Final Review
CS6360

• Final exam:
  – Comprehensive
• (Option 1) 12/09/2015, Wednesday,
  – 1:00pm – 2:15pm, ECSS 2.201
• (Option 2) 12/16/2015, Wednesday,
  – 2:00pm – 3:15pm, ECSS 2.201
• Midterm Review
• After Midterm Review
Midterm Review

- Major Database Concepts (Chap 1, 2)

- Conceptual Modeling
  - ER Model (Chap 7)
    - Exclude 7.8
  - EER (Chap 8)
    - Exclude 8.6, 8.7

- Relational Data Model
  - Concepts & Constraints (Chap 3.1-3.4)

- ER/EER-to-Relational Mapping (Chap 9)
Post-midterm Review

- Relational Data Model: Query languages
  - Relational algebra (6.1-6.5)
  - SQL (Chap 4)
  - More SQL (Views in SQL, etc) (Chap 5)
- DB Design Theory (Chap 15, 16)
  - Functional dependencies (Chap 15.1-15.2)
  - Normalization (Chap 15.3-15.5)
  - Relational DB Design Algorithm (Chap 16.1-16.3)
- TransX Processing Concepts (TPCs)
  - Introduction to TPCs (21.1)
    - What is transX?
    - Operations: r, w, begin, end, c, a
    - Why concurrency control and recovery?
  - TransX and System Concepts (21.2)
    - TransX states
    - System log
  - ACID properties (21.3)
  - Schedules & Serializability (Chapter 21.5)
Functional Dependency

• **Definition:** A *functional dependency* (FD) on a relation schema $R$ is a constraint $X \rightarrow Y$, where $X$ and $Y$ are subsets of attributes of $R$, such that for every pair of tuples, $t$ and $s$, in an instance of $R$, if $t$ and $s$ agree on all attributes in $X$ then they must agree on all attributes in $Y$

  – Key constraint is a special kind of functional dependency: all attributes are on LHS

  • $SSN \rightarrow SSN, \text{Name, Address}$
Mathematical Properties of FDs

• **Definition:** If $F$ is a set of FDs on schema $R$ and $f$ is another FD on $R$, then $F$ *entails* $f$ if every instance $r$ of $R$ that satisfies $F$ also satisfies $f$
  
  • Ex: $F = \{A \rightarrow B, B \rightarrow C\}$ and $f$ is $A \rightarrow C$

  • If Streetaddr $\rightarrow$ Town and Town $\rightarrow$ Zip then
    Streetaddr $\rightarrow$ Zip

  • **Definition:** The *closure* of $F$, denoted $F^+$, is the set of all FDs entailed by $F$

• **Definition:** $F$ and $G$ are *equivalent* if $F$ entails $G$ and $G$ entails $F$
Armstrong’s Axioms of FDs

- **Reflexivity (IR1):** If $Y \subseteq X$ then $X \rightarrow Y$ (trivial FD)
  
  - *Name, Address* $\rightarrow$ *Name*

- **Augmentation (IR2):** If $X \rightarrow Y$ then $XZ \rightarrow YZ$
  
  - If *Town* $\rightarrow$ *Zip* then *Town, Name* $\rightarrow$ *Zip, Name*

- **Transitivity (IR3):** If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$
  
  - E.g. HW3 pp328 10.18 (a, b)
Thus, $AB \rightarrow BD$, $AB \rightarrow BCD$, $AB \rightarrow BCDE$, and $AB \rightarrow CDE$ are all elements of $F^+$.
Attribute Closure

- Calculating attribute closure is a more efficient way of checking entailment.

- The *attribute closure* of a set of attributes, $X$, with respect to a set of functional dependencies, $F$, (denoted $X^+_F$) is the set of all attributes, $A$, such that $X \rightarrow A$
  
  - $X^+_F$ is not necessarily the same as $X^+_F$

- Checking entailment: Given a set of FDs, $F$, then $X \rightarrow Y$ if and only if $X^+_F \supseteq Y$

- E.g. Ex 10.20 (PP329)
Minimal Cover

• A *minimal cover* of a set of dependencies, $T$, is a set of dependencies, $U$, such that:
  – $U$ is equivalent to $T$ ($T^+ = U^+$)
  – All FDs in $U$ have the form $X \rightarrow A$ where $A$ is a single attribute
  – It is not possible to make $U$ smaller by
    • Deleting an FD
    • Deleting an attribute from an FD
Computing Minimal Cover

• **Example:** $T = \{ABH \rightarrow C, A \rightarrow D, C \rightarrow E, BGH \rightarrow F, F \rightarrow AD, E \rightarrow F, BH \rightarrow E\}$

• **step 1:** RHS of each FD is a single attribute.
  - $F \rightarrow AD$ replaced by $F \rightarrow A, F \rightarrow D$

• **step 2:** Eliminate unnecessary attributes from LHS.
  - Algorithm: If FD $XY \rightarrow Z \in T$ (where $Y$ and $Z$ are single attributes) and $X \rightarrow Z$ is entailed by $T$, then $Y$ was unnecessary
Normal Forms

• Each normal form is a set of conditions on a schema that guarantees certain properties (relating to redundancy and update anomalies)

• The two commonly used normal forms are third normal form (3NF) and Boyce-Codd normal form (BCNF)
BCNF

• **Definition:** A relation schema \( R \) is in BCNF if for every FD \( X \rightarrow Y \) associated with \( R \) either
  – \( Y \subseteq X \) (i.e., the FD is trivial) or
  – \( X \) is a superkey of \( R \)

• **Example:** \( \text{Person1}(\text{SSN}, \text{Name}, \text{Address}) \)
  – The only FD is \( \text{SSN} \rightarrow \text{Name, Address} \)
  – Since \( \text{SSN} \) is a key, \( \text{Person1} \) is in BCNF
Third Normal Form

• A relational schema $R$ is in 3NF if for every FD $X \rightarrow A$ associated with $R$ either:
  – $A \subseteq X$ (i.e., the FD is trivial) or
  – $X$ is a superkey of $R$ or
  – $A$ is part of some key

• 3NF is weaker than BCNF (every schema that is in BCNF is also in 3NF)
Second Normal form

• If every nonprime attribute A in R is not partially dependent on any key of R.
Transaction Processing Concept
(Chap 17.1 – 17.5)

• Ignore 17.6

• Transaction

• System log
  – Recovery

• Schedule
  – Serializability (17.5)
    • Serial, interleaved (serializable/leagal, non-serializable/non-legal) schedules
    • Local serializability
    • Conflict graph (CG)/precedent graph
  – Recoverability (17.3)
The System Log

• **Log or Journal**: The log keeps track of all transaction operations that affect the values of database items.
  - to permit recovery from transaction failures
  - kept on disk, not affected by any type of failure except for disk or catastrophic failure
  - In addition, the log is periodically backed up to archival storage (tape)

• **Types of Log Records**:
  1. `[start_transaction,T]`: Records that transaction T has started execution.
  2. `[write_item,T,X,old_value,new_value]`: Records that transaction T has changed the value of database item X from old_value to new_value.
  3. `[read_item,T,X]`: Records that transaction T has read the value of database item X.
  4. `[commit,T]`: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
  5. `[abort,T]`: Records that transaction T has been aborted.
Recovery Using Log Records

- **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - log contains a record of every *write* operation that changes the value of some database item,
  - **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.

- **Redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.
  - **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.
Characterizing Transactions and Schedules(17.3)

• Desirable Properties (ACID)
  – Atomicity: performed all actions or none
  – Consistency Preserving: moves database across consistent states
  – Isolation: T's updates are not visible to other Ts before commit
  – Durability: committed changes to DB are permanent

• Schedule of concurrently executing interleaved transactions
  – Trace from database
  – Def: schedule S (T₁, ..., Tₙ) = an ordering of operations of Tᵢ, s.t.
  – order of operations of Tᵢ in S = order of operations in Tᵢ for all Tᵢ.
  – ordering could be total or partial.
  – Ex. Fig. 17.3(a), (b) :: shorthand rᵢ(X), wᵢ(Y), cᵢ, aᵢ, ...
  – Committed projection: C(S) = operations from committed Ts
Exercises

• Q1? How many possible schedules for
  – Xactions in (a) Fig. 21.2
  – Xactions in (b) Fig. 21.8

• Q2? Count number of schedules for T1, ..., Tn,
  – Where \( N_i = \) number of operations in Ti
  – \( N = N_1 + N_2 + ... + N_n \)
Exercises

• Answer 1:
  – (a) \( \frac{9!}{6! \times 3!} = 84 \)
  – (b) \( \frac{13!}{5! \times 4! \times 4!} = 90900 \)

• Answer 2:
  – The ordering of \( N_i \) actions within \( T_i \) preserved = >
  – lose \( (N_i!) \) permutations for each valid schedule
  – \#schedules \( S(T_1, ..., T_n) = \frac{N!}{(N_1! \times N_2! \times ... \times N_n!)} \)
Serializability of Schedules (21.5)

• Why is serializability interesting?
  – It is the characterization of "correctness", legal interleavings

• Serial, Non-serial and serializable schedules (Fig. 21.5)
  – Serial S: no interleaving, i.e. all actions $T_i$ together in $S$
  – Nonserial $S$: interleaving, for some $T_i$, all actions are not together
  – Serializable $S$, if $S$ is eqv. to some serial schedule of same $Ts$

• ...used as correctness measure
Serializability of Schedules (Cont.)

• Conflict Serializable(S) iff conflict-equivalent(S, a serial schedule)

• Equivalence of schedules S1, S2
  – Result eqv.: if they produce the same final state of DB
    • not enough for correctness, see Fig. 21.6
  – Conflict eqv. order of any two conflicting operation is identical
    • < Wi(X),..., Wj(X) > or < Wi(X),...,Rj(X) > or < Ri(X),...,Wj(X) >
Testing Conflict serializability
(Algo. 21.1, pp 763)

• Look at only ReadItem (X) and WriteItem (X) operations

• Constructs a precedence graph
  – Nodes = transactions T1, T2, ..., Tn
  – Edges = conflists (read/write, write/read, write/write)
  – ...(Ti -- > Tj) if before(wi(X),rj(X)) in S
  – ...(Ti -- > Tj) if before(ri(X),wj(X)) in S
  – ...(Ti -- > Tj) if before(wi(X),wj(X)) in S

• No cycle in predence graph(S) IFF conflict serializable(S)
Recoverability Properties of Schedules

- **Strict schedule** ($S$): iff $(w_j(X) < c_j < r_i(X))$, $(w_j(X) < c_j < w_i(X))$
  - no concurrent read/write on any data item!
  - recovery is easy $\Rightarrow$ restore before image of $X$

- **Recoverable with cascaded rollback**
  - iff forall $Ti, Tj$ in $S$: reads_from($Ti, Tj, S$) $\Rightarrow$ ($c_j < c_i$)
  - $commit(Ti)$ waits for $commit(Tj)$
  - $Ti$ self-aborts if $Tj$ has aborted
  - Assumes $abort(Tj)$ can not rollback committed transactions.

- **No cascaded rollback**
  - $Cascadeless(S)$ iff reads_from($Ti, Tj, S$) $\Rightarrow$ ($c_j < r_i(X)$)
  - No dirty read (strong condition to avoid cascaded rollbacks)
Supplemental Review Slides

• For PhD Qualifier Exam (QE) students
Classification of CCTs

• 18.1 2PL
  – lock types, basic/2PL/strict
  – deadlocks
• 18.2 Timestamp Ordering
  – timestamps, ordering algo
• 18.3 Multiversion CCT
  – w time-stamps, 2-phase
• 18.4 Validation (Optimistic) based CCT
What is concurrent control?

• Def.: Protocols, Algorithms, Techniques to avoid problems
  – resulting from concurrent operations on shared data
  – Problems of lost update, incorrect summary …

• A Well-known problem in O.S., Client-Server, Shared Resource
  – Ex. Concurrent processes/threads in Operating Systems
    • Mutual Exclusion Protocols: semaphores, critical sections
Common Concurrency Control Techniques (CCTs)

• Goal:
  – Quickly ensure serializability
  – Fast filter to avoid violation of serializability
• may rule many possible serializable schedule
• Prescribes a Protocol to be observed by each transaction
• \(< \text{current state, new op. request} > \rightarrow \text{grant/delay op, abort T}\)
Simple CCTs

• 2 phase locking (2PL) protocol for transactions
  – Schedule: order Ti by the order of lock acquisition

• TSO: item-based violation test at read/write
  – Schedule: order Ti by their time-stamps

• Other schemes
  – Optimistic, 3 phases (read, validate, write)
    • Validate phase = conflict? w/committed/validated Tj
  – Multiversion - uses version semantics

32
Lock Based CCT : Basic Definitions

- \( \text{LOCK}[X] = \text{a variable describing status of data item } X \)  
  - wrt possible applicable operations without conflict.

- \( \text{Lock}(X, \text{op}) = \text{permission to Ti to perform op on } X \)  
  - NOTE Convention: \( \text{LOCK}(X) = \text{variable} \), \( \text{Lock}(X) = \text{permission} \)

- Operations on \( \text{LOCK}(X) \) requested by Ti to Lock Manager  
  - lock_item(X), lock_item(X, read), unlock(X), ...  
  - Protocol: Ti needs Lock(X, op) to perform op(X)  
  - Lock Manager: (operation on \( \text{LOCK}(X) \)) \( \rightarrow \) \{ grant, postpone, deny \}  
  - See Fig. 18.1, 18.2 (pp 643-645) for algorithms
Types of Locks

• **binary locks:**
  – Binary values (states): unlock= 0, lock=1
  – ops.: lock_item(X), unlock_item(X))
  – See protocol rules on pp 643

• **Read/Write (Shared/Exclusive) locks:**
  – 3 Values:
    • unlocked, read_locked (shared_locked), write_locked(exclusive_locked).
  – 3 ops:
    • read_lock(X) is shared: Multiple read locks can be allowed on X.
    • write_lock(X) exclusive, no other concurrent lock on X
    • unlock(X)
  – See pp644-646 the list of locking rules

• **Conversion of Locks**
  – Relax rule 4 and 5
  – Can convert the lock from one lock state to another
    • Upgrade lock: read_lock(X) → write_lock(X)
    • Downgrade: write_lock(X) → read_lock(X)
Implementing Serializability: Two-Phase Locking

• Locks are associated with each data item
• A transaction must acquire a read (shared) or write (exclusive) lock on an item in order to read or write it
• A write lock on an item conflicts with all other locks on the item; a read lock conflicts with a write lock
• If T1 requests a lock on x and T2 holds a conflicting lock on x, T1 must wait
Lock based CCT: 2 PL protocol

• Two-phase Locking (2PL): Ti follows 2PL if
  – all locking ops precede the first unlock op in T

• Two Phases relate to the text/execution of T
  – Phase 1: Expand / Grow the set of permissions (Locks)
  – Phase 2: Shrink - Release the Locks

• Variation of 2PL:
  – To get Strict Schedules (see pp622 17.4.2), Use Strict 2PL
  – Strict 2PL if T commits or aborts before any unlock() operations.
  – Strict 2PL does not avoid deadlocks.
Lock Release

Two-Phase locking: All locks are acquired before any lock is released

Strict: Transaction holds all locks until completion
Deadlock and Livelock Problems

- Livelocks or Deadlocks - possible with 2PL
- When a transaction can hold locks and request another lock (e.g., in two-phase locking), a cycle of waiting transactions can result:
  - `r_lock1(Y), r1(Y), r_lock2(X), r2(X), w_lock1(X), w_lock2(Y)`
  - pp 650 Fig 18.5
- A transaction in the cycle must be aborted by DBMS (since transactions will wait forever)
- DBMS uses deadlock detection algorithms or timeout to deal with this problem
Lock based CCT: Solving Deadlock Problems

• Priority based on age (timestamp of first submission)
  – E.g. TS(T1) < TS(T2)
• Solution 1: Prevent Deadlocks
  – Conservative 2PL if T acquires all locks before T starts execution.
  – No deadlocks, if all locks requested together (no hold & wait).
• Sol. 2: Detect and Resolve Deadlocks
  – Wait-for graph (nodes = Ts, edge = wait-for dependency)
  – cycle in WFG => deadlock
  – Break deadlock by aborting Ti to break cycles in WFG
• Sol. 3: Timeouts: Ti aborts if all locks are not granted within timeout!
  wait-die (new time-stamp) or wait-wound (no change in time-stamp)
CCT 2: Timestamp Ordering (pp594 18.2)

- IDEA: Make schedule eqv. to serial schedule
  - defined by the ordering of time-stamps (transX start time) of transactions
  - TS(T): timestamp of transX T
  - No deadlock

- Thus, Abort Ti operate on data-item X, such that
  - Ti reads /writes X with write_TS(X) > TS(Ti)
  - OR Ti writes X with read_TS(X) > TS(Ti)
  - WHERE
    - ...TS(Ti) = start time of a transaction Ti
    - ...read_TS(X) = largest TS(Ti) for any Ti that has already read X.
    - ...write_TS(X) = largest TS(Ti), any Ti has already written X.

- Timestamp Ordering ensures serializability
  - Note: Potential conflict detected before operation and avoided.
  - Conflict eqv. schedule = order Ti in S by their timestamps.

- Comparison of TSO, 2PL, CGS
  - 2PL/TSO limit concurrency: rules out some serializable schedules
  - 2PL and TSO produce different subsets of CGS