How Databases Save Your Data

Blue slides originally ©Silberschatz, Korth and Sudarshan

Modifications & additions by S Bird, J Bryson

Part -I: What's a Database

Blue slides originally ©Silberschatz, Korth and Sudarshan

Modifications & additions by S Bird, J Bryson

A first approximation...

- Databases are magic libraries that not only give you functionality but let you store information.
- Databases are servers, you can write clients that access them in many languages.
- Most databases are relational, and their functions are in SQL, more on that next lecture.

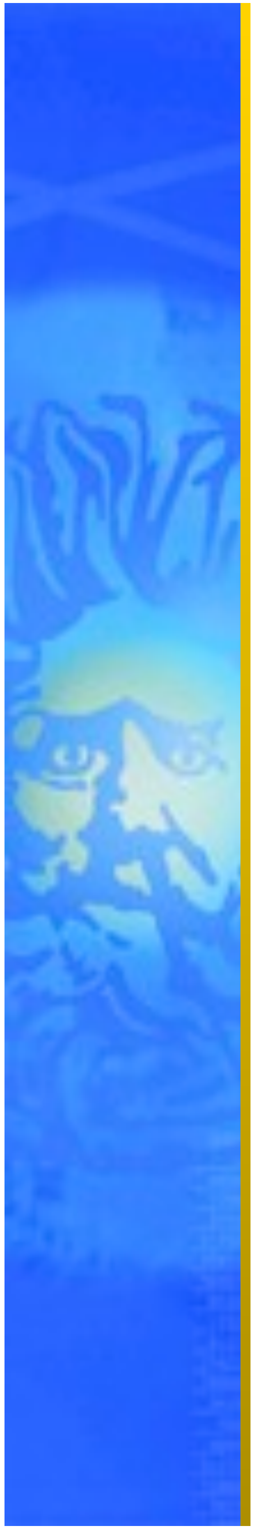
Part 1: What can go wrong?

General Risks (and solutions)

- Crackers: sabotage, theft;
  - backups, honest staff, secure software, up-to-date software, logging to detect suspicious activity, testing, ...
- Failures: lack of business, hardware, network, shipping, software;
  - reliability, redundancy, ...
- Policy changes: laws and taxes catching up with web commerce;
  - monitor legislation, join relevant associations.
- Growth: coping with demand;
  - handling concurrent requests, scalable design.

Security Threats (1 of 2)

- Exposure of confidential data
  - medical records, passwords, contact details, credit cards, ...
  - Solutions:
    - Don’t keep this data on the webserver.
    - Limit the privileges of the database account used by web-accessible scripts.
    - Require end-users to authenticate themselves, store encrypted passwords.
    - Use SSL (Secure Socket Layer, https).
    - Security of database files in the physical filesystem, firewalls, physical security of the server.
- Data loss
  - Web database might contain information that took many months to collect.
  - Loss can be malicious (cracker), inadvertent (admin error) or due to hardware failure.
  - Solutions: security, RAID drives, backups.



Security Threats (2 of 2)

- Data modification
  - Changes to an account balance, additional DB privileges.
  - Hard to detect.
  - Solutions: security, backups, monitoring.
- Denial of service
  - Actions designed to make a service inaccessible or very slow.
- Repudiation
  - One party to the transaction denies having taken part.
  - Solutions: password-based authentication, digital certificates, digital signatures, certification authorities.
- Software errors
  - Poor specifications, false assumptions by developers, poor testing.
  - Solutions: lots of testing, lots of user involvement, contingency plans.

Part 2: The physical safety of data

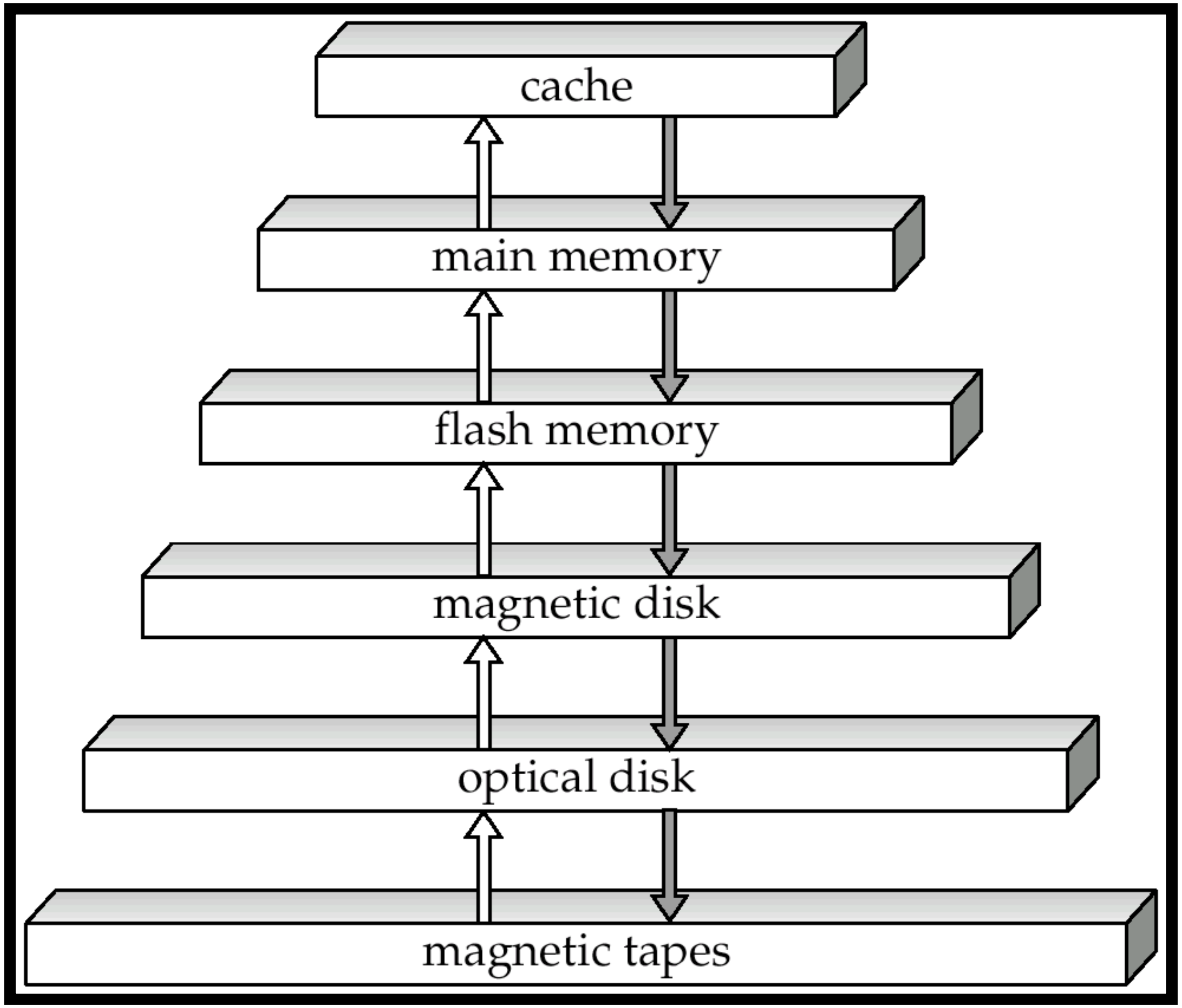
Types of Data Storage

- **Volatile storage**:
  - Does not survive system crashes.
  - Examples: main memory, cache memory.
- **Nonvolatile storage**:
  - Survives system crashes.
  - Examples: disk, tape, flash memory, non-volatile (battery backed up) RAM.
- **Stable storage**:
  - A mythical form of storage that survives all failures.
  - Approximated by maintaining multiple copies on different nonvolatile media stored in different locations.

Data Storage Hierarchy

1. **Primary storage:** Fastest media but volatile (e.g. cache, main memory).
2. **Secondary storage:** next level in hierarchy, non-volatile, moderately fast access time (e.g. flash, magnetic disks).
   - also called **on-line storage**
3. **Tertiary storage:** lowest level in hierarchy, non-volatile, slow access time (e.g. magnetic tape, optical storage).
   - also called **off-line storage**

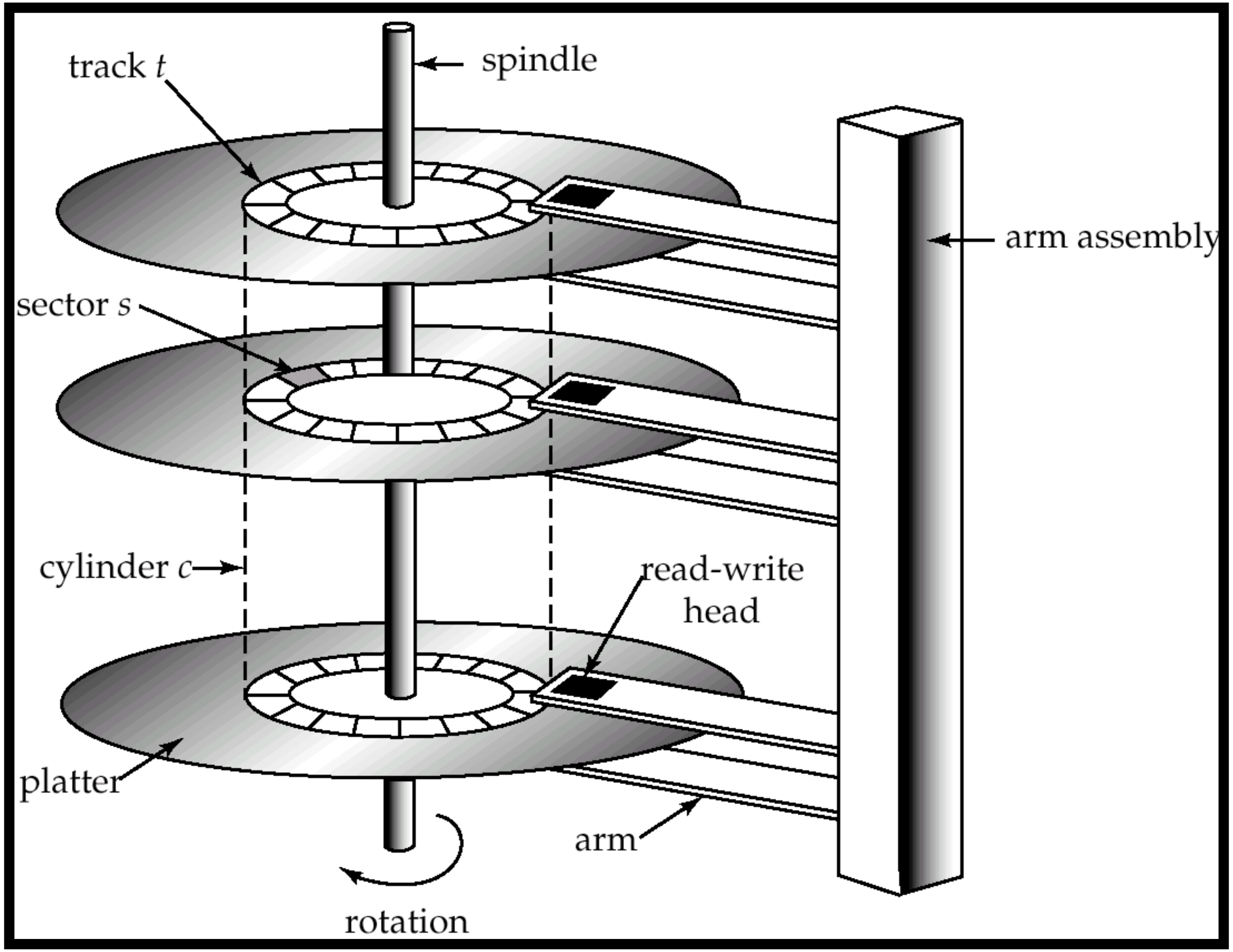
Types of Storage for Digital Data



Magnetic Disks

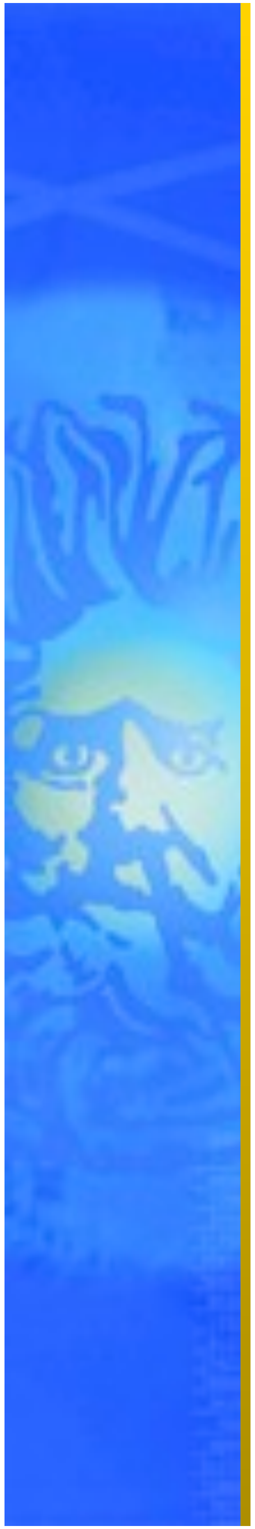
- Data is stored on spinning disk, and read/ written magnetically.
- Primary medium for the long-term storage of data; typically stores entire database.
- Much slower access than main memory, but much cheaper, and non-voliatile.
  - Data must be moved from disk to main memory for quick access by programs, then written back for storage.
- Survives most power failures and system crashes – non-volatile. Disk failure can destroy data, but relatively rare.

Magnetic Hard Disk Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

Part 3: Keeping Data Safe While You Change It



Stable-Storage Implementation

- Chunks of memory are called *blocks*.
- When you change something in a block, you need to make copies.
- Three possible outcomes of copying a block:
  - **Successful completion**,
  - **Partial failure** – destination block has incorrect information, or
  - **Total failure** – destination block was never updated.
- Keeping data safe requires detecting & correcting failures.

Fidelity Through Transactions

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
- A transaction starts with a consistent database.
- During transaction execution the database may be inconsistent.
- A transaction isn’t **committed** (done) until you know the database is consistent.
- Two main issues to deal with:
  - Failures, e.g. hardware failures and system crashes.
  - Concurrency, for simultaneous execution of multiple transactions.
    - Remember this from the threading lectures.

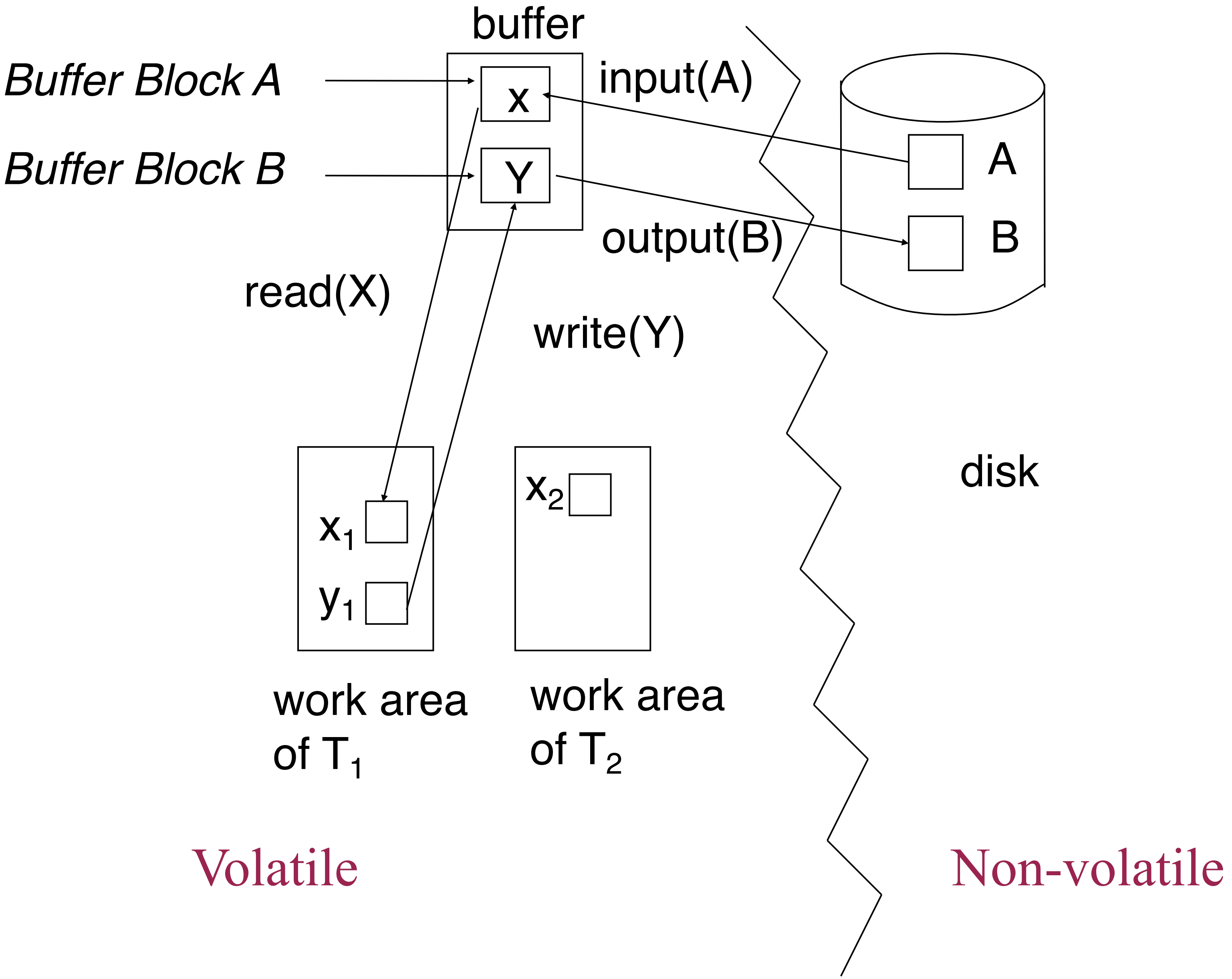
Moving Data Around: Definitions

- **Physical blocks:** blocks residing on the disk.
- **Buffer blocks:** blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
    - **input**(*B*) transfers the physical block *B* to main memory.
    - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.

Moving Data Around

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is called x_i.
- Here we assume (for simplicity) that each data item is stored in a single block.
  - It doesn't have to be, there are simple algorithms for fixing this.
- Transactions are just like the areas we protected in Java using synch() or locking, see the ATM example in lecture

Sample Data Access Diagram



buffer

Buffer Block A

Buffer Block B

x

Y

input(A)

output(B)

A

B

read(X)

write(Y)

disk

x_1

y_1

x_2

work area of T_1

work area of T_2

Volatile

Non-volatile

Moving Data Around (Cont.)

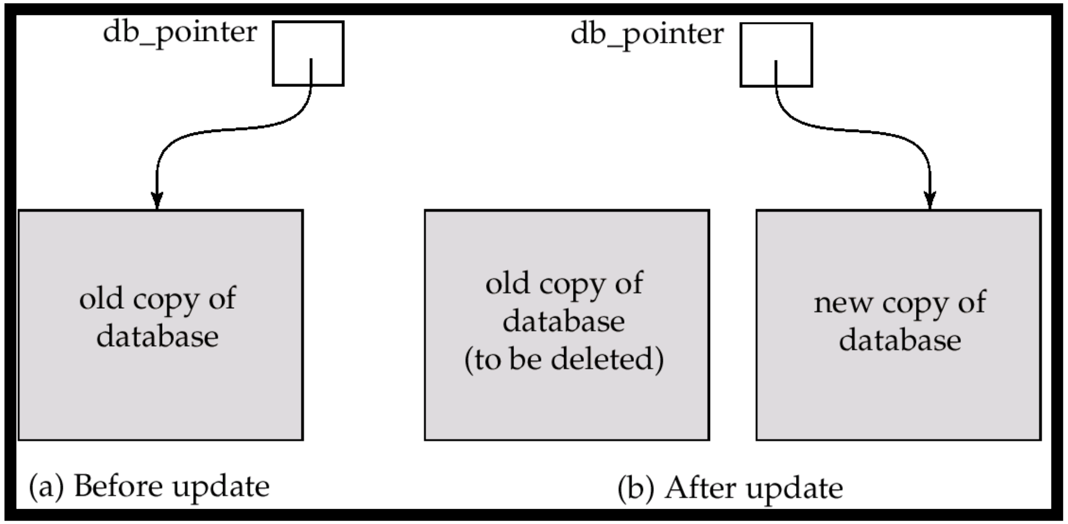
- A transaction transfers data items between system buffer blocks and its private work-area.
- Transactions
  - Perform **read**(X) while accessing X for the first time;
  - All subsequent accesses are to the local copy.
  - After last access, transaction executes **write**(X).
- **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.
- Reminder: Volatile memory is faster, but more vulnerable!
- But until B_X is updated on disk, it’s not safe, so the transaction isn’t finished (committed).

Recovery from Failures

- To ensure data is really saved on non-volatile memory before commitment,
  - first output a description of the modifications to stable storage *without* modifying the database itself.
  - Then update the database.
- Two ways to do this:
  - **shadow-paging (naïve)**, and
  - **log-based recovery.**

Shadow Database

- Assume only *one* transaction is active at a time.
- db_pointer always points to the current consistent copy of the database.
- Updates made on a *copy* of the database. Pointer moved to updated copy *after* transaction reaches partial commit & pages written.
- On transaction failure, old consistent copy pointed to by db_pointer is used, and the shadow copy is deleted.



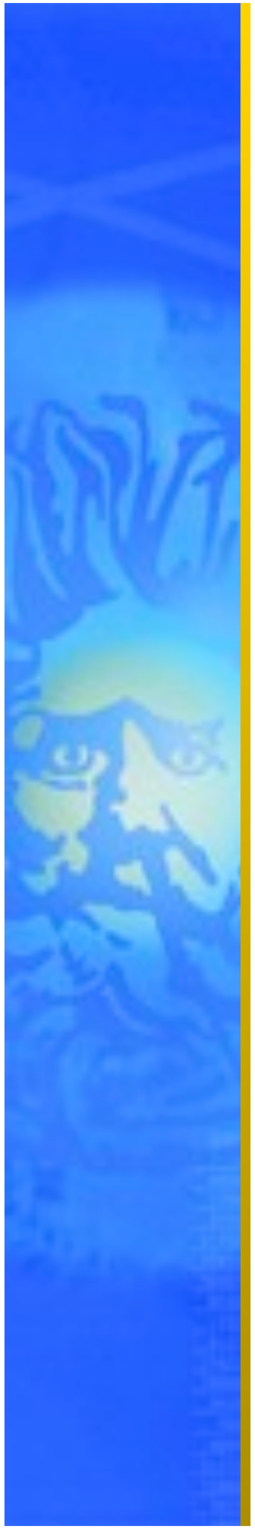
- Assumes disks don't fail.
- Useful for text editors, but extremely inefficient for large database -- executing a single transaction requires copying the *entire* database!

Log-Based Recovery

- A **log** is kept on stable storage.
- A log is a sequence of **log records**, which record the update activities on the database.
  - When transaction T_i starts, it registers itself by writing a $<T_i$ **start**$>$ log record.
  - *Before* T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - When T_i finishes its last statement, the log record $<T_i$ **commi**t$>$ is written. *This* ***is*** *when the transaction T_i is committed!*
  - Periodically output entire database, then you can truncate the log or just write this fact to the log.
- Log records must be written directly to stable storage (they can’t be buffered).

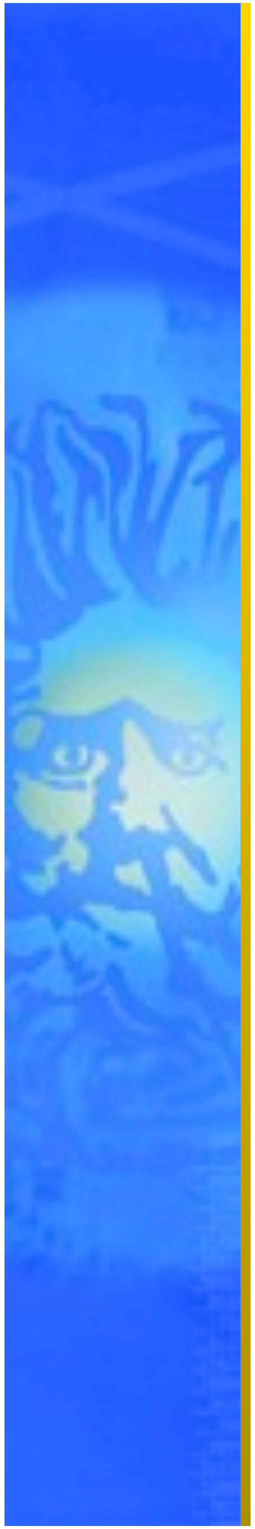
Deferred DB Modification (1/4)

- **Deferred database modification** scheme records all modifications to the log, but defers all **write**s to after a *partial* commitment.
- Transaction starts by writing <T_i ***start***> record to log.
- A **write**(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X.
  - Note: old value is not needed for this scheme.
- The real write is not performed on X at this time, but is deferred.
- When T_i *partially* commits, <T_i **commit**> is written to the log.
- Finally, the log records are used to actually execute the previously deferred writes.



Deferred DB Modification (2/4)

- During recovery, a transaction needs to be redone if and only if both <T_i **start**> and<T_i **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while:
  - the transaction is executing the original updates, or
  - while recovery action is being taken



Deferred DB Modification (3/4)

- Crashes can occur while:
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- Example: T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (A)

A: - A - 50

write (A)

read (B)

B:- B + 50

write (B)

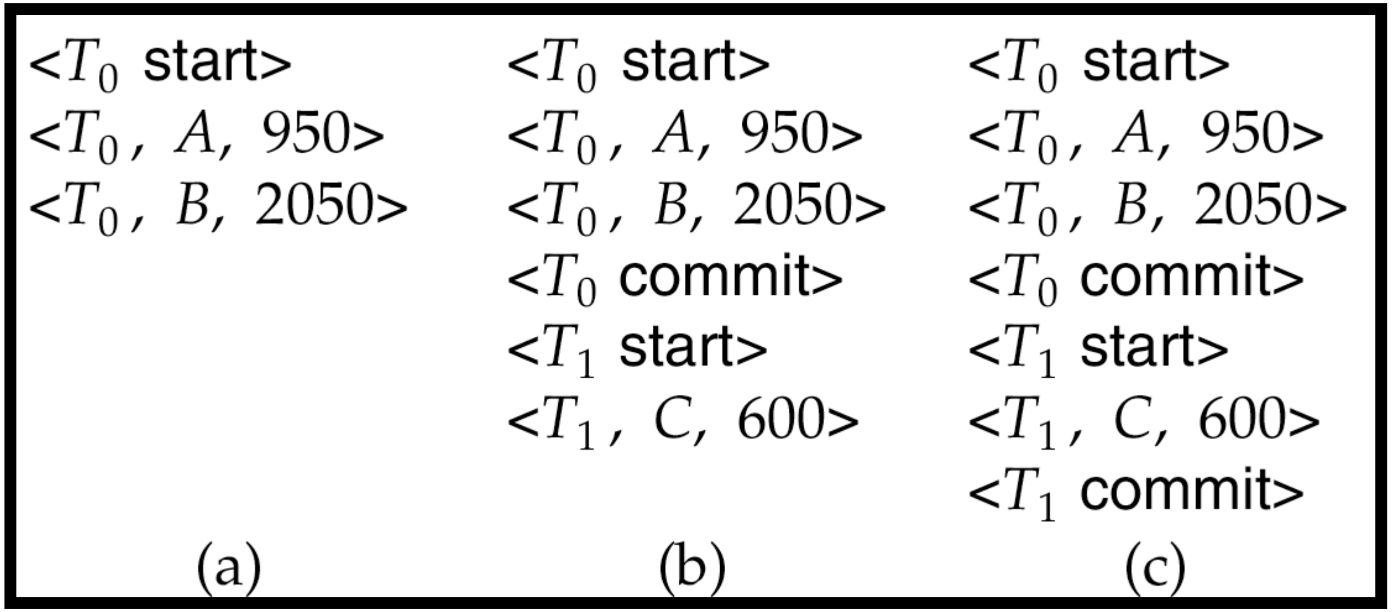
T_1 : **read** (C)

C:- C- 100

write (C)

continued…

Crash Recovery with a Log



(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>

- Assume the disk version of the database has not been updated.
- If log on stable storage at time of crash:
  - (a) No redo actions need to be taken.
  - (b) redo(T_0) must be performed since <T_0 **commi**t> is present.
  - (c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i **commit**> are present.