Theory

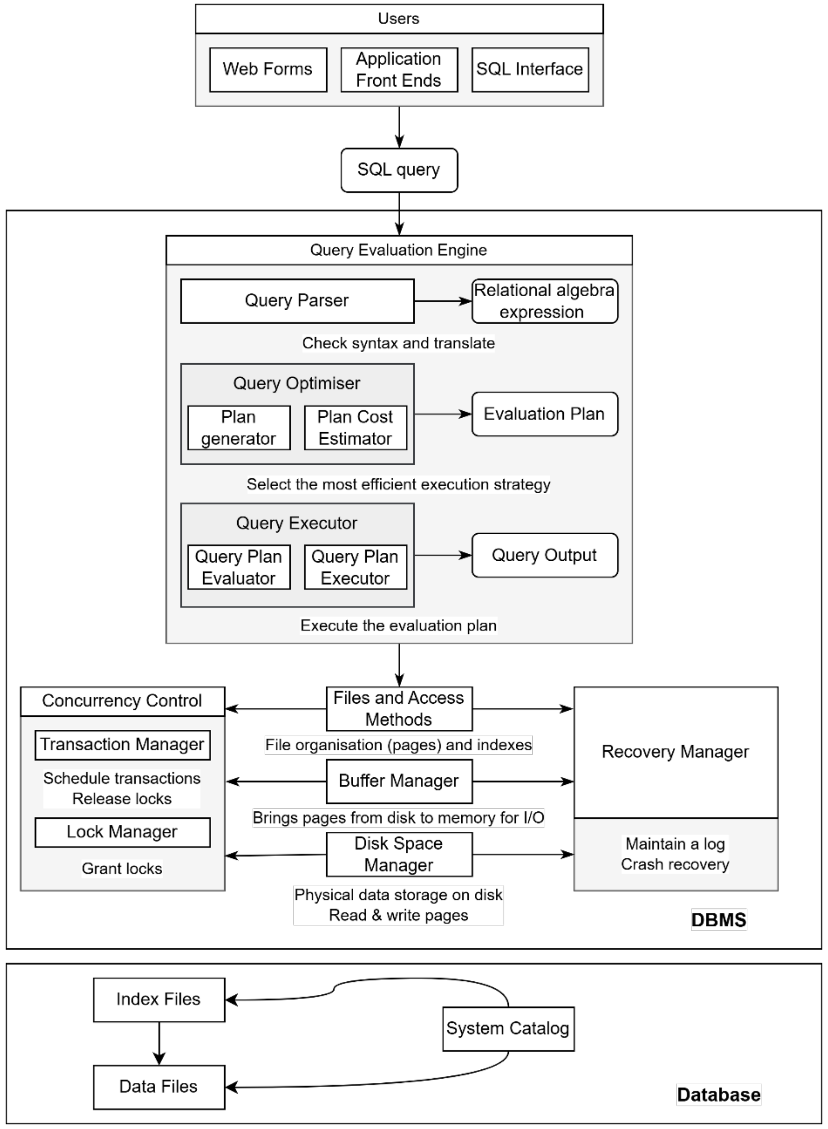
Lecture 1 Overview

Database

- Definition: an organised collection of data that models a Universe of Discourse (UoD)
- Characteristics:
  - Structured data: organised in a specific way to allow easy retrieval
  - Persistent data: data remains stored after system is turned off or restart
  - Efficient access: quick access of data
- Database design sequence: Requirement analysis – Conceptual design (ERD) – Logical design (RMD) – Schema refinement (normalisation) – Physical design (indexing and clustering) – App & security design (UML)

DBMS

- Definition: software to manage databases (bridge between the user and the data)
- Functions:
  - Data independence: abstract view to hide details of data representation and storage (applications not concerned about how data is structured and stored)
    - Logical data independence: changes in the logical (conceptual) structure of data does not impact the application/view
    - Physical data independence: changes in the physical layout does not impact the application
  - Declarative querying: query language to allow easy data access
    - Data Definition Language (DDL): define the structure of the data & relations
      - Create (CREATE TABLE), Alter (ALTER TABLE), Drop (DROP TABLE)
    - Data Manipulation Language (DML): manipulate the data in the database
      - Select (SELECT), insert (INSERT INTO), update (UPDATE), delete (DELETE)
    - Data Control Language (DCL): access authentication & security
      - Grant (GRANT), Revoke (REVOKE)
  - Transaction management: transaction must be ACID
  - Concurrency control: concurrent access to shared data to ensure data consistency (locking)
- Three-layer architecture:
  - Presentation: user interface that users interact with, e.g. web pages
    - User issues a query
  - Application: contains the business logic or rules of the application
    - Parser parses query – Query optimiser produces execution plan (a tree of relational operators) – Relational operators evaluate queries posed against data
    - File and access methods layer implements relational operators to keep track of pages – Buffer manager brings pages in from disk to main memory – Disk and space manager allocate, deallocate, read and write pages
    - Transaction manager produces locking protocol and schedules transaction execution – Lock manager keeps track of requests for locks and grants locks – Recovery manager maintains a log and restore the system after crash
  - Database: data storage and management



Lecture 2 Conceptual Modelling

Entity relationship model

Entity schema

- Entity schema: Entity set + Entity's attributes + Attribute's domain + Key attributes + Constraints
  - Entity: represents an individual object from the UoD
  - Entity set: a collection of entities that share common properties or characteristics
  - Entity's attributes: a descriptive property possessed by all members of an entity set
  - Attribute's domain: all possible values of an attribute (e.g. string < 20 char, int 1-10)
    - Single vs composite attributes
    - Single-valued vs multi-valued attributes
    - Derived attributes
  - Entity's key attribute: a minimal set of attributes whose values uniquely identify an entity in the set
    - Super key: set of attributes that uniquely identifies an entity in the set
    - Candidate key: minimal set of attributes that uniquely identifies an entity in the set
    - Primary key: the chosen candidate key (one entity can only have one PK)

Relationship schema

- Relationship schema: Relationship set + Relationship's attributes + Cardinality
  - Relationship: an association among two or more entities
  - Relationship set: a set of similar relationships
  - Relationship's attribute: additional properties of the relationship
  - Degree of relationships: unary (recursive), binary, ternary
  - Relationship role: entity can have an explicit role (useful for unary relationship)
  - Relationship's key (not mapped in ERD, in RM diagram only)
    - Super key: primary keys of participating entities
    - See below for cases where we don't need both primary keys

Weak entities

- Definition: does not have a self-contained primary key (cannot be uniquely identified by its attributes alone)
- Restrictions: total participation + key constraint (relate to the identifying strong entity via a N:1 relationship)
- Discriminator / partial key: the set of attributes that distinguishes among all the entities of a weak entity type related to the same owning entity (i.e. strong entity's PK + discriminator uniquely identifies an entity)

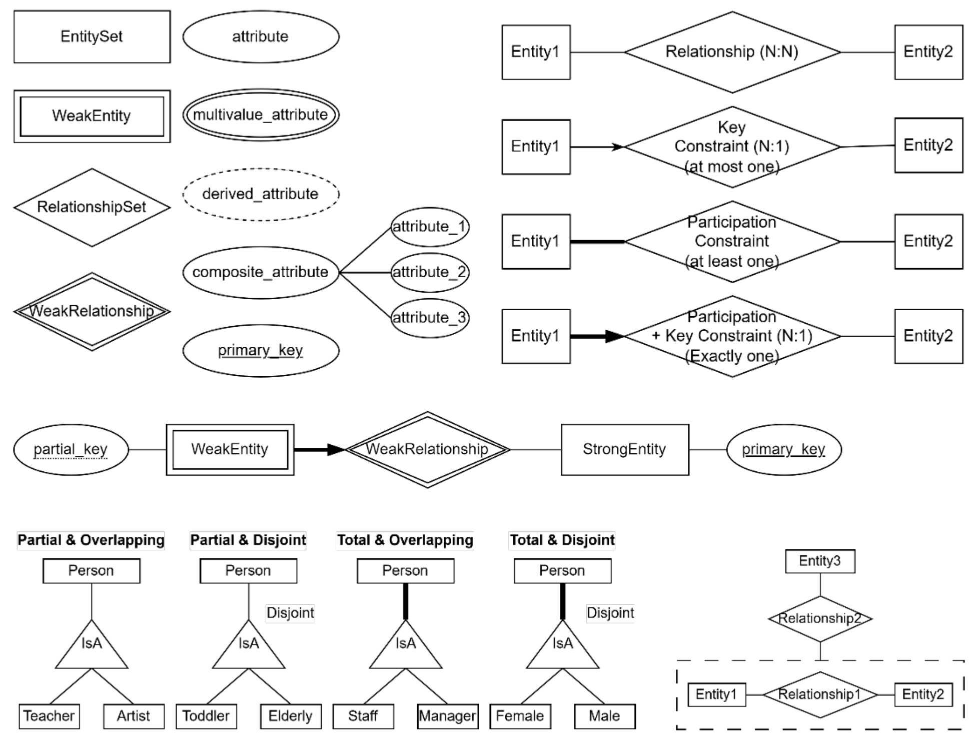
Class hierarchies – Generalisation / Specialisation / Inheritance

- Generalisation: entities sharing common features can be generalised into a superclass
- Two entity types E and F are in a IsA-relationship (F is a E) if:
  - The set of attributes of F is a superset of the set of attributes of E (F contains all attributes of E) and
  - The entity set F is a subset of the entity set of E (each f is an e)
- F is E; F is a subclass, E is a superclass; F inherit E's attributes and relationship
- Constraints: e.g. A and B are X
  - Overlapping constraints: whether subclasses are allowed to contain the same entity
    - Overlapping (default): an entity can belong to more than one subclass (an X can be both A and B)
    - Disjoint: an entity can belong to only one subclass (an X can be either A or B but not both)
  - Covering constraints: whether the entities in the subclasses collectively include all entities in the superclass
    - Partial (default): an entity need not belong to one of the subclasses (an X can be A, B or C)
  - Total: an entity must belong to one of the subclasses (an X must be A or B)

Aggregation

- A relationship participates in another relationship set (treat relationships as an entity)

ERD notations



Potential questions

- ERD analysis: given a description and ERD, identify errors
- ERD design: given a description, draw the ERD
- Common descriptions:
  - Cardinality:
    - At most one (can't have more than one): Key constraint = thin arrow
    - At least one (one or more, must, total participation): Participation constraint = thick line
    - Exactly one: Key + Participation constraint = thick arrow
  - Attribute:
    - Identified by, unique: primary key
    - Multiple values: multivalued attribute
    - Calculated as: derived attribute
  - Specialties:
    - Not interested in A if X doesn't exist, A depend on X: weak entity (X must be unique for each A)
    - F is E and has additional attributes: IsA
  - Generally: if something has attributes or has relationships with other entities, map as an entity

Lecture 3 Relational Data Model

Relation

- Relation schema: table structure (columns)
  - Relation's name (must be unique) + Name of each field/column/attribute (must be unique within the same relation) + Domain of each field (domain name and associated values)
- Relation instance: the table (rows)
  - A set of tuples/records/rows (all tuples must have the same structure and satisfy the domain constraints)
  - Terms: Degree/arity: number of fields in a relation, Cardinality: number of tuples in a relation

Relation's key

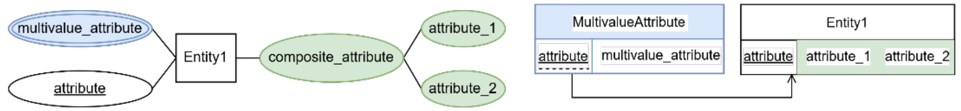
- Candidate key: the minimal set of attributes in a relation that uniquely identify each row of that relation
  - Uniqueness: no two distinct tuples in a legal instance can have the same values in all key fields
  - Minimality: any subset of the key cannot be a candidate key
  - Key can be simple or composite, a relation can have multiple keys, or no key (the whole tuple is a key)
- Super key: a set of attributes that contains a key (unique but not minimal, i.e. candidate key + other attribute)
- Primary key: the candidate key chosen by the database designer or DBA (only one per table)
- Foreign key: attribute referencing the primary key of the referenced relation (dependent to parent relation)

Integrity constraints

- Definition: a condition specified on a database schema and restricts the data that can be stored in an instance of the database
- Legal instance: a database instance that satisfies all integrity constraints specified on the database schema
- Types of constraints:
  - Key constraints: one or more columns in a table must uniquely identify a tuple
    - Entity integrity (primary key constraints): primary key has unique and non-null values
    - Referential integrity (foreign key constraints): foreign key values always refer to valid, existing rows in the related table (column name can be different, can be null) – no dangling reference
    - Unique constraints (candidate keys): if specified, cannot have duplicate values (but can be null)
  - Not null constraints: if specified, cannot have null values
  - Domain constraints: values must be within the domain of that column (i.e. the type of that field)
    - Semantic integrity constraints: enforces business rules or complex conditions on the data, such as limits on values or logical constraints (implemented using triggers or assertions)

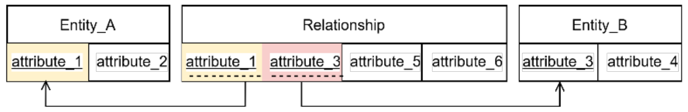
ERD to relations

Entities

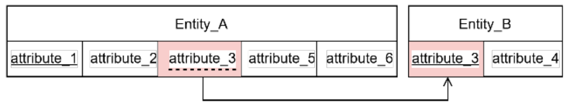


Relationships

- Many to Many: separate table for relationship

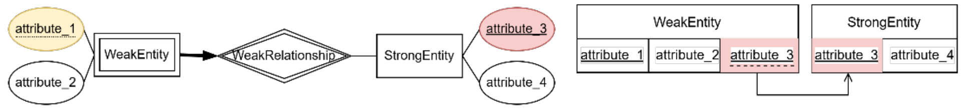


- Many to one: merge relationship with the "many" side



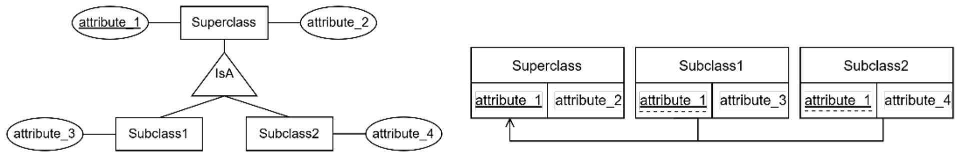
Weak entities

- Weak (N:1): merge relationship with A, FK NOT NULL ref B's PK, PK = A's discriminator + B's PK



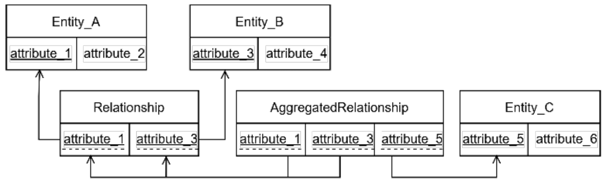
IsA relationship

- IsA: subclasses PK = FK ref superclass's PK



Aggregation

- Aggregation: PK = PKs of entity & relationship



Potential questions

- Convert ERD to RM:
  - Composite attribute: separate columns for each component attribute
  - Multivalued attribute: separate table, PK = entity's PK, FK = entity's PK
  - No constraint: separate table for relationship, PK = (A's PK, B's PK), FK = A's PK, FK = B's PK
  - Participation constraint: add NOT NULL, PK = (A's PK, B's PK), FK = A's PK, FK NOT NULL = B's PK
  - Key constraint on one side: combine relationship with A, A's FK = B's PK
  - Key constraint on both sides: combine relationship with A, A's FK UNIQUE = B' PK
  - Key & Participation + Key constraint: combine relationship with A, FK UNIQUE NOT NULL = B's PK
  - Weak relationship: combine relationship with A, PK = (A's discriminator, B's PK), FK NOT NULL = B's PK
  - IsA: one table for each superclass and subclasses, subclass's PK = superclass's PK, FK = superclass's PK
  - Aggregation: separate table, PK = relationship's PK + entity's PK, FK = relationship's PK + entity's PK
  - Note: if any of the PK's are composite, ensure FK references all attributes in the PK
- Write DDL based on RM:
  - One table for each entity set
  - Choose the appropriate data types for attributes
  - Identify PKs and FKs correctly
  - Implement integrity constraints

Lecture 4 Relational Algebra

- Operator types:
  - Unary vs Binary:
    - Unary: operators that focus on parts of one single relation
      - Selection, Projection, Rename
    - Binary: operators that combine tuples form two relations, and set operators
      - Cross-product, Join
      - Union, Intersection, Difference
  - Basic vs Derived:
    - Basic: Selection, Projection, Rename, Cross-product, Union, Difference
    - Derived: Join $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$, Intersection $R \cap S = R - (R - S)$
- Condition θ: output a Boolean combination of terms
  - Each term has the form attribute <op> constant, or attribute1 <op> attribute2
  - Op: $<, >, \leq, \geq, \neq, =$
  - Terms are connected by logical connectives: And $\wedge$, Or $\vee$
  - Reference to attributes: relation.name or name
  - E.g. $student.name = staff.name \wedge student.country = 'AUS'$
- Selection: $\sigma_{\theta}(R)$
  - Operation: Select rows meeting the selection criteria
  - Output: Filtered rows with same schema
  - SQL equivalent: `WHERE`
- Projection: $\pi_{column(s)}(R)$
  - Operation: Remove columns not in projection and remove duplicate rows
  - Output: Table with only projected columns
  - SQL equivalent: `SELECT DISTINCT`
- Rename: $\rho_{new\ table\ name[(col\ new\ names)]}\ (old\ table\ name)$
  - Operation: Rename a table and/or a column
  - Output: nothing, change of name
  - SQL equivalent:
    - Rename table only:
      ```
      SELECT * FROM <old table name> AS <new table name>
      ```
    - Rename table and attributes:
      ```
      SELECT <old col name> AS <new col name>, <another_col>
      FROM <old table name> AS <new table name>
      ```
- Cross-product: $R1 \times R2$
  - Operation: Combines two relations by pairing every row of the R1 with every row of R2
  - Output: Table with columns from both relations (duplicated columns are kept, with rename required)
  - SQL equivalent:
    ```
    SELECT * FROM <relation 1>, <relation 2>
    SELECT * FROM <relation 1> CROSS JOIN <relation 2>
    ```
  - No restriction on size or R1 or R2, A x B = B x A except column order is different
- Join: $R1 \bowtie_{\theta} R2 = \sigma_{\theta}\ (R1 \times R2)$
  - Operation: Join tables based on condition
    - Theta-join: any condition, output – all columns from both relations, duplicated columns kept
      ```
      R1 JOIN R2 ON <condition>
      ```
    - Equi-join: condition operator must be =, output – columns from both relations, duplication kept
      ```
      R1 JOIN R2 ON R.attr = S.attr
      ```
    - Natural join: equi-join using same-name columns, duplicated columns removed
      ```
      R1 NATURAL JOIN R2
      R1 JOIN R2 USING (<column list>)
      ```
- Set operators: take two input relations R1 and R2 of the same number of attributes and same attribute domains (union-compatible)
  - Operation: default is first eliminating duplicates from the input tables and then do the set operation
    - Union: $R1 \cup R2 = \{\, t \mid t \in\ R1 \vee t \in R2\}$
      - Returns all tuples in R1 plus R2
      ```
      SELECT * FROM R1 UNION SELECT * FROM R2
      ```
    - Intersection: $R1 \cap R2 = \{t \mid t \in R1 \wedge t \in R2\}$
      - Returns all tuples in R1 that are matched in R2 (both in R1 and R2)
      ```
      SELECT * FROM R1 INTERSECT SELECT * FROM R2
      ```
    - Difference: $R1 - R2 = \{t \mid t \in R1 \wedge t\ \notin R2\}$
      - Returns tuples in R1 that have no match in R2 (in R1 but not in R2)
      ```
      SELECT * FROM R1 EXCEPT SELECT * FROM R2
      ```
    - Retain duplicates by using UNION ALL, INTERSECT ALL, EXCEPT ALL
  - Output: a table with the same schema as R1 and R2, rows depend on operator
  - Duplicates in set tuple: a occurs r times in R, and s times in S (r, s > 0)
    - a occurs 0 times in R EXCEPT S
    - a occurs 1 time in R UNION S & R INTERSECT S
    - a occurs r + s times in R UNION ALL S
    - a occurs min(r, s) times in R INTERSECT ALL S
    - a occurs max(0, r-s) times in R EXCEPT ALL S

Potential questions

- Convert text to relational algebra
- Convert SQL to relational algebra
- Convert relational algebra to SQL
- Decomposing:
  - List xxx: projection,
  - Identify the selection condition: or – use $\vee$ in θ, and (same column) – select separately and use ∪
  - Not in, never: use – (All – the excluded rows)
  - Then determine which tables have those columns in projection list and selection criteria, join as needed
  - Note use natural join if the column has same name, otherwise need to specify join column

Lecture 8 Schema Refinement and Normalisation

Functional dependencies

- Definition: value of attribute X determines the value of attribute Y – X functionally determines Y ($X \to Y$)
- Armstrong's axioms: sound and complete way to derive FDs
  - Reflexivity – trivial (if $B \subseteq A, then\ A \to B$)
  - Augmentation (if $A \to B, then\ AC \to BC$)
  - Transitivity (if $A \to B\ and\ B \to C, then\ A \to C$)
- Closure of a set F (F^+): the set of all FDs deduced (logically implied) from set F using Armstrong's axioms

$$F^+ = \{A \to B\ |F\ | = A \to B\}$$

- Attribute closure (X^+): the set of all attributes determined by X, given a set F of FDs and a set X of attributes
  - Used to find keys of a relation $K^+ \to R$
- Normal forms: Determine if a relation needs to be decomposed into smaller relations (normalisation)

Decomposition

- Dependency preservation: all FDs maintained after decomposition (no FDs are lost)
- Lossless join: re-joining a decomposition of R should give back R without losing or creating incorrect tuples

Possible questions

- Convert text to FD: A rel at most one/the same B (A → B), A cannot rel > 1 B with same C & D (A, C, D → B)
- Find possible FDs / find illegal instances: for a given FD, the same LHS cannot have two different RHS
- Attribute closure: given a relation and functional dependencies, compute attribute closure for given attributes
- Finding candidate keys: Given a $R = \{A_1, A_2, A_3, A_4\}$, find the candidate key
  - Draw the table out, including all columns, and pointers from FD's LHS to RHS
  - Start with the attributes that never appear on the RHS: these must be part of all keys
  - Do attribute closure on these attributes using Armstrong's axioms

Implied attributes	Updated closure	Comment
RHS attributes	{attributes + RHS}	From LHS attributes via FD <no> using axiom

  - If the attribute closure on the attributes = R, then we have a super key
  - Then do attribute closure on the subsets of the super key, to prove that the key is minimal – CK
- Determining which normal form the table is in: check for violations (note a table can have many keys)
  - 1NF: no multivalued attributes (atomic)
  - 2NF: no part of key to non-key (partial dependency: LHS – key's proper subset, RHS – not part of key)
  - 3NF: no non-key to non-key (not transitive dependency if: LHS – super key, RHS – key's proper subset)
  - BCNF: no non-key to part of key (LHS must be super key)
  - 4NF: no multiple flattened multivalued attributes (A - PK, B & C - multivalued attribute flattened)
    - MVD: $A \twoheadrightarrow B, A\ \twoheadrightarrow C$ if relationship between A & B is independent from A & C's relationship
    - Check permutations: for a key in A, B has x possible values, C has y, num rows of key = x * y
- Decompose tables into the desired normal form:
  - Identify the violations; Split out LHS and RHS to separate tables, PK of the new table = LHS
  - Original table keeps LHS in, if still has violations, repeat the process
- Check for dependency preservation: preserved if $F' = F_1 \cup F_2 \cup \ldots \cup F_n = F$ or $F'^+ = F^+$
- Check for lossless join:
  - Check for common attribute between any pair of decomposed relations (intersection is key to a relation)
  - For a violation caused by $X \to Y, X \cap Y = \emptyset$, the decomposition of R into R-Y and XY is lossless
    $R_i \cap R_j \to R_j\ \ or\ R_j \cap R_i \to R_j$, i.e. the intersection is a key to at least one of the resulting relation

Lecture 9 Transaction Management

ACID properties

- Atomicity: A transaction is either performed entirely or not performed at all (committed or aborted)
- Consistency: The database remains consistent after transaction (all integrity constraints are satisfied and intended effect of transaction is achieved)
- Isolation: Each transaction should execute in isolation from other concurrent transactions (transactions can run concurrently without violating the correctness and consistency of the database caused by concurrent access (i.e. interleaving of operations))
- Durability: a committed transaction's changes are permanent and must never be lost, even in the event of system failures (log)

Concurrent access issues

- Temporary update problem: Occurs when one transaction updates an item but fails. Another transaction reads this temporary value, which is later changed back
- Incorrect summary problem: Occurs when one transaction is computing an aggregate (e.g. sum), but other transactions are modifying the data simultaneously
- Lost update problem: two transactions update the same data, and one's update is lost
  - The transaction that writes last is the one that's kept (other changes are lost)
- Phantom read: one transaction re-executes a query and sees additional rows inserted by other transactions

Isolation levels

- Read uncommitted: Transactions can read uncommitted changes made by other transactions
- Read committed: Transactions cannot read uncommitted data, solves WR
- Repeatable read: If a transaction reads a value, it read the same value during the transaction, solves RW
- Serializable: Transactions executed concurrently has the same effect as in a serial order, solves WW

Isolation level	WR (dirty read / temporary update problem)	RW (non-repeatable read, incorrect summary problem)	WW (lost update), phantom read
Read uncommitted	No	No	No
Read committed	Yes	No	No
Repeatable read	Yes	Yes	No
Serializable	Yes	Yes	Yes

Serializability

- Serial vs interleave schedule:

Concurrent transactions			Serial schedule (one after another)			Interleaved schedule	
T1	T2		T1	T2		T1	T2
R(A)	R(A)		R(A)			R(A)	
W(A)	W(A)		W(A)				R(A)
				R(A)		W(A)	
				W(A)			W(A)

- Serializable schedule: maybe interleaved but is equivalent to some serial schedule
  - Equivalent: S1 is equivalent to S2 if for any database state, the effect of executing S1 is identical to the effect of executing S2 (check if the interleaved schedule has the result as serial execution)

Conflict serializable

- Conflict: two operations a_i, a_j conflict if (e.g. R1(A) and W2(A), or W1(A) and W2(A))
  - They access the same data item
  - They come from different transactions
  - At least one of the operations is a write

- Conflict serializable: if a schedule is conflict equivalent to a serial schedule (If a schedule is conflict serializable, then it must be serializable, but not vice versa)
  - They involve the same set of operations of the same transactions
  - They order every pair of conflicting operations the same way

Lock-based concurrency control

- Types of lock:
  - Shared lock / Read lock (S): Multiple transactions can acquire a shared lock to read the same data item
    - Prevents read-write conflicts
  - Exclusive Lock / Write lock (X): Only one transaction can hold an exclusive lock on a data item at a time. It prevents both read and write access by other transactions while the lock is held
    - Prevents write-write conflicts
- Locking protocol: lock an item before using so another transaction requesting the same item needs to wait
- Two-phase locking (2PL): insist that all locks be granted before any are released
  - Process: Obtain locks, Perform computations, Release locks, then commit
  - Ensures conflict-serializability (assuming no deadlocks and failures)
  - Phases of 2PL: i.e. cannot have lock after unlocking in a transaction
    - Growing phase: the number of locks may only increase but not decrease (request more locks)
    - Shrinking phase: the number of locks can only decrease until no more lock exists (no acquiring after releasing)
- Strict 2PL: all locks are released after commit (not before)
  - Allows only serializable schedules (no interleaves)
- Deadlocks: If there exists a cycle where transactions waiting for each other to release locks
  - T1 holds a lock on an item A and is requesting a (conflicting) lock on an item B and
  - T2 holds a lock on item B and is requesting a (conflicting) lock on item A. (A and B can be the same)
  - Prevention: static 2PL (request all locks, and get all or none)
  - Detection: one transaction must abort to release lock

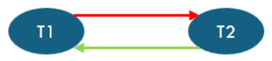
Possible questions

- Identify possible concurrent access issues and which isolation level solves the issue:
  - Temporary update (T2 read a temporary update by T1) – Read committed
  - Incorrect summary (T1 reads and calculates an aggregation, but T2 updates it later) – Repeatable read
  - Lost update (T1 and T2 both update but only T2's is kept) – Serializable
- Turn a SQL query into R W schedule:
  - SELECT FROM WHERE: read
  - UPDATE: write (note if value set to be based on original value = R then W)
- Evaluate the result of a schedule / SQL query:
  - Determine the isolation level: initial value of A = x
    - READ UNCOMMITTED: T1 UPDATE A = y, T2 SELECT A COMMIT (T2 reads A = x)
    - READ COMMITTED: T1 SELECT A, T2 UPDATE A = y COMMIT, T1 SELECT A COMMIT (T1 first read A = x, second read A = y in the same transaction)
    - REPEATABLE: T1 SELECT SUM(col B), T2 INSERT A into col B COMMIT, T1 SELECT SUM(col B) COMMIT (1st & 2nd sum not the same; same issue if SFW & where condition return different rows)
    - SERIALIZABLE: same operation as above but T2 may need to wait or rollback
  - Determine when the transactions are committed
    - Identify if the query has errors or explicit rollbacks, if so, the database should be unchanged
- Checking conflict serialisability:
  - Option 1: use non conflicting swapping and check if the result is serial
    - Example: swap 3&4 (both read), swap 5&6 (different item), then swap 4&5 (different item) → Result if serial (T1, T2), conflict serializable

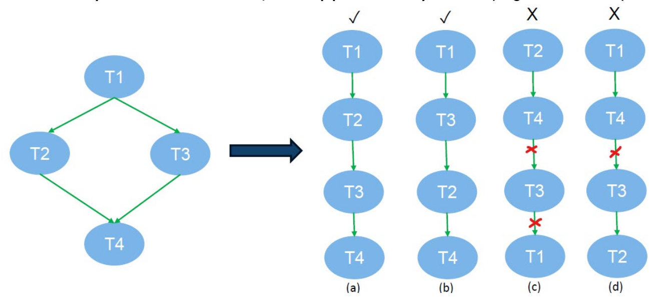
	Original		Step 1		Step 2		Step 3	
	T1	T2	T1	T2	T1	T2	T1	T2
1	R(A)		R(A)		R(A)		R(A)	
2	W(A)		W(A)		W(A)		W(A)	
3		R(A)	R(B)		R(B)		R(B)	
4	R(B)			R(A)		R(A)	W(B)	
5		W(A)		W(A)	W(B)			R(A)
6	W(B)		W(B)			W(A)		W(A)
7		R(B)		R(B)		R(B)		R(B)
8		W(B)		W(B)		W(B)		W(B)

  - Option 2: use Precedence Graph (Serialization Graph Testing) and check for cycle
    - Find all conflict pairs: same item, different transactions, at least one operation is write
    - For each pair, point from first to second (for each object, list all conflicts in order of appearance)
    - Example: T1 and T2 have 3 conflict pairs [R1(A),W2(A)], [R2(A),W1(A)], [W2(A),W1(A)] → not conflict serializable

T1	R1(A)			W1(A)
T2		R2(A)	W2(A)	



    - If SGT graph is acyclic, then can obtain serial schedules from a topological sorting (linear order)
      - If two transactions are connected (direct or indirect), must follow the order (e.g. T2/T3 must go after T1, T4 must go after T3/T2/T1)
      - If they are not connected, can appear in any order (e.g. T2 and T3)



- Checking serializability (view serializable): effect is the same is executing the schedule in either serial order
  - Check blind write (write without read), if no blind write, not view-serializable
  - Draw dependence graph: if no cycle, view serializable
    - Initial read: Check first read, this transaction must read from database – no W before first read
    - Final write: Check last write, this transaction must be after all transactions (last transaction)
    - Updated read: check the closest W before each R, this order should be preserved
- Determine if a schedule is 2PL: check no lock after unlocking in the same transaction
- Determine if a schedule is strict 2PL: check no lock after unlocking and no unlock before commit
- Determine if there's a deadlock: convert R and W to S and X locks and check for waiting cycle

Lecture 11 Storage Indexing

Physical data storage

- ❖ Data is stored on external (physical) storage devices and fetched into main memory when needed for processing (e.g. disks)
- ❖ Structure of disk: sector – block – track – cylinder
- ❖ Block/page: a contiguous sequence of bytes (the unit of information read from or written to disk)
  - ➢ Stores multiple records or tuples (divided by software once in buffer)
  - ➢ Size: 4KB or 8KB (usually more than one sector), set by operating system
    - ▪ 1 byte = 8 bits, 1KB = 1024 bytes, 1MB = 1024*1024 bytes
  - ➢ Blocking factor: $b = \frac{block\ size}{record\ size}$, Record spanning occurs when b < 1
- ❖ I/O: reading or writing in a disk: if an item on the block is needed, transfer the whole block to main memory
  - ➢ Input: read from disk to main memory for processing
  - ➢ Output: written from memory to disk for persistent storage
  - ➢ Access time (cost) for disk I/O: seek time (find the track) + rotational delay (find the start of block) + transfer time (read the whole block)

Buffer management

- ❖ Buffer management: access strategies to guess which blocks the application will ask for
- ❖ Buffer replacement strategy: which block gets evicted from buffer when there's not enough space
  - ➢ LRU: victim is the block used the longest time ago
  - ➢ MRU: victim is the block most recently used

Redundancy

- ❖ Disk failure: mean time to failure (MTTF) describes the average operating time until the disk fails (e.g. MTTF = 1m hour, 1m disks, disk failure is expected to occur once per hour; 1000 disks, disk failure every 1000 hours)
- ❖ RAID: uses data stripping and redundancy to increase performance and reliability
  - ➢ Level 0: non-redundant data stripping (data segmented and distributed into multiple disks in parallel)
  - ➢ Level 1: mirrored disks (two copies of data on two different disks)
  - ➢ Level 2: memory-style error-correcting codes (strip at bit level and have log n parity check disks)
  - ➢ Level 3: bit-interleaved parity (strip at bit level and single parity check disk)
  - ➢ Level 4: block-interleaved parity (strip at block level and single parity check disk)
  - ➢ Level 5: block-interleaved distributed parity (strip block level and distribute parity check over all disks)
  - ➢ Level 6: P+Q redundancy (block level with two copies of parity check distributed over all disks)

Physical data organisation

- ❖ File: multiple pages, Page = block – multiple records, Records – row

File organisation – Access paths

- ❖ File organisation / access paths: a method of arranging the records stored on disk
  - ➢ Heap Files (linear scan): unordered – useful for file scan or insertion is more important than retrieval
    - ▪ Worst case: B, Average case: B/2
  - ➢ Sorted Files (binary search): sorted by search key – fast data access, but expensive to update
    - ▪ Worst case: log B
  - ➢ Indexes (index scan): uses B+ Tree or Hashing to create index for faster lookup and updates
    - ▪ Worst case: height of tree (level of index pages + 1)

B+ Tree

- ❖ Building a B+Tree index: Sort all records in all pages, Create one index entry for each page, Fit index entries into pages (bottom to top level, until reaching the root)

Potential questions

- ❖ RAID levels: Level 0 – lowest cost, Level 0+1 – best for small writes, Level 3 – lowest overhead, Level 5 – best performance for read and large writes, Level 6 – highest reliability
- ❖ Buffer replacement strategy:
  - ➢ The buffer size = 3 blocks, 4 records in the relation (c1, c2, c3, c4), one record fits into one block
  - ➢ Operation: joining the relation with another relation using nested for loops
  - ➢ MRU is more efficient than LRU for table scan: 2 page faults vs 4 for the shown operations
    - ▪ For LRU, the record we need next is always selected as the victim in the previous round

	Next	LRU			MRU		
		Least		Most	Least		Most
0	C4	C1	C2	C3	C1	C2	C3
1	C1	C2	C3	C4	C1	C2	C4
2	C2	C3	C4	C1	C1	C2	C4
3	C3	C4	C1	C2	C1	C2	C4
4	C4	C1	C2	C3	C1	C2	C3

- ❖ Data organisation: space needed for a table
  - ➢ Table has 2,000,000 records (rows), each record is 200 bytes, including a primary key tuple-key of 4 bytes, an attribute 1 of 4 bytes, along with other attributes
  - ➢ Each page is 4k bytes, of which 250 bytes are reserved for header and array of record pointers
  - ➢ Calculations:
    - ▪ Calculate size of each record: 200 bytes
    - ▪ Records per page (assuming no record spanning): assume 100% occupancy
      - • Space available for data storage = 4 * 1024 – 250 = 3846 bytes
      - • Number of records per page: $\left\lfloor\frac{3846}{200}\right\rfloor = \lfloor 19.23\rfloor = 19$ records per page
      - • Remaining space of each page: $3846 - 19 * 200 = 46$ bytes
    - ▪ Number of pages for the table
      - • Number of pages: $\left\lceil\frac{2{,}000{,}000}{19}\right\rceil = \lceil 105{,}263.16\rceil = 105{,}264$
      - • Space: $105{,}264\ pages * 4KB = 421{,}056\ KB \approx 411\ MB$
      - • Number of records on last page: $2{,}000{,}000 - 105{,}263 * 19 = 3$
    - ▪ Overhead:
      - • Total size of required pages: $105{,}264 * 4 * 1024 = 431{,}161{,}34$ bytes
      - • Actual table size: $2{,}000{,}000 * 200 = 400{,}000{,}000$ bytes
      - • Overhead = $\frac{(431{,}161{,}344 - 400{,}000{,}000)}{400{,}000{,}000} = 7.79\%$
    - ▪ Pages with a given fill factor: 75% on average
      - • Updated record per page: $\left\lfloor\frac{3846}{200} * 0.75\right\rfloor = \lfloor 14.42\rfloor = 14$ records per page
      - • Total pages: $\left\lceil\frac{2{,}000{,}000}{14}\right\rceil = \lceil 142{,}857.14\rceil = 142{,}858$ pages
      - • Total size: $142{,}858 * 4 * 1024 = 585{,}146{,}368$ bytes
      - • Overhead: $\frac{585{,}146{,}368 - 400{,}000{,}000}{400{,}000{,}000} = 46.28\%$
- ❖ B+ Tree construction: file has 140,351 pages
  - ➢ Create a logically sorted file: All records within a page are physically sorted on the search key
  - ➢ Create one index entry for each page: <search key, rowid>
    - ▪ Size of entry: size of search key 4 bytes + size of rowid 4 bytes = 8 bytes
  - ➢ Fit index entries into pages:
    - ▪ Number of index entries in each page: $\left\lfloor\frac{3846}{8}\right\rfloor = 480$ entries per page

        - Number of entries with fill factor (75%): $480 * 0.75 = 360$ entries
        - Number of index pages: $\frac{number\ of\ pages}{360} = \left\lceil\frac{140{,}351}{360}\right\rceil = 390$ pages
    - Build B+ Tree index:
        - Pages required for 2nd level: $\frac{number\ of\ entries\ in\ level\ down}{360} = \lceil\frac{390}{360}\rceil = 2$
        - Pages required for 3rd level: $\left\lceil\frac{2}{360}\right\rceil = 1$ → reached 1, this is the root level
        - Total number of index pages: $390 + 2 + 1 = 393$
        - Increase: $\frac{393}{140{,}351} = 0.2\ \%$ increase in the number of pages required
    - Final tree:
        - Root index: 1 page storing 2 index pointers
        - 1st level index: 2 pages storing 390 index pointers
        - 2nd level index: 390 pages storing 140,351 pointers to pages
        - 3rd level data: 140,351 pages of data
- I/Os needed for different access path: if total pages in a file = 140,351 and page size = 4KB
    - Linear scan on heap:
        - Equality search:
            - If search key is unique and exists: $\left\lceil\frac{140{,}351}{2}\right\rceil = 70{,}176$ I/Os
            - If non-unique or no match: 140,351 I/Os
        - Range search: 140,351 I/Os
    - Binary search on sorted files:
        - Height $\log 140{,}351$ + additional pages containing retrieved records
    - Index scan on B+ Tree:
        - Equality search: height of B+ Tree (index pages) + additional pages containing retrieved records
        - Range search if first item is search key: height of B+ Tree + records matched with subsequent attributes + additional pages containing retrieved records
            - Table R(A,B,C) has 100,000 records, A has 100 different values, B has 1000, C has 50,000
            - B+ Tree index on composite key (A,B), 2 levels of index + record pages
            - Fits 160 records per page, file has 625 pages, reading a page takes 150 msec
            - Calculate time to run: SELECT C FROM R WHERE A = "x" AND (B BETWEEN 'y' AND 'z'); selectivity of range condition on B is 10%
            - Time to locate the data: go through one page at each index page level: 2 pages
            - Page containing the record:
                - 100,000 records & 100 different values of A: assume $\frac{100{,}000}{100} = 1000$ records = 'x'
                - Of the 1000 pages, 10% matches the range search on B: $1000 * 10\% = 100$ records
                - The 100 records fit onto: $\left\lceil\frac{100}{160}\right\rceil = 1$ page
            - Total page to read: 2 + 1 = 3 pages
            - Total time = 3 pages * 150 msec = 450 msec

Lecture 12 Query Processing

Logical query plan – Heuristic rules

- Algebraic transformation rules:
    - Selections:
        - Cascade of Selections: $\sigma_{\theta 1 \wedge \theta 2}(R) = \sigma_{\theta 1}\left(\sigma_{\theta 2}(R)\right)$
        - Commutative of selections: $\sigma_{\theta 1}(\sigma_{\theta 2}(R)) = \sigma_{\theta 2}\left(\sigma_{\theta 1}(R)\right)$
    - Projections:
        - Cascade of projections: if attributes B1, …Bn are a subset of A1,…An, then

$$\pi_{B1,\dots Bn}\left(\pi_{A1,\dots An}(R)\right) = \pi_{B1,\dots Bn}(R)$$

    - Joins:
        - Commutative Rule for Joins: $R1 \bowtie R2 = R2 \bowtie R1$
        - Associative Rule for Joins: $(R1 \bowtie R2) \bowtie R3 = R1 \bowtie (R2 \bowtie R3)$
    - Commute selection with projection: if selection involves only attributes retained by the projection (i.e. all attributes in selection condition are in projection attributes)

$$\pi_{A1,\dots An}(\sigma_\theta(R)) = \sigma_\theta\left(\pi_{A1,\dots An}(R)\right)$$

    - Commute selection with join: if all the attributes in the selection condition θ involve only the attributes of one of the relations being joined (i.e. selection attribute is only in R1)

$$\sigma_\theta(R1 \bowtie R2) = \left(\sigma_\theta(R1)\right) \bowtie R2$$

        - If c1 involves both R and S, c2 involves only attributes of R, and c3 involves only attributes of S

$$\sigma_{c1 \wedge c2 \wedge c3}(R \bowtie S) = \sigma_{c1}\left(\sigma_{c2}(R) \bowtie \sigma_{c3}(S)\right)$$

    - Commute projection with join: if the projection includes all attributes needed for the join (all attributes in join condition are in projection)

$$\pi_{a1,a2}(R \bowtie_c S) = \pi_{a1}(R) \bowtie_c \pi_{a2}(S)$$

        - If join condition c needs a1 and a2 from R and a3 from S, final projection only needs a1 and a3

$$\pi_{a1,a3}(R \bowtie_c S) = \pi_{a1,a3}\left(\pi_{a1}(R) \bowtie_c \pi_{a2}(S)\right)$$

- Strategy: pick a sequence of operations that can minimise the size of intermediate results
    - Selections: select before joining – reduce tuples
        - If condition span across multiple tables, do selection within each table, then join the filtered tables
    - Projections: do projection as soon as possible – reduce attributes
        - Consider which attributes we must need for join and result attributes

Physical query plan – Cost estimate

- Select the algorithm that minimise I/O costs

Optimising joins

- Nested-loop join: For each tuple in outer table, scan all tuples in inner table (page R, tuple r, page S, tuple s)
- Block-nested loop join: for every page of in, pair with every page of out (page R, page S, tuple r, tuple s)
- Index-nested loop join: For each tuple r in outer table R, use the index on inner table S to look up tuples in S that satisfy the join condition with tuple r (page R, tuple r, tuple s = index(s))
    - Must be equi-join or natural join, and there's index for the inner table's join attribute

Optimising sorting

- External merge-sort:
    - Create sorted runs (sorted subset of records): ⌈N/B⌉ runs (load one run at a time and sort in buffer)
    - Merge each contiguous groups of B-1 runs into 1 run: (B-1)-way merge (1 output + B-1 input pages)
    - After each merge pass, the number of runs reduced by a factor of B-1

Query execution

- Materialisation (set-at-a-time): Evaluate one operation at a time (high cost)
  - Store the result of each evaluation in a temporary relation for subsequent use (i.e. output of one operator written to disk and next operator read from disk)
    - Compute and store new table for σ
    - Compute and store the materialised join with other table
    - Then compute projection on the joined table
- Pipelining (tuple-at-a-time / on-the-fly-processing): Evaluate several operations in a pipeline (no extra space)
  - Output of one operator directly input to next operator)
    - Find next tuple matching σ
      - Join matching tuple with all tuples from the other table
      - Project sname from the joined tuples
      - Repeat for all join results
    - Repeat for next selection result

Possible questions

- Logical query plan: optimise relational algebra
- Calculate the I/O for different join types: Two tables R is the outer table, S is the inner table
  - R has |R| number of tuples, stored in b_R pages, S has |S| number of tuples, stored in b_S pages
  - Nested-loop join: $b_R + |R| * b_S$ (outer table: each page read once, inner table: each page read once for every tuple of R)
  - Block-nested loop join: $b_R + b_R * b_S$ (outer table – each page is read once, inner table – each page is read once for every page of R)
  - Index-nested loop join: $b_R + |R| * c_1$ (index available on S, traverse the index tree and read all matching S tuple for each tuple of R)
    - C1 = index level in S's B+ Tree + number of pages with records matching the condition
      - Num pages = (for each value of join condition in S, number of tuples satisfying the other conditions) / (record / page)
    - Note if we don't have the index on S, this takes the same amount of time as nested-loop join
- Perform an external merge-sort
- Calculate I/O for external merge-sort: N=108 pages, B = 5
  - Number of passes: $\left\lceil \log_{B-1}\frac{N}{B} \right\rceil + 1 = \left\lceil \log_4 \frac{108}{5} \right\rceil + 1 = \lceil 2.21 \rceil + 1 = 4$
  - Pass 0: create sorted run
    - Size of sorted runs: 5 = size of buffer
    - Number of sorted runs: $\left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{108}{5} \right\rceil = \lceil 21.6 \rceil = 22$ (last run is 3 pages)
  - Pass 1: B-1 four-way merge to produce sorted runs of bigger size
    - Size of sorted run: 4 input slots * 5 pages in each sorted run = 20 pages
    - Number of sorted runs: $\left\lceil \frac{N}{B-1} \right\rceil = \left\lceil \frac{22}{4} \right\rceil = \lceil 5.5 \rceil = 6$ (last run is 8 pages)
  - Pass 2: B-1 four-way merge to produce sorted runs of bigger size
    - Size of sorted run: 4 input slots * 20 pages in each sorted run = 80 pages
    - Number of sorted runs: $\left\lceil \frac{N}{B-1} \right\rceil = \left\lceil \frac{6}{4} \right\rceil = \lceil 1.5 \rceil = 2$ (last run is 28 pages)
  - Pass 3: last merge to produce the sorted file
    - Number of sorted runs: $\left\lceil \frac{N}{B-1} \right\rceil = \left\lceil \frac{2}{4} \right\rceil = \lceil 0.5 \rceil = 1$
  - Total costs: 2 (read & write) * number of passes * N (number of pages read and write in each pass) = 2 * 4 * 108 = 864 I/Os

Possible SQL questions

- Find the name of the manager whose bank has any accounts with a balance greater than $10,000

```
SELECT DISTINCT b.manager_name # use distinct as a bank may have multiple acccounts
FROM Bank b JOIN Account a ON b.bank_id = a.bank_id
WHERE a.balance > 10000;
```

- Find names of students who enrolled in both COMP9001 and COMP9120:

```
SELECT name FROM Student WHERE sid IN
(SELECT sid FROM Enrolled WHERE uos = 'COMP9001'
INTERSECT sid FROM Enrolled WHERE uos = 'COMP9120);
```

- Find students who have taken COMP9001 but not COMP9120

```
SELECT sid FROM Enrolled WHERE uosCode = 'COMP9001' AND grade IS NOT NULL
EXCEPT
SELECT sid FROM Enrolled WHERE uosCode = 'COMP9120'
```

- Find names of students who did not enrol in COMP9121

```
SELECT name FROM Student
WHERE sid NOT IN (SELECT sid FROM Enrolled WHERE uos = 'COMP9121);
```

- Find the number of students from each country other than AUS, that have more students than from AUS:

```
SELECT country, COUNT(sid) FROM Student WHERE country != 'AUS'
GROUP BY country HAVING COUNT(sid) >=
(SELECT COUNT(sid) FROM Student WHERE country = 'AUS');
```

- Find the names of students who have enrolled in more than one uos:

```
SELECT name FROM Students WHERE sid IN
(SELECT sid FROM Enrolled GROUP BY sid HAVING COUNT(sid) > 1);
```

- Find all the banks id that have the lowest total balance across all of their accounts

```
SELECT bank_id FROM Account GROUP BY bank_id
HAVING SUM(balance) <= ALL (SELECT SUM(balance) FROM Account GROUP BY bank_id);
```

- Find the employee with the highest salary in a given department:
  - Using ALL:

```
SELECT employee_id FROM employees WHERE department_id = 10
AND salary >= ALL (SELECT salary FROM employees WHERE department_id = 10);
```

  - Using EXISTS:

```
SELECT employee_id FROM employees e1 WHERE department_id = 10
AND NOT EXISTS (SELECT 1 FROM employees e2
WHERE e2.department_id = e1.department_id
AND e2.salary > e1.salary);
```

  - Using MAX

```
SELECT employee_id FROM employees WHERE department_id = 10
AND salary = (SELECT MAX(salary) FROM employees WHERE department_id = 10);
```

  - Using LIMIT: not for if there's tie

```
SELECT employee_id FROM employees WHERE department_id = 10
ORDER BY salary DESC LIMIT 1;
```

- Find the employee with the highest salary in each department:

```
SELECT e1.department_id, e1.employee_id FROM employees e1
WHERE e1.salary >= ALL(SELECT e2.salary FROM employees e2
                       WHERE e1.department_id = e2.department_id)
```

- Return titles of film released in 2004 and all actors who played in those films, return N/A if film has no actor

```
SELECT title, COALESCE(first_name, 'N/A') AS actor_firstname,
FROM Film LEFT JOIN Film_Actor USING (film_id) LEFT JOIN Actor USING (actor_id)
WHERE release_year = '2004';
```

- Find all actors who has the same first and last name as another actor:

```
SELECT A1.actor_id, A1.first_name, A1.last_name
FROM Actor A1 CROSS JOIN Actor A2
WHERE A1.first_name = A2.first_name AND A1.last_name = A2.last_name
AND A1.actor_id <> A2.actor_id
```

- Find the name of libraries that have at least one database book

```
SELECT library_name FROM Library NATURAL JOIN Book NATURAL JOIN Contains
WHERE Book_title ILIKE '%Database%'
```

SQL

Data Definition Language (DDL) – Schema

Data types

- Common type: CHAR(n), VARCHAR(n), INTEGER, NUMERIC(p, s), DATE, BOOL, SERIAL
- Numeric type:
  - Whole numbers: INTEGER or INT (useful for ID and count), SERIAL (auto-increment integer when inserting)
  - Exact precision and scale: NUMERIC(p, d) or DECIMAL(p, d) (p=total digit, d=digit after decimal) (useful for monetary values) NUMERIC(5,2)
- Character types:
  - Variable length: VARCHAR(n) (max length is n)
  - Fixed length: CHAR(n) (length is n, if not enough, pad with spaces)
  - Unlimited length: TEXT
- Date and time:
  - DATE, TIME, TIMESTAMP (date and time)
  - INTERVAL: a span of time (seconds, minutes, hours, days, months, years)
- Boolean: TRUE, FALSE or NULL

INTEGER	10, -5	VARCHAR(4)	‘1111’	DATE	‘2024-11-05’
NUMERIC(5,2)	123.45, -10.00	CHAR(4)	‘N ’, ‘NONE’	TIME	‘05:20:35.331369+00’
SERIAL	Like integer Auto increment	TEXT	‘abcdefghijkl’	TIMESTAMP	‘2024-11-05 05:20:35.331369+00’
BOOLEAN	True, False, Null			INTERVAL	'3 hours 15 minutes'

Tables

- Create table:
```
CREATE TABLE <table_name> (<col_name1> <type1>, <col_name2>, <type2>…);
```
- Drop table:
```
DROP TABLE [IF EXISTS] <table_name> [,<table_name2>…] [CASCADE | RESTRICT];
```
- Rename table:
```
ALTER TABLE <table_name_old> RENAME TO <table_name_new>;
```

Columns

- Add column:
```
ALTER TABLE <table_name> ADD COLUMN <col_name> <type> [<constraints>];
```
- Drop column:
```
ALTER TABLE <table_name> DROP COLUMN <col_name> [CASCADE];
```
- Rename column:
```
ALTER TABLE <table_name> RENAME COLUMN <col_name_old> TO <col_name_new>;
```
- Modify column:
  - Default value:
    - Modify default value:
```
ALTER TABLE <table_name> ALTER COLUMN <col_name> SET DEFAULT <value>;
```
    - Drop default value:
```
ALTER TABLE <table_name> ALTER COLUMN <col_name> DROP DEFAULT;
```
  - Modify data type:
```
ALTER TABLE <table_name> ALTER COLUMN <col_name> TYPE <type>;
```

Constraints

- Constraints that can be written as table constraints:
  - Add constraint:
    - Primary key constraints:
```
ALTER TABLE <table_name>
ADD PRIMARY KEY (<col_name> [, <col_name2>…]);
```
    - Foreign key constraints: note other_table can be the table itself (for self-referencing)
      - On delete options: (NO ACTION, RESTRICT – immediate) – prevent, CASCADE – delete, (SET NULL, SET DEFAULT) – update value
      - The referenced attribute must be a PK or CK (UNIQUE) in the other table
```
ALTER TABLE <table_name>
ADD FOREIGN KEY (<cols>) REFERENCES <ref_table> [(<ref_cols>)]
[ON DELETE <option>] [ON UPDATE <option>];
```
    - Unique constraints:
```
ALTER TABLE <table_name>
ADD CONSTRAINT <constraint_name> UNIQUE (<col_name> [, <col_name2>]);
```
    - Domain constraints:
      - Condition: anything that evaluates to a Boolean
```
ALTER TABLE <table_name>
ADD CONSTRAINT <constraint_name> CHECK (<condition>);
```
      - Or create a domain:
```
CREATE DOMAIN <domain_name> <data_type> [DEFAULT <value>]
[CONSTRAINT constraint_name] {NOT NULL | NULL | CHECK (condition)};
```
  - Drop constraint:
```
ALTER TABLE <table_name> DROP CONSTRAINT <constraint_name>;
```
  - Modify constraint: drop then add: fail and raise error if exists data that violates the new constraint
- Not null constraints:
  - Add not null:
```
ALTER TABLE <table_name> ALTER COLUMN <col_name> SET NOT NULL;
```
  - Drop not null:
```
ALTER TABLE <table_name> ALTER COLUMN <col_name> DROP NOT NULL;
```
- Timing of constraints: if set as deferred, allow to check later after transaction finishes
```
ADD CONSTRAINTS <constraint_name> <constraint>
[NOT DEFERRABLE | DEFERRABLE INTITIALLY {DEFERRED | IMMEDIATE}]
SET CONSTRAINTS <constraint_name> {IMMEDIATE | DEFERRED};
```
- Different ways to add constraint:
  - During table creation:
    - In line:
```
CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) UNIQUE,
    salary NUMERIC(10, 2) DEFAULT 50000 CHECK (salary >= 30000),
    department_id INT REFERENCES departments(department_id));
```
    - At the bottom:
```
CREATE TABLE employees (
    employee_id SERIAL,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    salary NUMERIC(10, 2),
    department_id INT,
    PRIMARY KEY (employee_id),
    UNIQUE (last_name),
    FOREIGN KEY (department_id) REFERENCES departments(department_id),
    CHECK (salary >= 30000),
    CONSTRAINT salary_default DEFAULT 50000 FOR salary);
```
    - At the bottom (with names): `CONSTRAINT <constraint name> <constraint detail as above>`
  - Added later: using alter table as shown above

Data Manipulation Language (DML) – Insertion

- Insert tuples:
  - If column name not specified, the order of values must match with the column order

    ```
    INSERT INTO <table_name> [(<col1, col2…>)] VALUES
    (attr1a, attr2a,…), [(attr1b, attr2b,…),…];
    ```
  - Can use DEFAULT for value to access default value: ('1234', 'John', `DEFAULT`, 8)
  - Copying data from existing table:

    ```
    INSERT INTO <table_name> [(<col1, col2…>)]
    SELECT <attr1, attr2,…> # can mix with hardcoded values
    FROM <table_name2> WHERE condition;
    ```
- Delete tuples:

  ```
  DELETE FROM <table_name> [WHERE condition];
  ```
- Update tuples:

  ```
  UPDATE <table_name> SET <col_name> = <value> [WHERE condition];
  ```

Data Manipulation Language (DML) – Query

Basic query

- SELECT…[FROM…WHERE…GROUP BY…HAVING… WINDOW… ORDER BY…LIMIT…OFFSET…]
- Order of execution: FROM-WHERE-GROUP BY-HAVING-WINDOW-SELECT-ORDER BY-LIMIT-OFFSET

SELECT

- Select one or more columns: `SELECT col1, [col2, col3…]`
- Select column and give alias: `SELECT col1 AS col_alias`
  - Still need to refer to the original column name in the query, except for in the ORDER BY clause
- Select all columns (wildcard): `SELECT *`
- Select hardcoded number/string (create a new column with same values): `SELECT 1` or `SELECT 'str'`
  - Useful with existence checks, or placeholder column with a more meaningful value
- Select with arithmetic operation on column result: `SELECT num_col * 2 +1` or `SELECT abs(col)`
- Select an aggregate (whole table or for a certain group using GROUP BY): `SELECT AVG(num_col)`
- Select unique rows (remove duplicated rows): `SELECT DISTINCT col1, col2`
  - If selecting more than one column, remove duplicated combinations (i.e. check duplicated (col1, col2))
- Select with conditional expressions: `CASE…WHEN`, `COALESCE`, `GREATEST`, `LEAST`
- Select window function: `rank() | AVG() OVER(PARTITION BY ORDER BY)`

WHERE

- Comparison: works for numeric and string values
  - Comparison operators: `=, >, >=, <, <=, !=, <>`
  - Within a range (inclusive): `BETWEEN val1 AND val2`
  - Set comparison:
    - Specify multiple possible values: `IN (val1, val2, val3…)`
    - At least one row in a subquery: `EXISTS (SELECT 1 FROM <table> WHERE <condition>)`
    - At least one value in subquery satisfies the comparison (or): `op ANY (subquery)`
    - All values in subquery satisfy the comparison (and): `op ALL (subquery)`
- Logical operators: `NOT, AND, OR`
  - Can mix multiple operators together, use () to indicate order of evaluation
- Pattern matching:
  - Match a keyword: `LIKE` (case-sensitive) or `ILIKE` (case-insensitive)
    - Or use `LOWER(col) LIKE 'str'` for case-insensitive matching
  - Match any sequence (zero or more) `%`, match one character `_`
    - `LIKE 'str%'` (match words starting with str)
    - `LIKE '%str%'` (match words containing str)
    - `LIKE 'str_'` (match a single character)
    - `LIKE 'str___'` (matches three characters after str)
  - SIMILAR TO: `SIMILAR to <pattern>`
    - Pattern regular expressions: `|` either, `*` repetition (0 or more), `+` repetition (1 or more), `?` repetition (0 or 1), `()` group items into a single logical item
    - E.g. match words starting with either Advanced or Data: `SIMILAR TO '(Advanced|Data)%'`

FROM

- From a single table: `FROM table_name`
- From multiple tables: use join
  - Join type: how **rows** are joined (LEFT, RIGHT, FULL, INNER)
    - Combine same rows, pad with null if not exist in other tables
      - Note INNER JOIN sometimes can achieve similar result as IN
    - `R1 INNER JOIN R2 USING (<col>)`: keep tuples both in R1 and R2
    - `R1 LEFT JOIN R2 USING (<col>)`: keep tuples in R1
    - `R1 RIGHT JOIN R2 USING (<col>)`: keep tuples in R2
    - `R1 FULL JOIN R2 USING (<col>)`: keep tuples in R1 + R2
  - Join condition: how **columns** are joined (JOIN ON, JOIN USING, NATURAL JOIN)
    - `R1 JOIN R2 ON (R1.col1 op R2.col2)` or `R1, R2 WHERE R1.col1 op R2.col2`
      - Useful when condition is not =, or col names are different (if same name, must use table name to distinguish), keep duplicated columns
    - `R1 JOIN R2 USING (<cols>)` or `R1 NATURAL JOIN R2`:
      - Must have same name column(s), remove duplicate columns (for USING, only the specified columns are removed duplicates)
  - Cross join `R1 CROSS JOIN R2` or `R1, R2`: permutation (output = R1's col + R2's col, R1's row * R2's row)
- Give table alias: `FROM table1_name [AS] table1_alias`
  - If an alias is given, the original table name cannot be used and must refer to the alias in the query
  - Table name/alias can be omitted if omission doesn't cause confusion

Null values

- Null: unknown value, not the same as '' (empty string) or 0
- Check for null: SELECT…FROM…`WHERE col IS NULL` or `WHERE col IS NOT NULL`
- Three-valued logic: output when involving null values
  - Arithmetic: Null
  - Comparison operator: Null
  - OR – return max, AND – return min, NOT – return (1-), where True is 1, Null is 0.5, False is 0
- Make null values visible in a query: COALESCE(col, <value if null>)
  - E.g. return manager_id or N/A if none: `SELECT COALESCE(manager_id, 'N/A') FROM employees`
- Turn certain values to NULL: NULLIF(col, <value>) return NULL if col value = 'value'
  - E.g. turn all 'NaN' in salary column to NULL: `SELECT NULLIF(salary, 'NaN') FROM employees`

String functions and operators:

- String operators:
  - || concatenation, SUBSTRING(text [FROM start num] [FOR len num]) extract substring
  - LOWER(), UPPER()
- String functions: LENGTH(text) number of characters

Data and time

- TIME (time), DATE (date), TIMESTMAP (date & time), TIMESTAMP WITH TIME ZONE (incl time zone)
- Constants: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `NOW()` (function for current time)
  - Given time in the correct time zone: TIME AT TIME ZONE 'AEST'
- Extract individual time fields: `EXTRACT(<component> FROM <value>)`
  - Date-related component (DATE and TIMESTAMP): `DAY, MONTH, YEAR`
  - Time-related component (TIME and TIMESTAMP): `HOUR, MINUTE, SECOND, TIMEZONE`
  - Day of the year: `DOY`, Day of the week (Mon): `ISODOW`, Week of the year: `WEEK`
- Make date using year, month and day: `MAKE_DATE(year, month, date)`
- Convert from string to date: `TO_DATE(<string>, <format of string>)`
  - e.g. `TO_DATE(12||'-'||'December'||'-'||2024, 'DD-Month-YYYY')`

Mathematical functions and operators

- Arithmetic operators:
  - + addition, - subtraction, * multiplication, / division, % modulo, ^ exponentiation

2+3	2-3	2*3	2/3	2.0/3.0	2%3	2^3
5	-1	6	0	0.66666666666666666667	2	8

- Mathematical functions:
  - abs(num) absolute value, sqrt(num) square root, exp(num), factorial(num), log(num) base 10 log, pi()

abs(-1)	sqrt(2)	exp(2)	factorial(5)	log(100)	pi()
1	1.4142135623730951	7.38905609893065	120	2	3.141592653589793

  - ceil(num) round up, floor(num) round down, round(num, d) round to d decimal

ceil(5.0/3.0)	floor(5.0/3.0)	round(5.0/3.0,2)	trunc(5.0/3.0,2)
2	1	1.67	1.66

ceil(-5.0/3.0)	floor(-5.0/3.0)	round(-5.0/3.0,2)	trunc(-5.0/3.0,2)
-1	-2	-1.67	-1.66

  - trunc(num, d) remove decimals after d decimal, mod(num1, num2) remainder of 1/2

trunc(5.0/3.0)	mod(5.0, 3.0)
1	2

- Special values for floating point data: '-infinity' < any number < 'infinity' < 'NaN'

Aggregation

- COUNT, SUM, AVG, MAX, MIN, e.g. COUNT(col)
- Only perform the functions on non-null values
  - Exception is `COUNT(*)`, which counts null values as well (total number of rows in a table)
- SUM, AVG only works on numeric values; COUNT, MAX, MIN works on string as well
- Can use DISTINCT with COUNT to only count distinct values: `COUNT(DISTINCT col)`
- Special string aggregate on function: `STRING_AGG(col, sep [ORDER BY cols])`
  - Combine all strings in a set (group) by concatenating into a single string split by separator characters

Grouping

- Calculate a value for different subsets of the data, partition a query result into groups of values (rows in the same group has the same value for certain attributes) and do aggregate function on the group
- Sequence: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY

```
SELECT <col1>, aggregate<col> # col1 must be in the GROUP BY list, or any aggregate
FROM <table>
WHERE <condition> # condition can be on any col in FROM table, can't use aggregate
GROUP BY <col1> [, <col2>…]
[HAVING aggregate<col> <condition> AND|OR <col1> <condition>]; # col restriction
```

- Any col in the SELECT/HAVING clause must be in the grouping list, but aggregate can use any column
  - So ensure all columns in SELECT are included in GROUP BY

HAVING

- Further filters the aggregated result of GROUP BY:

```
SELECT department, AVG(salary) FROM employees GROUP BY department
HAVING AVG(salary) < 50000 # only department with average salary < 50k will be returned
```

ORDER BY

- Order the result by one or more columns: `ORDER BY col1, [col2, col3…]`
  - If more than one column is given, latter columns are used to break ties
  - Order in ascending order (default): ASC
  - Order in descending order: DESC
- E.g. sort by last name from A-Z, then by salary DESC: `ORDER BY last_name ASC, salary DESC`

LIMIT and OFFSET

- To restrict the size of the output to a pre-defined number of tuples (e.g. top three)

```
LIMIT {count | ALL} OFFSET start_num
```

  - Count: the maximum number of rows to return (ALL if not specified)
  - Start: the number of rows to skip before starting to return (0 if not specified)
- ORDER BY + LIMIT: Can be used for highest (when want to return just one, ignoring ties)

```
SELECT…FROM…WHERE…ORDER BY…LIMIT <positive int>
```

Window functions

- Performs a calculation across a set of table rows that are somehow related to the current row
  - Window: the set of rows related to the current row for the function's calculation
  - Preserves row-level detail, while adding a new column with aggregation of the window (related rows)

```
<window_function>() OVER (PARTITION BY <cols> ORDER BY <cols>)
```

  - Window functions: RANK(), ROW_NUMBER(), SUM(), AVG() etc
  - OVER(): how the rows are grouped, window frame
  - PARTITION BY: divides the result into partitions (groups) of rows, and apply window function separately within each partition (similar to GROUP BY without collapsing rows into a single value per group)
    - E.g. Show department, employee and salary (individual), and a sum of salary by each department (e.g. employees within the same department will have the same value in dept_sal)

```
SELECT dept, employee, salary, sum(salary) OVER (PARTITION BY dept) AS dept_sal
```

  - ORDER BY: defines how rows are ordered within each partition
    - Function like rank() and row_number() must have ORDER BY
    - E.g. calculate the running sum of salary ordered by employee name (employee 1's sum will show their salary, employee 2's sum column will show their salary + employee 1's salary)

```
SELECT employee, salary, sum(salary) OVER (ORDER BY employee)
```

- Rank with the rank number: rank() + ORDER BY + LIMIT
  - Rank(): produces a numerical rank for each distinct ORDER BY value in the current row's partition

```
SELECT col1, col2, col3, rank() OVER (ORDER BY col2 DESC)
```

- E.g. select the top-5 categories by number of films

```
SELECT rank() OVER (ORDER BY COUNT(film_id) DESC) AS rank # produce the rank
name AS category, COUNT(film_id) AS films # other columns as requested
FROM Film_Category JOIN Category USING (category_id)
GROUP BY name # needed because we're returning COUNT(film_id) and using it in rank()
ORDER BY rank, category # order of the final result
LIMIT 5; # select the top n only
```

Conditional expressions

CASE

- CASE: retrieve certain values based on a conditional expression in the SELECT clause

```
SELECT col1, col2, CASE WHEN <condition1> THEN <expr>
WHEN <condition2> THEN <expr> ELSE <expr> END
```

- Useful for giving more meaningful description of column result WHEN col = 'value' THEN 'alias'

COALSECE and NULLIF

- See null value section

GREATEST and LEAST

- Select the largest or smallest value from a list of any number of expressions (ignoring null values)
- Used to find the largest or smallest value in a row (cf. MAX and MIN which returns a value for a column)
  - E.g. find the highest and lowest score for student's exam scores

```
SELECT sid, GREATEST(math, biol, phys), LEAST(math, biol, phys) FROM scores;
```

Views

- A virtual table that can be queried:
- Create view: `CREATE VIEW <view_name> AS <query: SELECT…FROM…WHERE>`
- Use view: `SELECT * FROM <view_name>`
- Delete view: `DROP VIEW <view_name>`

Nested query

- Can nest queries in SELECT, FROM, WHERE or HAVING (if use in FROM, need to use AS to give a name)
  - Co-related subquery: the inner query references attributes in the outer query (that are not in the inner query – check: can the inner query be run independently?) – run once for every row in outer query
    - Useful for row-specific comparisons or filtering
  - Non co-related subquery: the inner query can be run independently – inner query run only once
- Useful for:
  - Finding highest/lowest: especially where there are ties (otherwise LIMIT 1 would work)

```
WHERE <condition1> AND col1 = (SELECT MAX(col1) FROM table WHERE <condition1>)
```

  - Finding above average:

```
WHERE col1 > (SELECT AVG(col1) FROM table)
```

  - Check existence: each row in the outer query that doesn't satisfy inner query is removed from the result

```
WHERE EXISTS (SELECT 1 FROM t2 WHERE t2.col1 = t1.col1 AND t2.col2 = 'x')
```

EXISTS

- EXISTS and NOT EXISTS
- Useful for checking existence of related records in another table (ignore NULL values in inner query)

```
SELECT t1.cols FROM t1
WHERE EXISTS (SELECT 1 FROM t2 # t2 is related to t1
              WHERE t1.col = t2.col # how t1 and t2 are related (common column)
              AND <condition in t2> # some condition based on columns in t2
```

- Difference with IN: IN checks for specific value (return the values), EXISTS checks for existence (return T or F)

Set Operations

- Combine results (rows) of different queries (removes duplicate first)
  - Union: `(query1) UNION (query2)`
  - Intersection: `(query1) INTERSECT (query2)`
  - Difference: `(query1) EXCEPT (query2)`
- Keep duplicated rows: UNION ALL, INTERSECT ALL, EXCEPT ALL

Relational division (For-All)

- Can use set difference to replicate relational division (for-all)
  - E.g. find all students who have taken all COMP units (NOT EXISTS + EXCEPT)
    - For each student, find all COMP units taken and compare with all COMP units, exclude if different

```
SELECT sid, name FROM Student S WHERE
EXISTS (SELECT uos FROM UnitofStudy WHERE uos LIKE 'COMP%') AND
# ensure there are units starting with COMP, otherwise the first select return empty, so except
return empty, not exist always return True
NOT EXISTS ((SELECT uos FROM UoStudy WHERE uos LIKE 'COMP%') # Get all COMP units
EXCEPT # result is all COMP units not taken by the student, if >0, return false
(SELECT uos FROM Enrolled E WHERE E.sid = S.sid AND E.uos LIKE 'COMP%')) # Get
list of COMP units taken
```

- Or compare the size of the involved sets

```
SELECT sid, name FROM Student NATRUAL JOIN Enrolled WHERE uos LIKE 'COMP%'
GROUP BY sid, name
HAVING COUNT(grade) = (SELECT COUNT(*) FROM UoStudy WHERE uos LIKE 'COMP%')
```

Testing for set equality

- Set A = Set B if A-B = None and B-A = None
- Check if two relations have the exact same content R(a,b) and S(c,d)

```
SELECT 'R and S are the same'
WHERE NOT EXISTS ((SELECT a, b FROM R) EXCEPT (SELECT c, d FROM S))
AND NOT EXISTS ((SELECT c, d FROM S) EXCEPT (SELECT a, b FROM R))
```

- Check if same value from R and S have the same dependent values (c) in T R(a,…), S(a,…) and T(a,c)

```
SELECT R.a, S.a FROM R, S WHERE NOT EXISTS (
(SELECT c FROM T WHERE T.a = R.a) EXCEPT (SELECT c FROM T WHERE T.a = S.a)) AND
NOT EXISTS (
(SELECT c FROM T WHERE T.a = S.a) EXCEPT (SELECT c FROM T WHERE T.a = R.a))
```

Transactions

- Create a transaction:

```
BEGIN;
[SET TRANSACTION ISOLATION LEVEL {READ COMMITED | REPEATABLE | SERIALIAZABLE};
[LOCK TABLE <table name> IN <lock_mode>;]
<SQL read or write statements>;
COMMIT | ROLLBACK;
```

Locks

- Row-level lock: FOR SHARE (another transaction can read but not modify), FOR UPDATE

```
BEGIN; SELECT…FROM…WHERE… <lock_mode>; <update_statement>; COMMIT;
```

- Table-level lock: ACCESS SHARE, ROW SHARE, EXCLUSIVE…

```
LOCK TABLE <table_name> IN <lock_mode>;
```

- Timeout for deadlock: `SET lock_timeout = '5s';`

Indexes

- Create: `CREATE INDEX <index_name> ON <table_name> (cols)`
- Delete: `DROP INDEX [IF EXISTS] <index_name>`
- Check indexes: `SELECT * FROM PG_INDEXES`

Stored functions

❖ Return a value or table for use in SQL queries

```
CREATE [OR REPLACE] FUNCTION function_name(input1 datatype1, input2 datatype2…)
RETURNS <value or table> AS $$
[DECLARE variable variable_type;]
BEGIN
<query>;
END; $$ LANGUAGE plpgsql;
```

➢ Return value:

```
RETURNS return_datatype AS $$
BEGIN RETURN <value statement>; or
IF…THEN RETURN <val1>; ELSIF…THEN RETURN <val2>; ELSE…RETURN <val3>; END IF;
END;
```

➢ Return table:

```
RETURNS TABLE(col1 col1_type, col2 col2_type,…) AS $$
BEGIN
RETURN QUERY SELECT…FROM…WHERE…; END;
```

❖ Calling a stored function: `SELECT function_name(inputs)`

Stored procedures

❖ Perform actions without returning a value

```
CREATE [OR REPLACE] PROCEDURE procedure_name (input1 datatype1, input2
datatype2,…)
LANGUAGE plpgsql AS $$
BEGIN
    -- Procedure logic here
END; $$;
```

❖ Calling a stored procedure: `CALL procedure_name(inputs)`

Triggers

❖ Create function: stored procedure

➢ OLD.col: value before update or deleted; NEW.col: new value inserted or updated

```
CREATE [OR REPLACE] FUNCTION <function_name()>
RETURNS TRIGGER AS $$
BEGIN
<actions>
END;
$$ LANGUAGE plpgsql;
```

➢ Raise exception: raise an exception if new inserted value violates condition

```
IF NEW.<col> <condition> THEN RAISE EXCEPTION 'message', NEW.<col>;
END IF;
RETURN NEW;
```

➢ Insert and return old values:

```
INSERT INTO employee (employee_id, deleted_at)
VALUES (OLD.employee_id, NOW());
RETURN OLD;
```

❖ Create trigger:

```
CREATE [OR REPLACE] TRIGGER <trigger_name> # create or update
{BEFORE | AFTER | INSTEAD OF} # timing
{INSERT | UPDATE | DELETE} [OR…] # event, if want more than one, use OR
ON <table_name>
[REFERENCING {OLD|NEW} TABLE AS table_alias]
[FOR EACH {ROW | STATEMENT}] # granularity, =statement if omitted
[WHEN (condition)] # condition, optional
EXECUTE {FUNCTION | PROCEDURE} <function_name(arguments)> # action
```

➢ Trigger on data changes: check input is valid and log the person who changed it

```
CREATE FUNCTION emp_stamp() RETURNS TRIGGER AS $emp_stamp$ BEGIN
          IF NEW.salary IS NULL THEN
              RAISE EXCEPTION '% cannot have null salary', NEW.empname;
          END IF;

          NEW.last_date := current_timestamp;
          NEW.last_user := current_user;
          RETURN NEW;
END; $emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

➢ Trigger to implement semantic constraints: an employee can't work in more than 3 departments

```
CREATE FUNCTION department_limit() RETURNS TRIGGER AS $$ BEGIN
    IF (SELECT COUNT(*) FROM works_in
        WHERE employee_id = NEW.employee_id >= 3 THEN
        RAISE EXCEPTION 'Employee cannot work in more than 3 departments'
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER department_limit_trigger BEFORE INSERT OR UPDATE ON works_in
FOR EACH ROW EXECUTE FUNCTION department_limit();
```

Assertions

❖ Create assertions: test for validity on every update (usually using not exists)

```
CREATE ASSERTION <assertion_name> CHECK <condition>;
```

❖ Simulating assertion with check constraint:

➢ Where to place the check: if the restriction is to have less than 10 sailor+boat for all clubs, the following check would fail as it's only triggered on update in the Sailor table, but we can add more boats – Solution: add the check in boats as well

```
CHECK((SELECT COUNT(sailor) FROM Sailors)+(SELECT COUNT(boat) FROM Boats) < 10)
```

➢ Threshold of check: as the check is triggered before each insert, if we have the same threshold, we can insert one more than the allowed threshold – Solution: reduce the threshold by 1

```
CHECK((SELECT COUNT(sailor) FROM Sailors)+(SELECT COUNT(boat) FROM Boats) < 9)
```

❖ Assertion: each loan has at least one borrower who has an account with at least $1000

```
CREATE ASSERTION balance_constraint CHECK (NOT EXISTS
    (SELECT 1 FROM Loan WHERE NOT EXISTS
        (SELECT borrower.loan_id, borrower.balance
        FROM borrower NATURAL JOIN depositor NATURAL JOIN account
        WHERE loan.loan_id = borrower.loan_id AND account.balance >= 1000)));
```

❖ Simulating with a stored function: less than 50 employees and sum(project & investment type) <10

➢ Assertion:

```
CREATE ASSERTION sme_company_check CHECK(
(SELECT COUNT(*) FROM Employee) < 50) AND
((SELECT COUNT(*) FROM Project) + (SELECT COUNT(*) FROM Investment) < 10)));
```

➢ Check constraint:

```
CREATE OR REPLACE FUNCTION check_sme_company() RETURNS BOOLEAN
LANGUAGE plpgsql AS $$ BEGIN
IF (SELECT COUNT(*) FROM Employee) < 49)
AND ((SELECT COUNT(*) FROM Project) + (SELECT COUNT(*) FROM Investment) < 9)))
THEN RETURN TRUE; ELSE RETURN FALSE; END IF; END;$$;
```

▪ Using the constraint:

```
CREATE TABLE Employee (employee_id INTEGER PRIMARY KEY, name CHAR(50),
CHECK (check_sme_company()));
```

❖ Check constraint: every class has a maximum enrolment of 30 students

```
CREATE TABLE Enrolled (snum INT, cname CHAR(20), PRIMARY KEY (snum, cname),
FOREIGN KEY (snum) REFERENCES Student, FOREIGN KEY (cname) REFERENCES Class,
CHECK((SELECT COUNT(snum) FROM Enrolled GROUP BY cname) <= 30);
```