Transactions

Lecture topics

- properties of transactions
- failures and concurrency
- transactions in SQL
- implementation of transactions
- degrees of isolation
- transactions in distributed systems

References:

- text 3rd edition: Chapter 5; Chapter 19; Chapter 20, sections 1, 5, 7-8; Chapter 21, sections 1, 2, 7-8; Chapter 24, section 4-5
- text 4th edition: Chapter 13, sections 1–6; Chapter 17; Chapter 18, sections 1, 5, 7-8; Chapter 19, sections 1, 2, 7-8; Chapter 25, section 4-5

Problems caused by failures

Accounts

Anum	CId	BranchId	Balance

```
update Accounts
  set Balance = Balance + 5
  where BranchId = 12345
```

- if system crashes during processing, some, but not all, tuples with BranchId = 12345 may have been updated
- DB state is unpredictable; may be inconsistent

...continued

- transfer money between accounts:

```
update Accounts
set Balance = Balance - 100
where Anum = 8888

update Accounts
set Balance = Balance + 100
where Anum = 9999
```

- if system crashes between these updates, balance for 8888 might be reduced without increasing balance for 9999 (i.e., funds withdrawn but not redeposited)

Problems caused by concurrency

- transaction 1:

```
update Accounts
set Balance = Balance - 100
where Anum = 8888

update Accounts
set Balance = Balance + 100
where Anum = 9999
```

- transaction 2:

```
select Sum(Balance)
from Accounts
```

- result of transaction 2 may not reflect the true sum of the account balances

Transaction properties

- transactions are **durable**, **atomic** units of work
  - **Atomic**: indivisible, all-or-nothing
  - **Durable**: survives failures

- a transaction occurs either entirely, or not at all
- if a transaction occurs, its effects will not be erased or undone by subsequent failures

...continued

- concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order

- if T_i and T_j are concurrent transactions, then either:
  - T_i will appear to precede T_j, meaning that T_j will "see" any updates made by T_i, and T_i will not see any updates made by T_j, or
  - T_i will appear to follow T_j, meaning that T_i will see T_j's updates and T_j will not see T_i's.

The ACID properties of transactions

- DBMS guarantees transactions have "ACID" properties:

  Atomicity: all-or-nothing execution

  Consistency: execution preserves database integrity

  Isolation: a transaction's updates are not visible until it commits (finishes successfully)

  Durability: updates made by a committed transaction will not be destroyed by subsequent failures.

Abort and commit

- a transaction terminates by aborting, or by committing:
  - when a transaction **commits**, any updates become durable and visible to other transactions
  - when a transaction **aborts**, any updates are undone (erased), as if the transaction never ran at all

- atomicity:
  - commit is the “all” in “all-or-nothing” execution
  - abort is the “nothing” in “all-or-nothing” execution

- a transaction that has started, but not yet aborted or committed, is **active**

Transactions in SQL

- a transaction begins when an application first executes an SQL command

- two SQL commands are available to terminate a transaction:

  `commit [work]`: commit the transaction

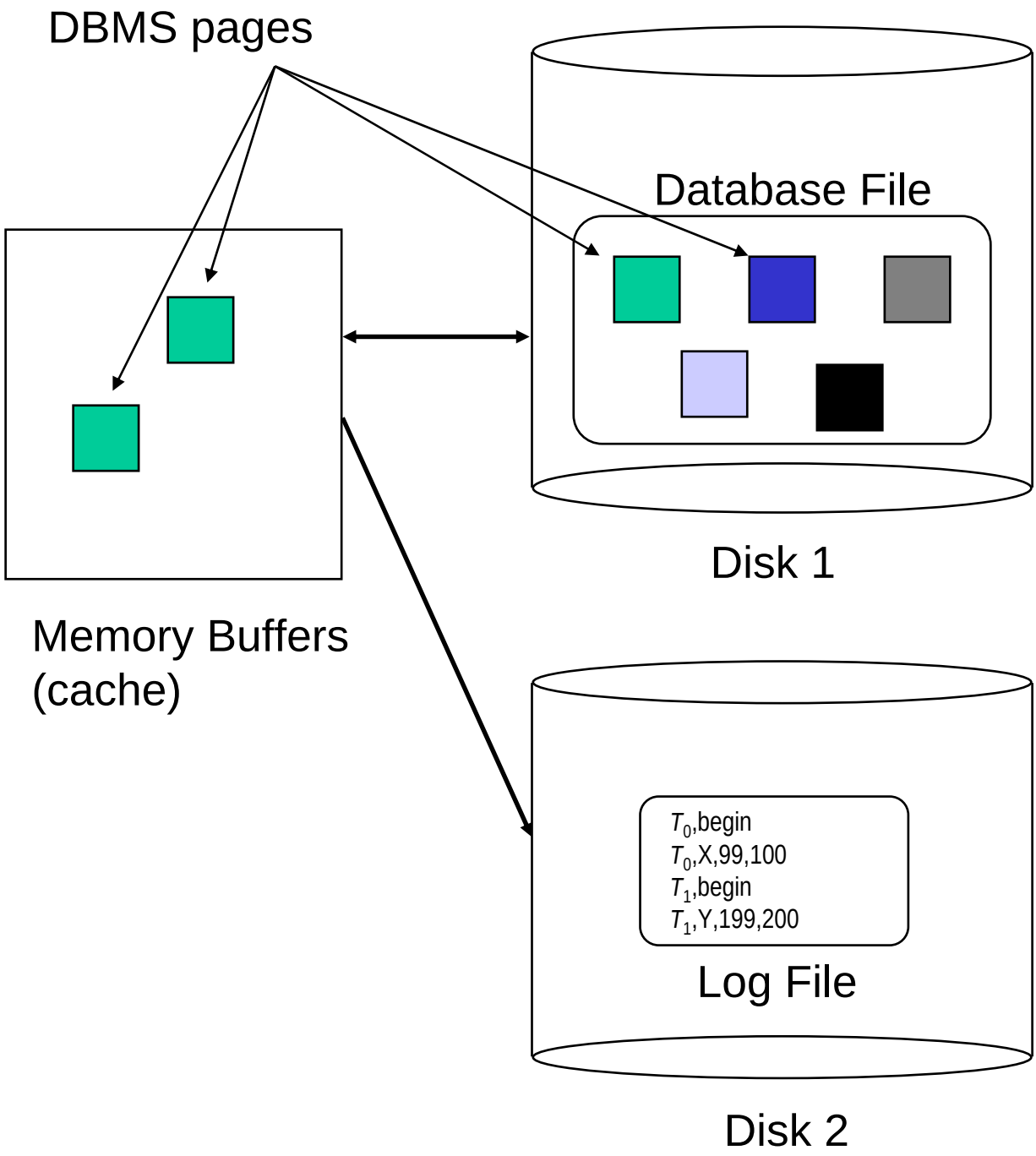
  `rollback [work]`: abort the transaction

- a new transaction begins with the next SQL command after **commit work** or **rollback work**

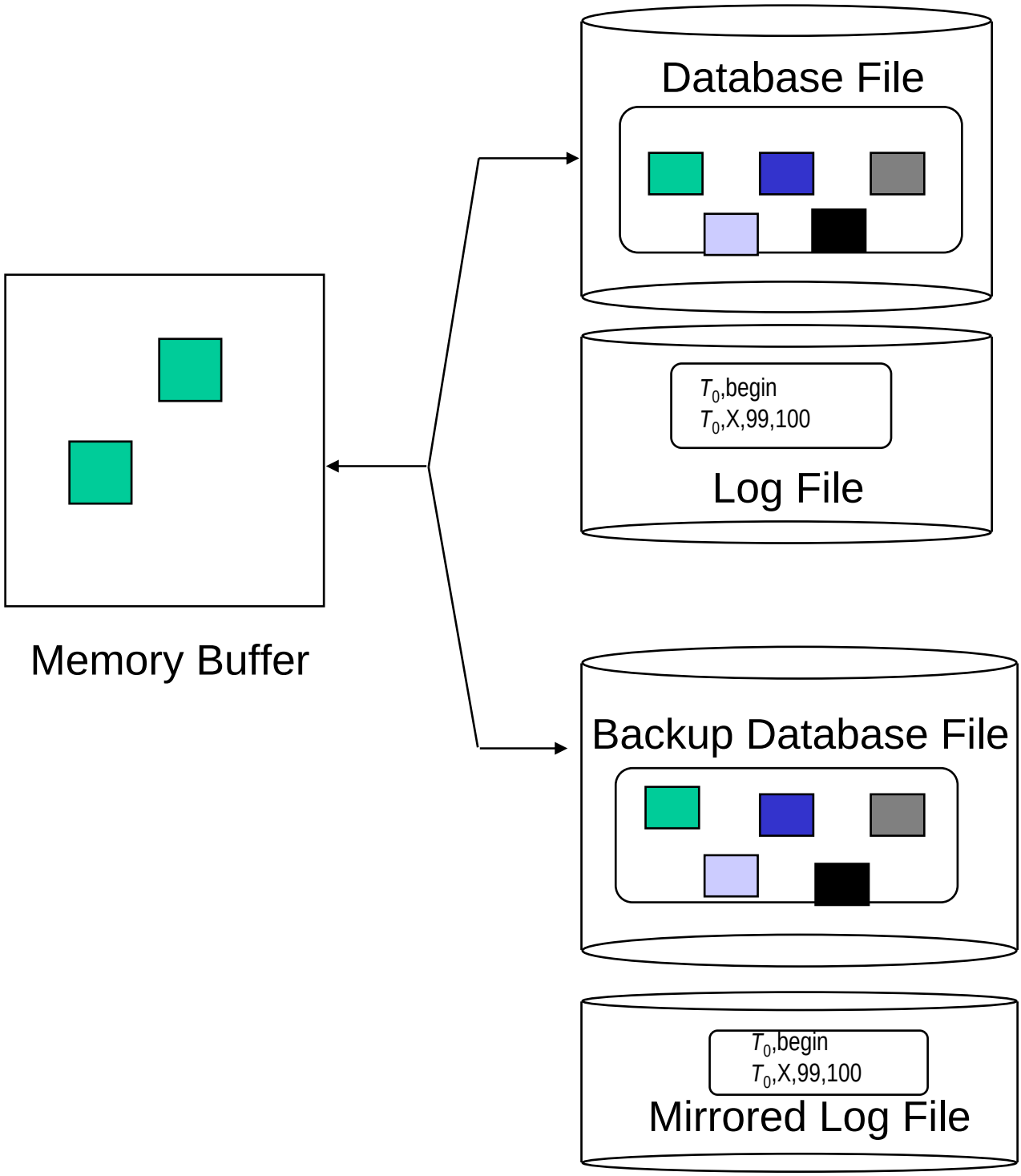
Using transactions

- use of transactions in a DBMS has two aspects:
  - **concurrency control**: guarantees that committed transactions appear to execute sequentially
  - **recovery management**: guarantees that committed transactions are durable, and that aborted transactions have no effect on the database

A DBMS storage model



Backups, mirroring, and multiple disks



Storage management

- Disks are *persistent storage*; permanent, infinite size (relatively) but slow
- Memory is *transient storage*; temporary, limited size, but fast
- DBMS fetches database contents from disk into cache memory
  - DB instance organized into **pages**
  - need **page cache management** strategy
  - handle “out of cache memory” problems, eg LRU, FIFO
  - need to decide when to write from cache back to disk (*page flushing*)
- *In-place* versus *shadow* update
  - in-place rewrites the disk block in the same place
  - shadow creates multiple copies
  - implications for implementation of abort/commit
  - in-place is used in most systems

Deferred vs immediate update

- Deferred update: wait (at least) until transaction commits before updating disk pages
  - changes in memory cache & written to log (but could be *buffered write*)
  - abort is easy
  - commit must ensure all log records written before cache pages are written
  - **force** flushes as soon as committed
- Immediate update: cache manager may write pages as required
  - called **stealing**
  - abort is more complex, must be able to undo
  - must write log records whenever page is flushed
- In either case (assuming in-place update), use *write-ahead logging*

Write-ahead logging

- Used to ensure the log is consistent with the main database
- Two basic rules:
  1. log record must be written before corresponding page is flushed
  2. all log records must be written before commit
- Rule 1 for atomicity
  - so that each operation is known and can be undone if necessary
- Rule 2 for durability
  - so that the effect of a committed transaction is known

Logfile structure

- recovery management is usually done with a database **log**
- database log is a read/append data structure, normally stored in a file
- when DBMS processes transactions, **log records** are appended to the log
- Log record types:

  UNDO information:

  "before" copy of objects that are modified by a transaction. UNDO information is used to undo database changes made by transactions that abort

  REDO information: "after" copy of objects that are modified by a transaction. REDO records are used to redo the work done by a transaction that commits

  BEGIN/COMMIT/ABORT: record transaction boundaries

Example of a log

(oldest part of the log)

log head → T_0,begin

T_0,X,99,100

T_1,begin

T_1,Y,199,200

T_2,begin

T_2,Z,51,50

T_1,M,1000,10

T_1,commit

T_3,begin

T_2,abort

T_3,Y,200,50

T_4,begin

T_4,M,10,100

log tail → T_3,commit

(newest part of log)

Using the log for commit & abort

- To commit a transaction T_i
  - append **T_i,COMMIT** to log
  - ensure all log records actually written (in case of buffering)
  - inform transaction manager commit complete, and end transaction
- To abort a transaction T_i
  - scan log backwards looking for items updated by **T_i**
  - restore old value
  - append **T_i,ABORT** to log
  - inform transaction manager abort complete, and end transaction

Using the log for recovery

- to recover from a *system failure*, the DBMS uses the log:
  - to determine which transactions were active when the failure occurred, and to undo their database updates
  - to recreate the committed updates that may have been lost

- method:
  - scan the log from tail to head (backwards in time):
    - create a list of committed transactions
    - create a list of rolled-back transactions
    - undo updates of active transactions
  - scan the log from head to tail (forwards in time):
    - redo updates of committed transactions
    - ignore rolled-back transactions
  - maybe restart active transactions

...continued

- after recovering from *failure with media damage*:
  - restore database from backup
  - use log to determine which transactions had been committed since the backup
  - redo committed transaction database updates

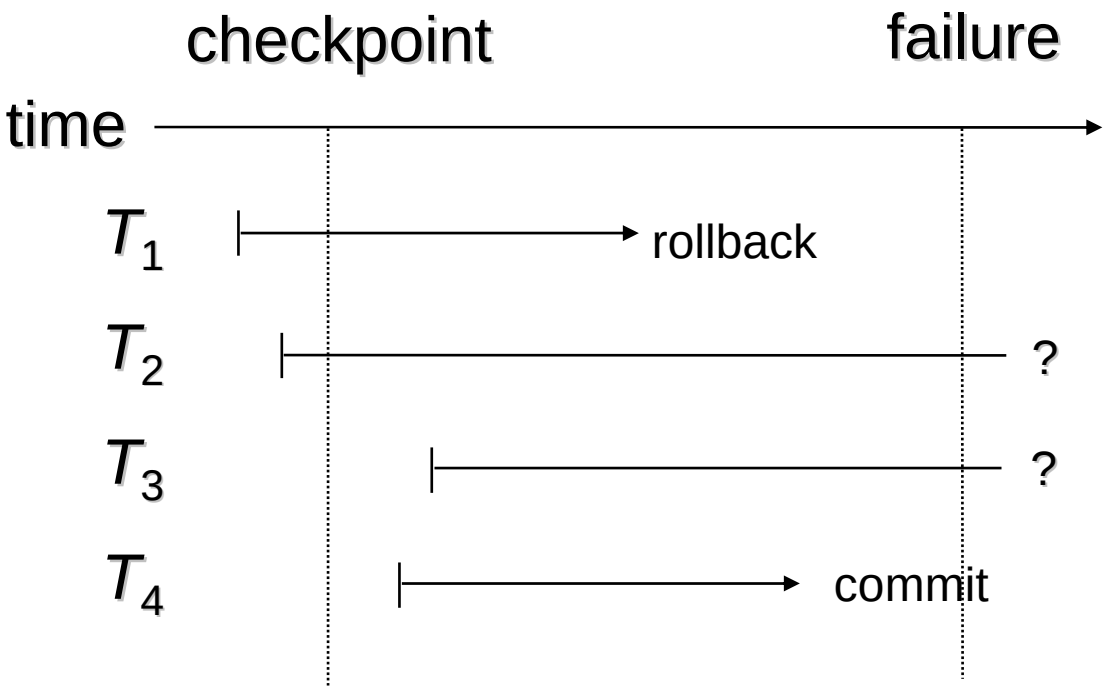
Checkpoints

- as the log grows, the time required to recover from a failure also grows
- **checkpoints** are used to reduce the amount of log data that must be scanned after a system failure

- a simple checkpoint algorithm:
  - prevent new transactions from starting, wait for active transactions to finish
  - copy modified blocks from memory buffer to database files
  - write a CHECKPOINT record in the log
  - allow new transactions to begin

- problems: time-consuming, unacceptable downtime
  - more sophisticated algorithms can improve performance

Example



- scan backward:
  - committed: T_4
  - rolled-back: T_1
  - T_2, T_3 are active, undo
- scan forward
  - redo T_4
  - ignore T_1
- maybe restart T_2, T_3

Concurrency control

- server-oriented DBMS typically processes several transactions simultaneously. This is generally much faster than processing transactions **serially**, i.e., one at a time.
- DBMS must ensure that concurrent transactions appear to be processed serially
- an interleaved execution of a set of transactions is **serializable** if it is equivalent to a serial execution of the same transactions
- notation:
  - $r_i[x]$ means that transaction T_i reads object x
  - $w_i[x]$ means that transaction T_i writes (modifies) object x

Example

- transactions:

  T_1: $r_1[x]$, $x \leftarrow x+1$, $w_1[x]$

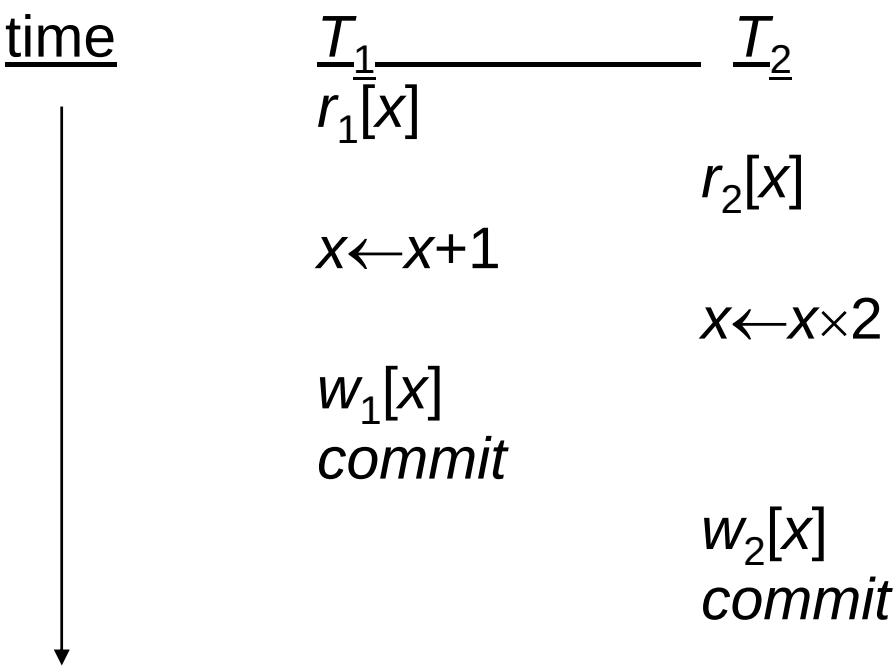
  T_2: $r_2[x]$, $x \leftarrow x \times 2$, $w_2[x]$

- serial executions:

  T_1 then T_2: x is $2 \cdot (x+1)$

  T_2 then T_1: x is $2 \cdot x + 1$

- an interleaved execution:



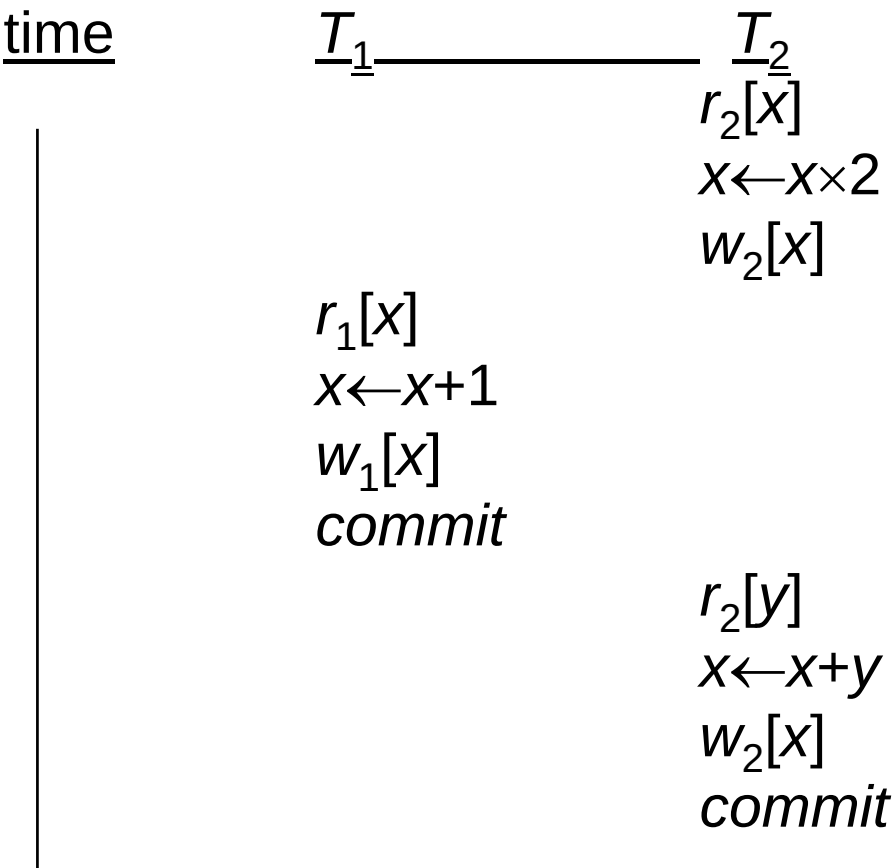
- result x is $2 \cdot x$ (“lost update”, “dirty write”)

Example

- T_2: $r_2[x]$, $x \leftarrow x \times 2$, $w_2[x]$, $r_2[y]$, $x \leftarrow x+y$, $w_2[x]$
- serial executions:

  T_1 then T_2: x is $2 \cdot (x+1)+y$

  T_2 then T_1: x is $2 \cdot x+y+1$
- another execution:



- result x is $2 \cdot x+y$
- T_1 has done a “dirty read”

Serializability

- def^{n}: a **schedule** for a set of transactions is an arbitrary ordering of read and write operations (preserving relative orders within each)
- transactions:
  $T_1 = w_1[x]\ w_1[y]$
  $T_2 = r_2[x]\ r_2[y]$
- some schedules:
  $S_a = w_1[x]\ w_1[y]\ r_2[x]\ r_2[y]$
  $S_b = w_1[x]\ r_2[x]\ w_1[y]\ r_2[y]$
  $S_c = w_1[x]\ r_2[x]\ r_2[y]\ w_1[y]$

- S_a is a serial schedule (T_1, T_2)
- S_b is serializable because it is equivalent to S_a
- S_c is not serializable

Two-phase locking

- most DBMSs use **locking** to guarantee that only serializable executions occur

- before a transaction may read or write an object, it must have a lock on that object
  - a **shared lock** is required to read an object
  - an **exclusive lock** is required to write an object

- there is no “lock” command in SQL -- locks are acquired automatically by the database system

...continued

- rules for locks:
  - any number of transactions may acquire and hold shared locks on the same object
  - only one transaction may acquire an exclusive lock on an object
  - if an exclusive lock is held on an object, no locks by other transactions are permitted
  - a transaction's locks are not released until it commits or aborts, i.e., until it is finished

- this algorithm is called **(strict) two-phase locking (2PL)**
  - growing phase: locks are acquired
  - shrinking phase: locks are released, no new locks are acquired
- text: conservative, basic, strict, rigorous
- two-phase locking guarantees transaction executions to be serializable

Transaction blocking

- a transaction must have a lock on each object it wishes to read or write
- what if a transaction cannot acquire a lock?
- E.g., consider schedule of transactions T_1 and T_2:

  $S = r_1[x]\ w_2[x]$

  T_2 cannot be given the necessary lock on x because of the rule prohibiting a shared and exclusive lock on the same object by different transactions
- when a transaction cannot obtain a lock, it is **blocked** (made to wait) until the lock can be obtained
- in the example above, T_2 must wait until T_1 commits or rolls-back

Deadlocks

- when two-phase locking is used, **deadlocks** may occur

- E.g., consider schedule
  $S = r_1[x]\ r_2[y]\ w_2[x]\ w_1[y]$

- T_1 obtains a shared lock on object x
  T_2 obtains a shared lock on object y
  T_2 requests an exclusive lock on object x, but is blocked
  T_1 requests an exclusive lock on object y but is blocked

- if deadlock occurs, the DBMS must abort one of the transactions involved: called an **involuntary abort**

Isolation levels

- For some applications, the guarantee of serializable executions may carry a heavy price. Performance may be poor because of blocked transactions and deadlocks.
- SQL allows serializability guarantees to be relaxed, if necessary. Four **isolation levels** are supported, with the highest being serializability:
  - Level 3: (serializability)
    - read and write locks are acquired and held until end of transaction
  - Level 2: (repeatable read)
    - identical to Level 3 unless insertion and deletion of tuples is considered
    - “phantom tuples” may occur

...continued

- Level 1: (cursor stability)
  - shared (read) locks are not held until the end of the transaction
  - exclusive (write) locks are held until the end of the transaction
  - non-repeatable reads are possible, i.e., a transaction that reads the same object twice may read a different value each time
- Level 0:
  - no read nor write locks
  - no updates, insertions, or deletions are permitted
  - transaction may read uncommitted updates (of other transactions)

Transactions in distributed servers

- a transaction is officially committed when its commit log record is written
- in distributed DBMS, a transaction may execute at several sites, each with its own log
- a single transaction must not commit at some sites and abort on others
- distributed DBMSs must use an **agreement protocol** to ensure that all sites agree on the fate of each transaction
- most systems use an agreement protocol called **two-phase commit**

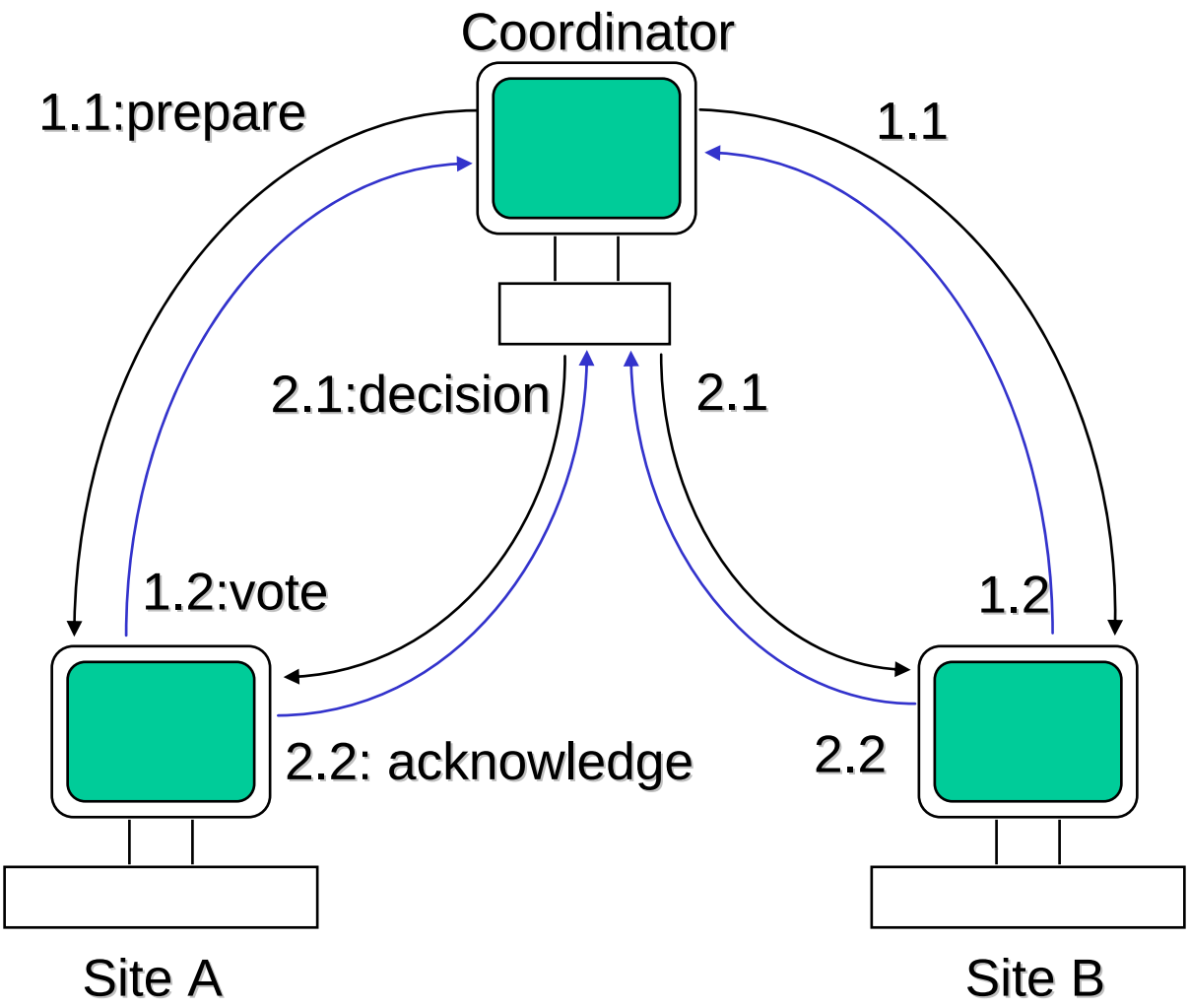
The two-phase commit protocol

- One site acts as the coordinator. The following steps are taken to commit a transaction:
  - the coordinator sends a “prepare” message to the other sites
  - each site decides whether it wants to commit or abort the transaction and sends its vote to the coordinator
  - if abort, it writes an abort record in its log, and votes for abort
  - if commit, it writes a prepare record in its log, and votes for commit

...continued

- If all sites vote commit, the coordinator writes a commit record in its log, otherwise it writes an abort record. The coordinator sends its decision to all of the sites.
- Each site commits or aborts, according to the message from the coordinator. Some versions of 2PC send an acknowledgement to the coordinator.

...continued


Coordinator
1.1:prepare
1.1
2.1:decision
2.1
1.2:vote
1.2
2.2: acknowledge
2.2
Site A
Site B