Redundant Attributes on the Left Sides

• Let \( D = (A \rightarrow B), (AB \rightarrow C) \)

• Let \( D' = (A \rightarrow B), (A \rightarrow C) \)

• \( D \) and \( D' \) are equivalent

• Hence, the attribute \( B \) is redundant in \( (AB \rightarrow C) \)

• When considering whether \( A \) can be removed, the question we must ask is:
  – Can \( C \) be reached from \( B \), without \( A \)?
Testing for Redundant Attributes on the Left Sides

- Let $D$ be a set of dependencies
- Suppose $(AX \rightarrow Y)$ is in $D$, and $A$ is a single attribute
- Then we can replace with $(X \rightarrow Y)$ iff:
  - $Y$ belongs to $X^+$
  - *All* the rules of $D$ are used for this closure
UNIT 7: TRANSACTION PROCESSING

Two parts:

- Recovery management
- Concurrency control
Transactions

- Transaction is a sequence of database operations issued by an application program
- Treated as a complete unit
- One application may generate many transactions
- Often we view transactions abstractly, as a sequence of read and write operations to the database
- Example of two transactions:
  - T1: read x, read y, write z
  - T2: read z, write z
The Fate of a Transaction

- Ultimately, a transaction either commits, or aborts (is rolled back)
- Committed transactions are guaranteed to have completed all their operations
- Aborted transactions must leave no trace on the database
Basic Requirements for Transactions

• Acronym = ACID = Atomicity, Consistency, Isolation, Durability

• Atomicity (indivisibility):
  – Effect of transaction should either be the whole thing (in case of commit), or none of it (in case of abort)
  – I.e., it should never be that only part of the transaction is ultimately applied to the database
  – Example: Suppose T1 = write x, write y. Then if write x, crash, the effect of write x must be undone
• Consistency:
  – If the database starts in a “consistent” state, then the effect of the (whole) transaction should leave the database in a consistent state

• Isolation:
  – Concurrent executions of multiple transactions should not interfere with each other
  – To each application, it should appear that the transaction it issues operations in isolation from all other transactions

• Durability:
  – A committed transaction should have a permanent effect on the database
How are A, C, I and D to be ensured?

• Consistency is the responsibility of the application program, so database theory and systems don’t have much to say

• A, I and D should be *transparently* assured to the application by the database system

• Failures can break A and D:
  – The recovery subsystem will guarantee A and D in the presence of failures

• Concurrency can break I:
  – The concurrency-control subsystem guarantees I
Recovery Manager

• Problem: without a recovery mechanism, failures can break:
  – Atomicity (half-completed transaction)
  – Durability (if committed data gets lost)

• First, we consider recovery alone, by assuming that transactions do not interact
Example: Money Transfer

- Transfer $50 from account a to b
  1. transaction start
  2. read a into a’ (local variable)
  3. a’ := a’ - 50
  4. write a’ onto a
  5. read b into b’
  6. b’ := b’ + 50
  7. write b’ onto b
  8. transaction end
• If initial values are $a = 180$, $b = 100$, then after execution $a = 130$ and $b = 150$

• If RAM is destroyed between instructions 1 and 8, it is not known which was the last instruction executed

• Thus in general, neither of the following naive recovery procedures will work:
  – re-execute the transaction
    Wrong, because if the transaction crashed after step 4, incorrect values will exist in the database
  – do not re-execute
    Wrong, because if crashed before step 7, incorrect values will exist in the database
FAILURES

Several modes of failure:

1. Loss of magnetic (permanent) storage
2. Loss of contents of RAM (e.g., power outage)
3. Program fails in a detectable way (e.g., division by zero)

We focus on the most common scenarios, (2) and (3).
General Forms of Storage

- Permanent (magnetic), stable storage: tapes, disks
- Volatile RAM
- Database and auxiliary ("log") files, on stable storage
- Buffer cache stored in RAM
Performance Notes

• Better for the database system to manage files and buffers, than the O.S.:
  – Use extents for the files
  – Use real, not virtual, memory for the database buffers

• Because sequential, writes to logs are cheaper than writes to database

• Improve performance by:
  – Keeping log on separate disk
  – Putting frequently used tables on different disks
**Two General Principles for Recovery**

1. For an uncommitted transaction, all of the old values must be held in stable storage

2. For a committed transaction, all of the new values must be held in stable storage
The Log File

• Sequential file, that describes actions on the database before they are executed on the database

• Transaction-start record: “t starts”

• Data-write record:
  – Transaction id
  – Variable name
  – Old value
  – New value

• Commit record: “t committed”
Schemes for Updating the Database

1. Store the old values in the log, and force the new values to the database
   - Performance is terrible, because of disk seeks

2. Keep old values in the database, and write new values to log; at commit, transfer new values to database
   - Problem: at commit time, many seeks

3. Write the old values and the new values to the log, and also write the new values to the buffer cache
   - Better because: writes to the log are cheap, and writes to database under control of cache manager
Basic Recovery Algorithm

• From the “beginning of time”, scan the log file:
  – For every transaction that committed, apply all of the writes (i.e., redo)
  – For every transaction that didn’t commit, undo all of the writes

• Can be a long process, if log file is long

• What if the system crashes during recovery?
  – Recovery should be “idempotent”, i.e., repeated recoveries should have the same effect as a single recovery
Checkpointing: Speeds up Recovery

- Periodically, e.g., every 20 minutes:
  - Force all log buffers to the log
  - Force database buffers to the database
  - Force a checkpoint record to the log
- If desired, records before the checkpoint can then be removed from the log
- Revised recovery algorithm:
  - Undo all transactions that have not committed
  - Redo all transactions that have committed after checkpoint
**Other Types of Failures**

- **Program error**
  - Undo transaction
  - Send message to user — transaction bad
  - Mark the log

- **Disk error**
  - restore database from backup copy
  - redo transactions from the log
  - therefore the log may need to be replicated or backed up