EEL-4930/5934
Principles of Computer System Design

Lecture Slides 21

Textbook Chapter 9

Atomicity Logs

Introduction

- Addressing performance implications associated with all-or-nothing actions
- Key idea:
  - Optimize for the common case (reads, writes)
  - Recovery must be possible, but it is the uncommon case – not the main focus of performance optimization

Atomicity Logs

- How to benefit from
  - All-or-nothing of journal storage
  - Performance of cell storage
- Write data twice
  - Journal storage (Log) – atomicity
  - Cell storage (install) – fast reads
- Don't you just double the work?

Logging

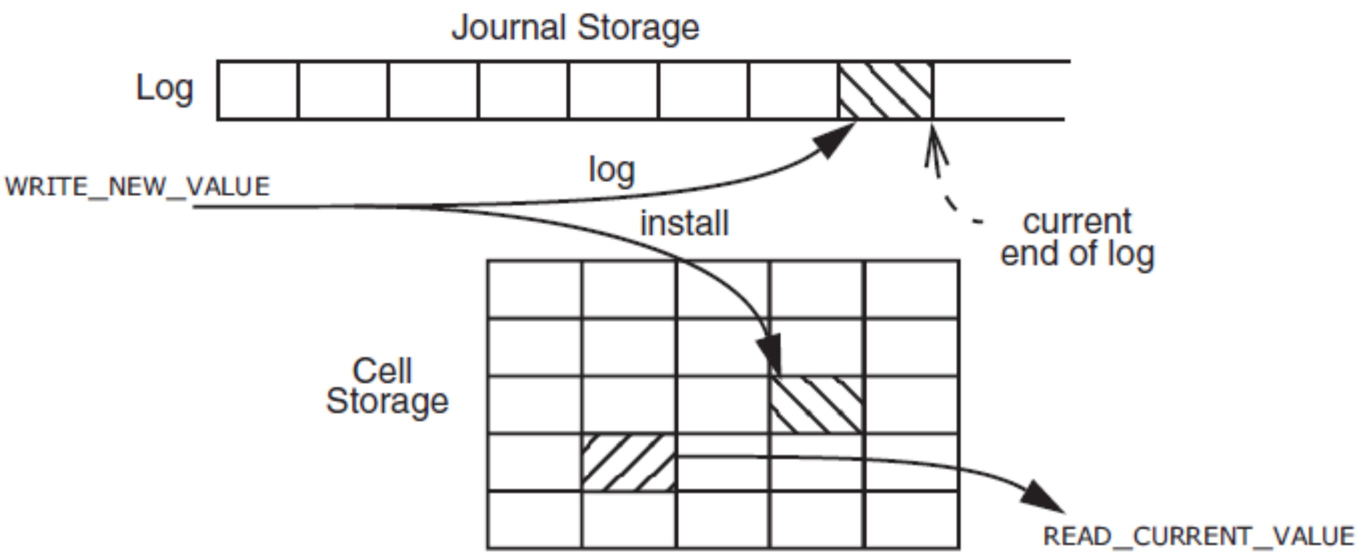
- Optimize writing of multiple objects by mapping to single log
- Key benefit: writes are sequential
  - Single pointer for all updates
  - Dedicated disk storage for logging: minimize seek times, significantly improving performance
- Drawback
  - Histories of different objects are interleaved
  - Longer lists - lookup is slower; recovery however is uncommon

Cell storage

- “Installing” a new version is an overwrite
- Faster subsequent reads
- No concern about overwriting a single copy, if it’s been already logged
  – Cache that allows bypassing overhead of traversing log on reads
  – Also avoids seek overhead by keeping the arm in place at log’s end
- Authoritative copy always in log
  – Cell storage – can always be reconstructed from log

Primitives

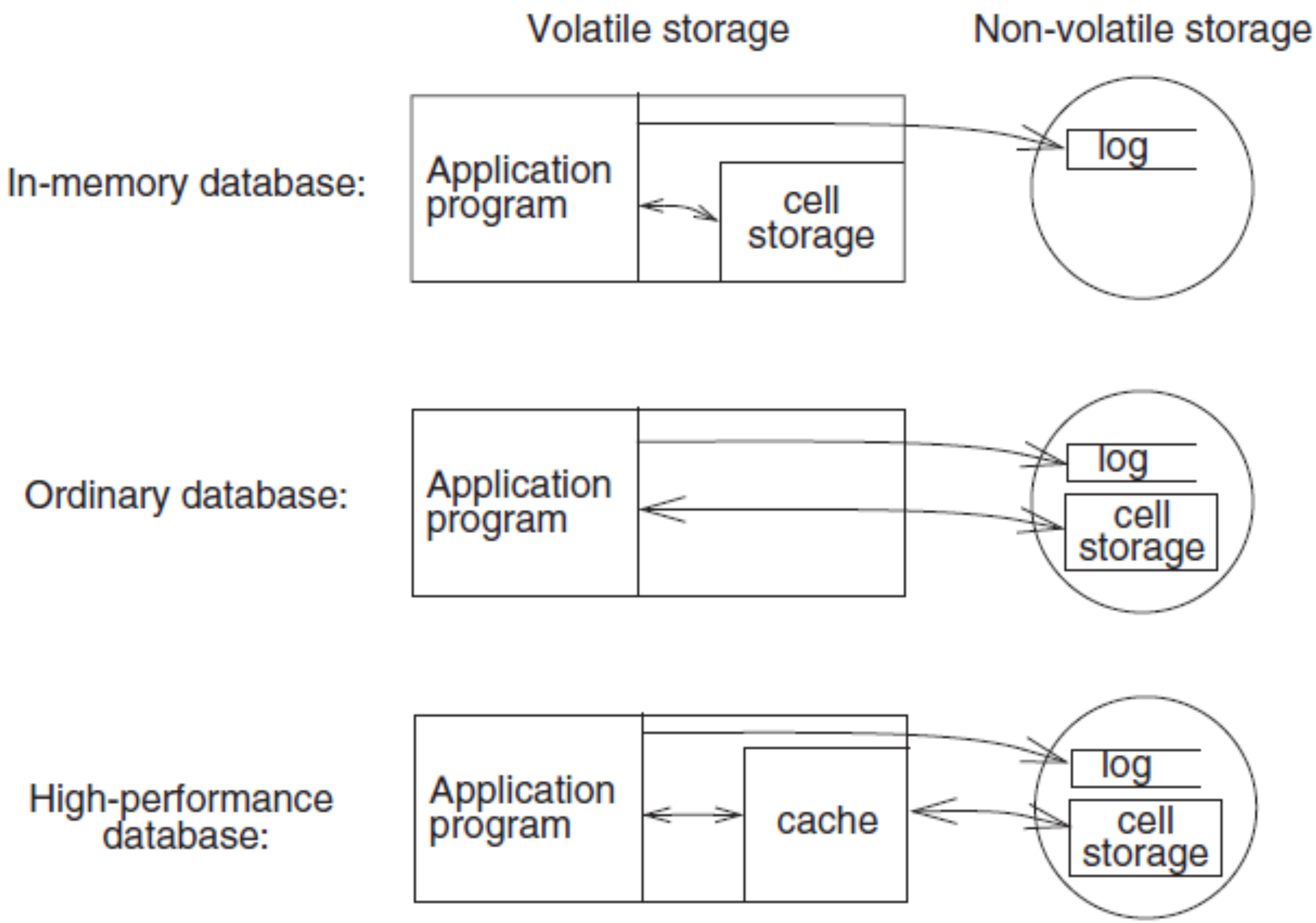
- WRITE_NEW_VALUE()
  - LOG
    - Append entry to end of log
  - INSTALL
    - Overwrite cell storage with new value



Summary

- Writing twice can *improve* performance
  - Reduce seek overhead on log operations for WRITE_NEW_VALUE
  - Faster READ_CURRENT_VALUE – no need to traverse log
- Writing twice *maintains* all-or-nothing nature of operations
  - Log is written all-or-nothing, as we did before
  - Cell storage is just a copy

Logging strategies



Issues

- Failures can happen between writes
  - Log should be authoritative copy
  - Cell storage may not be properly installed
  - Note that cell storage may be non-volatile
- Enforce ordering
  - Log first, install after logged
  - Write-ahead log
- Run recovery procedure after crash
  - Recall check_and_repair

Logging protocols

- Basic element – *log record*
- Record type (e.g. CHANGE)
- ID of all-or-nothing action performing update
- Action that, if performed, installs intended value in cell storage (“redo”)
- Action that, if performed, reverses the effect in cell storage (“undo”)

Redo/Undo actions

- Redo:
  - Part of the log record that is invoked when a recovery procedure needs to install committed data into cell storage
  - Crash that happens before a committed value gets to be installed
- Undo:
  - Part of the log record that is invoked when an action aborts after installing into cell storage (possibly during recovery procedure)

Primitives

- LOG:
  - Appends log record to log storage
  - Must itself be all-or-nothing
  - Again, can bootstrap from single-sector primitive (ALL_OR_NOTHING_PUT)
- WRITE_NEW_VALUE:
  - LOG then INSTALL (PUT)
- READ_CURRENT_VALUE:
  - Read (GET) from cell storage

Primitives

- NEW_ACTION:
  - Build upon LOG primitive
  - Logs a BEGIN record that establishes the new action's identity
- Action's writes log CHANGE records through its pre-commit phase
- COMMIT/ABORT:
  - Log OUTCOME records
  - Commit point – becomes authoritative outcome or action

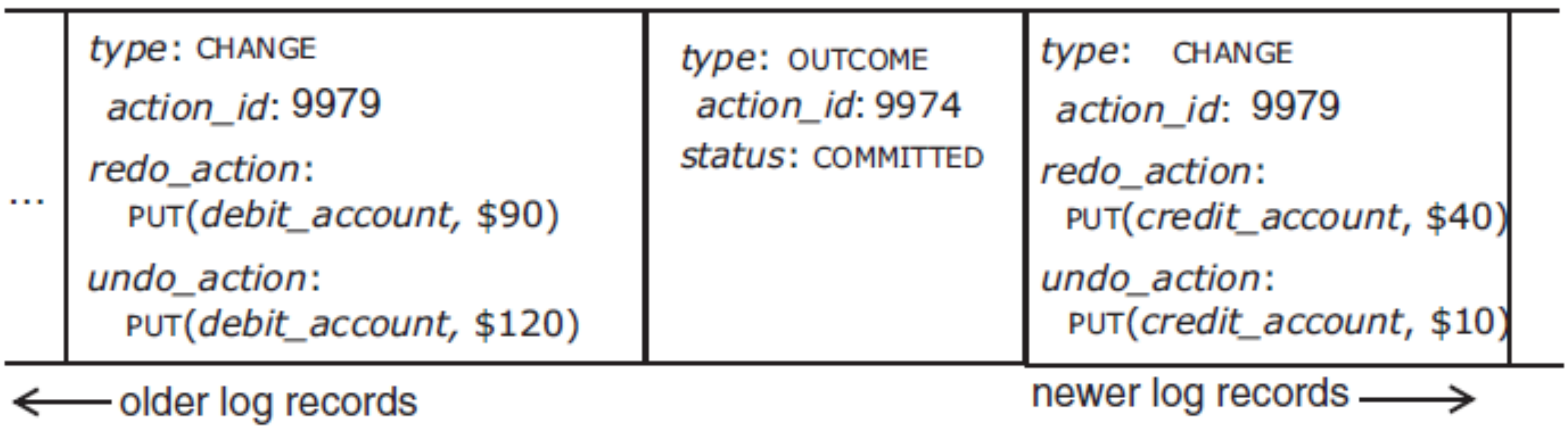
Account transfer revisited

```
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)
```

Notes

- INSTALL of a value
  - Must occur after LOG of a CHANGE
  - But not necessarily after LOG of OUTCOME
- BEGIN, CHANGE, OUTCOME, END
  - Most general
  - Implementations may combine (first CHANGE means BEGIN; OUTCOME +END)

Log records


...
type: CHANGE
action_id: 9979
redo_action:
PUT(debit_account, $90)
undo_action:
PUT(debit_account, $120)
type: OUTCOME
action_id: 9974
status: COMMITTED
type: CHANGE
action_id: 9979
redo_action:
PUT(credit_account, $40)
undo_action:
PUT(credit_account, $10)
older log records
newer log records

ABORT Procedure

- An action may install values in cell storage that need to be restored
  - Not necessarily by thread that started action (e.g. thread deadlocks)
  - Can't leave an action pending indefinitely
- ABORT procedure
  - Scan log backwards for aborted action
  - Perform logged "undo" action for every CHANGE log record
  - Stop at BEGIN record
- Expensive (but uncommon)

ABORT procedure

```
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)
```

Recovery Procedure

- Must be executed after any crash and before application uses data
- Bring cell storage contents in accordance to log
- System can crash during recovery procedure itself
  - Idempotent recovery
- Cell storage in volatile memory simplifies recovery

Recovery for in-memory db

```
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
```

Backward scan looks for records with OUTCOME
Forward scan applies redo actions
As if all all-or-nothing actions that committed before the crash had run to completion, and as if those that didn't commit (aborted, pending) never existed

Recovery for non-volatile cell

- Cell storage in volatile memory simplifies recovery
  – All cell storage data is lost on a crash; start from clean slate
- However, the database may be too large to fit in memory
- In general, need cell storage to also be in non-volatile storage
  – Performance enhancement – a cache in volatile memory

Recovery for non-volatile cell

- The additional complexity now is that installs persist crashes
  - There may be all-or-nothing actions that installed changes, but were still pending when the system crashed
  - Those changes will persist – recovery needs to undo them
- Furthermore, reinstalling the whole database in cell storage from the log upon recovery would be too slow for large databases

Modified recovery procedure

- Scan backwards, but instead of looking for “winners” (actions that had an outcome), build a set of “losers”
  - Those that were pending
  - I.e., on backward scan, if the first record found for an action is *not* END, it is a loser
- Perform undo action for loser set
  - Mark them as ENDed so to not undo in future recoveries
- Then, forward scan and perform redo of committed actions

Recovery

```
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.
```

Improvement

- Scanning forward and redoing every install – slow
- Cannot know if all installs have been done (even by committed action) until an END record is found
  – Write-ahead log protocol specifies all installs must be done when an END is logged
- Any committed action that has logged END has completed its installs
  – Can modify code from previous page to avoid those installs

Rollback logging

- Can avoid the need for redoing any installs if application is required to perform all of its installs before it logs an OUTCOME record
- Recovery procedure needs to only undo installs of losers, and skip forward scan

Rollback recovery

```
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.
```

Before-or-after

- With logs, when an action installs data in cell storage, it is visible
  - Intermediate steps would be visible to other concurrent threads
- Our assumption - single thread
  - Data used later in time (after commit) or, on a crash, after recovery process
- Supporting multiple threads requires dealing with before-or-after atomicity

Simple serialization

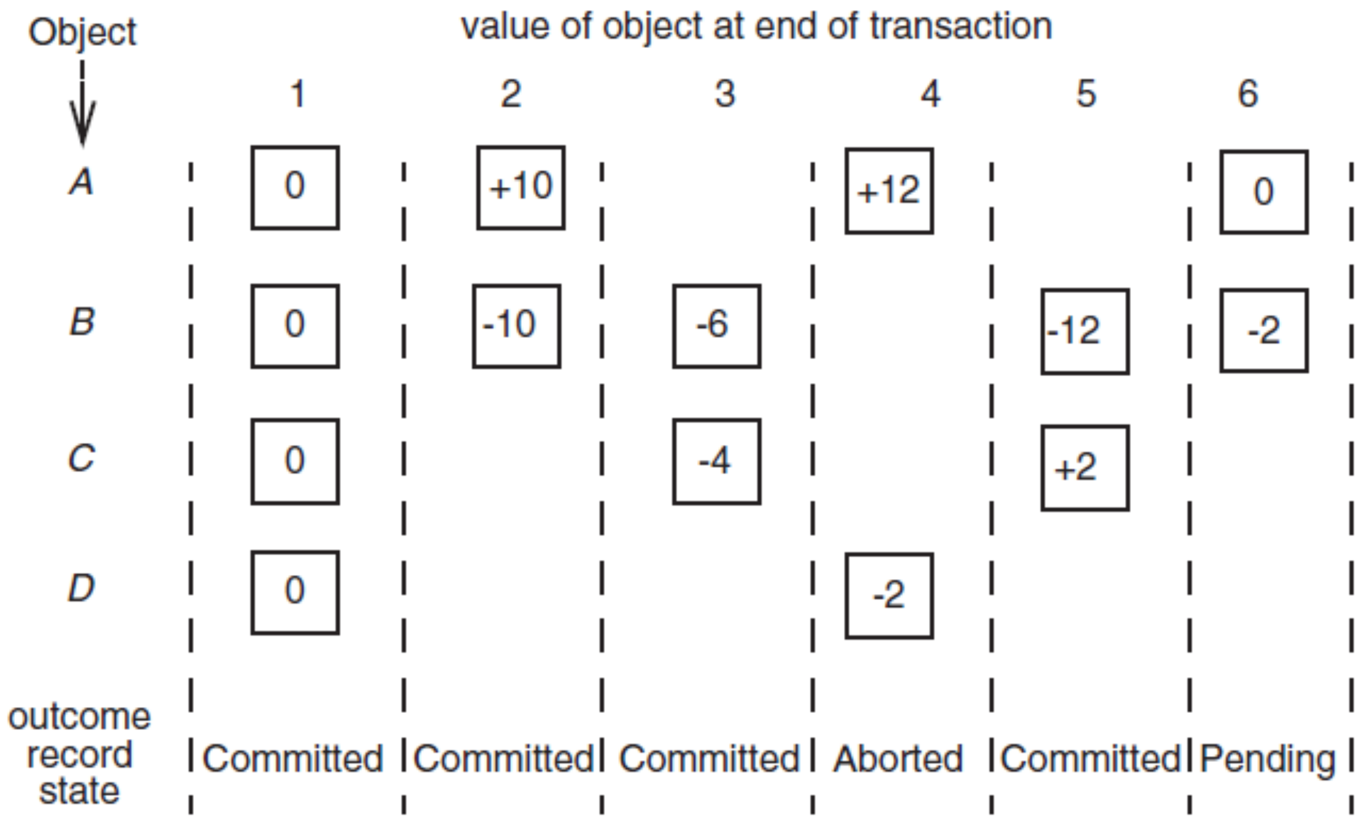
- Assign monotonically-increasing transaction IDs
- Transaction n+1 does not read/write data until transaction n has committed or aborted
- Enforces that *transactions* run in serial order
  - Threads can still execute concurrently outside transaction boundaries

Simple serialization

- Simple and correct; conservative
  - Transactions n, n-1 may work on completely independent data, yet one blocks the other
  - If higher performance is needed, there are alternatives – more efficient and complex

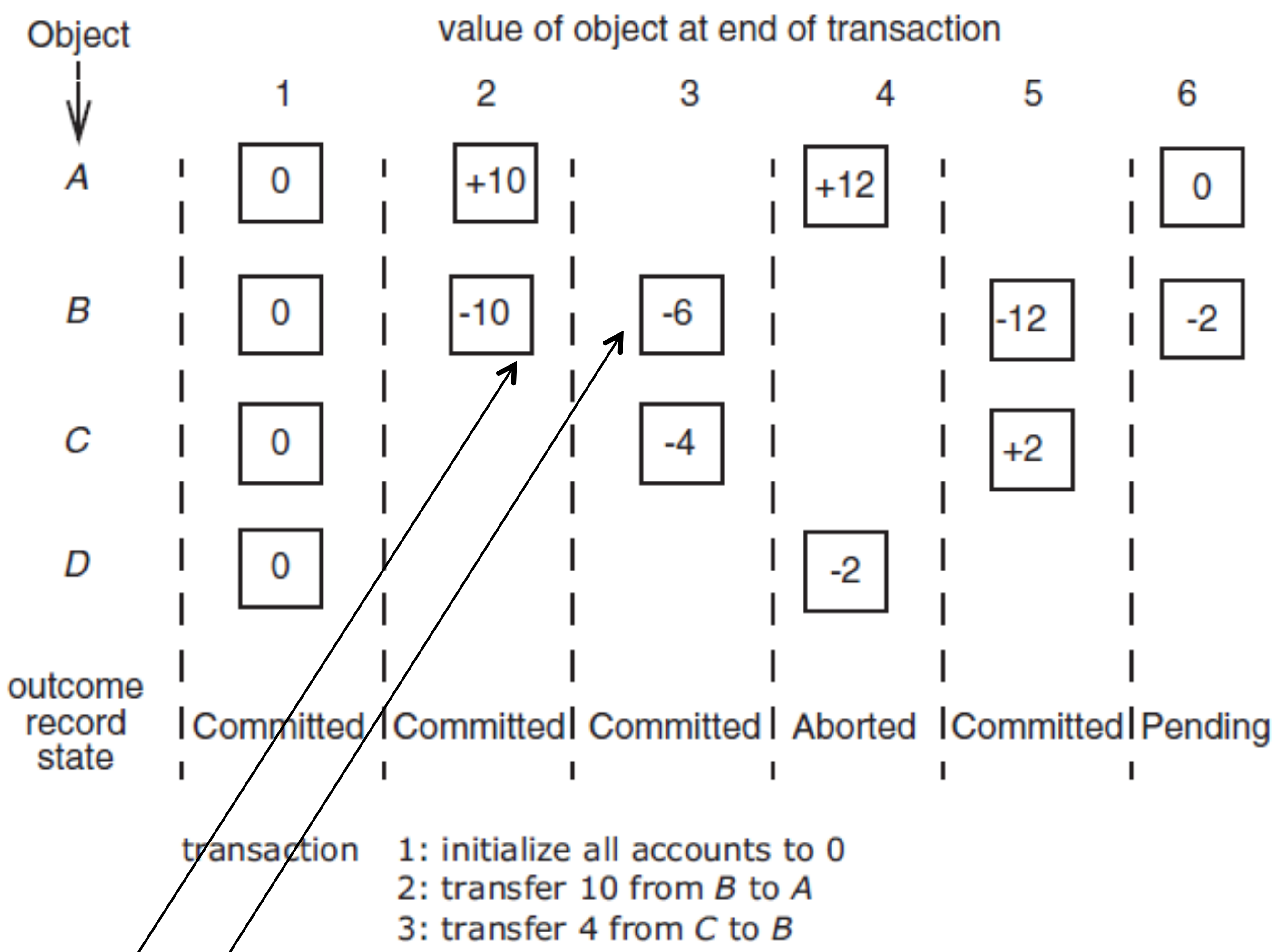
```
procedure BEGIN_TRANSACTION ()
    id ← NEW_OUTCOME_RECORD (PENDING)          // Create, initialize, assign id.
    previous_id ← id – 1
    wait until previous_id.outcome_record.state ≠ PENDING
    return id
```

Opportunities for Concurrency



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*

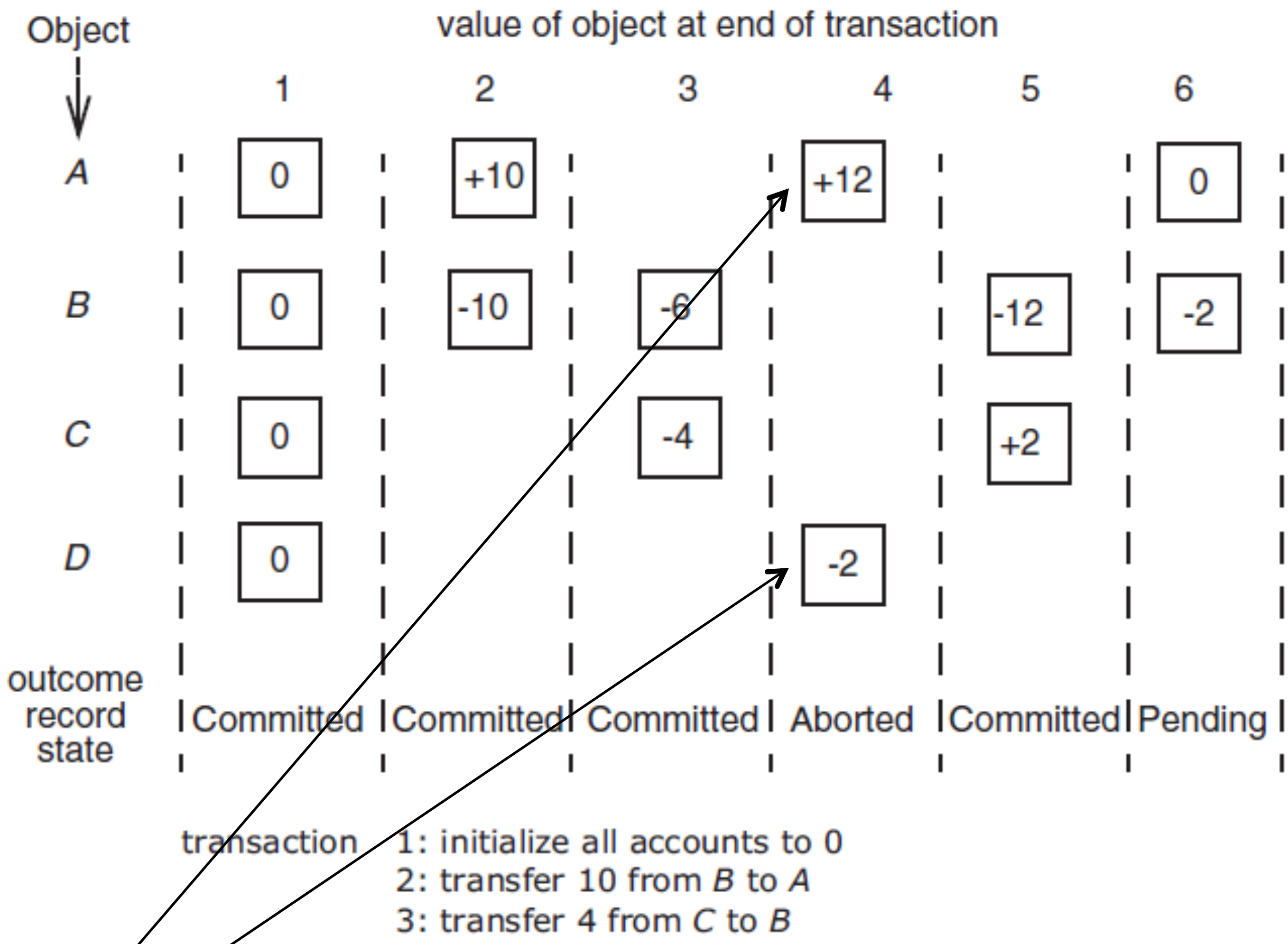
Opportunities for Concurrency



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*

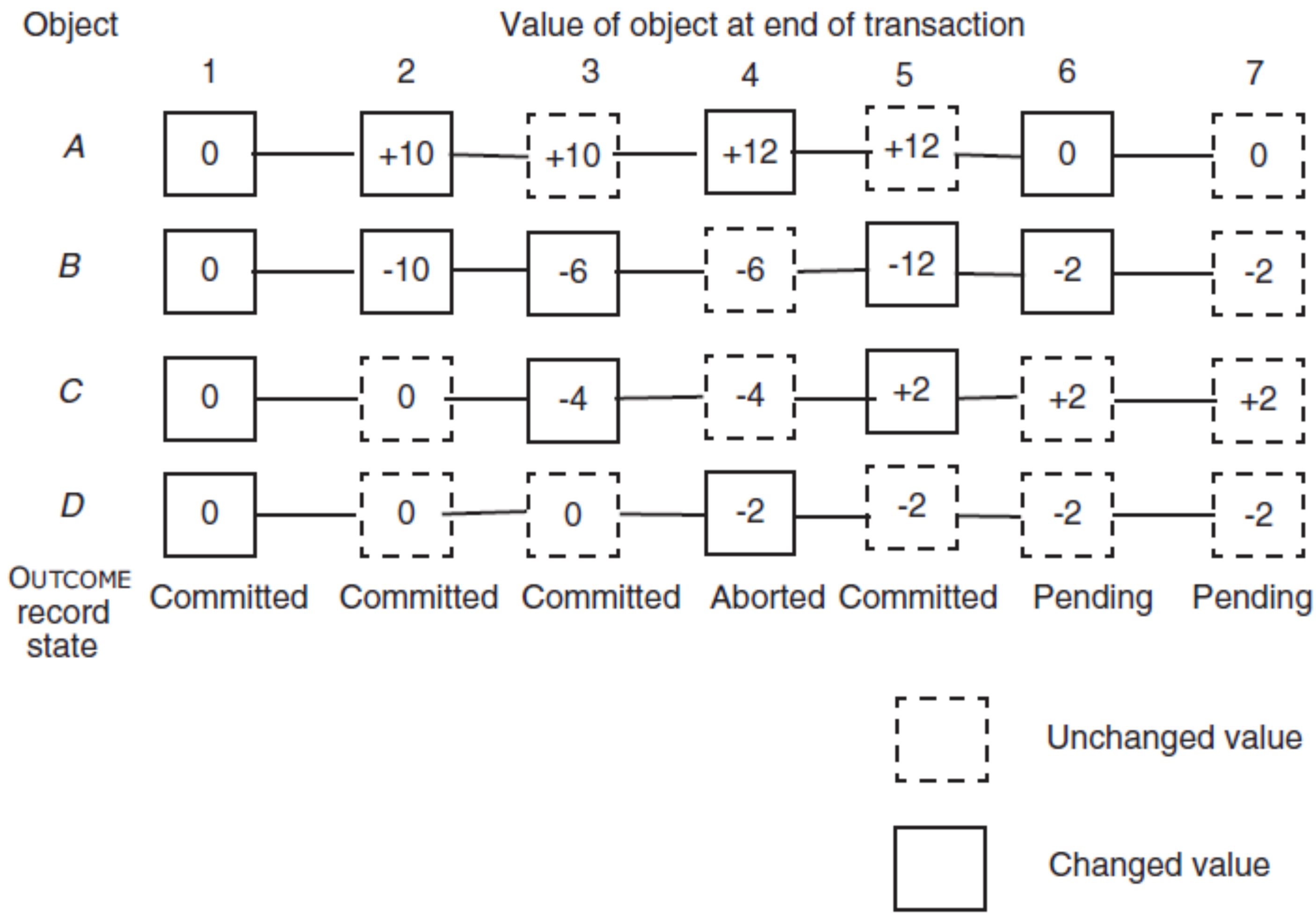
3 must wait for 2

Opportunities for Concurrency

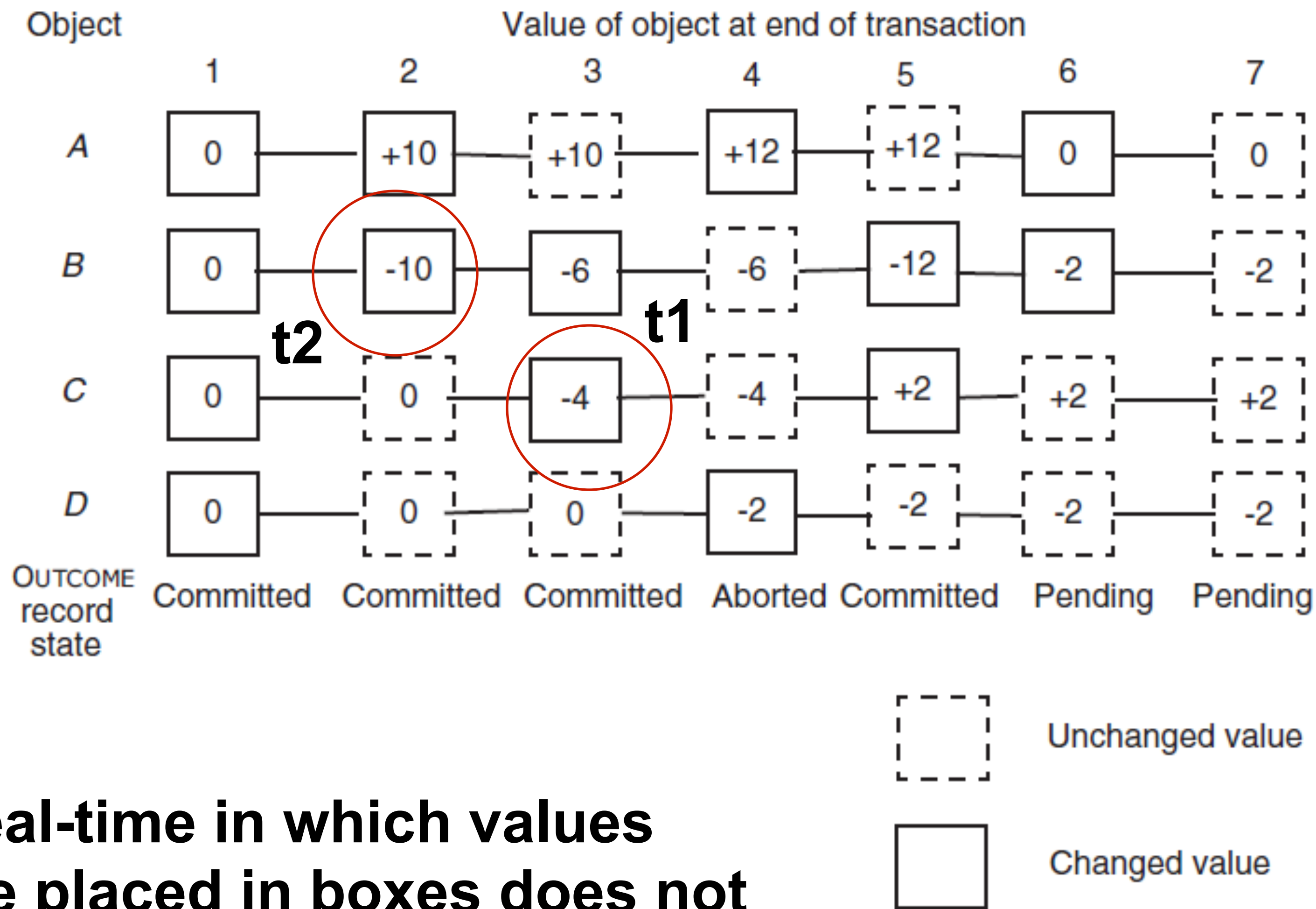


4 does not need to wait for 3; must wait for 2

Opportunities for Concurrency

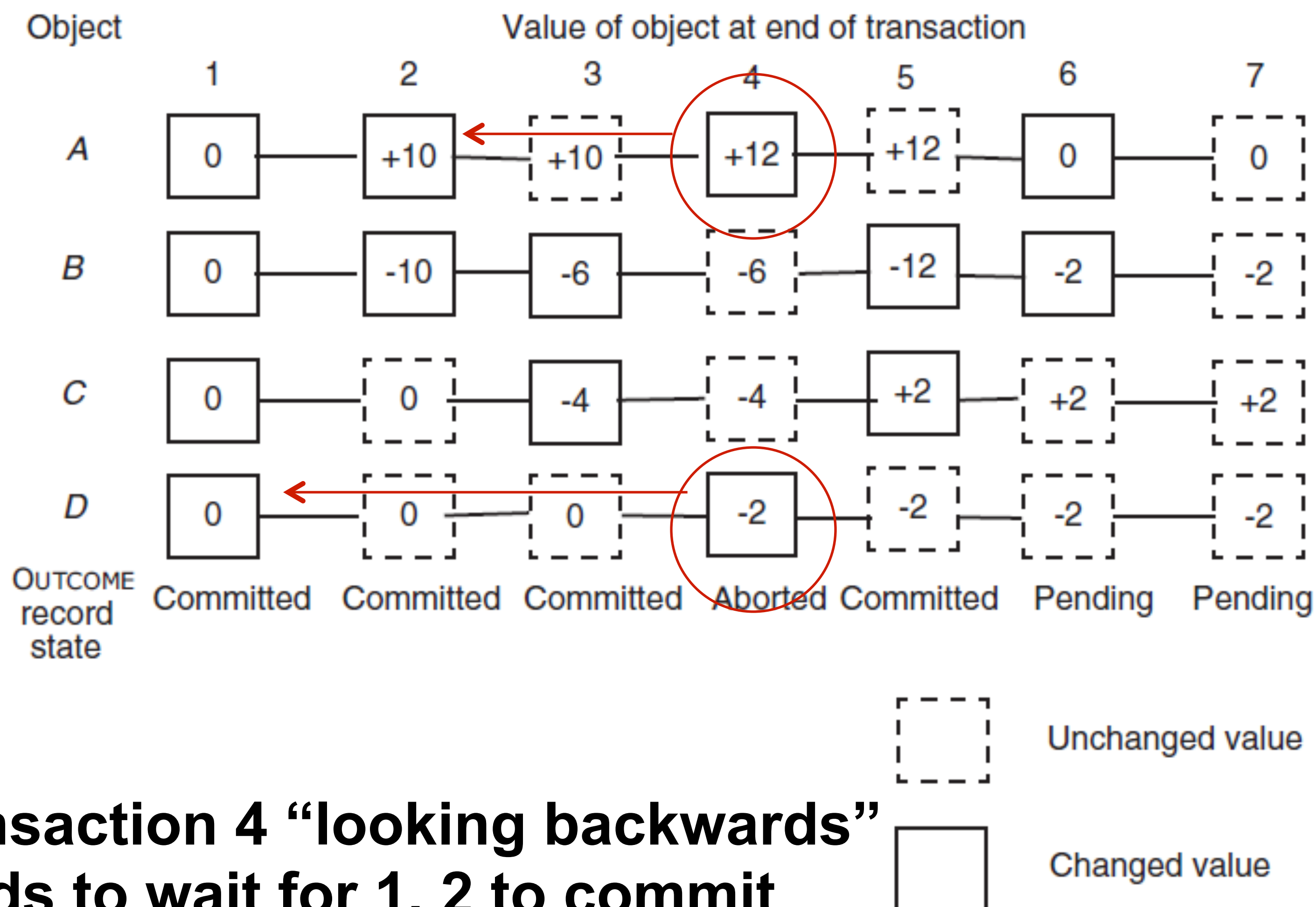


Opportunities for Concurrency



Real-time in which values are placed in boxes does not need to be strictly serial – as long as the results are the same as one would get with serial execution

Opportunities for Concurrency



Transaction 4 "looking backwards" needs to wait for 1, 2 to commit (not 3)

Mark Point Discipline

- Recall version history/journal discussion
  - READ_CURRENT_VALUE should not expose any pending, non-committed value
    - All values should be exposed at once at commit
    - No intermediate values visible (expect for self)
      - READ_MY_CURRENT_VALUE
- Additional requirement:
  - A thread must *wait* until a pending value from an earlier thread that it needs to read is committed

Mark Point Discipline

- READ_CURRENT_VALUE
  - Skip “later” transaction (larger IDs)
    - Concurrent execution - later threads may be creating their own versions
  - Wait for pending values from “earlier” transactions (smaller IDs)
- How does a transaction know which pending values to wait for?
  - Explicitly programmed in the application

Mark Point Discipline

- Each transaction explicitly creates new, pending versions of each object it intends to modify
  - "Marking" an object
  - WRITE_NEW_VALUE: NEW_VERSION (mark), WRITE_VALUE (update)
- Transaction also announces that it has done so (for *all* objects)
  - The mark point
  - MARK_POINT_ANNOUNCE

Mark Point Discipline

- Enforce mark point discipline at the beginning of a transaction
- BEGIN_TRANSACTION – wait until preceding transaction has reach mark point (or no longer pending)
- By design – transactions must reach mark point in transaction ID order
  - But they can overlap

BEGIN / MARK

Block until previous ID marked (or not pending)

```
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
```

Bootstrap before-or-after

Dispense one monotonically increasing outcome record at a time

More state in outcome record

WRITE_NEW_VALUE

```
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!"))
```

Just a placeholder (before mark point)

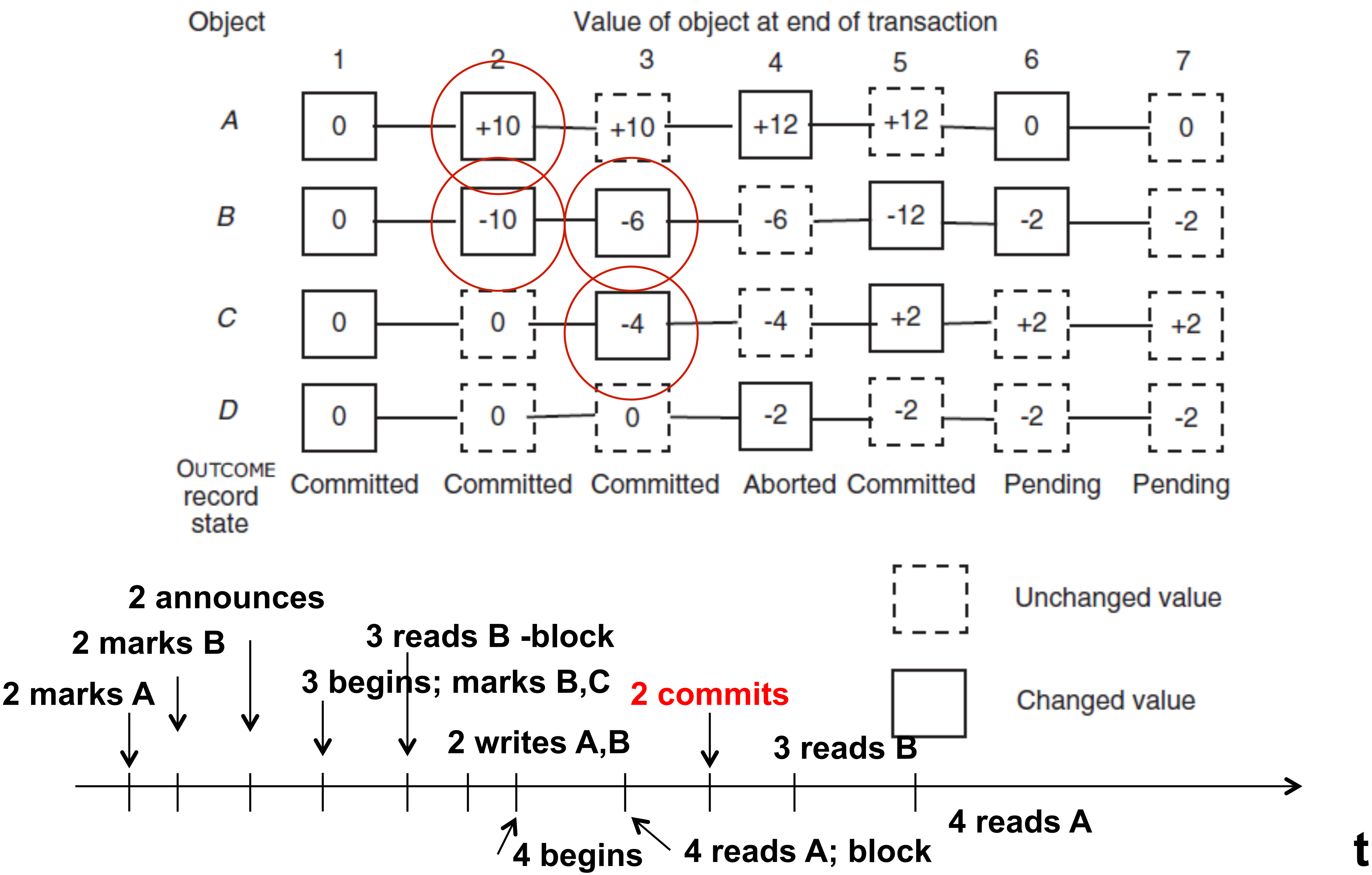
Actual write (after mark point)

READ_CURRENT_VALUE

```
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")
```

Block until outcome Record no longer pending (e.g. from a previous ID's NEW_VERSION)

Opportunities for Concurrency



Updated transfer function

```
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.")
```

Example – architecture

- Register renaming
- Legacy architectures have few registers
  - E.g. x86 – 8 registers
- For high instruction-level parallelism
  - Need to overlap many instructions
  - Implementation – 128 physical registers
- Reorder buffer maps architecture register (1 of 8) to physical register (1 out of 128) holding actual value

Example- architecture

- Instruction issues – assigned next sequential slot in reorder buffer
  - Begin transaction
  - NEW_OUTCOME_RECORD (its position in the buffer) and NEW_VERSION
- Commits
  - Write to physical register
  - WRITE_NEW_VALUE and COMMIT

Register Renaming

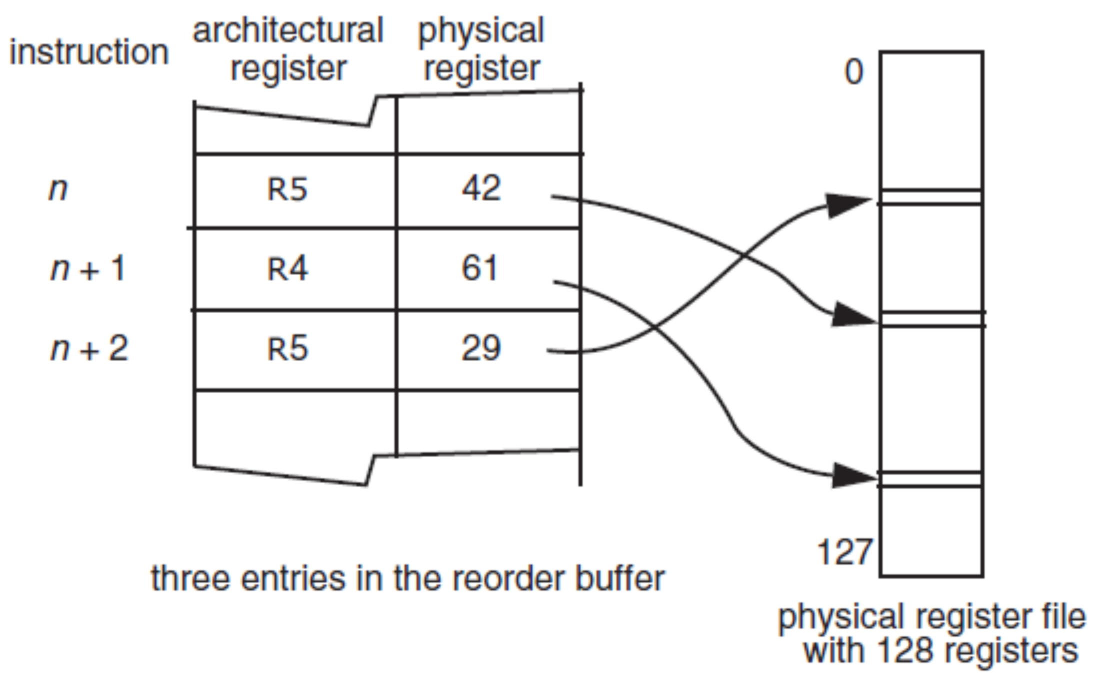


FIGURE 9.34

Example showing how a reorder buffer maps architectural register numbers to physical register numbers. The program sequence corresponding to the three entries is:

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.

Instructions n and $n + 2$ both write into register R5, so R5 has two versions, with mappings to physical registers 42 and 29, respectively. Instruction $n + 2$ can thus execute concurrently with instructions n and $n + 1$.