(*xid-of a* := *insert* (*lock-for a*) (*locks* (*xid-of a*))))›
⟨*proof*⟩

lemma *s2pl-suffix*: ‹*valid-locks locks* $\Longrightarrow$ *s2pl locks* (*s* @ *t*) $\Longrightarrow$
∀ *a* ∈ *set s*. ∃ *b* ∈ *set t*. *eq-xid a b* $\Longrightarrow$
∃ *locks'*. *valid-locks locks'* ∧ (∀ *xid*. *locks xid* ⊆ *locks' xid*) ∧ *s2pl locks' t*›
⟨*proof*⟩

lemma *set-drop*: ‹$l \leq$ *length xs* $\Longrightarrow$ *set* (*drop l xs*) = *nth xs* ‘ {*l..<length xs*}›
⟨*proof*⟩

lemma *drop-eq-Cons*: ‹$i <$ *length xs* $\Longrightarrow$ *drop i xs* = *xs* ! *i* # *drop* (*Suc i*) *xs*›
⟨*proof*⟩

theorem *s2pl-conflict-serializable*: ‹*s2pl* (λ-. {}) *s* $\Longrightarrow$ *conflict-serializable s*›
⟨*proof*⟩

corollary *s2pl-serializable*: ‹*s2pl* (λ-. {}) *s* $\Longrightarrow$ *serializable s*›
⟨*proof*⟩

6 Example Executing S2PL

To make the S2PL check executable regardless of the transaction id type, we restrict the quantification to transaction ids that are occurring in the schedule.

fun *s2pl-code* :: ‹(*'xid* ⇒ *'addr lock set*) ⇒ (*'xid*, *'addr*) *schedule* ⇒ *bool*› **where**
‹*s2pl-code locks* [] = *True*›
| ‹*s2pl-code locks* (*a* # *s*) =
(*let xid* = *xid-of a*; *addr* = *action.addr-of a*
in if ∃ *xid′* ∈ *xid-of* ‘ *set s*. *xid′* ≠ *xid* ∧ (*X addr* ∈ *locks xid′* ∨ *isWrite a* ∧ *S addr* ∈ *locks xid′*)
then False
else s2pl-code (*locks*(*xid* := *if xid* ∉ *xid-of* ‘ *set s then* {} *else locks xid* ∪ {*lock-for a*})) *s*)›

lemma *s2pl-code-cong*: (⋀*xid*. *xid* ∈ *xid-of* ‘ *set s* $\Longrightarrow$ *f xid* = *g xid*) $\Longrightarrow$
(⋀*xid*. *xid* ∉ *xid-of* ‘ *set s* $\Longrightarrow$ *f xid* = {}) $\Longrightarrow$
s2pl f s = *s2pl-code g s*
⟨*proof*⟩

lemma *s2pl-code*[*code-unfold*]: *s2pl* (λ-. {}) *s* = *s2pl-code* (λ-. {}) *s*
⟨*proof*⟩

definition *TB* = (*0* :: *nat*)
definition *TA* = (*1* :: *nat*)
definition *TC* = (*2* :: *nat*)
definition *AX* = (*0* :: *nat*)
definition *AY* = (*1* :: *nat*)
definition *AZ* = (*2* :: *nat*)

Good example involving a lock upgrade by *TA* and *TB*

lemma ‹*s2pl* (λ-. {})
[*Write TB AZ*, *Read TA AX*, *Read TB AY*, *Read TC AX*, *Write TB AY*, *Write TC AY*, *Write TA AX*, *Write TA AY*]›
⟨*proof*⟩

Bad example: TC cannot acquire exclusive lock for AY, which is already held by TA

lemma ‹¬ *s2pl* (λ-. {})
 [*Read TA AX*, *Read TB AX*, *Read TC AX*, *Write TA AY*, *Write TC AY*, *Write TB AY*, *Write TA AZ*]›
 ⟨*proof*⟩

hide-const *TB TA TC AX AY AZ*
hide-fact *TB-def TA-def TC-def AX-def AY-def AZ-def*

References

[1] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill Higher Education, 3rd edition, January 2003.