Supplemental Notes:
Practical Aspects of Transactions

THIS MATERIAL IS OPTIONAL
Buffer Manager Policies

• **STEAL or NO-STEAL**
  – Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?

• **FORCE or NO-FORCE**
  – Should all updates of a transaction be forced to disk before the transaction commits?

• Easiest for recovery: NO-STEAL/FORCE
• Highest performance: STEAL/NO-FORCE
Solution: Use a Log

• Enables the use of STEAL and NO-FORCE

• **Log: append-only file containing log records**

• For every update, commit, or abort operation
  – Write **physical, logical, physiological** log record
  – Note: multiple transactions run concurrently, log records are interleaved

• After a system crash, use log to:
  – Redo some transaction that did commit
  – Undo other transactions that didn’t commit
Write-Ahead Log

• All log records pertaining to a page are written to disk before the page is overwritten on disk

• All log records for transaction are written to disk before the transaction is considered committed
  – Why is this faster than FORCE policy?

• Committed transaction: transactions whose commit log record has been written to disk
ARIES Method

- Write-Ahead Log

- Three pass algorithm
  - **Analysis pass**
    - Figure out what was going on at time of crash
    - List of dirty pages and running transactions
  - **Redo pass (repeating history principle)**
    - Redo all operations, even for transactions that will not commit
    - Get back state at the moment of the crash
  - **Undo pass**
    - Remove effects of all uncommitted transactions
    - Log changes during undo in case of another crash during undo
ARIES Method Illustration

Figure 3: The Three Passes of ARIES Restart

[Figure 3 from Franklin97]
ARIES Method Elements

• Each page contains a pageLSN
  – Log Sequence Number of log record for the latest update to that page
  – Will serve to determine if an update needs to be redone

• Physiological logging
  – page-oriented REDO
    • Possible because will always redo all operations in order
  – logical UNDO
    • Needed because will only undo some operations
ARIES Method Data Structures

- **Transaction table**
  - Lists all running transactions (active transactions)
  - With `lastLSN`, most recent update by transaction

- **Dirty page table**
  - Lists all dirty pages
  - With `recoveryLSN`, LSN that caused page to be dirty

- **Write ahead log** contains log records
  - LSN
  - `prevLSN`: previous LSN for same transaction
Checkpoints

• Write into the log
  – Contents of transactions table
  – Contents of dirty page table

• Enables REDO phase to restart from earliest recoveryLSN in dirty page table
  – Shortens REDO phase
Analysis Phase

• Goal
  – Determine point in log where to start REDO
  – Determine set of dirty pages when crashed
    • Conservative estimate of dirty pages
  – Identify active transactions when crashed

• Approach
  – Rebuild transactions table and dirty pages table
  – Reprocess the log from the beginning (or checkpoint)
    • Only update the two data structures
  – Find oldest recoveryLSN (firstLSN) in dirty pages tables
Redo Phase

- **Goal:** redo all updates since firstLSN
- **For each log record**
  - If affected page is not in the Dirty Page Table then **do not update**
  - If affected page is in the Dirty Page Table but recoveryLSN > LSN of record, then **no update**
  - Else if pageLSN > LSN, then **no update**
    - Note: only condition that requires reading page from disk
  - Otherwise perform update
Undo Phase

- **Goal:** undo effects of aborted transactions
- Identifies all loser transactions in trans. table
- Scan log backwards
  - Undo all operations of loser transactions
  - Undo each operation unconditionally
  - All ops. logged with compensation log records (CLR)
  - *Never undo a CLR*
    - Look-up the UndoNextLSN and continue from there
Handling Crashes during Undo

Figure 4: The Use of CLRs for UNDO

[Figure 4 from Franklin97]
Implementation: Locking

- Can serve to enforce serializability
- Two types of locks: Shared and Exclusive
- Also need two-phase locking (2PL)
  - Rule: once transaction releases lock, cannot acquire any additional locks!
  - So two phases: growing then shrinking
- Actually, need strict 2PL
  - Release all locks when transaction commits or aborts
Phantom Problem

- A “phantom” is a tuple that is invisible during part of a transaction execution but not all of it.

- Example:
  - T0: reads list of books in catalog
  - T1: inserts a new book into the catalog
  - T2: reads list of books in catalog
    - New book will appear!

- Can this occur?
- Depends on locking details (eg, granularity of locks)
- To avoid phantoms needs predicate locking
Deadlocks

- Two or more transactions are waiting for each other to complete

- **Deadlock avoidance**
  - Acquire locks in pre-defined order
  - Acquire all locks at once before starting

- **Deadlock detection**
  - Timeouts
  - Wait-for graph (this is what commercial systems use)
Degrees of Isolation

- Isolation level “serializable” (i.e. ACID)
  - Golden standard
  - Requires strict 2PL and predicate locking
  - But often too inefficient
  - Imagine there are few update operations and many long read operations

- Weaker isolation levels
  - Sacrifice correctness for efficiency
  - Often used in practice (often default)
  - Sometimes are hard to understand
Degrees of Isolation

• Four levels of isolation
  – All levels use **long-duration exclusive locks**
  – **READ UNCOMMITTED**: no read locks
  – **READ COMMITTED**: short duration read locks
  – **REPEATABLE READ**:
    • Long duration read locks on individual items
  – **SERIALIZABLE**:
    • All locks long duration and lock predicates

• **Trade-off: consistency vs concurrency**
• Commercial systems give choice of level
Lock Granularity

- **Fine granularity locking** (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
- **Coarse grain locking** (e.g., tables)
  - Many false conflicts
  - Less overhead in managing locks
- **Alternative techniques**
  - Hierarchical locking (and intentional locks)
  - Lock escalation
The Tree Protocol

• An alternative to 2PL, for tree structures
• E.g. B-trees (the indexes of choice in databases)

• Because
  – Indexes are hot spots!
  – 2PL would lead to great lock contention
The Tree Protocol

Rules:

• The first lock may be any node of the tree
• Subsequently, a lock on a node A may only be acquired if the transaction holds a lock on its parent B
• Nodes can be unlocked in any order (no 2PL necessary)
• “Crabbing”
  – First lock parent then lock child
  – Keep parent locked only if may need to update it
  – Release lock on parent if child is not full

• The tree protocol is NOT 2PL, yet ensures conflict-serializability!
Other Techniques

- DB2 and SQL Server use strict 2PL
- Multiversion concurrency control (Postgres)
  - Snapshot isolation (also available in SQL Server 2005)
  - Read operations use old version without locking
- Optimistic concurrency control
  - Timestamp based
  - Validation based (Oracle)
  - Optimistic techniques **abort** transactions instead of blocking them when a conflict occurs
Summary

• Transactions are a useful abstraction

• They simplify application development

• DBMS must be careful to maintain ACID properties in face of
  – Concurrency
  – Failures