Reflexivity

$\models \quad X \rightarrow X$

Augmentation

$X \rightarrow Y \models XZ \rightarrow Y$

$X \rightarrow Y \models XZ \rightarrow YZ$

Transitivity

$(X \rightarrow Y) \wedge (Y \rightarrow Z) \models X \rightarrow Z$

Additivity

$(X \rightarrow Y) \wedge (X \rightarrow Z) \equiv X \rightarrow YZ$

Projectivity

$X \rightarrow YZ \models X \rightarrow Y$

$X \rightarrow YZ \models X \rightarrow Z$

Pseudotransivity

$(X \rightarrow Y) \wedge (YZ \rightarrow W) \models XZ \rightarrow W$

Entity Integrity Rule

Any attribute that is part of a primary key can not have a NULL value.

Fully Functional Dependent

A non key attribute is fully functional dependent on the primary key if it is not functionally dependent on any subset of the primary key.

Keys

- A set of attributes is a candidate key iff it functionally determines all other attributes in a relation
- Primary Key $\in$ Candidate Keys
- Alternative Keys $\subseteq$ Candidate Keys
- Candidate Keys = Alternative Keys $\cup$ {Primary Key}
- Primary Keys are underlined in a Relation Schema

X^{+}_{F}, is the closure of X under the set of functional dependencies F and is the set of attributes functionally determined by X under F.

F_C is the irreducible cover for F:

- $F \equiv F_C$
- Right-hand side of all relations only involve one attribute
- Left-hand side of all relations is irreducible

Normalisation

- Avoid update problems
- Avoid redundancy
- Simplify update operations
    - Single operation for insert
    - Single operation for delete

1NF

- No composite domains
- All values have to be atomic
- No duplicate tuples
- Tuples are unordered
- Problems
    - Redundancy
    - Update not always single operation

2NF

- 1NF
- All non key attributes must be fully functionally dependent on the primary key
- Problems
    - Update problems if transitive dependencies on the primary key

3NF

- 2NF
- No non key attribute can have a transitive functional dependence on the primary key

Integrity

- Integrity constraints are constraints on the entry of data, i.e. UNIQUE, BOUNDS, FOREIGN KEY
- Constraints may be either deferred (satisfied only some of the time) or immediate (has to be satisfied all the time)

Referential Integrity

- Databases must never contain unmatched foreign key values. Suppose Table B has a foreign key that points to Table A. Referential integrity would prevent you from adding a record to Table B that cannot be linked to Table A.
- Foreign keys can have NULL values

Transaction

- Collection of operations that performs a single logical function
- Takes a consistent database and returns a consistent database
- A transaction is active in a history if it has neither committed or aborted

Serialisable

- Concurrent running of a set transactions that is equivalent to some serial running of the transactions
- Consistency guaranteed
- The order of operations is often represented as a directed graph (dag)
- A concurrent execution of a set of transactions is represented by a history (sometimes know as a schedule)
  - Since some of these operations may be in parallel, a history is defined as a partial order
  - A history must specify the order of all conflicting operations
    - Operations are said to be in conflict if one or more is a write operation
  - A committed project of a history is the history of all the transactions that have committed

Equivalence

- **View Equivalence**
  - Ti reads x from Tj in history H if
    - $w_j[x] < r_i[x]$
    - $a_j \nless r_i[x]$
    - If there is some $w_k[x]$ such that $w_j[x] < w_k[x] < r_i[x]$, then $a_k < r_i[x]$
  - Final writes
    - Given a history H, we define wi[x] to be a final write for x in H if $a_i \notin H$ and
      - for all $w_j[x]$ in H ($j \neq i$) either $a_j \in H$ or $w_j[x] < w_i[x]$
  - Two histories will be view equivalent if they have the same reads-from relationships and the same final writes
- **Conflict Equivalence**
  - The serialisation graph for H is a directed graph whose nodes are the transaction those that are committed in H and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that one of T_i's operations proceeds and conflicts with one of T_j's operations in H
  - A cycle in a serialisation graph indicates a deadlock
- Conflict Serialisable $\rightarrow$ View Serialisable
- View Serialisable $\nrightarrow$ Conflict Serialisable

ACID

- Atomicity
  - A transaction can have just one of two outcomes COMMIT or ABORT
- Consistency
  - Each transaction acts on a database that is in a consistent state and leaves the database in a consistent form
- Isolation
  - An executing Transaction cannot reveal its contents to other concurrent transactions before (while the database may not be consistent) its commitment
- Durability
  - Once a transaction commits, its results are permanent. The results can only be undone by running a compensating transaction

Locks

- To guarantee serialisability we need to ensure that a transaction should not release a lock until it is certain that it will not request another lock, or alternatively a transaction should not request a lock after it has released any locks
- Transactions go through two phases
  - Growing phase
    - Locks acquired
  - Shrinking phase
    - Locks released
- Basic 2PL
  - Lock can be released when the growing phase has finished
  - Difficult to implement (how do you know when no more lock will be acquired)
  - Can lead to cascading aborts
- Strict 2PL
  - Locks can only be released at the end of the transaction.
  - Most schedulers implement Strict 2PL
- Granularity
  - Low → high locking overhead
  - High → reduce concurrency

Deadlock

- Locking can lead to deadlock
- Livelock: constant abortion of transaction if priority not high enough
- Detection by timeout or wait-for-graph
  - Timeouts
    - Too long - takes too long to detect
    - Too short - may undo transaction unnecessarily
- Broken by choosing victim, which should be
  - Most recently started
  - Holding least locks

    - Made fewest changes
    - Most amount of work to finish
    - Avoid cyclic restart i.e. livelock
    - In the most deadlocks
- Can be avoided
    - Transactions declare all required locks, transactions then only get scheduled if all locks are available.
        - They get restarted if extra locks are needed
        - Tendency to over declare locks to prevent restart, which reduces concurrency
    - Timestamps
        - The older the transaction the smaller the time stamp
        - Aborted transaction uses its original timestamp
        - T_j holding lock
        - T_i requesting lock
        - Wait-Die
            - Requester older → requester waits
            - Requester younger → requester aborts
            - If $ts(T_i) < ts(T_j)$
                - Then T_i waits
                - Else T_i aborts
            - Can result in cyclic restart
            - Once you have a lock you never have to abort
        - Wound-Wait
            - Requester older → holder aborts
            - Requester younger → requester waits
            - If $ts(T_i) < ts(T_j)$
                - Then T_j aborts
                - Else T_i waits
            - Maximum one abortion
    - Timestamp Ordering
        - Under this scheme the scheduler rejects operations that are too late
        - Aborted transaction gets new timestamp (it is less likely to be rejected)
        - This is different from the timestamp approach as the timestamp is increased
        - Each data item has two timestamp values associated with it
            - max-r-ts(x) denotes the largest timestamp of any transaction that has executed r[x] successfully
            - max-w-ts(x) denotes the largest timestamp of any transaction that has executed w[x] successfully

        - The timestamp order scheme produces serialisable executions equivalent to a serial execution in which transactions appear in timestamp order.
        - When the scheduler receives an operation it compares the transactions timestamp with the timestamp of the data item being operated on
            - Either the transaction's timestamp is less than the data item's relevant timestamp the operation is completed and the data item's timestamp is updated
            - Or the transaction's timestamp is more than the data item's relevant timestamp and the transaction is reject

Recovery

- Three types of failure
    - Transaction failure i.e. deadlock
    - System failure, where main database left in tact, but main memory and I/O buffers lost
    - Media failure, where main database becomes corrupted
- Stable database
    - Secondary storage is permanent and will never be lost
- Normally information is only transferred to the stable database when buffers are full, although this can be done manually
- Both the buffers and the stable storage is divided into pages
- The buffers area is known as the volatile database and does not survive system crashes
- Both undo and redo are idempotent operations as all you do is copy before or after images and therefore can safely be done multiple times
- Transactions that have to be undo can be spotted by the fact that they have a start and no finish
- Transactions that have to be redone can be spotted by the fact that they have a start and a finish
- In order to redo we store information about each transaction in the log
- Information that is held in the log is
    - <begin-transaction> entry
    - name of the transaction
    - name of the item being updated
    - old value of the data item ('before-image')
    - new value of the data item ('after-image')
    - forward/backward pointers to the next/previous log entry for this transaction followed by a <commit>/<abort> entry for the transaction
- The log information would initially be held in log buffers in the main memory before being transferred to stable storage to form the stable log
- The log-write-ahead-protocol is to ensure we always update the stable log before updating the associated changes
- Recovery Procedure

- Transactions that have a <begin-transaction> without a <commit> or <abort> have to be undone
- Transactions that have a <commit> have to be redone
- Alternatively Checkpointing can be used
  - Checkpointing
    - At certain defined intervals the system takes a checkpoint which consists of
      - Forcing the contents of the log buffers to stable storage
      - Writing a <checkpoint> record to the log in stable storage
      - Forcing the contents of the database to stable storage
      - Writing the address of the <checkpoint> record just written to stable storage into a restart file
    - The checkpoint record contains
      - A list of all transaction active at the time of the checkpoint
      - For each such transaction the address in the log of the most recent log record for that transaction
    - Transactions will not perform updates either on buffer blocks or on the log while the checkpointing procedure is in progress

- **Restart Procedure**
  - The recovery manager obtains the address of the most recent checkpoint record from the restart file and locates that record
  - Two lists are setup
    - Undo List
      - Initially contains all transactions listed in the checkpoint record, i.e. all active transactions when the checkpoint was performed
    - Redo List
      - Initially empty
    - The restart process then searches forward through the log, starting from the checkpoint record
      - If it finds a <begin-T_i> record it adds T_i to the undo list
      - If it finds a <commit-T_i> record it moves T_i from the undo list to the redo list
  - The process works backwards through the log undoing completely all transactions in the undo list; then it works forwards from the checkpoint redoing all transaction in the redo list
  - A new checkpoint record is written to avoid loosing the work done by the restart process
  - In order to reduce the amount of material in the log various log compressions can be used
    - Remove info on aborted transactions (these have been undone)

    - Remove before images for committed transaction (never need to undo)
    - Keep only last after image for committed transactions
    - Consider granularity of transactions

- **Shadow Paging**
  - This is an alternative to log based recovery procedures
  - During the lifetime of a transaction two pages tables are maintained
    - The current page table with the latest versions
    - The shadow page table with the pages prior to the start of transactions
  - When the transaction commits, the current page table is written to stable storage and becomes the new shadow page table for the next transaction
  - Under this scheme neither undo nor redo operations are required
  - Problems
    - Related data is fragmented over different disks
    - Garbage collection of old shadow pages
- **Media Failure**
  - To recover from media failure we will need to refer to an earlier consistent version of the database. We will therefore have to take regular back-ups of the database; these should be held at a remove site
  - In order to restore the database we will, in addition, require details of all changes made to the database between the time of the latest back-up and the time of the failure
  - These changes are available in the log. It is therefore essential that regular back-ups are also taken of the log and held at a remove site
  - We would also require some procedure for rescheduling those transactions that were in progress at the time of the failure.

Entity-Relationship Modelling

- Enables the semantics of data to be captured
- Three basic notations
  - Entity sets (rectangles)
    - Set of entities that share the same set attributes
    - Also called the extension of the entity type
  - Attributes (ellipses)
    - Each attribute has an associated domain
    - Types
      - Simple i.e. cannot be divided
      - Composite (double ellipses)
      - Single-valued i.e. one entry
      - Multi-valued i.e. one or more entries (double ellipses)
      - Derived (dashed ellipses)
      - Null i.e. not applicable or unknown
  - Relationship sets (diamonds)
    - Associations between entity sets
    - Can have descriptive attributes
    - Mapping cardinality
    - Mapping constraints
      - Dominant entity
        - Other entities depends on its existence
      - Subordinate entity (double line)
        - Existence depends on another entity
      - Weak entity (double diamond, double rectangle, double line)
        - Cannot be uniquely identified using own attributes
      - Strong entity
        - Can be uniquely identified using own attributes
      - Inheritance (ISA in inverted triangle)
        - Inherits all attributes
    - Weak Entity $\rightarrow$ Subordinate Entity
    - Subordinate Entity $\nrightarrow$ Weak Entity

Relational Algebra

- R1 UNION R2
  - R1 ∪ R2
- R1 MINUS R2
  - R1 – R2
- R1 INTERSECT R2
  - R1 ∩ R2
- R1 TIMES R2
  - R1 × R1
- DEFINE ALIAS S FOR R
- Select (Select Row)
  - R WHERE C
- Project (Select Column)
  - R[A1,A2]
- R1 JOIN R2
  - R1 ⋈ R2
- R1 DIVIDEDBY R1
  - R1 ÷ R2

Relational Calculus

- Variables are tuple variables: RANGE OF X is R1, R2, R3
  - Calculus
    - SX.S# WHERE SX.CITY = London
  - Algebra
    - (S WHERE CITY = London)[S.S#]

SQL

- SELECT attributes (projection)
- FROM relations (product)
- WHERE conditions (selection)
  - WHERE part may be omitted
  - SQL
    - SELECT A1, …, An
    - FROM R1, …, Rm
    - WHERE C

- Algebra
  - ((R1 × … × Rm) WHERE C)[A1, …, An]
- No operation for division
- Use of *
  - SELECT *
  - FROM P
  - WHERE CITY = ‘London’
- Use of IN and NOT IN
  - SELECT Sname
  - FROM S
  - WHERE City IN [‘London’, ’Paris’]

  - SELECT Sname
  - FROM S
  - WHERE City NOT IN [‘Rome’, ‘Athens’]
- Removing duplicate answers
  - SELECT DISTINCT Status
  - FROM S
- Sorting
  - SELECT Sname, City, Status
  - FROM S
  - ORDER BY Status (*default ascending*)

  - SELECT S#, P#, Qty
  - FROM SP
  - ORDER BY P# ASC, Qty DESC
- Built in aggregate functions
  - COUNT
  - SUM
  - AVG
  - MAX
  - MIN

- Built in aggregate examples
  - Counts rows
    - SELECT COUNT(*)
    - FROM S
  - Counts distinct rows
    - SELECT COUNT(DISTINCT S#)
    - FROM S
  - Sum of all quantities on order for each part number
    - SELECT P#, SUM(Qty)
    - FROM SP
    - GROUP BY P#
- Multiple Tables
  - Get supplier names for suppliers who supply part P2
    - SELECT S.Sname
    - FROM S, SP
    - WHERE S.S# = SP.S# AND
      SP.P# = ‘P2'
    - SELECT S.Sname
    - FROM S NATRUAL JOIN SP
    - WHERE SP.P# = ‘P2’
  - Get part number of parts that are either stored in London or supplied by S1 or both
    - SELECT P.P#
    - FROM P
    - WHERE P.City = ‘London’
    - UNION
    - SELECT SP.P#
    - FROM SP
    - WHERE SP.S# = ‘S1’

- o Renaming
  - SELECT SP1.S#
  - FROM SP SP1, SP SP2
  - WHERE SP2.S# = S1
  - AND SP1.P# = P1
  - AND SP2.P# = P1
  - AND SP1.Qty > SP2.Qty
- Subqueries
  - o SELECT S.Sname
  - o FROM S
  - o WHERE S.S# IN
    - (SELECT SP.S#
    - FROM SP
    - WHERE SP.P# = ‘P2’)
  - o Joins or products can be used instead of all subqueries but in some cases subqueries have a more ‘natural’ feel, for example those involving EXISTS and NOT EXISTS
- EXISTS and NOT EXISTS
  - o Get the supplier names of those suppliers which supply part P2
    - SELECT S.Sname
    - FROM S
    - WHERE EXISTS
      - o (SELECT *
      - o FROM SP
      - o WHERE S.S# = SP.S#
      - o AND SP.P# = ‘P2’)
  - o Get the suppliers names for suppliers who do not supply part P2
    - SELECT S.Sname
    - FROM S
    - WHERE NOT EXITS
      - o (SELECT *
      - o FROM SP
      - o WHERE S.S# = SP.S#

- AND SP.P# = ‘P2’)

- % can be used to indicate any string of letter i.e. ‘J%mie’ ≡ ‘%ie’ ≡ ‘J%e’
- - can be used to replace any single letter i.e. ‘J-mie’ ≡ ‘Ja--e' ≡ ‘-----‘
  - Use
    - SELECT Vintage, Quality
    - FROM Wine
    - WHERE Vineyard LIKE %nay

    - SELECT Vintage, Quality
    - FROM Wine
    - WHERE Vineyard LIKE ------

Relational Algebra	SQL
UNION	UNION
INTERSECT	INTERSECT
MINUS	MINUS
JOIN	NATURAL JOIN