Functional Dependency:

* Reasons for normalizing relational schema => removing update anomalies, facilitating maintenance of integrity constraints
* BCNF vs. 3NF:
  + 3NF guarantees lossless decomposition and dependency preservation
  + BNF does not
* Computing BCNF:
  + Find super keys (compute closure => F+)
    - Either it doesn't show up on RHS or do combinations
  + Check for each new relation if it is BCNF
  + If not, keep decomposing
* Is BCNF?
  + If for each FD: (1) Trivial or (2) LHS is a super key
* Is 3NF?
  + If for each FD: (1) Trivial or (2) LHS is a super key or (3) RHS is part of a super key
  + All relations that are BCNF are 3NF
  + Ex: R = ABCD, FD = (ABC -> D, D -> A)
    - Super keys are ABC, BCD
    - ABC -> D [ABC is a super key]
    - D -> A [A is part of super key ABC]
      * Yes, 3NF relation
* Is lossless?
  + If R is decomposed to R1 and R2, then is lossless if:
    - Attr(R1) union Attr(R2) = Attr(R)
    - Attr(R1) intersection Attr(R2) is not the empty set
    - Attr(R1) intersection Attr(R2) -> (R1 or R2)
  + Ex: R = ABCD, FD = (A -> BC)
    - Super key is AD
    - R1(AD), R2(ABC)
      * R1 union R2 = ABCD = R [check]
      * R1 intersect R2 = A is not empty set [check]
      * R1 intersect R2 -> R2 [check]
        + Yes, it is lossless
* Is BCNF decomposition dependency preserving?
  + Decomposition D = {R1, R2, …, Rn} of R is dependency preserving with respect to set F of FDs if for R1 with FD {F1} and R2 with FD {F2}:
    - F1 union F2 = F [Dependency preserving]
    - F1 union F2 = subset of F [Not dependency preserving]
  + Ex: R = ABCD, FD = {AB -> C, C -> D, D -> A}
    - R1(ABC), R2(CD)
    - Find closure of F1:
      * closure(A) = {A}, closure(B) = {B}, closure(C) = {C, A, D} (as D is not in R1) = {C, A} and thus C -> A
      * closure(AB) = {A, B, C} and thus AB -> C, closure(BC) = {A, B, C} and thus BC -> A
      * closure(AC) = {A, C}
        + From F1, AB -> C is preserved in R
    - Find closure of F2:
      * closure(C) = {C, D} and thus C -> D, closure(D) = {D}
        + From F2, C -> D is preserved in R
    - However, D -> A is not preserved, and thus this decomposition is not dependency preserving
* Is 1NF? => If the domains of all attributes are atomic
  + Atomic => Domains of elements are considered to be indivisible units
    - Not Ex: Phone number being broken up into area code, actual phone number, country code

=========================

Transactions, Concurrency, and Recovery:

* Transaction => A sequence of SQL statements that are executed "together" as one unit
  + Reasons:
    - Allowing concurrent execution of multiple transactions => Concurrency control problem
    - Reduce average response time, increase latency
    - Increase processor and disk utilization
  + ACID Properties:
    - Atomicity => all or nothing
    - Consistency => If DB is in consistent state before transaction, it is also after
    - Isolation => Even if multiple transactions are executing concurrently, it is as if they are executing sequentially
      * Transactions are unaware that other transactions are executing at the same time
    - Durability => If a transaction commits, then all of its changes remain permanently (even after a crash)
* Commit and rollback:
  + Commit => transaction completing all of its reads and writes
  + Rollback => not completing all of the reads and writes, thus abort
    - Note: After commit => a transaction can't be rolled back
* Transactions in SQL:
  + Explicit control => begin transaction … update … end transaction
  + Implicit control => Each SQL update is treated as a separate transaction
  + Autocommit mode off => obeys explicit commit and rollback statements
* Shadow database scheme (shadow copying) => Use of a pointer pointing to a copy of the DB
  + When a new version exists, set the pointer to the new view of the DB, delete the old one
  + Ex: Text editors (not good for large DBs)
* Shadow paging => variation of shadow copying (not good with concurrency)
  + Reduces copying:
    - Page table contains pointers to all pages
    - Page table and all updated pages are copied to the new location
    - Any page not updated by a transaction is not copied, but page table just stores a pointer to the original page
    - When transaction commits, db\_ptr points to the new page table copy
* Types of schedules:
  + Serial schedule => a schedule of instructions from various transactions, where all the instructions from a given transaction appear together
  + Conflict Serializable schedule => execution order that produces the same results as a serial one
  + Recoverable schedule => if transaction Tj reads data item written by Ti previously, then Ti must commit before Tj commits
  + Cascadeless schedule => if Tj reads data item written by Ti previously, then Ti must commit before Tj reads it
    - Dirty reads => a read from a transaction not yet committed (could lead to cascading rollback)
* Conflict Serializability => if it is conflict equivalent to a serial schedule
  + Rules (for different transactions acting on the same data item):
    - Read(Q) and Read(Q) => Don't conflict
    - Read(Q) and Write(Q) => Conflict
    - Write(Q) and Write(Q) => Conflict
  + How to handle:
    - Can swap non-conflicting instructions
    - Precedence (conflict) Graph => Directed graph with a node for each transaction
      * Directed edges show conflicts on the same items
      * For Ti -> Tj:
        + Ti executes write(Q) before Tj executes read(Q)
        + Ti executes read(Q) before Tj executes write(Q)
        + Ti executes write(Q) before Tj executes write(Q)
      * Schedule is serializable if => no cycles exist in the precedence graph
      * Ex: S = [r1(x) r1(y) w2(x) w1(x) r2(y)]
        + T1 -> T2 (as T1 reads X before T2 writes to it)
        + T2 -> T1 (as T2 writes X before T1 writes it)

As there is a cycle, it is not conflict serializable

* Concurrency Control Protocols:
  + Locking protocol
  + Timestamp-based protocol
* Transaction Isolation Levels:
  + Serializable => (Default) Ensuring serializable execution
  + Repeatable read => Only committed records to be read
    - Repeatable reads of the same record return same data
    - Not always serializable
  + Read committed => Only committed data to be read
    - Repeatable reads of the same record can return different values
  + Read uncommitted => Allows uncommitted data to be read
* Lock-based Protocols:
  + Lock => Mechanism to control concurrent access to a data item X
    - Two Types:
      * Exclusive(X) => Data item can be read or written (lock-X)
        + If a transaction holds a lock-X for a data item, no other transactions can access that data item
      * Shared(S) => Data item can only be read (lock-S)
        + Many transactions can simultaneously hold lock-S for a data item
* Two-phased locking:
  + Ensures conflict serializable schedules
    - Can see this by looking at the lock points => point where a transaction acquired its final lock
  + NOTE: Transactions that are rolled back, restart up again at randomized times picked by the DB
  + NOTE: Transactions waiting for another lock will not skip to another instruction until it acquires that lock that it is waiting for
    - Ex: Read C, then Write A
      * If the transaction is waiting for a lock for C, it will not grab a lock for A, while it waits
  + Phase 1 => Growing => Transactions can obtain locks, but may not release locks
  + Phase 2 => Shrinking => Transactions can release locks, but they may not obtain locks
  + Read operation:
    - If Ti has lock on D
    - Then
      * read(D)
    - else begin
      * If necessary wait until no other transaction has lock-X on D
      * Grant Ti lock-S on D
      * Read(D)
    - end
  + Write operation:
    - If Ti has lock-X on D
    - Then
      * write(D)
    - Else begin
      * If necessary wait until no other transaction has any lock on D
      * Grant Ti lock-X on D or upgrade a lock-S to a lock-X
      * write(D)
    - end
  + All locks for a transaction are released after commit/abort
* Lock conversion:
  + In growing phase => can upgrade lock-S to lock-X
  + In shrinking phase => can downgrade lock-X to lock-S
* 2PL Pros and Cons:
  + Pros => Conflict serializable schedules
  + Cons => Some conflict serializable schedules can't be obtained
    - Deadlock => When multiple transactions can't advance as they are waiting for locks held by others
      * Must rollback one in the deadlock state
    - Starvation => When a transaction doesn't make any progress
      * Ex: Keep getting picked to be rolled back
      * Ex: Waiting for a lock-X, while others keep getting lock-S in front of you
    - Non-recoverable schedules
* Restrictive 2PL:
  + Strict => release lock-X only at the end of the transaction
    - Excludes dirty reads, produces recoverable, cascadeless schedules
    - Deadlock can still occur
  + Rigorous => Release all locks (S and X) at the end of the transaction
    - Deadlock can still occur
  + Conservative => Transaction requests all locks at the beginning of the transaction
    - If it doesn't get all, it waits
    - Prevents deadlock, dirty reads can still happen
* Handling Deadlocks:
  + Use timestamp-based protocols
  + Detection and recovery
  + Prevention
    - Conservative 2PL
    - Wait-die
    - Wound-wait
    - Timeouts
* Detection and recovery:
  + Wait-for graph => G = (V, E) where V = set of transactions and E = set of arcs between transactions
    - A -> B => If A is waiting for B to release a lock on same data item
    - Deadlock => If a cycle exists (still prone to starvation)
      * Select one transaction in the cycle to rollback (to break deadlock)
      * Timeout => If a transaction has no progress after some time, assume deadlock and rollback
    - Types of rollback:
      * Partial => Rollback only as far as necessary
      * Total => Abort the whole transaction
        + Can prevent starvation by taking age into account or counting the number of times a transaction was rolled back, when deciding which one to choose to rollback
  + Timestamp-based deadlock prevention:
    - Prevents starvation
    - Wait-die:
      * Older transaction may wait for younger one to release data item
      * Younger transaction never waits for older one, and is rolled back
      * Transactions may die several times before getting the data
    - Wound-wait:
      * Older transactions wound (force rollback) of younger transactions
      * Younger transactions may wait for older ones
      * May be fewer rollbacks
    - Timeout => After waiting for certain time, rollback
      * Avoids deadlock, prone to starvation, hard to pick good timeout interval
    - NOTE: Rolled back transactions keep their original timestamps
      * This is important to prevent starvation
* Timestamp-based Protocols:
  + Assures serializability and prevents deadlock
  + Doesn't guarantee cascade-free nor recoverability
  + Each transaction is issued a timestamp
  + Each data item Q has 2 timestamp values:
    - W-timestamp(Q) => Largest (youngest) TS of any transaction that executed write(Q) successfully
    - R-timestamp(Q) => Largest (youngest) TS of any transaction that executed read(Q) successfully
  + Two cases:
    - Ti issues read(Q):
      * If TS(Ti) <= W-TS(Q) => value of Q that Ti wanted was already overwritten
        + Read operation is rejected, Ti is rolled back
      * If TS(Ti) >= W-TS(Q) => the read operation is executed
        + R-TS(Q) = max(R-TS(Q), TS(Ti))
    - Ti issues write(Q):
      * If TS(Ti) < R-TS(Q) => then the value of Q that Ti is producing was needed before
        + Write operation is rejected, and Ti is rolled back
      * If TS(Ti) < W-TS(Q) = then Ti is attempting to write obsolete value
        + Write operation is rejected, Ti is rolled back
        + NOTE: Thomas' Write Rule => Obsolete writes are ignored, and Ti is not rolled back
      * Otherwise, write operation is executed
        + W-TS(Q) = TS(Ti)
* Recovery:
  + Execution of a checkpoint forces the buffers for all running transactions to be written out to safe storage
  + Failure Modes:
    - Transaction:
      * Logical errors => Ex: Integrity constraint violation
      * System errors => DB must terminate active transaction (e.g. deadlock)
    - System crash => Power/software failure => Ensure recovery
    - Disk failure: Ex: Head crash
* Recovery Algorithm:
  + During normal transaction processing to ensure that enough info exists to recover
  + Actions taken after failure to recover
* Data Access:
  + Physical blocks => Blocks residing on disk
  + Buffer blocks => Blocks residing temporarily in main memory
  + Block movements between disk and main memory initiated by:
    - input(B) => transfers physical block B to main memory
    - output(B) => transers buffer block B to disk
* Recovery and atomicity => Use log in stable storage describing all DB modifications
  + Log-based recovery => Log kept in stable storage
  + Series of log records, recording all update activities in the DB
  + <Ti start> => When Ti starts, it registers itself
  + Before Ti executes write(X) => a log record <Ti, X, V1, V2> is written
  + <Ti commit> => When Ti finishes its last statement
  + <Ti abort> => When a transaction is aborted
* Two approaches:
  + Immediate DB Modification => Output log record to stable storage of any updates as it occurs in a transaction
    - Once full log for a transaction is on disk, write out the data to disk
  + Deferred => When transaction reaches commit:
    - Writes log to buffer and outputs to disk
    - Outputs updated data blocks
    - Ex: If no commit, no page was written => No undoes
* Transaction commit:
  + Committed if log record is output to disk
  + Undo => Writes V1 to X of <Ti, X, V1, V2>
    - Do when <Ti start> but not commit/abort
    - When completes, <Ti abort> is written to disk
  + Redo => Writes V2 to X
    - Do when start and commit/abort exist
* Checkpoints:
  + To streamline recovery procedure
  + Process:
    - Output all log records currently residing in main memory to disk
    - Output buffer blocks to disk
    - Write log record <checkpoint Li> to disk
    - Scan backwards from the end of the log until the most recent checkpoint is received
      * Only transactions in L or after checkpoint need to be redone/undone
      * Transactions committed/aborted before checkpoint are done
    - Checklist consists of transactions still active at the time of the checkpoint
  + Recovery algorithm:
    - Redo Phase:
      * Find last checkpoint record, and set undo list to L
      * Scan forward from the checkpoint record
        + When <Ti, X, V1, V2> found redo it
        + When <Ti start> found, add Ti to the undo list
        + When <Ti commit> or <Ti abort> found, remove Ti from undo list
    - Undo Phase:
      * Scan log backwards
        + When <Ti, X, V1, V2> found, and Ti is in the undo list:

Do undo (write V1 to X)

Write log record <Ti, X, V1>

* + - * + When <Ti start> is found, and Ti is in the undo list:

Write <Ti abort>

Remove Ti from the undo list

* + - * + Stop when the undo list is empty (i.e. when <Ti start> is found for every transaction in the undo list)
* Commit Protocols:
  + Used to ensure atomicity across distributed sites
  + Must either be committed or aborted at all sites
  + Types:
    - Two-phase
    - Three-phase
* Two Phase Commit:
  + Phase 1: Prepare
    - After each cohort has locally completed its transaction, it sends "Done" to coordinator
    - When the coordinator receives "Done" message from all cohorts, it sends "Prepare" message to cohorts
    - Cohorts vote on whether they want to commit or abort:
      * If commit => They send "ready" message
      * If abort => They send "Not ready" message
        + Could happen if it has conflicting concurrent transactions or timeout
  + Phase 2: Commit/Abort
    - After coordinator has received "Ready" message from all cohorts:
      * Coordinator sends out "Global commit" message to the cohorts
      * Cohorts apply the transaction and send "Commit ACK" to the coordinator
      * When coordinator receives "Commit ACK" from all cohorts, it considers the transaction committed
    - After receiving one "Not ready" message:
      * Coordinator sends "Global abort" message
      * Cohorts abort the transaction and send "Abort ACK" to the coordinator
      * When coordinator receives "Abort ACK" from all, it considers the transaction aborted
* Three Phase Commit:
  + Phase 1: Prepare
  + Phase 2: Prepare to commit
    - Coordinator sends "Enter prepare state" to all cohorts
    - Cohorts vote "OK" in response
  + Phase 3: Commit/Abort
    - No need for ACKs
* Disadvantage of 2PC:
  + Blocking protocol => If coordinator fails permanently, some cohorts will never resolve their transactions
    - Ex: After cohort sends its "Ready/Not ready", it will block until commit/abort message
      * Could result in deadlock as some cohorts get the commit/abort and grab locks, while others never receive it and block => deadlock
* Pros of 3PC:
  + Achieves consensus
  + Non-blocking => automatically commit/abort after timeout
  + Can tolerate node failures, even if coordinator dies, carry on with the commit

=========================

ER Model:

* Three components: (1) Entity Set, (2) Relationship Set, (3) Attributes
* Types of attributes: simple, composite, single-valued, multi-valued, derived
* Roles => specifying how entities may interact in a relationship set (optional)
* Mapping cardinalities:
  + One-to-one
  + One-to-many => Ex: Loan associated with several customers
  + Many-to-one => Ex: Many students associated to at most 1 advisor
  + Many-to-many
    - Constraints (-> points to 1, and – points to many)
* Keys:
  + Super key => Set of 1 or more attributes whose values uniquely determine each entity
  + Candidate key => minimal super key
  + Primary key => The 1 selected candidate key by the DB designer
* Degree of the relationship set => The number of entity sets participating in a relationship set
  + For ternary => Constraints: At most 1 arrow
    - If more than one => Each pair of entities from (A, B) is associated with a unique C entity and each pair of entities from (A, C) is associated with a unique B entity
* Weak entity sets => Doesn't have a primary key (must participate in total participation to the relationship set)
  + Identifying entity set => Existence of a weak entity set depends on this
  + Discriminator (partial key) => Set of attributes distinguishing among all entities in a weak entity set
  + Primary key => Formed by combining the primary key of the strong entity set and the partial key of the weak entity set
* Specialization => Top-down design process, where we designate subgroupings within an entity set
  + Depicted by a triangle (ISA)
  + Attribute inheritance => Lower-level entities inherit all attribute and relationship participation of the higher-level entity set it is linked to
* Generalization => Bottom-up design process, combining a number of entity sets that share the same features into a higher-level entity set
* Constraints:
  + Disjoint => Entity can belong to at most 1 lower-level entity set
  + Overlapping => Entity can belong to more than 1 lower-level entity set
  + Completeness:
    - Total => Entity must belong to 1 of the lower level entity sets
    - Partial => Entity need not belong to 1 of the lower level entity sets
* Aggregation:
  + Allows relationships with relationships
  + Eliminates redundancy
  + Abstractions of relationship into a new entity
* Reduction of ER to tables:
  + Strong entity sets => have table with the same primary key
  + Weak entity set => Table that includes a column for primary key of the identifying strong set
  + Relationship set => Table with columns as keys of the participating entities (with the candidate key based off of the cardinality with more)
  + Specialization:
    - Form separate table for the higher and lower level entities (have access to multiple tables)
    - Form a table for each entity set with all local and inherited attributes (redundant)
  + Can combine tables with the same key => To get minimal tables
  + Aggregation => Table with the primary key of the aggregated relationship, primary key of the associated entity set, and any descriptive attributes