Area	All-or-nothing atomicity	Before-or-after atomicity
database management	updating more than one record	records shared between threads
hardware architecture	handling interrupts and exceptions	register renaming
operating systems	supervisor call interface	printer queue
software engineering	handling faults in layers	bounded buffer

procedure TRANSFER (*debit_account*, *credit_account*, *amount*)

GET (*dbdata, debit_account*)

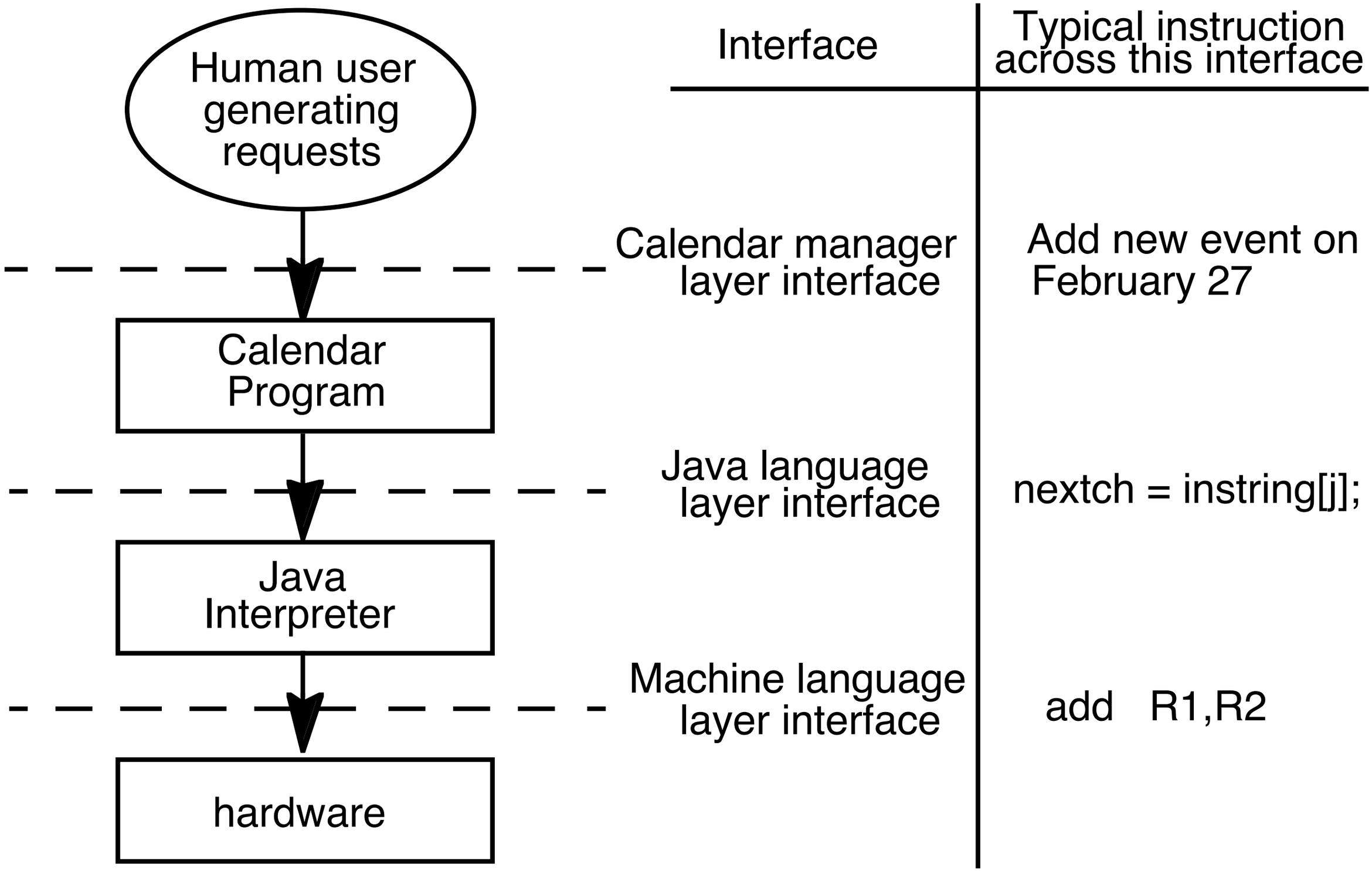
dbdata ← *dbdata* - *amount*

PUT (*dbdata*, *debit_account*)

GET (*crdata, credit_account*)

crdata← *crdata* + *amount*

PUT (*crdata, credit_account*)



All-or-nothing atomicity

A sequence of steps is an *all-or-nothing action* if, from the point of view of its invoker, the sequence always either

completes,

or

aborts in such a way that it appears that the sequence had never been undertaken in the first place. That is, it *backs out.*

Before-or-after atomicity

Concurrent actions have the *before-or-after* property if their effect from the point of view of their invokers is the same as if the actions occurred either *completely before* or *completely after* one another.

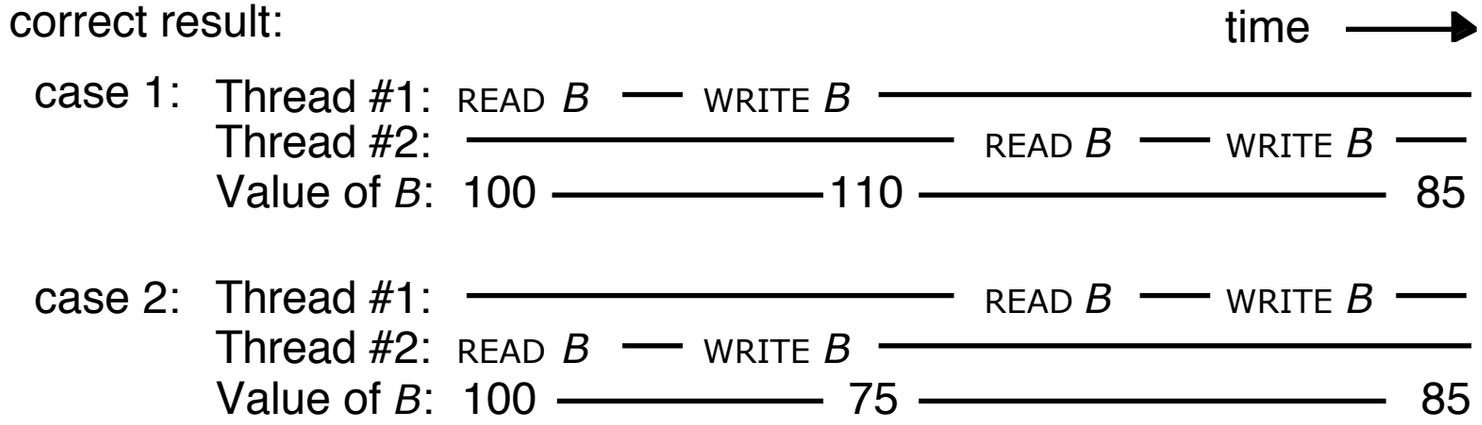
procedure TRANSFER (**reference** *debit_account*, **reference** *credit_account*, *amount*)
    debit_account ← *debit_account* - *amount*
    credit_account ← *credit_account* + *amount*

TRANSFER (*A*, *B*, $10)

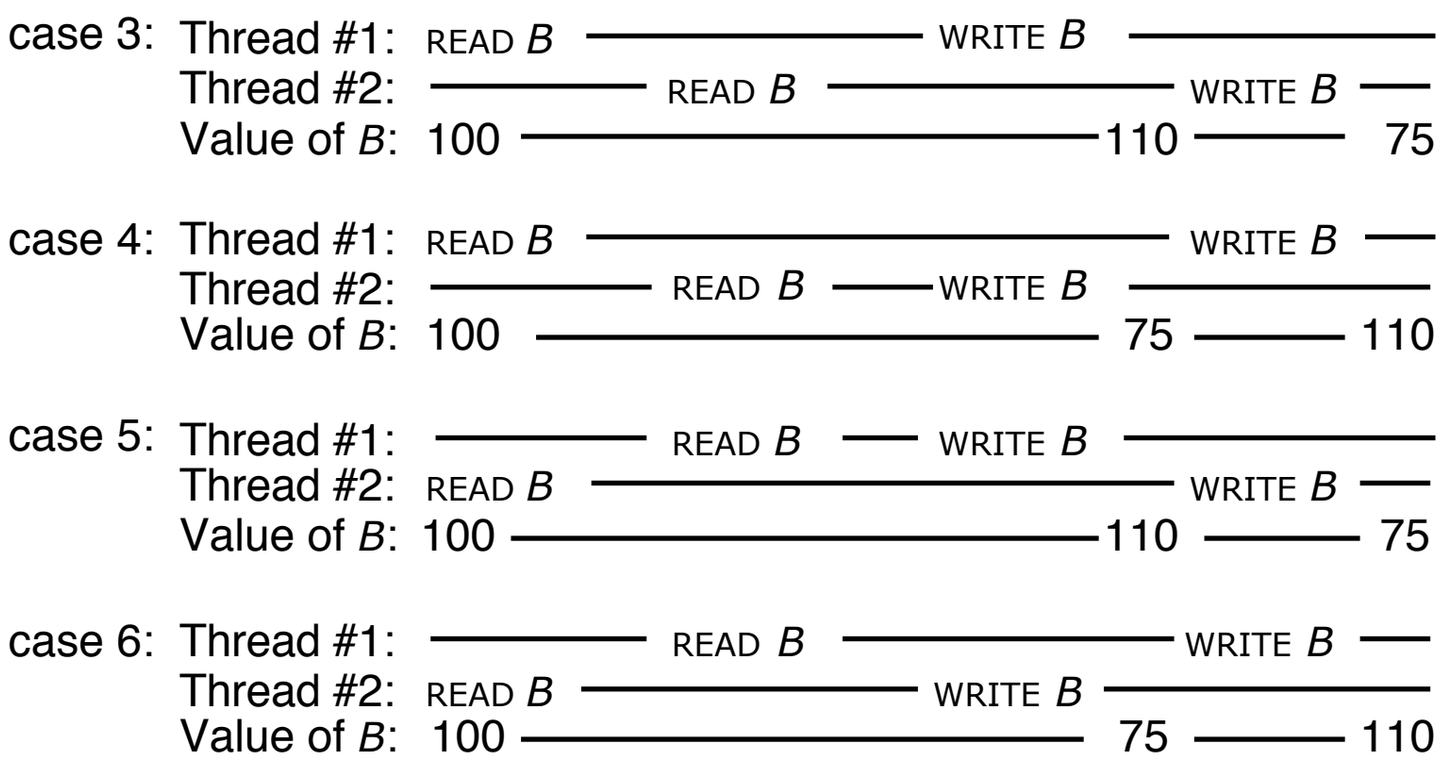
TRANSFER (*B*, *C*, $25)

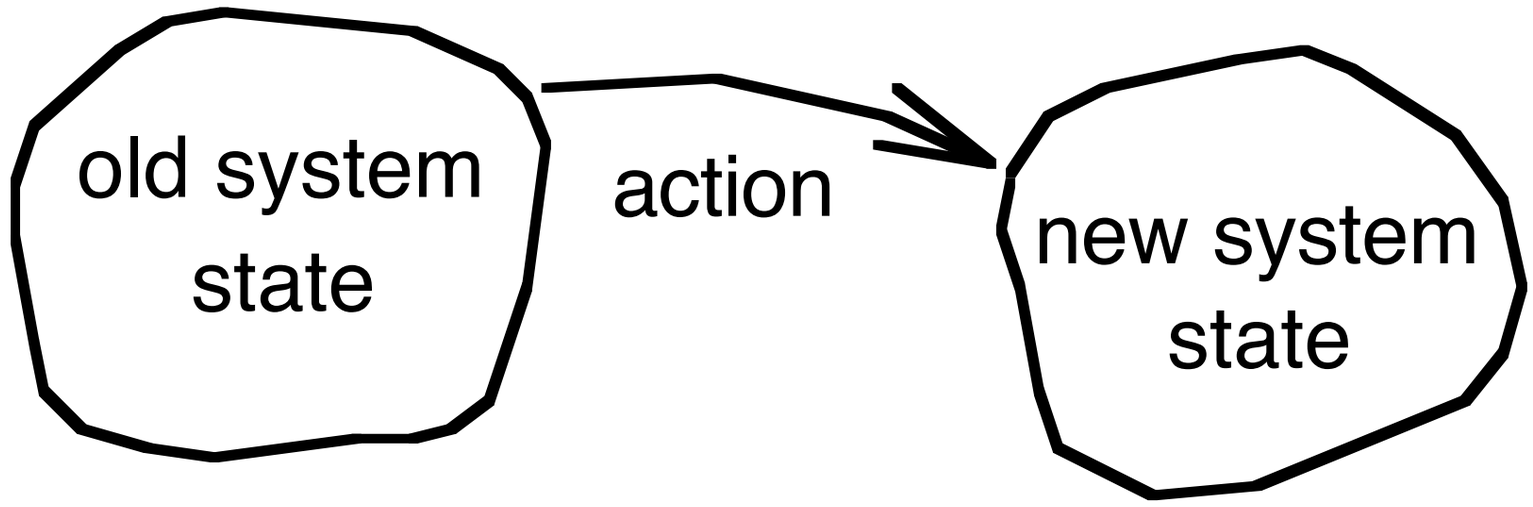
Thread #1 (*credit_account* is *B*)	Thread #2 (*debit_account* is B)
1–1 READ *B*	2–1 READ *B*
.	.
.	.
.	.
1–2 WRITE *B*	2–2 WRITE *B*

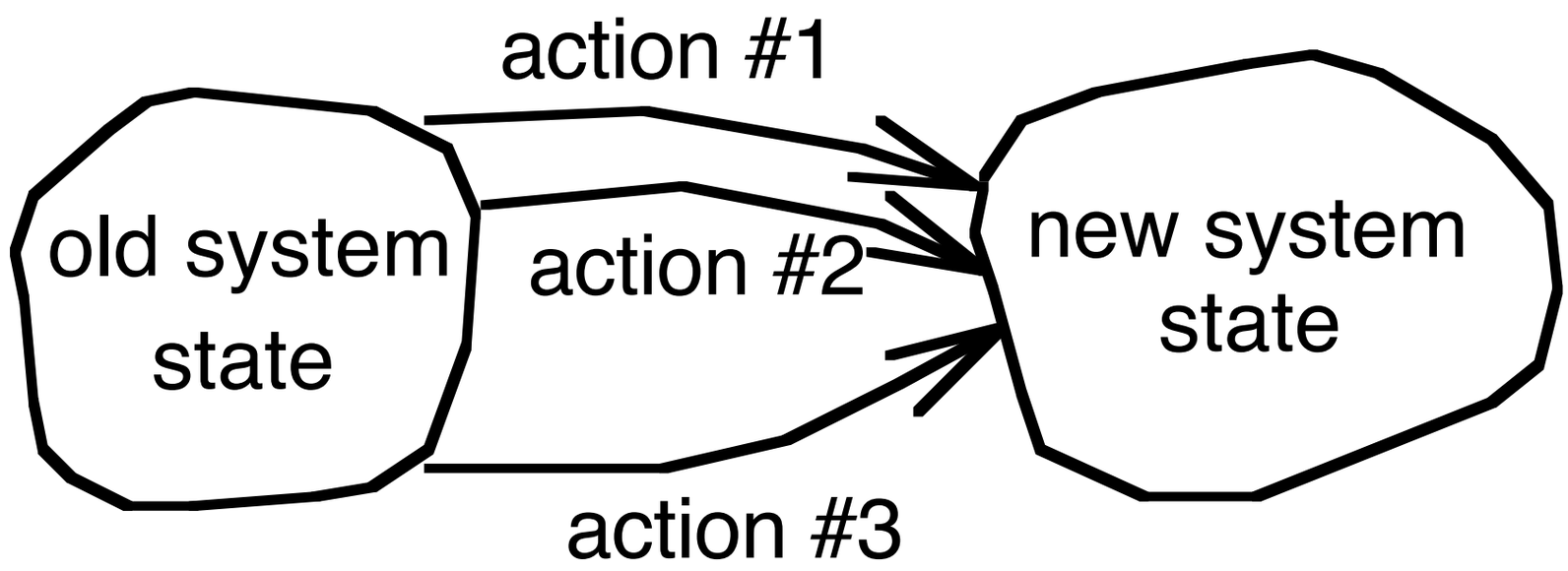
correct result:

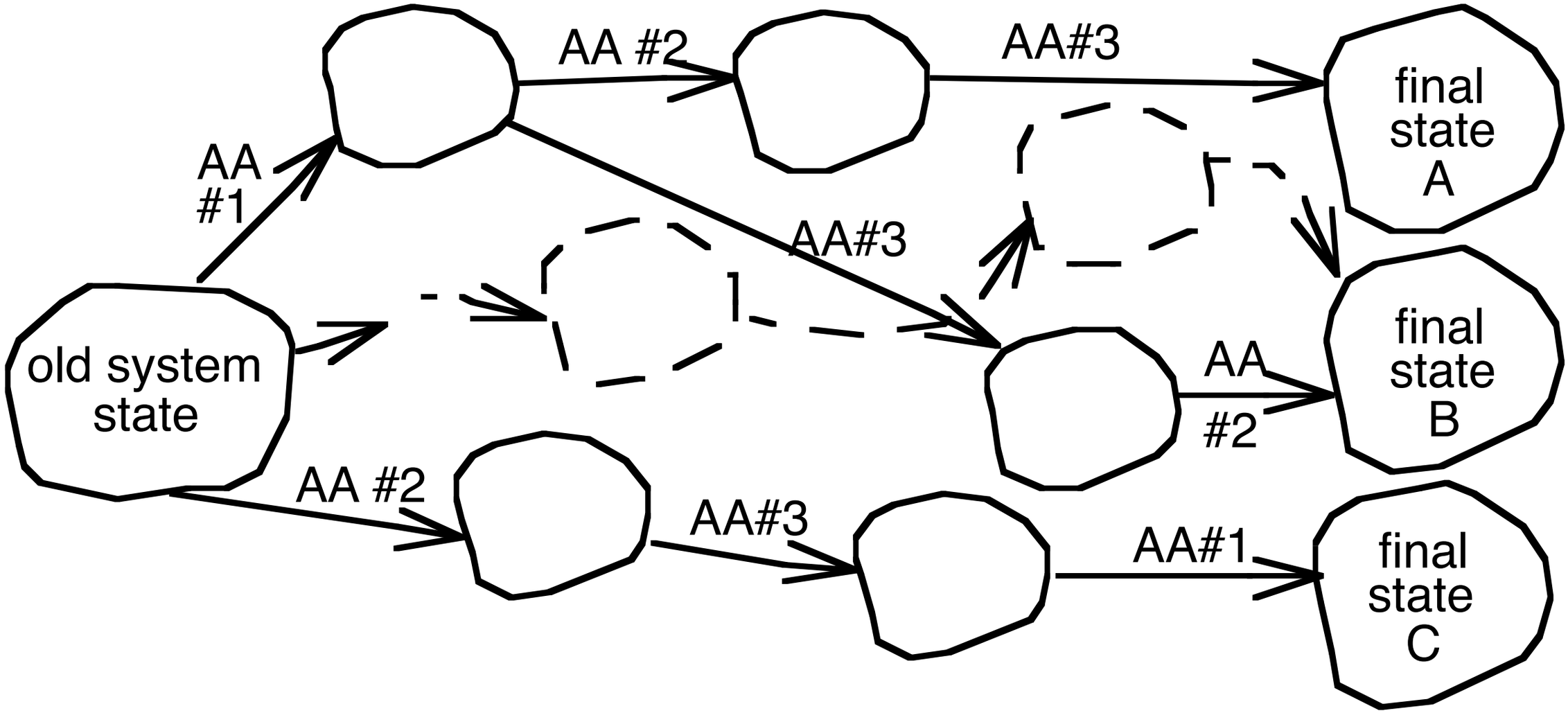


wrong results:









```
procedure AUDIT()
    sum ← 0
    for each W ← in bank.accounts
        sum ← sum + W.balance
    if (sum ≠ 0) call for investigation
```

```
// TRANSFER, in thread 1

    debit_account ← debit_account - amount
   ...
    credit_account ← credit_account + amount
```

```
// in thread 2

...
AUDIT()
...
```

Atomicity

An action is atomic if there is no way for a higher layer to discover the internal structure of its implementation.

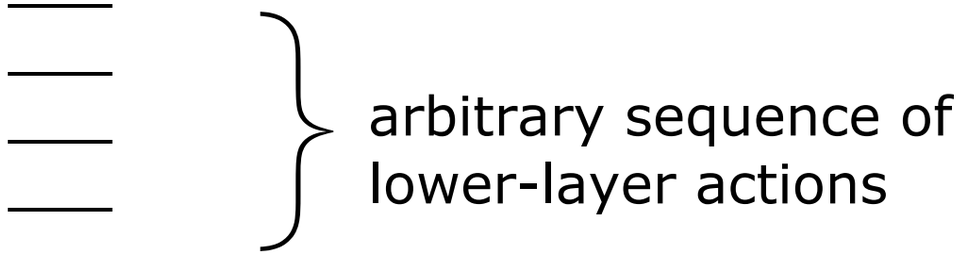
```
procedure ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
    CAREFUL_PUT (data, all_or_nothing_sector.S1)
    CAREFUL_PUT (data, all_or_nothing_sector.S2)          // Commit point.
    CAREFUL_PUT (data, all_or_nothing_sector.S3)

procedure ALL_OR_NOTHING_GET (reference data, all_or_nothing_sector)
    CAREFUL_GET (data1, all_or_nothing_sector.S1)
    CAREFUL_GET (data2, all_or_nothing_sector.S2)
    CAREFUL_GET (data3, all_or_nothing_sector.S3)
    if data1 = data2 then data ← data1                    // Return new value.
    else data ← data3                                     // Return old value.
```

```
procedure ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
    CHECK_AND_REPAIR (all_or_nothing_sector)
    ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)

procedure CHECK_AND_REPAIR (all_or_nothing_sector)// Ensure copies match.
    CAREFUL_GET (data1, all_or_nothing_sector.S1)
    CAREFUL_GET (data2, all_or_nothing_sector.S2)
    CAREFUL_GET (data3, all_or_nothing_sector.S3)
    if (data1 = data2) and (data2 = data3) return    // State 1 or 7, no repair
    if (data1 = data2)
        CAREFUL_PUT (data1, all_or_nothing_sector.S3) return   // State 5 or 6.
    if (data2 = data3)
        CAREFUL_PUT (data2, all_or_nothing_sector.S1) return   // State 2 or 3.
    CAREFUL_PUT (data1, all_or_nothing_sector.S2)      // State 4, go to state 5
    CAREFUL_PUT (data1, all_or_nothing_sector.S3)      // State 5, go to state 7
```

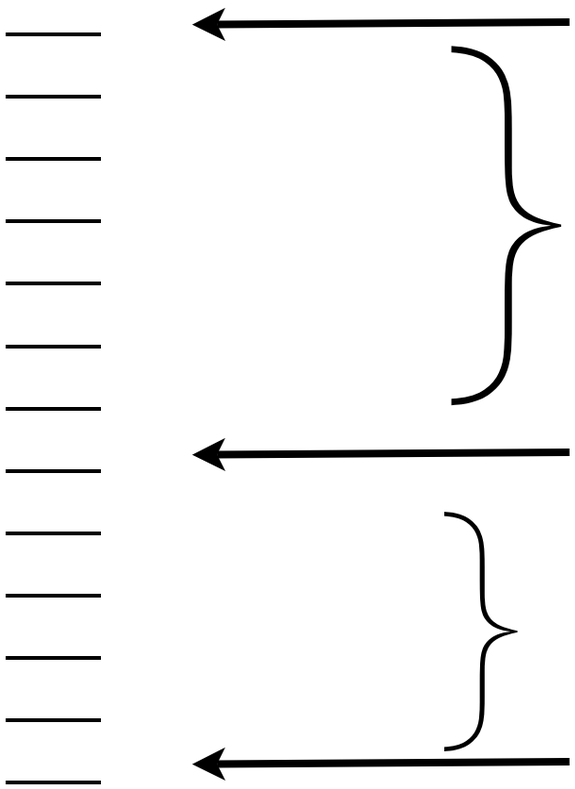
---

data state:	1	2	3	4	5	6	7
sector *S1*	old	bad	new	new	new	new	new
sector *S2*	old	old	old	bad	new	new	new
sector *S3*	old	old	old	old	old	bad	new

begin all-or-nothing action



end all-or-nothing action



first step of all-or-nothing action

Pre-commit discipline: can back out, leaving no trace

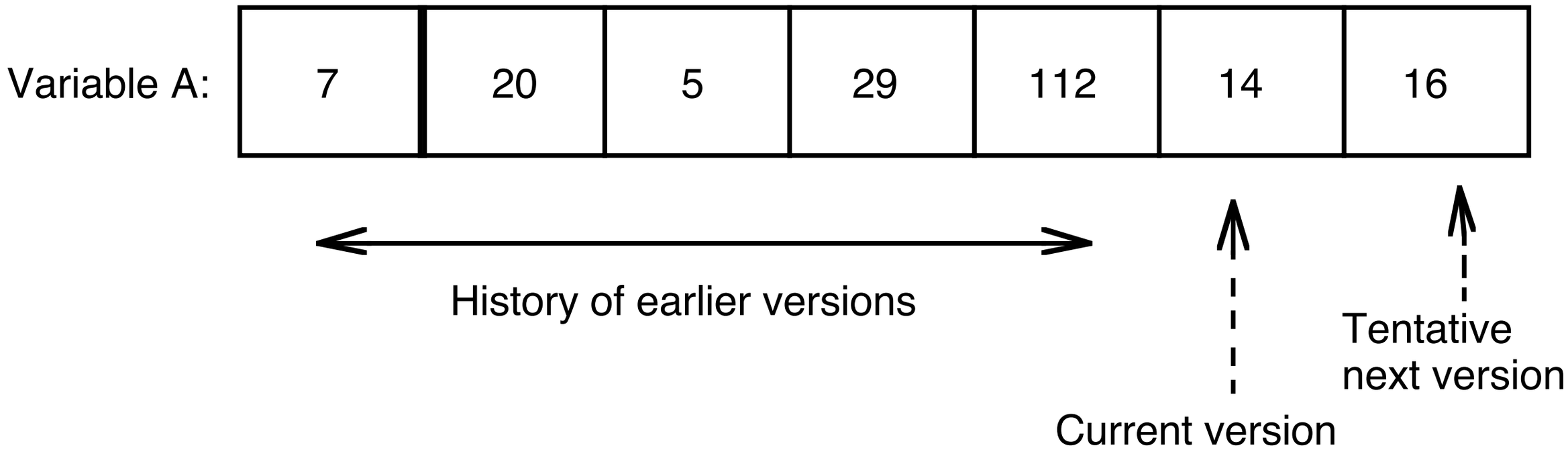
Commit point

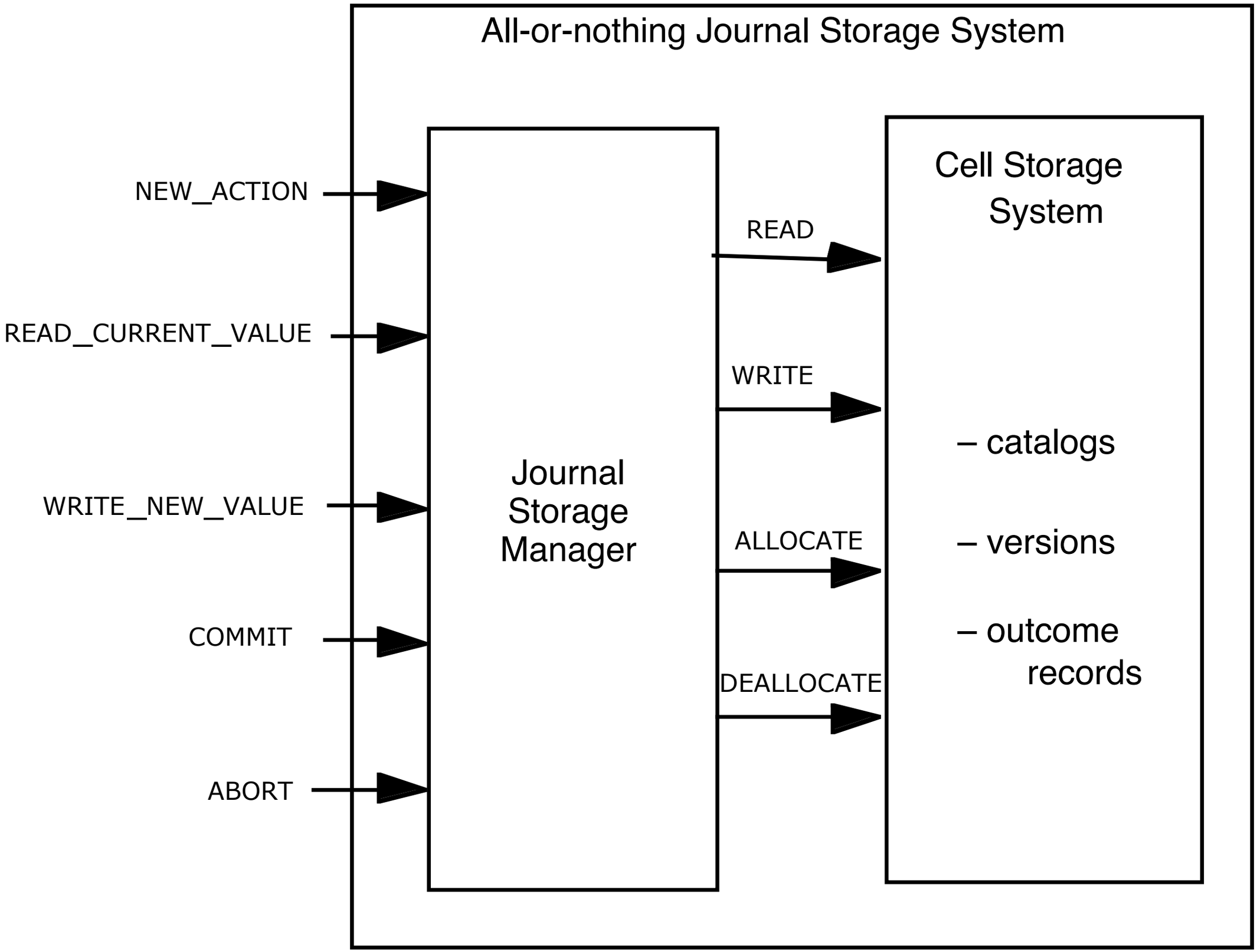
Post-commit discipline: completion is inevitable

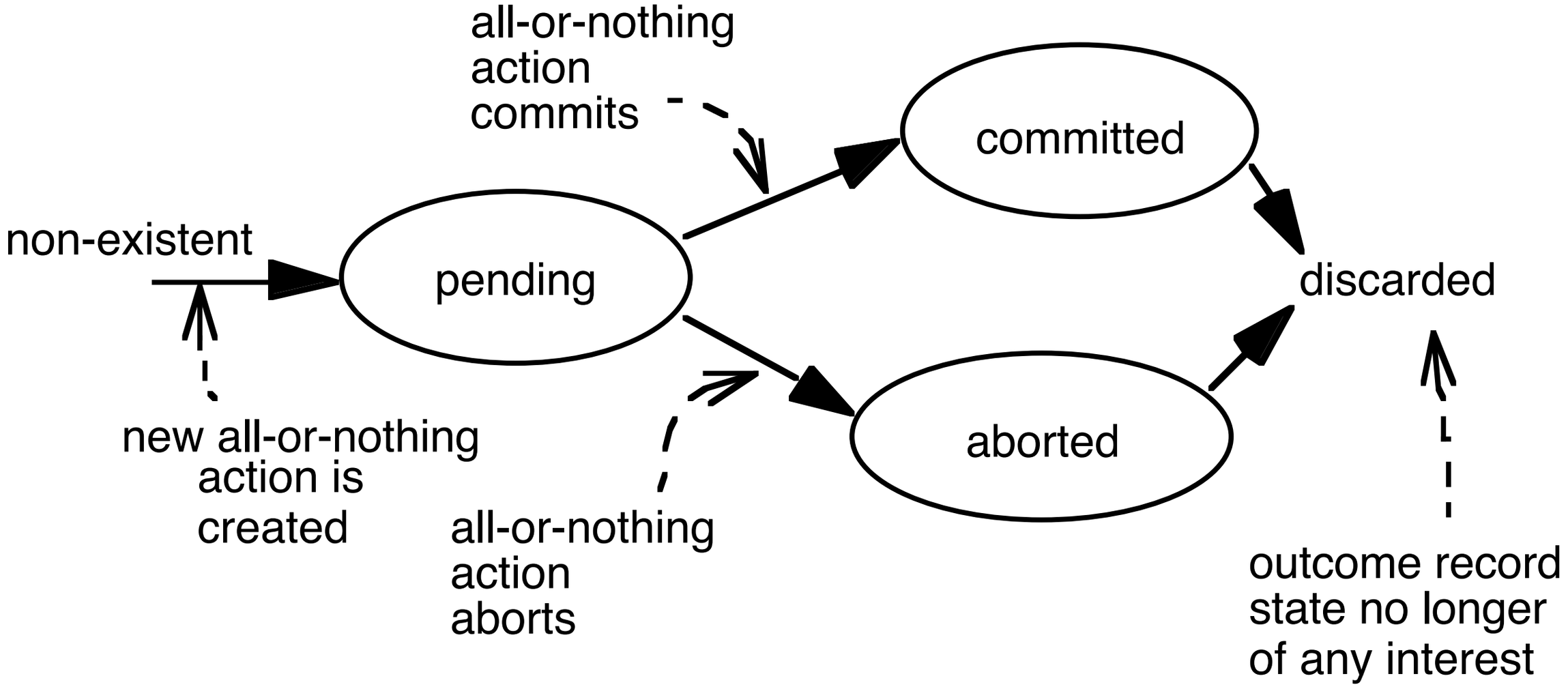
last step of all-or-nothing action

The golden rule of atomicity

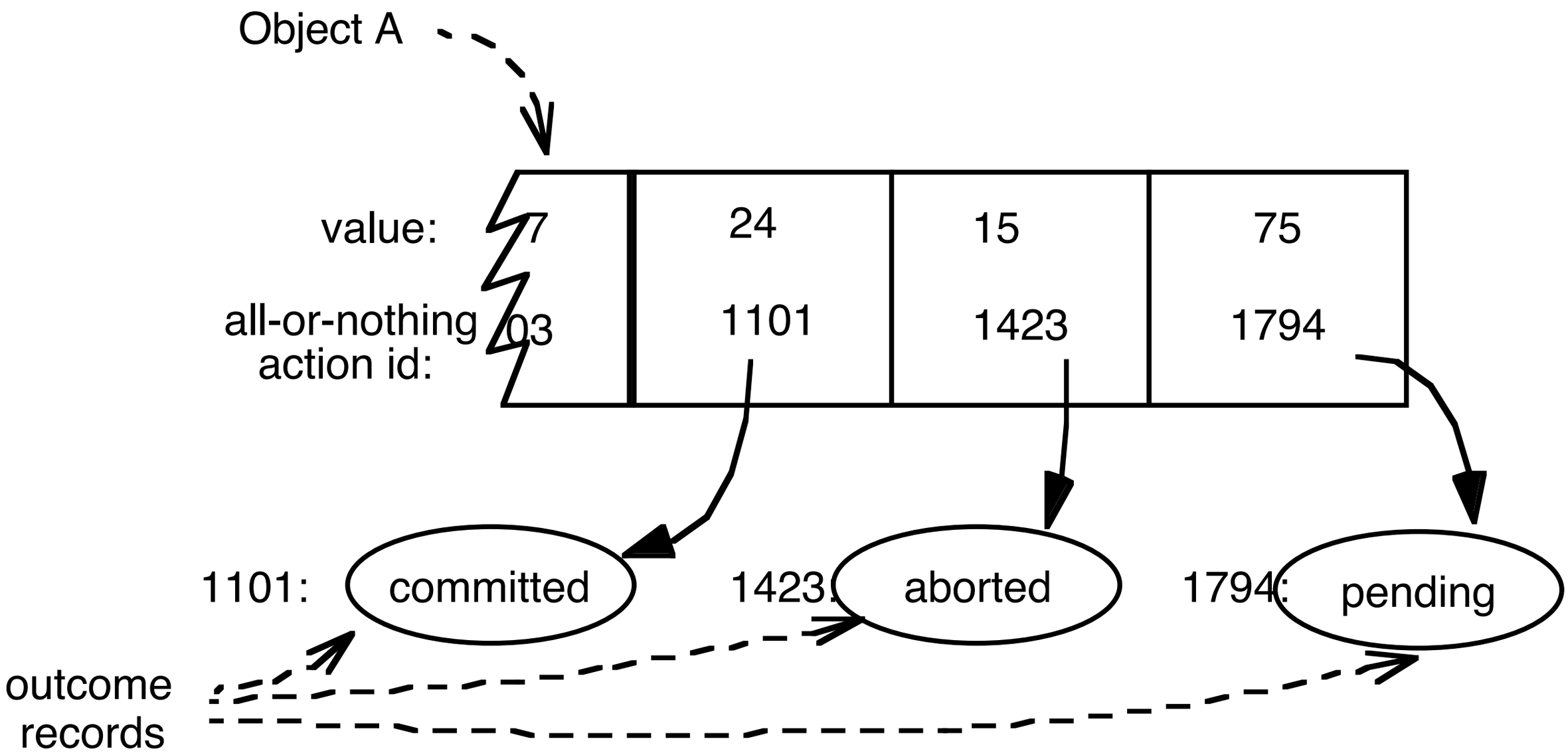
Never modify the only copy!





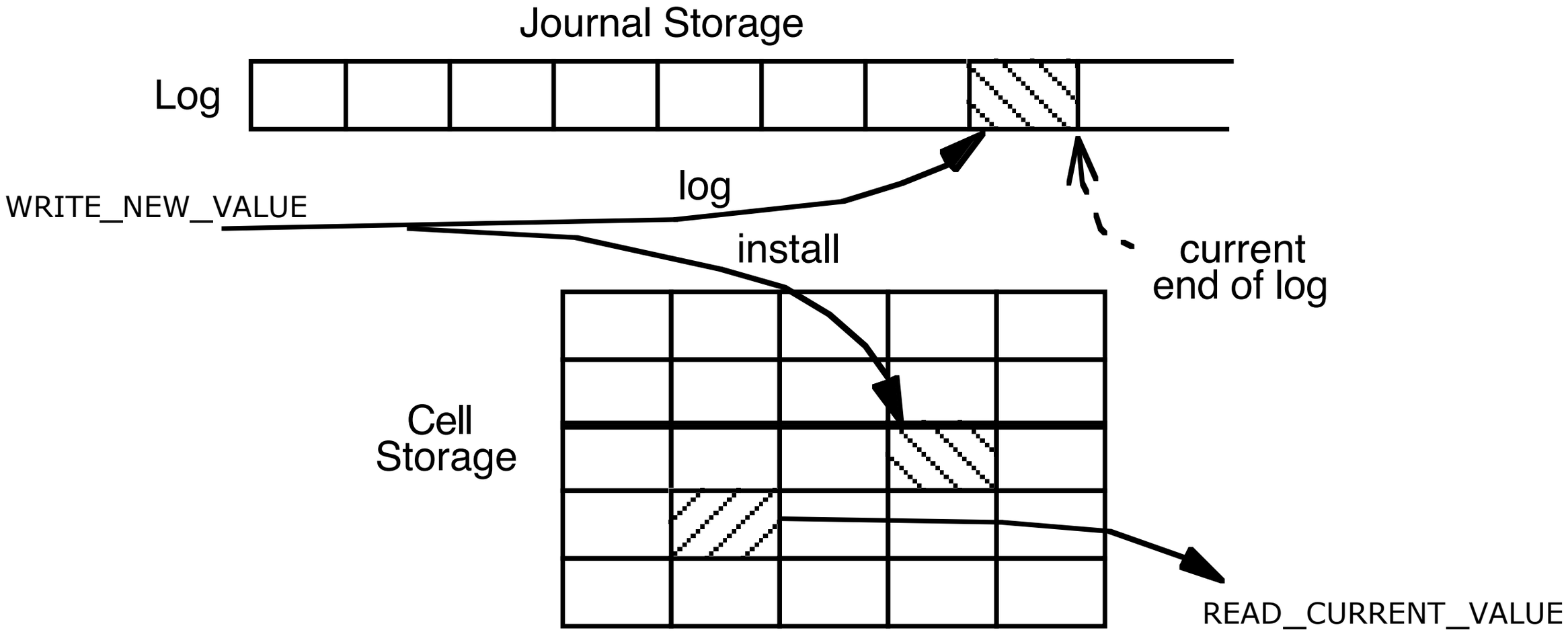


```
procedure NEW_ACTION ()
    id ← NEW_OUTCOME_RECORD ()
    id.outcome_record.state ← PENDING
    return id

procedure COMMIT (reference id)
    id.outcome_record.state ← COMMITTED

procedure ABORT (reference id)
    id.outcome_record.state ← ABORTED
```



```
procedure READ_CURRENT_VALUE (data_id, caller_id)
    starting at end of data_id repeat until beginning
        v ← previous version of data_id        // Get next older version
        a ← v.action_id  // Identify the action a that created it
        s ← a.outcome_record.state             // Check action a's outcome record
        if s = COMMITTED then
            return v.value
        else skip v                            // Continue backward search
    signal ("Tried to read an uninitialized variable!")

procedure WRITE_NEW_VALUE (reference data_id, new_value, caller_id)
    if caller_id.outcome_record.state = PENDING
        append new version v to data_id
        v.value ← new_value
        v.action_id ← caller_id
                    else signal ("Tried to write outside of an all-or-nothing action!")
```

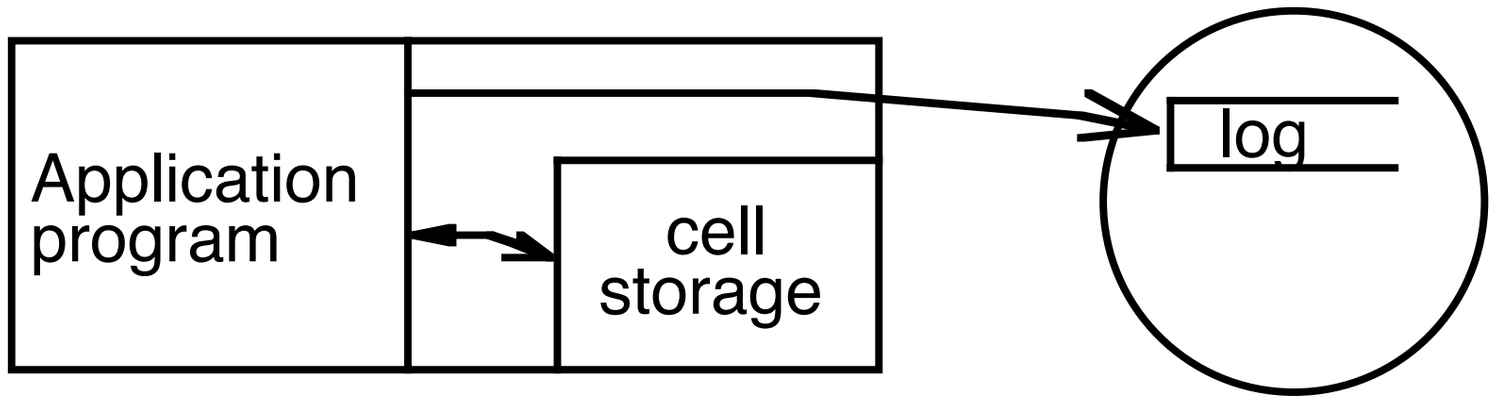
```
procedure TRANSFER (reference debit_account, reference credit_account, amount)
    my_id ← NEW_ACTION ()
    xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
    xvalue ← xvalue - amount
    WRITE_NEW_VALUE (debit_account, xvalue, my_id)
    yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
    yvalue ← yvalue + amount
    WRITE_NEW_VALUE (credit_account, yvalue, my_id)
    if xvalue > 0 then
        COMMIT (my_id)
    else
        ABORT (my_id)
        signal("Negative transfers are not allowed.")
```



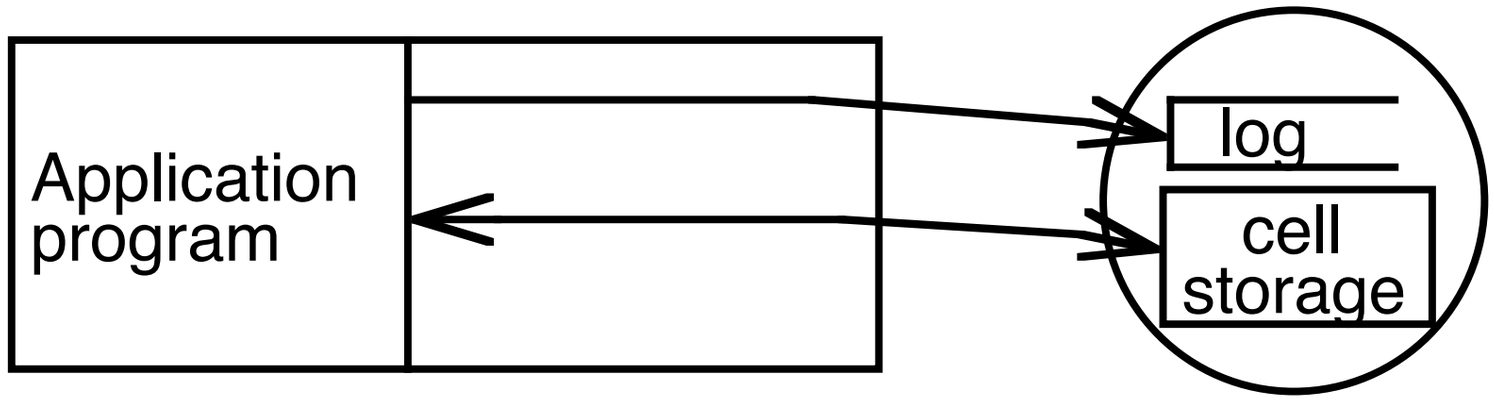
Volatile storage

Non-volatile storage

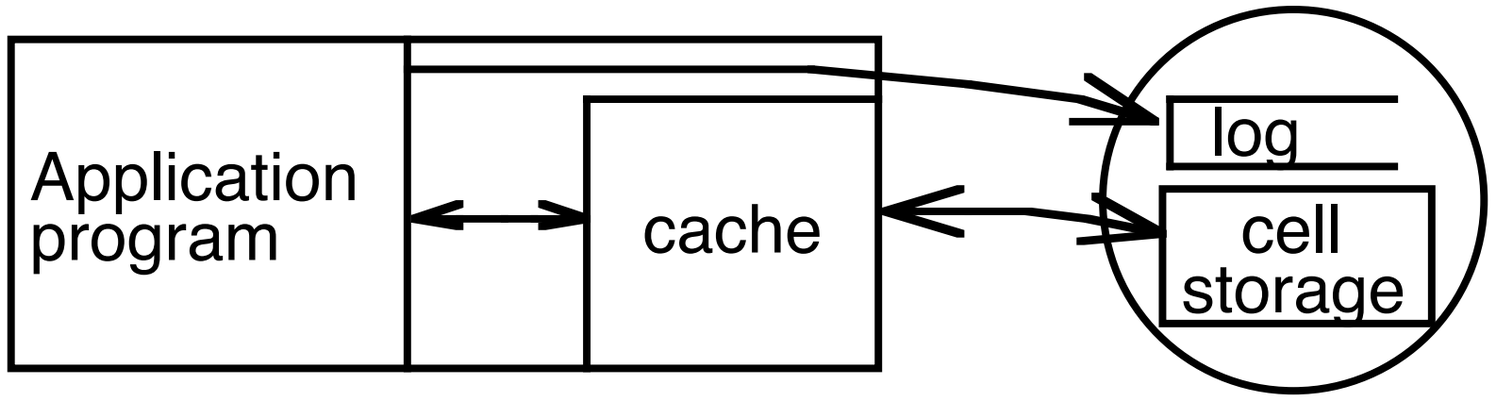
In-memory database:



Ordinary database:



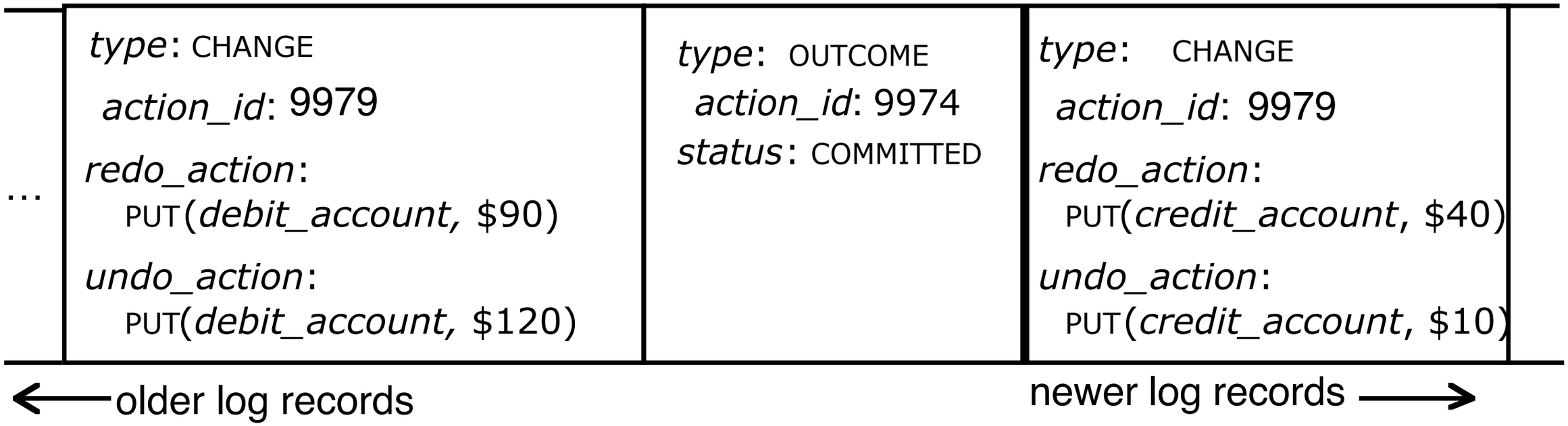
High-performance database:



Write-ahead-log protocol

Log the update *before* installing it.

```
procedure TRANSFER (debit_account, credit_account, amount)
    my_id ← LOG (BEGIN_TRANSACTION)
    dbvalue.old ← GET (debit_account)
    dbvalue.new ← dbvalue.old - amount
    crvalue.old ← GET (credit_account, my_id)
    crvalue.new ← crvalue.old + amount
    LOG (CHANGE, my_id,
        "PUT (debit_account, dbvalue.new)",                    //redo action
        "PUT (debit_account, dbvalue.old)" )                   //undo action
    LOG ( CHANGE, my_id,
        "PUT (credit_account, crvalue.new)"                    //redo action
        "PUT (credit_account, crvalue.old)")                   //undo action
PUT (debit_account, dbvalue.new)                               // install
PUT (credit_account, crvalue.new)                              // install
if dbvalue.new > 0 then
    LOG ( OUTCOME, COMMIT, my_id)
else
    LOG (OUTCOME, ABORT, my_id)
    signal("Action not allowed. Would make debit account negative.")
LOG (END_TRANSACTION, my_id)
```



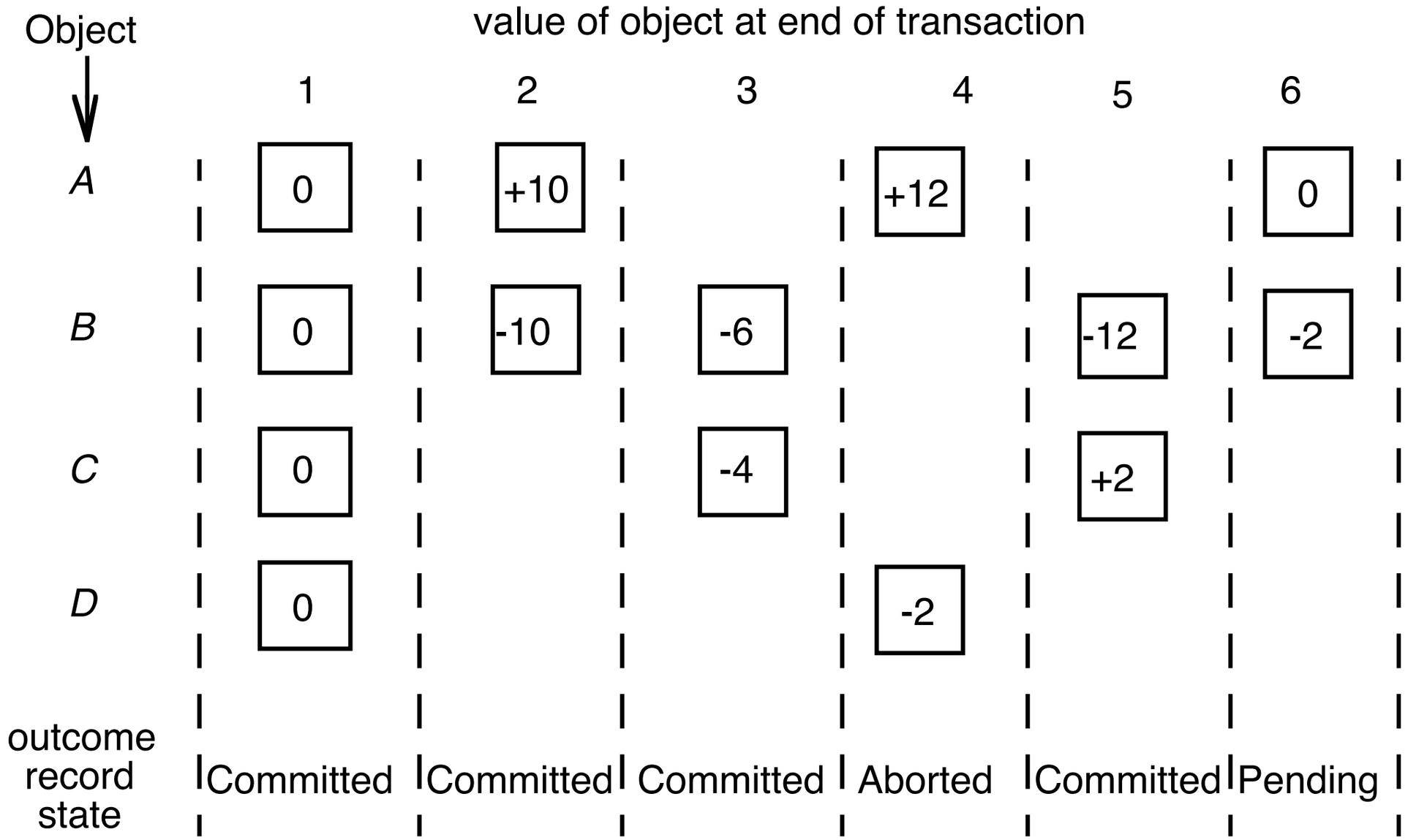
```
procedure ABORT (action_id)
    starting at end of log repeat until beginning
        log_record ← previous record of log
        if log_record.id = action_id then
            if (log_record.type = OUTCOME)
                then signal ("Can't abort an already completed action.")
            if (log_record.type = CHANGE)
                then perform undo_action of log_record
            if (log_record.type = BEGIN)
                then break repeat
LOG (action_id, OUTCOME, ABORTED)                    // Block future undos.
LOG (action_id, END)
```

```
procedure RECOVER ()// Recovery procedure for a volatile, in-memory database.
    winners ← NULL
    starting at end of log repeat until beginning
        log_record ← previous record of log
        if (log_record.type = OUTCOME)
            then winners ← winners + log_record                    // Set addition.

starting at beginning of log repeat until end
    log_record ← next record of log
    if (log_record.type= CHANGE)
        and (outcome_record ← find (log_record.action_id) in winners)
        and (outcome_record.status = COMMITTED) then
        perform log_record.redo_action
```

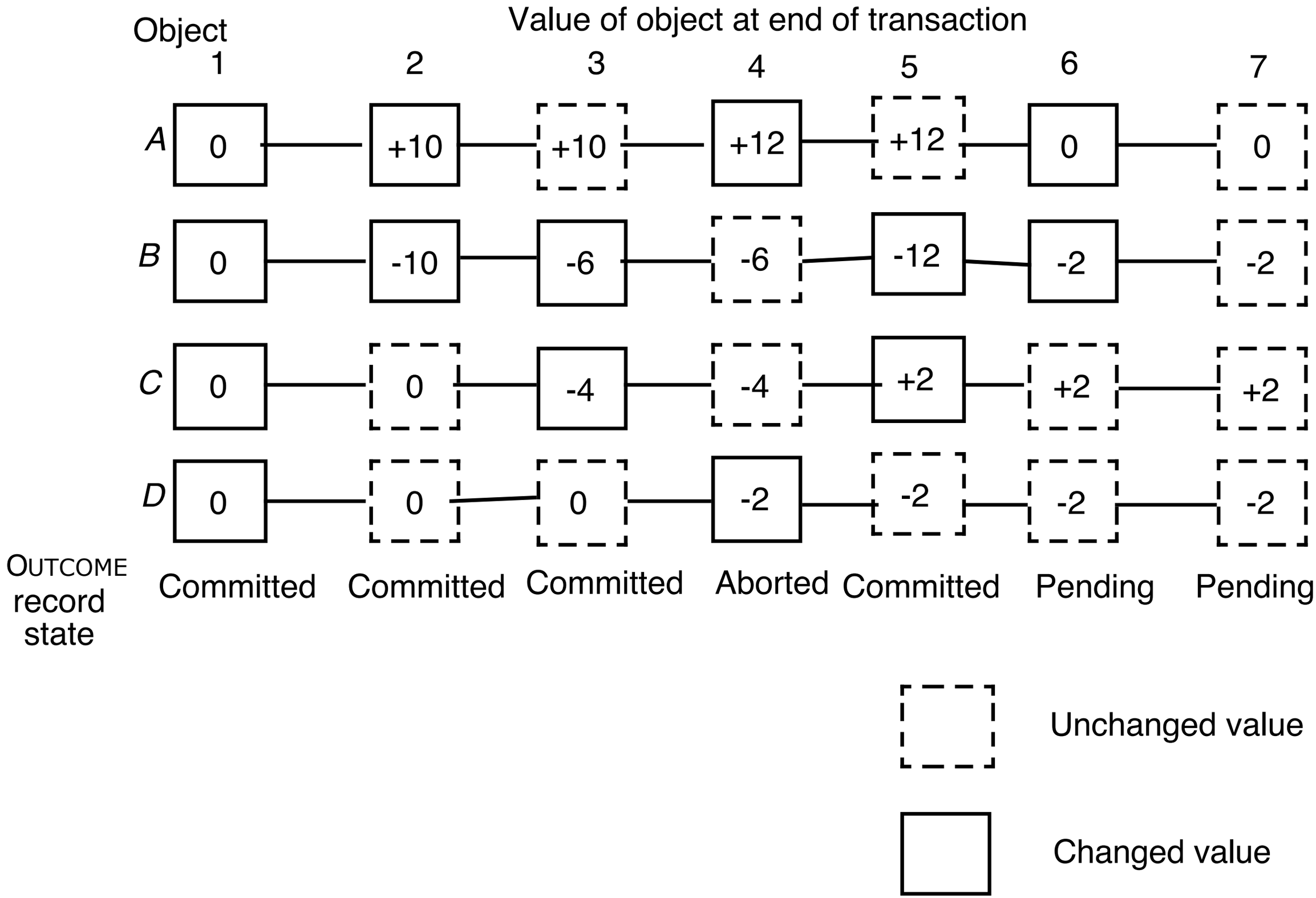
```
procedure RECOVER ()// Recovery procedure for non-volatile cell memory
    completeds ← NULL
    losers ← NULL
    starting at end of log repeat until beginning
        log_record ← previous record of log
        if (log_record.type = END)
            then completeds ← completeds + log_record          // Set addition.
        if (log_record.action_id is not in completeds) then
            losers ← losers + log_record                       // Add if not already in set.
            if (log_record.type = CHANGE) then
                perform log_record.undo_action

    starting at beginning of log repeat until end
        log_record ← next record of log
        if (log_record.type = CHANGE)
            and (log_record.action_id.status = COMMITTED) then
            perform log_record.redo_action

    for each log_record in losers do
        log (log_record.action_id, END)                        // Show action completed.
```

```
procedure RECOVER ()            // Recovery procedure for rollback recovery.
    completeds ← NULL
    losers ← NULL
    starting at end of log repeat until beginning        // Perform undo scan.
        log_record ← previous record of log
        if (log_record.type = OUTCOME)
            then completeds ← completeds + log_record    // Set addition.
    if (log_record.action_id is not in completeds) then
        losers ← losers + log_record                     // New loser.
        if (log_record.type = CHANGE) then
            perform log_record.undo_action

for each log_record in losers do
    log (log_record.action_id, OUTCOME, ABORT)           // Block future undos.
```

```
procedure BEGIN_TRANSACTION ()
    id ← NEW_OUTCOME_RECORD (PENDING)          // Create, initialize, assign id.
    previous_id ← id – 1
    wait until previous_id.outcome_record.state ≠ PENDING
    return id
```



transaction
1: initialize all accounts to 0
2: transfer 10 from *B* to *A*
3: transfer 4 from *C* to *B*
4: transfer 2 from *D* to *A* (aborts)
5: transfer 6 from *B* to *C*
6: transfer 10 from *A* to *B*



```
procedure READ_CURRENT_VALUE (data_id, this_transaction_id)
    starting at end of data_id repeat until beginning
        v ← previous version of data_id
        last_modifier ← v.action_id
        if last_modifier ≥ this_transaction_id then skip v        // Keep searching
        wait until (last_modifier.outcome_record.state ≠ PENDING)
        if (last_modifier.outcome_record.state = COMMITTED)
            then return v.state
            else skip v                                           // Resume search
signal ("Tried to read an uninitialized variable")
```

```
procedure NEW_VERSION (reference data_id, this_transaction_id)
    if this_transaction_id.outcome_record.mark_state = MARKED then
        signal ("Tried to create new version after announcing mark point!")
    append new version v to data_id
    v.value ← NULL
    v.action_id ← transaction_id

procedure WRITE_VALUE (reference data_id, new_value, this_transaction_id)
    starting at end of data_id repeat until beginning
        v ← previous version of data_id
        if v.action_id = this_transaction_id
            v.value ← new_value
            return
    signal ("Tried to write without creating new version!"))
```

```
procedure BEGIN_TRANSACTION ()
    id ← NEW_OUTCOME_RECORD (PENDING)
    previous_id ← id - 1
    wait until (previous_id.outcome_record.mark_state = MARKED)
        or (previous_id.outcome_record.state ≠ PENDING)
    return id

procedure NEW_OUTCOME_RECORD (starting_state)
    ACQUIRE (outcome_record_lock)        // Make this a before-or-after action.
    id ← TICKET (outcome_record_sequencer)
    allocate id.outcome_record
    id.outcome_record.state ← starting_state
    id.outcome_record.mark_state ← NULL
    RELEASE (outcome_record_lock)
    return id

procedure MARK_POINT_ANNOUNCE (reference this_transaction_id)
    this_transaction_id.outcome_record.mark_state ← MARKED
```

```
procedure TRANSFER (reference debit_account, reference credit_account,
                                                              amount)
    my_id ← BEGIN_TRANSACTION ()
    NEW_VERSION (debit_account, my_id)
    NEW_VERSION (credit_account, my_id)
    MARK_POINT_ANNOUNCE (my_id);
    xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
    xvalue ← xvalue - amount
    WRITE_VALUE (debit_account, xvalue, my_id)
    yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
    yvalue ← yvalue + amount
    WRITE_VALUE (credit_account, yvalue, my_id)
    if xvalue > 0 then
        COMMIT (my_id)
    else
        ABORT (my_id)
        signal("Negative transfers are not allowed.")
```

Value of object at end of transaction

1 2 3 4 5 6 7

A: 0 | HWM=2 | +10 | +12 | HWM=6 | 0 | HWM=7 | +2

B: 0 | HWM=2 | -10 | HWM=3 | -6 | HWM=5 | -12 | HWM=6 | -2

C: 0 | HWM=3 | -4 | HWM=5 | +2

D: 0 | HWM=4 | -2 | HWM=7 | -4

Committed | Committed | Committed | Aborted | Committed | Pending | Pending

Outcome state record

Conflict: Must abort!

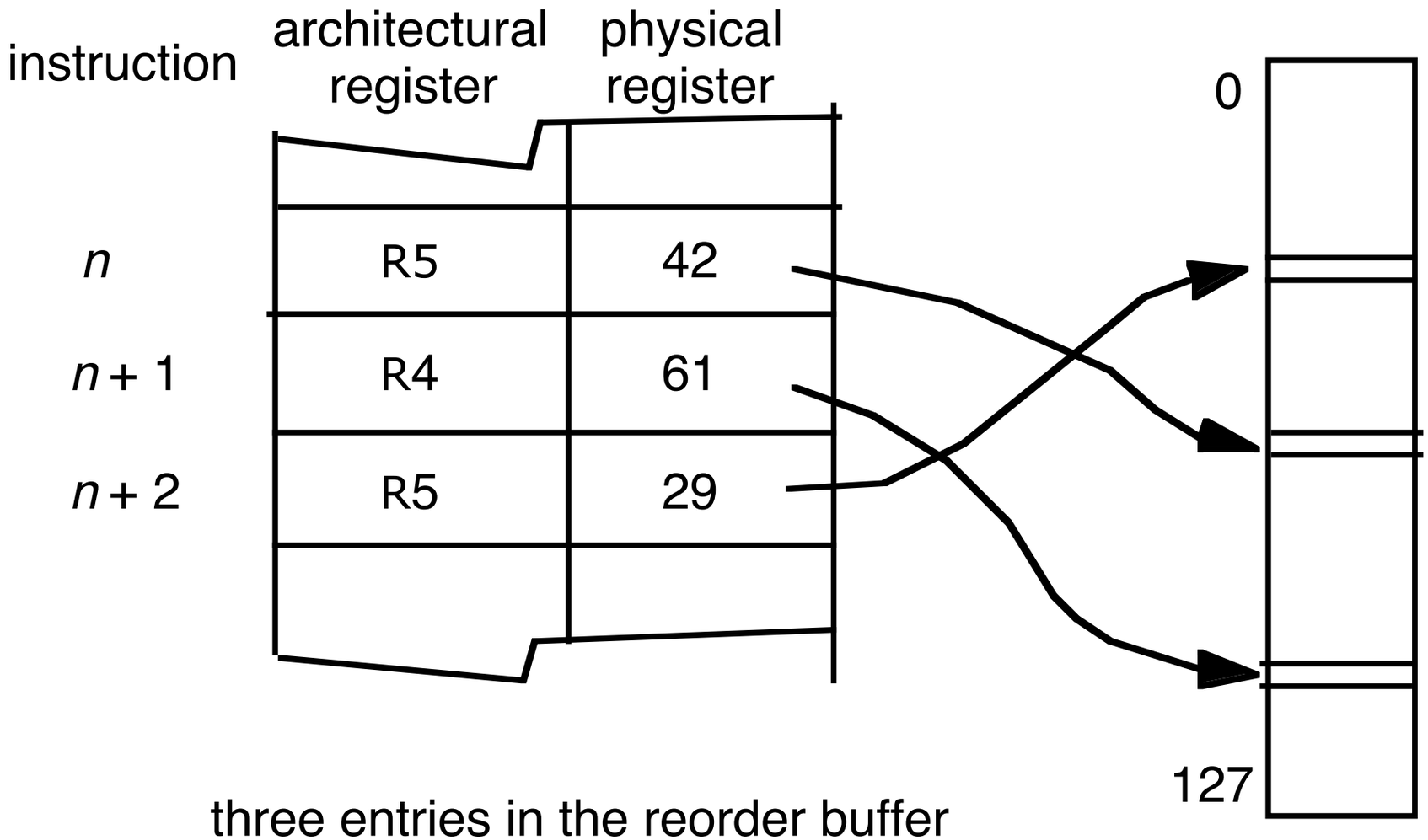
HWM= High-water mark

Conflict

Changed value

```
procedure READ_CURRENT_VALUE (reference data_id, value, caller_id)
    starting at end of data_id repeat until beginning
        v ← previous version of data_id
        if v.action_id ≥ caller_id then skip v
        examine v.action_id.outcome_record
            if PENDING then
                WAIT for v.action_id to COMMIT or ABORT
                if COMMITTED then
                    v.high_water_mark ← max(v.high_water_mark, caller_id)
                    return v.value
                else skip v                                  // Continue backward search
signal ("Tried to read an uninitialized variable!")
```

```
procedure NEW_VERSION (reference data_id, caller_id)
    if (caller_id < data_id.high_water_mark)            // Conflict with later reader.
    or (caller_id < (LATEST_VERSION[data_id].action_id))    // Blind write conflict.
    then ABORT this transaction and terminate this thread
    add new version v at end of data_id
    v.value ← 0
    v.action_id ← caller_id

procedure WRITE_VALUE (reference data_id, new_value, caller_id)
    locate version v of data_id.history such that v.action_id = caller_id
            (if not found, signal ("Tried to write without creating new version!"))
    v.value ← new_value
```




```
n       R5 ← R4 ×  R2          // Write a result in register five.
n + 1   R4 ← R5 + R1           // Use result in register five.
n + 2   R5 ← READ (117492)     // Write content of a memory cell in register five.
```

```
procedure PAY_INTEREST (reference account)
    if account.balance > 0 then
        interest = account.balance * 0.05
        TRANSFER (bank, account, interest)
    else
        interest = account.balance * 0.15
        TRANSFER (account, bank, interest)

procedure MONTH_END_INTEREST:()
    for A ← each customer_account do
        PAY_INTEREST (A)
```

MONTH_END_INTEREST

outcome:	PENDING
superior:	none

PAY_INTEREST$_1$ (*1st invocation*)

outcome:	COMMITTED
superior:	MONTH_END_INTEREST

PAY_INTEREST$_2$ (*2nd invocation*)

outcome:	PENDING
superior:	MONTH_END_INTEREST

TRANSFER$_1$

outcome:	COMMITTED
superior:	PAY_INTEREST$_1$

TRANSFER$_2$

outcome:	PENDING
superior:	PAY_INTEREST$_2$

creator: TRANSFER$_1$

newest version of account *bank*

OK for TRANSFER$_2$ to read?

From: Alice
To: Bob
Re: my transaction 91

if (Charles does Y **and** Dawn does Z) **then do** X, please.

From:Alice
To: Bob
Re: my transaction 271

Please do X as part of my transaction.

From:Bob
To: Alice
Re: your transaction 271

My part X is ready to commit.

Two-phase-commit message #1:

From:Alice
To: Bob
Re: my transaction 271

PREPARE to commit X.

Two-phase-commit message #2:

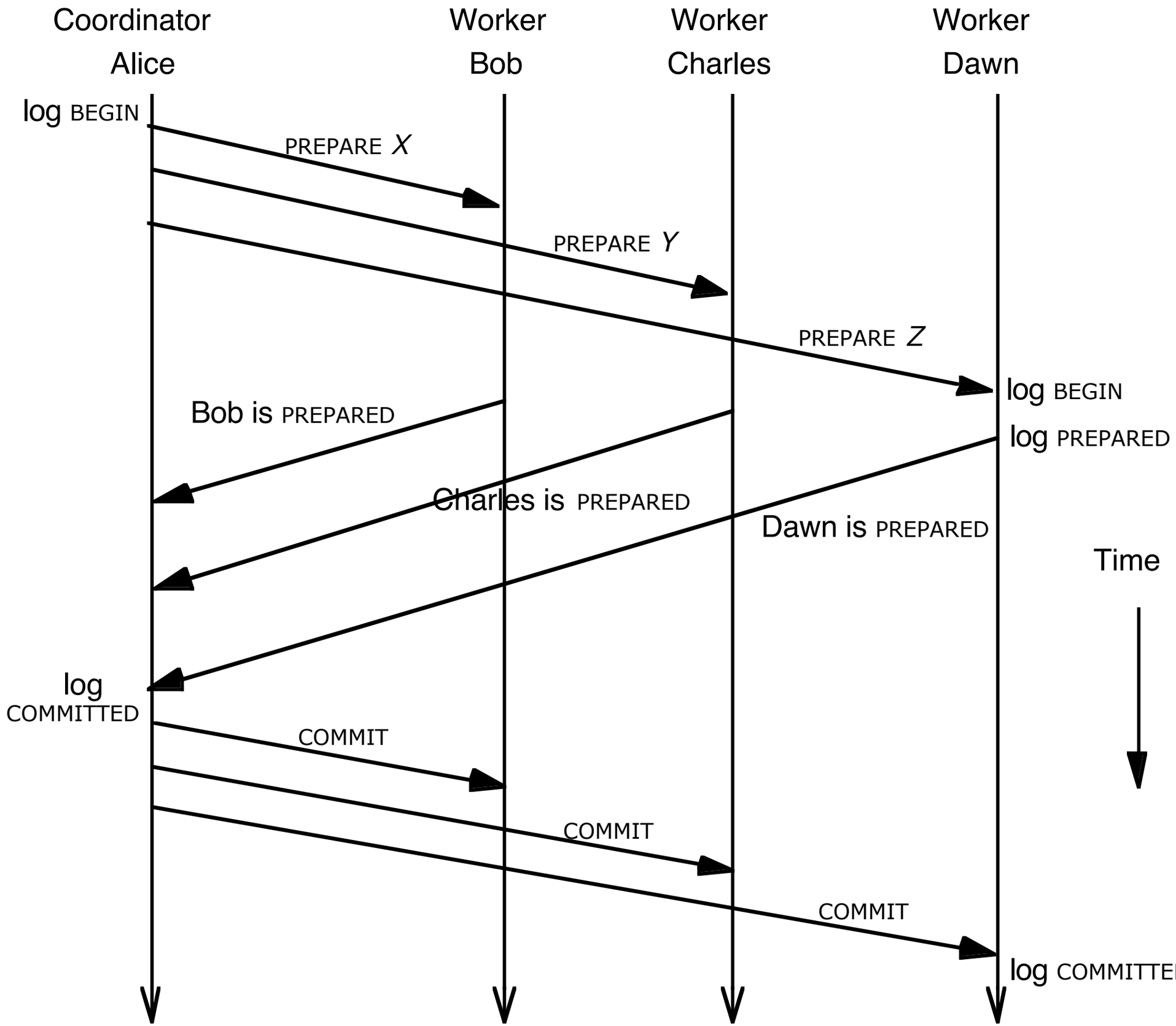
From:Bob
To:Alice
Re: your transaction 271

I am PREPARED to commit my part. Have you decided to commit yet?

Two-phase-commit message #3

From:Alice
To:Bob
Re: my transaction 271

My transaction committed. Thanks for your help.



From:Julius Caesar
To:Titus Labienus
Date:11 January

I propose to cross the Rubicon and attack at dawn tomorrow. OK?

From:Titus Labienus
To:Julius Caesar;
Date:11 January

Agreed, dawn on the 12th.

From:Julius Caesar
To:Titus Labienus
Date:11 January

The die is cast.

or

From:Titus Labienus
To: Julius Caesar
Date:11 January

No. I am awaiting reinforcements from Gaul.

```
procedure ALL_OR_NOTHING_DURABLE_GET (reference data, atomic_sector)
    ds ← CAREFUL_GET (data, atomic_sector.D0)
    if ds = BAD then
        ds ← CAREFUL_GET (data, atomic_sector.D1)
    return ds

procedure ALL_OR_NOTHING_DURABLE_PUT (new_data, atomic_sector)
    SALVAGE(atomic_sector)
    ds ← CAREFUL_PUT (new_data, atomic_sector.D0)
    ds ← CAREFUL_PUT (new_data, atomic_sector.D1)
    return ds

procedure SALVAGE(atomic_sector)      //Run this program every T_d seconds.
    ds0 ← CAREFUL_GET (data0, atomic_sector.D0)
    ds1 ← CAREFUL_GET (data1, atomic_sector.D1)
    if ds0 = BAD then
        CAREFUL_PUT (data1, atomic_sector.D0)
    else if ds1 = BAD then
        CAREFUL_PUT (data0, atomic_sector.D1)
    if data0 ≠ data1 then
        CAREFUL_PUT (data0, atomic_sector.D1)
```