Transaction

Finite sequence of database operations and constitutes the smallest logical unit of work from the application perspective

ACID properties: of transaction T

1. Atomicity: either all effects of T are reflected in the database or none
2. Consistency: T guarantees to yield the correct state of the database
3. Isolation: T is isolated from the effect of concurrent transactions
4. Durability: after a commit of T, its effects are permanent

Equivalence: same effect on the data

Serializability: execution is equivalent to some serial execution of the same set of transactions

Relations

Set of tuples (or records) represented by R(A_1, A_2, …, A_n) where every instance of the schema is a relation which is a subset of $\{(a_1, a_2, \dots, a_n) | a_i \in dom(A_i) \cup \{NULL\}\}$

Domain: set of atomic values whose values are either in the domain or NULL

Key constraints

Superkey: subset of attributes that uniquely identifies a tuple in a relation

Key: superkey that is also minimal (i.e. no proper subset of the key is a superkey)

Candidate keys: set of all keys

Primary key: one selected key

Prime attributes: attributes of a candidate key

Properties:

1. If (A, B, C) is a superkey, then (A, B, C, D) is a superkey
2. If (A, B, C) is a key, then (A, B, C) is a superkey
3. If (A, B) is a superkey, then (A, B, C) is NOT a key
4. If (A, B) is a key, then it is possible that (B, C, D) is a key
5. Every relation has at least one superkey

Foreign key constraints

Subset of attributes of relation R_1 that refers to the primary key of relation R_2 where R_1 is the referencing relation and R_2 is the referenced relation

$$(R_1.A_{i1}, R_1.A_{i2}, \dots) \rightsquigarrow (R_2.A_{j1}, R_2.A_{j2}, \dots)$$

Criteria: each foreign key in R_1 must satisfy one of the following

1. Appear as primary key in R_2
2. Be a NULL value (or a tuple containing at least one NULL value)

Three-Valued logic

c_1	c_2	$c_1 \wedge c_2$	$c_1 \vee c_2$	$\neg c_1$
F	F	F	F	T
F	T	F	T	T
F	N	F	N	T
T	F	F	T	F
T	T	T	T	F
T	N	N	T	F
N	F	F	N	N
N	T	N	T	N
N	N	N	N	N

Operations with NULL

If NULL on any side, then result is NULL unless

c_1	c_2	$c_1 \equiv c_2$	$c_1 \not\equiv c_2$
N	N	T	F
c_1	N	F	T
N	c_2	F	T
c_1	c_2	$c_1 = c_2$	c_1 <> c_2

Relational algebra

Closure property: set of values closed under the set of operators if any combination of the operators produces only values in the given set

- All relations are closed under relational algebra

Selection: $\sigma_{[c]}(R)$ selects all tuples from a relation that satisfy the condition c

- Results have the same schema as the input relation
- Number of rows often smaller
- c must specify only attributes in R

Projection: $\pi_{[l]}(R)$ keeps only the columns specific in the ordered list l and in the same order

- l must not contain any operations or duplicate attributes
- Number of rows may be smaller since relations are a set of tuples
- l must specify only attributes in R

Renaming: $\rho_{[\mathfrak{R}]}(R)$ renames the attributes listed in $\mathfrak{R}$

- $\mathfrak{R}$ is an unordered collection of $B_i \leftarrow A_i$ (new <- old)
- Order of columns remain unchanged, and rows remain the same
- No two different attributes can be renamed to the same name
- No attributes can be renamed more than once in a single operation

Union compatibility: relation R and S are union-compatible if they both have the same number of attributes AND have the corresponding attributes (order matters) have the same or compatible domains

Cross product (R x S)

Relation formed by combining all pairs of tuples from the input relations

$$R \times S = \{(a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n) | (a_1, a_2, \dots, a_n) \in R \wedge (b_1, b_2, \dots, b_n) \in S\}$$

- Attributes in R and S cannot have the same names
- |R x S| = |R| x |S|
- Paired with selection operation to remove unnecessary rows and projection to remove unnecessary/duplicate columns

Join operators

Combines cross product, selection, and (optionally) projection into a single operator

Inner join: includes only tuples that satisfy the condition

1. θ-join: $\bowtie_{[\theta]} = \sigma_{[\theta]}(R \times S)$
2. Equi join: $\bowtie_{=}$
   - Only relational operator allowed is equality
3. Natural join: $\bowtie\, = R \bowtie S = \pi_{[l]}(R \bowtie_{[\theta]} S)$
   - Join performed over all common attributes (same name and domain)
   - Common attribute columns appear once only
   - If no common attributes, then a cross product occurs

Outer join: includes tuples that do not satisfy the condition

1. Left outer join: $⟕_{[\theta]} = R ⟕_{[\theta]} S \cup (\text{dangle}(R \bowtie_{[\theta]} S) \times \{\text{null}(S)\})$
2. Right outer join: $⟖_{[\theta]} = R ⟖_{[\theta]} S \cup (\{\text{null}(S)\} \times \text{dangle}(S \bowtie_{[\theta]} R))$
3. Full outer join: $⟗_{[\theta]} = R ⟗_{[\theta]} S \cup (\text{dangle}(R \bowtie_{[\theta]} S) \times \{\text{null}(S)\}) \cup (\{\text{null}(S)\} \times \text{dangle}(S \bowtie_{[\theta]} R))$
4. Natural outer join
   - Same criteria as natural join

Equivalence

$Q_1 \equiv Q_2$ if for any input, both relations either both produce error or both produce the same result (strong equivalence)

Weak equivalence: if there is no error, then both produces the same result

Common strong equivalence:

Selection

- $\sigma_{[c_1]}(\sigma_{[c_2]}(R)) \equiv \sigma_{[c_2]}(\sigma_{[c_1]}(R))$
- $\sigma_{[c_1]}(\sigma_{[c_2]}(R)) \equiv \sigma_{[c_1 \wedge c_2]}(R)$

Projection

- $\pi_{\ell_1}(\pi_{\ell_2}(R)) \not\equiv \pi_{\ell_1}(R)$ *(unless $\ell_1 \subseteq \ell_2$)*

Cross Product and Joins

- $R \times S \not\equiv S \times R$ *(different column order*)*
- $R \bowtie S \not\equiv S \bowtie R$ *(i.e., non-commutative)*
- $R \times (S \times T) \equiv (R \times S) \times T$
- $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ *(i.e., associative)*
- $R \bowtie_{[\theta_1]} (S \bowtie_{[\theta_2]} T) \not\equiv (R \bowtie_{[\theta_1]} S) \bowtie_{[\theta_2]} T$

  *(if θ_1 uses T **and** θ_2 uses R)*

Combined

- $\pi_{[\ell]}(\sigma_{[\theta]}(R)) \not\equiv \sigma_{[\theta]}(\pi_{[\ell]}(R))$ *(unless θ uses only attributes in ℓ)*
- $\sigma_{[\theta]}(R \times S) \not\equiv \sigma_{[\theta]}(R) \times S$ *(unless θ uses only attributes in R)*

Common weak equivalence:

Selection

- $\sigma_{[c_1]}(\sigma_{[c_2]}(R)) \equiv \sigma_{[c_2]}(\sigma_{[c_1]}(R))$
- $\sigma_{[c_1]}(\sigma_{[c_2]}(R)) \equiv \sigma_{[c_1 \wedge c_2]}(R)$

Projection

- $\pi_{\ell_1}(\pi_{\ell_2}(R)) \equiv \pi_{\ell_1}(R)$

Cross Product and Joins

- $R \times S \not\equiv S \times R$ *(different column order*)*
- $R \bowtie S \not\equiv S \bowtie R$ *(i.e., non-commutative)*
- $R \times (S \times T) \equiv (R \times S) \times T$
- $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ *(i.e., associative)*
- $R \bowtie_{[\theta_1]} (S \bowtie_{[\theta_2]} T) \equiv (R \bowtie_{[\theta_1]} S) \bowtie_{[\theta_2]} T$

Combined

- $\pi_{[\ell]}(\sigma_{[\theta]}(R)) \equiv \sigma_{[\theta]}(\pi_{[\ell]}(R))$
- $\sigma_{[\theta]}(R \times S) \equiv \sigma_{[\theta]}(R) \times S$

Additional rules: ***

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections
$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right)$$
2. Selection operations are commutative
$$\sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right) = \sigma_{\theta_2}\left(\sigma_{\theta_1}(E)\right)$$
3. Only the last in a sequence of projections is needed
$$\pi_{L_1}\left(\pi_{L_2}\left(\dots\left(\pi_{L_n}(E)\right)\dots\right)\right) = \pi_{L_1}(E)$$
4. Selection can be combined with Cartesian products and theta joins
$$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$$
$$\sigma_{\theta_1}\left(E_1 \bowtie_{\sigma_{\theta_2}} E_2\right) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$
5. Theta joins and natural joins are commutative
$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$
6. Natural joins are associative
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$
7. Theta joins are associative where θ_2 involves attributes in E2 and E3 only
$$\left(E_1 \bowtie_{\theta_1} E_2\right) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} \left(E_2 \bowtie_{\theta_2} E_3\right)$$
8. Selection operation distributes over theta join when all attributes in selection involve only the attributes of one of the expressions being joins
$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = \left(\sigma_{\theta_0}(E_1)\right) \bowtie_\theta E_2$$
9. Selection operation distributes over theta join when each selection condition only involves attributes of its own relation
$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = \left(\sigma_{\theta_1}(E_1)\right) \bowtie_\theta \left(\sigma_{\theta_2}(E_2)\right)$$
10. Projection distributes over theta join when theta involves only attributes in $L_1 \cup L_2$
$$\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \left(\pi_{L_1}(E_1)\right) \bowtie_\theta \left(\pi_{L_2}(E_2)\right)$$
11. Projection distributes over theta join (alternate)
$$\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1 \cup L_2}\left(\left(\pi_{L_1 \cup L_3}(E_1)\right) \bowtie_\theta \left(\pi_{L_2 \cup L_4}(E_2)\right)\right)$$
    - where L1 and L2 be sets of attributes from E1 and E2 respectively; L3 be attributes of E1 that are involved in the join condition but not in L1 U L2; L4 be attributes of E2 that are involved in the join condition but not in L1 U L2

12. Union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

13. Set difference is not commutative

$$E_1 - E_2 \neq E_2 - E_1$$

14. Union and intersection are associative

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

15. Selection operation distributes over union, intersection, and difference

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2 = \sigma_P(E_1) - \sigma_P(E_2)$$

16. Project distributes over union

$$\pi_L(E_1 \cup E_2) = \left(\pi_L(E_1)\right) \cup \left(\pi_L(E_2)\right)$$

Invalid relational expressions

1. Attribute no longer available after projection
2. Attribute no longer available after renaming
3. Incompatible attribute types

Valid but redundant relational expressions

1. Cross product + attribute selection = join
2. Unnecessary operators
3. Unoptimized query

ER Model

Diagrammatic representation of database schema; cannot encode constraints like not-NULL, unique, data types

Entity: representation of real-world objects that are distinguishable from other objects

Entity set: collection of entities of the same type (mapped to tables); typically plural non-proper noun names

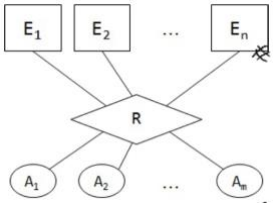
Attributes: information describing an entity (mapped to column)

1. Key attributes (underlined): uniquely identify each entity (mapped to primary key)
2. Composite attributes: composed of multiple other attributes
3. Multivalued attributes (double lined): one or more values for a given entity
4. Derived attributes (dashed line): derived from other attributes (don't appear in implementation)

Relationship: association among one or more entities

Relationship set: collection of relationships of same type; may contain own attributes; typically verb names

- Degree: number of entity sets involved in relationship set; no limit but binary is most common



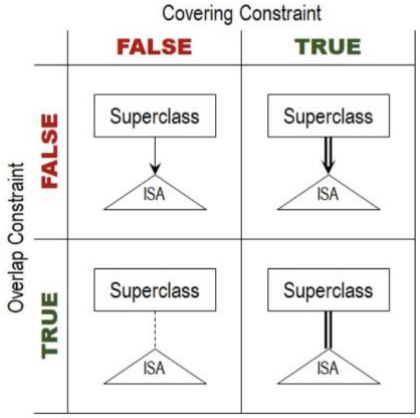
Relationship constraints:

1. Cardinality: upper bound (1 (key constraint) or inf) to number of times an entity can participate in a relationship
2. Participation: lower bound (0 (partial participation) or 1 (total participation)) to number of times an entity can participate in a relationship
3. Dependency: entity set that contains partial keys (i.e. composite key with owning entity); must have identifying relationship set connecting to owning entity set

Cardinality constraint: many-to-many, many-to-one, one-to-one

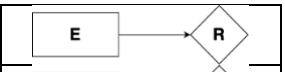
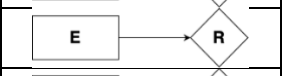
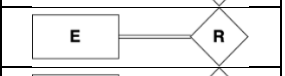
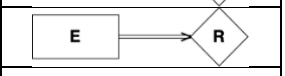
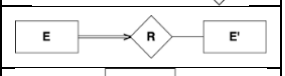
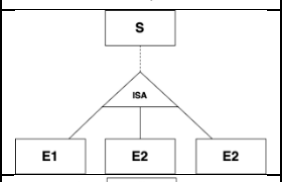
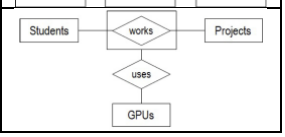
ISA hierarchy: supporting superclass/subclass relationships

- Every entity in a subclass inherits its primary key of the superclass
- Subclass key is not shown in ER diagram
- Subclass contains own attributes or relationships
- Overlap constraint: can a superclass entity belong to multiple subclasses?
- Covering constraint: must a superclass entity belong to at least one subclass?



Aggregation: abstraction that treats relationships as higher-level entities; behaves as both relationships and entities

ER relationship diagrammatic representation:

Relationship	Implication	Diagram
Unconstrained	Each instance of E may participate in 0 or more instances of R	
Key constraint	Each instance of E participates in at most 1 instance of R	
Total participation	Each instance of E participates in at least 1 instance of R	
Key + total participation	Each instance of E participates in exactly 1 instance of R	
Weak entity + identifying relationship	E is a weak entity set with identifying owner E' and identifying relationship set R	
ISA	S is a superclass of E1, E2, E3	
Aggregation	Works is an aggregation; arrows that connect directly to its diamond contribute primary keys to the relationship; arrows that connect directly to its box treat it like an entity	

Mapping to SQL:

ER representation	SQL representation
Key attribute	Primary key (only one)
Composite attributes	Set of single-valued columns
Multi-valued attributes	Additional table with foreign key constraint (increases queries required by 1)
Relationship set	Foreign key
Many-to-many	Relationship set becomes separate table with foreign key to participating entity sets
Many-to-one	Relationship set becomes separate table (not advised), OR Add foreign key to entity that participates at most once that references other entity
One-to-one	Dual foreign key constraint (use deferrable constraints to allow insert), OR Set referencing attribute to be UNIQUE
Key + total participation	Many-to-one with one entity containing non-NULL foreign key reference to many entity
Weak entity	Many-to-one with weak entity containing non-NULL foreign key reference to owning entity; include ON DELETE/UPDATE CASCADE; primary key is partial key + owning key
ISA relationship	One relation per superclass and subclass; subclass has PRIMARY KEY/FOREIGN KEY reference to superclass with ON DELETE CASCADE; only overlapping & covering constraints do not require triggers to implement
Aggregation	Aggregation becomes relation where primary keys and foreign keys to the entities that contribute a primary keys (forming composite primary key); relations that use the aggregation must include the composite primary key

Normal form

Definition of minimum requirements to a) reduce data redundancy, and b) improve data integrity

Redundancy: data that is unnecessarily stored multiple times

Name	NRIC	PhoneNumber	HomeAddress
Alice	1234	67899876	Jurong East
Alice	1234	83848384	Jurong East
Bob	5678	98765432	Pasir Ris

- Increases susceptibility to abnormalities
  - Updates only affect one record
  - Deletes do not properly remove all version so the data
  - Inserts could alter the repeated data accidentally

Functional dependency: $X \rightarrow Y$ if X uniquely decides Y

- For all X, the same Y can be derived
- Dependent on type of data and table
- Decomposed: RHS has only one attribute (decompose using rule of decomposition)
- Non-trivial: $X \rightarrow Y$ and $Y \not\subseteq X$ (opposite of reflexivity) (RHS doesn't appear on LHS)

Functional dependency reasoning: given a set of FDs, figure out what other FDs they can imply through Armstrong's Axioms or closures

Armstrong's Axioms:

1. Axiom of reflexivity: $T \rightarrow S$ where $S \subseteq T$
2. Axiom of augmentation: $A \rightarrow B \Rightarrow AC \rightarrow BC$ for any C
3. Axiom of transitivity: $A \rightarrow B \wedge B \rightarrow C \Rightarrow A \rightarrow C$
4. (Extension) Rule of decomposition: $A \rightarrow BC \Rightarrow A \rightarrow B \wedge A \rightarrow C$
5. (Extension) Rule of union: $A \rightarrow B \wedge A \rightarrow C \Rightarrow A \rightarrow BC$

Closures: $\{A_1,\ A_2,\ A_3, \ldots,\ A_n\}^+$ derives the set of attributes that can be decided by given attributes

Algorithm

1. Initialize the closure to given attributes
2. If there is an FD such that $X \rightarrow B$ and $X \subseteq C$ where C is the current closure, put B into the closure
3. Repeat until no more attributes can be added

- If closure of $\{X\}^+$ contains Y, then $X \rightarrow Y$ holds (inverse is true)

Deriving keys using FD: a key is a minimal set of attributes that decides all other attributes

Algorithm

1. Consider every subset of attributes of T
2. Derive the closure of each subset
3. Identify all superkeys based on closures (closure contains every attribute of T)
4. Identify all keys from superkeys

Tricks
- If the smallest possible closure is already a key, then any supersets do not need to be checked (they will be superkey)
- If attribute does not appear on RHS of all FDs, then any key must include this attribute

Non-trivial & Decomposed FD

FD whose RHS does not appear on LHS and RHS only has one attribute

Algorithm
1. Consider all attribute subsets of relation
2. Compute the closure of each subset
3. Remove the trivial attribute (RHS)
4. Derive all non-trivial and decomposed FD from each closure

BCNF (Boyce-Codd NF)

A table R is in BCNF, if every non-trivial and decomposed FD has a superkey as its LHS
- All RHS attributes can depend only on superkeys
- If RHS depends on non-superkey, then LHS can repeat which implies redundancy

Algorithm (Long)
1. Find all non-trivial and decomposed FD (from previous algorithm)
2. Check if all non-trivial and decomposed FDs satisfies the BCNF requirement
3. If all satisfies, then R is in BCNF
- Use the "more but not all" condition instead

More but not all condition: find if any non-trivial and decomposed FD is not a superkey
- If any closure of an attribute subset contains more than the initial attributes but not all attributes
- Note that FD decomposition does not need to be found

Good properties:
1. No anomalies
2. Small redundancy
3. Original table can be reconstructed from decomposed tables (lossless join)

Bad properties:
1. Dependencies may not be preserved

BCNF decomposition
- Non-unique
- Removes at least one BCNF violation every decomposition
- Recursive and multiple layers

Algorithm
1. Find a X where $X \subseteq R$ such that its closure satisfies the "more but not all" condition
2. Decompose R into R_1 and R_2 where:

a. R_1 contains all attributes in the closure
b. R_2 contains all attributes in X and those not in the closure

3. Repeat if either R_1 or R_2 is not in BCNF using projection of closures

Tricks
- If a table has only two attributes, then it must be in BCNF

Projection of closures

To find the closure of R_i which is a decomposition of R, remove all attributes that do not appear in R_i from the closures of R

Lossless join decomposition

Algorithm
1. Derive intersecting attributes of decomposition
2. Check if the closure of intersecting attributes is equals to either decomposed table (i.e. superkey of either decomposed table)

Tricks
- Check if joining all decompositions gives the full relation

Dependency preservation

Let S be the given set of FDs on the original table and S' is the set of FDs on the decomposed tables. Decomposition preserves all FDs iff S and S' are equivalent
- Show equivalence by
  - For all FDs in S', show that the closure of all LHS can be derived from S (shows S' can be derived from S) and vice versa
- Avoids inappropriate updates (lost FDs can prevent certain constraints from being enforced)
- Alternative check: $X \rightarrow Y$ is preserved in relation R if R contains all the attributes of X and Y (all $X \rightarrow Y$ must be preserved across all R)

Algorithm
1. For each decomposed relation, fin the closure of all subsets of attributes
2. Each closure contains the dependencies that are preserved
3. With all FDs found, derive the original set of FDs and see if it can be covered

Tricks
- If R contains all attributes of $X \rightarrow Y$, then $X \rightarrow Y$ is preserved

3NF

A table satisfies 3NF iff for every non-trivial and decomposed FD, either

a. LHS is a superkey, OR
b. RHS is a prime attribute (attribute that appears in a key) (all attributes of the RHS must be a prime attribute)

- 3NF is more lenient than BCNF (inverse and converse errors)
  - BCNF implies 3NF
  - Not 3NF implies not BCNF

Good properties:
1. Small redundancy
2. Lossless join property
3. Preserves all FDs

Bad properties:
1. Not as strict as BCNF
2. May have update/delete anomalies in rare cases

Algorithm
1. Derive the keys of R
2. For each given FD (non-trivial and decomposed), check criteria for 3NF above
3. If all given FDs satisfies these conditions, then R is in 3NF

3NF decomposition
- Linear and single layer

Algorithm
1. Derive a minimal basis of the set of FDs
2. In the minimal basis, combine the FDs whose LHS are the same
3. Create a table for each FD remaining
4. If none of the tables contains a key of the original table, create a table that contains a key of the original table – only 1 needs to be in 1 of the tables to skip this step
5. Remove redundant tables, i.e. those contained in other tables

Minimal basis/Minimal cover

Simplified version of the set of FDs (S)

Conditions:
1. Every FD in the minimal basis can be derived from S and vice versa
2. Every FD in the minimal basis is a non-trivial and decomposed FD
3. No FD in the minimal basis is redundant, i.e. cannot be derived from other FDs in the minimal basis
4. For each FD in the minimal basis, none of the attributes on the LHS is redundant, i.e. removing an attribute from the LHS results in a FD that cannot be derived from the original set of FDs

Algorithm
1. Decompose the FDs so that each RHS contains only one attribute
2. Remove redundant attributes on the LHS of each FD
   a. Try removing one attribute at a time and checking if the new closure contains the RHS
   b. If it does not contain, then the removed attribute is NOT redundant
   c. Otherwise, the removed attribute does not need to exist in the minimal basis FD
3. Remove redundant FDs
   a. Try removing one FD at a time and checking if the closure of the LHS of the removed FD contains the RHS
   b. If it does not, then the removed FD is NOT redundant
   c. Otherwise, the removed FD does not need to exist in the minimal basis

Tricks
- If the RHS of all FDs contain different attributes, then none of them can be removed (skip step 3)

Common Extensions

|R| = m, |S| = n, m > n > 0

Operation	Min	Max
$R \cup S$	m	m+n
$R \cap S$	0	n
$R - S$	m-n	m
$R \bowtie S$	0	mn
$R \rtimes S$	m	mn

SQL

Principle of acceptance: perform the operation if the condition evaluates to True
- Used in WHERE

Principle of rejection: reject the insertion if a condition evaluates to False
- Used in integrity constraints

x	x IS NULL	x IS NOT NULL
Not NULL	False	True
NULL	True	False

x	y	x IS DISTINCT FROM y	x IS NOT DISTINCT FROM y
NN	NN	x <> y	x = y
NN	N	True	False
N	NN	True	False
N	N	False	True

Foreign key actions:

Keyword	Action
NO ACTION	Reject delete/update if it violates constraint (default)
RESTRICT	Similar to NO ACTION except cannot be deferred
CASCADE	Propagates delete/update to the referencing tuple
SET DEFAULT	Updates foreign key of referencing tuple to some default value
SET NULL	Updates foreign key of referencing tuple to NULL

Foreign key considerations:
1. SET NULL doesn't work with prime attributes
2. SET DEFAULT does not work if there are no default values
3. CASCADE can create a chain of propagation
4. CASCADE can significantly affect overall performance

Deferrable constraints

Defers constraint checks till the end of a transaction (right before COMMIT)

- Available for UNIQUE, PRIMARY KEY, FOREIGN KEY
- Allows for cyclic foreign key constraints and overcomes constraint check performance bottlenecks
- Makes troubleshooting harder; data definition becomes ambiguous; performance penalty

```
ALTER TABLE Teams ADD CONSTRAINT
  eid_fkey FOREIGN KEY (eid)
  REFERENCES Employees (id)
  DEFERRABLE <type>;
```

Types:

1. NOT DEFERRABLE (default): cannot be deferred at all
2. INITIALLY DEFERRED: defers by default
3. INITIALLY IMMEDIATE: used with SET CONSTRAINT <constraint> DEFERRED at the top of transactions to defer

4. Equivalent subqueries:

WHERE <expr> IN <subquery>	WHERE <expr> = ANY <subquery>
WHERE <el1> <op> ANY (SELECT <e2> FROM <rel> WHERE <cond>)	WHERE EXISTS (SELECT * FROM <rel> WHERE <cond> AND <e1> <op> <e2>)

Scoping rules

1. Table alias declared in (sub-)query Q can only be used in Q or subqueries nested within Q
2. If the same table alias is declared in both inner and outer query, the declaration in the inner scope applies

Inner scope can access outer scope variables but not vice versa

Aggregation

Computes single value from a set of tuples: MIN, MAX, AVG, COUNT, SUM

Query	Interpretation
SELECT [MIN | MAX | AVG | SUM | COUNT](A) FROM R	Aggregate of non-NULL values in A; NULL if no values
SELECT COUNT(*) FROM R	Count of all rows in R; 0 if no values
SELECT [AVG | SUM | COUNT](DISTINCT A) FROM R	Aggregate of distinct non-NULL values in A

Order of clause execution

FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY > LIMIT/OFFSET

Procedures

- Behavior without return values

```
CREATE OR REPLACE PROCEDURE/FUNCTION <procedure name>
(<parameters>)
AS [(SETOF) TRIGGER/RETURN TYPE]$$
DECLARE
  <variable declaration>
BEGIN
  <function body>
END;
$$ LANGUAGE plpgsql;
```

<parameters> types:

- IN – input parameters
- OUT – output parameters (used when returning new tuples)
  - Order matters
- INOUT – input parameters that are also used as output parameters

Execute procedures: CALL <procedure name> (<arguments>);

Functions

- Behavior with specific return values

<return type> types:

- [SETOF] <table name> – returns existing tuple(s) of table
- <data type> – returns single value of data type (see common data types)
- [SETOF] RECORD – returns new tuple(s) (see OUT parameter type)
- TABLE (<columns>) – replaces OUT + SETOF RECORD return
- TRIGGER – creates a trigger function to be used on tables

Execute functions: SELECT <function name> (<arguments>);

Variable declaration

- Goes inside DECLARE block

```
DECLARE
  <variable name> <data type> [NOT NULL] [:= <initial value> | <subquery>];
BEGIN
```

- Reassignment in BEGIN must be done with walrus operator (**:=**)

Control flow

- Goes inside BEGIN block

If logic: ELSEIF can be used as many times as you wish

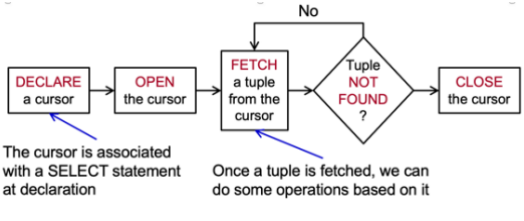
Loops:

<exit condition>: supports multiple conditions with AND and OR

- NOT FOUND – if CURSOR reaches the end of the table
- <boolean predicate> – when predicate is true, stop the loop

Cursors

- Access individual rows of a SELECT statement
- Declared in DECLARE block (along with RECORD) and used in BEGIN block



Record: represents a row of data from the SELECT table

Alternate cursor movements:

- PRIOR FROM – previous row
- FIRST FROM – first row
- LAST FROM – last row
- ABSOLUTE <n> FROM – n^{th} row

Triggers

- Type of functions (where return type is TRIGGER)
- Enforces behavior before/after INSERT and/or UPDATE and/or DELETE

Trigger

```
CREATE TRIGGER <trigger name>
<timing> <action> ON <table name>
<trigger level>
[WHEN (<trigger condition>)]
EXECUTE FUNCTION <trigger function name>();
```

<action> types: can be chained with OR; INSERT, UPDATE, DELETE

<timing> types:

- AFTER – trigger executes after operation is done (cannot override behavior of operation)
- BEFORE – trigger executes before operation is performed (can override behavior of operation)
- INSTEAD OF – trigger executes in place of operation (only for views)
  - Only on row-level trigger

<trigger level> types: primary difference seen in bulk INSERT

- FOR EACH ROW – row-level trigger that executes trigger for every tuple
- FOR EACH STATEMENT – statement-level trigger that executes trigger once per statement
  - Ignores values returned by trigger function
  - OLD/NEW not available*

<trigger condition>: only executes trigger if condition is satisfied

- Has access to OLD/NEW depending on <action>
- Cannot use SELECT
- Cannot be used with INSTEAD OF

Trigger function: note the lack of input parameters*

- Trigger functions can access:
  - TG_OP – operation (INSERT/UPDATE/DELETE) that activated the trigger
  - TG_TABLE_NAME – table name that caused trigger activation
  - OLD – previous tuple (available only when UPDATE/DELETE)
  - NEW – new tuple (available only when INSERT/UPDATE)

Return values: depending on <timing> of trigger can influence behavior of operation (assuming *t* is a non-null tuple returned)

- If row-level trigger returns NULL for a single row, subsequent triggers are omitted

Operation	BEFORE		AFTER	
	NULL	*t*	*NULL*	*t*
INSERT	No insertion	*t* inserted	No effect	No effect
UPDATE	No update	*t* for update	No effect	No effect
DELETE	No delete	Delete	No effect	No effect

INSTEAD OF trigger: behavior on view is executed on table instead

- Return NULL: ignore operation
- Return non-NULL: proceed as normal

Deferred triggers: avoids execution of trigger until end of transaction (upon COMMIT)

```
CREATE CONSTRAINT TRIGGER <trigger name>
AFTER <action> ON <table name>
DEFERRABLE <defer timing>
FOR EACH ROW
EXECUTE FUNCTION <trigger function name>();
```

- Must include CONSTRAINT and DEFERRABLE keywords
- Only works with AFTER and FOR EACH ROW

<defer timing>:

- INITIALLY DEFERRED – by default, trigger is deferred
- INITIALLY IMMEDIATE – by default, trigger is not deferred
  - Used with SET CONSTRAINTS <trigger name> DEFERRED at start of transaction to defer transaction

```
BEGIN TRANSACTION
  SET CONSTRAINTS <trigger name> DEFERRED;
COMMIT;
```

Order of trigger activation: within each category, triggers activate alphabetically

1. BEFORE statement-level
2. BEFORE row-level
3. AFTER row-level
4. AFTER statement-level

Raising messages

RAISE NOTICE: prompts with message

RAISE EXCEPTION: prompts with message and terminates operation early; used to terminate statement-level triggers

SQL injection

- Type of attacks on dynamic SQL
- SQL injected into concatenated user-provided parameters and executed

Solutions:

1. Function/procedures: strictly treats the parameters as the given type
2. Prepared statements: SQL statement is compiled when prepared and treats any inputs as constants

SQL

SQL problem solving

1. Can you find a simpler problem to solve or solve a partial problem?
   a. If so, solve the simpler/partial problems then solve the main problem
2. Deconstruct the problem
   a. What columns/attributes to find?
   b. Which relations contains the attributes?
   c. What are the conditions to be satisfied?

Create table

```
CREATE TABLE <table_name> (
  <attr1> <type1> [<column_constraint>],
  <attr2> <type2> [<column_constraint>],
  …
  <attrn> <typen> [<column_constraint>],
  [<table_constraint>], -- comment
  [<table_constraint>], /* comment */
  …
  [<table_constraint>]  -- no comma
);
```

Inserting data

```
INSERT INTO <table_name> [(attr_1, attr_2, ..., attr_n)]
  VALUES
    (<val_1_1>, <val_2_1>, ..., <val_n_1>),
    (<val_1_2>, <val_2_2>, ..., <val_n_2>),
    …
    (<val_1_m>, <val_2_m>, ..., <val_n_m>);
```

- Either all or none are inserted
- Attributes can be out of order
- Missing values are replaced with NULL (if allowed) or default values (if specified)

Deleting data

```
DELETE FROM <table_name>
  [ WHERE <condition> ];
```

- No "where" clause => WHERE TRUE by default
- Based on principle of acceptance

Principle of acceptance: perform the operation if the condition evaluates to True

- Used in WHERE

Integrity constraints

```
[column definition] [CONSTRAINT <constraint_name>] <constraint>
[<constraint arguments>]
```

Principle of rejection: reject the insertion if a condition evaluates to False

- Used in integrity constraints

x	x IS NULL	x IS NOT NULL
Not NULL	False	True
NULL	True	False

x	y	x IS DISTINCT FROM y	x IS NOT DISTINCT FROM y
NN	NN	x <> y	x = y
NN	N	True	False
N	NN	True	False
N	N	False	True

Location of specification:

- Column constraint: only applicable to single column; specified at column definition
- Table constraint: applies to one or more columns; specified after column definition

Types:

1. NOT NULL
2. UNIQUE
3. PRIMARY KEY (same as NOT NULL + UNIQUE)
4. FOREIGN KEY (columns) REFERENCES <referencing table> (columns) [actions]
5. CHECK ([check condition])
   a. Column must satisfy condition (i.e. TRUE or NULL)

Foreign key actions:

Keyword	Action
NO ACTION	Reject delete/update if it violates constraint (default)
RESTRICT	Similar to NO ACTION except cannot be deferred
CASCADE	Propagates delete/update to the referencing tuple
SET DEFAULT	Updates foreign key of referencing tuple to some default value
SET NULL	Updates foreign key of referencing tuple to NULL

Foreign key considerations:

5. SET NULL doesn't work with prime attributes
6. SET DEFAULT does not work if there are no default values
7. CASCADE can create a chain of propagation
8. CASCADE can significantly affect overall performance

Altering table

```
ALTER TABLE <table_name>
  [ALTER / ADD / DROP] [COLUMN / CONSTRAINT] <name>
  <changes>;
```

```
ALTER TABLE Projects ALTER COLUMN name TYPE VARCHAR(200);
ALTER TABLE Projects ALTER COLUMN start_year SET DEFAULT 2021;
ALTER TABLE Projects ALTER COLUMN name DROP DEFAULT;
ALTER TABLE Projects ADD COLUMN budget NUMERIC DEFAULT 0.0;
ALTER TABLE Projects DROP COLUMN budget;
ALTER TABLE Teams ADD CONSTRAINT
  eid_fkey FOREIGN KEY (eid) REFERENCES Employees (id);
ALTER TABLE Teams DROP CONSTRAINT eid_fkey;
```

Dropping table

```
DROP TABLE
  [IF EXISTS]   -- no error if table does not exist
  <table_name>[, <table_name> [, <table_name> [...]]] -- multiple table
  [CASCADE];    -- also delete referencing table
```

Deferrable constraints

Defers constraint checks till the end of a transaction (right before COMMIT)

- Available for UNIQUE, PRIMARY KEY, FOREIGN KEY
- Allows for cyclic foreign key constraints and overcomes constraint check performance bottlenecks
- Makes troubleshooting harder; data definition becomes ambiguous; performance penalty

```
ALTER TABLE Teams ADD CONSTRAINT
  eid_fkey FOREIGN KEY (eid)
  REFERENCES Employees (id)
  DEFERRABLE <type>;
```

Types:

5. NOT DEFERRABLE (default): cannot be deferred at all
6. INITIALLY DEFERRED: defers by default
7. INITIALLY IMMEDIATE: used with SET CONSTRAINT <constraint> DEFERRED at the top of transactions to defer

Selection

```
SELECT [DISTINCT] <target-list>
FROM    <relation-list>
WHERE <conditions>
ORDER BY <attribute> [(ASC) | DESC]
OFFSET <g>
LIMIT <k>
```

- Duplicates are not removed by default so DISTINCT is used to do so
  - Using IS DISTINCT FROM to do so
- <target-list> as * includes all attributes

Pattern matching: <column> [NOT] LIKE <pattern matching>

- _ -> any single character
- % -> any sequence of 0 or more characters

Ordering: rows are unsorted by default

- Default ordering if specified is ASC
- Sorting by multiple attributes specified in order

Ranking:

- LIMIT k: returns first k rows
- OFFSET g: index of first row (starts with 1)

Operators

Mathematical	+, *, -, %, /, ^, |/ (square root)
String	|| (concatenate), LOWER(s), UPPER(s)
Date time	+, NOW()

Set operations

Only works on union-compatible tables

Eliminate duplicates	Keep duplicates
UNION	UNION ALL
INTERSECT	INTERSECT ALL
EXCEPT	EXCEPT ALL

Join operations

```
SELECT <target-list>
FROM <relation 1> AS <alias>, <relation 2> AS <alias>
WHERE <selection criteria>
```

```
SELECT <target-list>
FROM <relation 1> AS <alias>
  [INNER | LEFT | RIGHT | FULL | NATURAL] JOIN <relation 2> AS <alias>
    ON <selection criteria>
```

- Table name (after renaming) can only appear at most once from the FROM clause

Subqueries

Scalar subquery: query that returns at most a single value; dynamically checked at runtime; can be used as a value in queries; no rows returned = NULL

[NOT] IN:

```
WHERE <expr> [NOT] IN [<subquery> | <tuple>]
```

- <tuple> form: (value, value, value, …)
- Subquery MUST be return just one column
- IN: TRUE if <expr> matches any subquery row; can be replaced with inner joins
- NOT IN: TRUE if <expr> matches no subquery row; can be replaced with outer joins

ANY:

```
WHERE <expr> <op> ANY <subquery>
```

- Subquery MUST return exactly one column
- <expr> compared to each row from subquery using <op>
- TRUE if comparison evaluates to TRUE for at least one row in the subquery
  - No row => FALSE

ALL:

```
WHERE <expr> <op> ALL <subquery>
```

- Subquery MUST return exactly one column
- <expr> compared to each row from subquery using <op>

- TRUE if comparison evaluates to TRUE for all rows in the subquery
  - No row => TRUE

[NOT] EXISTS:

```
WHERE [NOT] EXISTS <subquery>
```

- Subquery may return multiple columns; convention to return just value of 1
- EXISTS: TRUE if subquery returns at least one row
- NOT EXISTS: TRUE if subquery returns no rows

Equivalent subqueries:

WHERE <expr> IN <subquery>	WHERE <expr> = ANY <subquery>
WHERE <el1> <op> ANY (SELECT <e2> FROM <rel> WHERE <cond>)	WHERE EXISTS (SELECT * FROM <rel> WHERE <cond> AND <e1> <op> <e2>)

Scoping rules

3. Table alias declared in (sub-)query Q can only be used in Q or subqueries nested within Q
4. If the same table alias is declared in both inner and outer query, the declaration in the inner scope applies
5. Inner scope can access outer scope variables but not vice versa

Aggregation

Computes single value from a set of tuples: MIN, MAX, AVG, COUNT, SUM

Query	Interpretation
SELECT [MIN | MAX | AVG | SUM | COUNT](A) FROM R	Aggregate of non-NULL values in A; NULL if no values
SELECT COUNT(*) FROM R	Count of all rows in R; 0 if no values
SELECT [AVG | SUM | COUNT](DISTINCT A) FROM R	Aggregate of distinct non-NULL values in A

Grouping

```
GROUP BY attr1, attr2, …
HAVING <condition>
```

- Aggregation applied over each group

Restrictions to SELECT clause: if column C of table R appears in SELECT clause with GROUP BY, then

1. C appears in GROUP BY clause, OR
2. C appears as input of an aggregation function in the SELECT clause, OR
3. Primary key of R appears in the GROUP BY

HAVING: condition checks for each group in GROUP BY; must be used with GROUP BY; involves aggregate functions

- Similar restrictions as GROUP BY

Order of clause execution

FROM > WHERE > GROUP BY > HAVING > SELECT > ORDER BY > LIMIT/OFFSET

Conditional expressions

CASE: like switch statements

```
CASE
  WHEN <condition/value 1> THEN <result 1>
  WHEN <condition/value 2> THEN <result 2>
  …
  ELSE <default>
END
```

- ELSE is optional: defaults to NULL if no ELSE
- Condition can contain aggregate functions

COALESCE: Returns first non-NULL value or NULL if all values are NULL

```
COALESCE(<value 1>, <value 2>, …, <value n>)
```

NULLIF: returns NULL if value 1 = value 2, else return value 1

```
NULLIF(<value 1>, <value 2>)
```

Common Table Expressions (CTEs)

Compute a temporary table for a query

```
WITH
  CTE_1 (<attr>, <attr>, …, <attr>) AS (Q_1),
  CTE_2 AS (Q_2),
  …,
  CTE_n AS (Q_n)
-- main SELECT
```

- Each CTE can access the ones that came before it

Views

Permanently named query (aka virtual table) where computation to retrieve table is done every time the table is accessed

```
CREATE VIEW <name> (<attr>, <attr>, …, <attr>) AS
  <query>
;
```

- Allows only parts of a table to be selected or accessed
- No restrictions for SELECT statements
- Restrictions for INSERT/UPDATE/DELETE
  - Only one entry in FROM
  - No WITH, DISTINCT, GROUP BY, HAVING, LIMIT, OFFSET
  - No UNION, INTERSECT, EXCEPT, ALL
  - No aggregate function

Recursive queries

```
WITH RECURSIVE
  CTE_name AS (
    Q_1  -- base case
    UNION [ALL]
    Q_2 (CTE_name)
  )
Q_0 (CTE_name)
```

- Query stops evaluating when no more rows added

Universal quantification

$$\forall x: Exists(x) \Leftrightarrow \neg\neg\forall x: Exists(x) \Leftrightarrow \neg\exists x: \neg Exists(x)$$

PSM

Dynamic SQL: SQL statement to be executed is not fixed; stored in string variable, compiled, and executed at runtime.

Advantages of functions/procedures: code reuse; ease of maintenance; performance; security

Procedures

- Behavior without return values

```
CREATE OR REPLACE PROCEDURE <procedure name> (<parameters>)
AS $$
DECLARE
  <variable declaration>
BEGIN
  <function body>
END;
$$ LANGUAGE plpgsql;
```

<parameters> types:

- IN – input parameters
- OUT – output parameters (used when returning new tuples)
  - Order matters
- INOUT – input parameters that are also used as output parameters

Execute procedures:

```
CALL <procedure name> (<arguments>);
```

Functions

- Behavior with specific return values

```
CREATE OR REPLACE FUNCTION <function name> (<parameters>)
RETURNS <return type> AS $$
DECLARE
  <variable declaration>
BEGIN
  <function body>
END;
$$ LANGUAGE plpgsql;
```

<return type> types:

- [SETOF] <table name> – returns existing tuple(s) of table
- <data type> – returns single value of data type (see common data types)
- [SETOF] RECORD – returns new tuple(s) (see OUT parameter type)
- TABLE (<columns>) – replaces OUT + SETOF RECORD return
- TRIGGER – creates a trigger function to be used on tables

Execute functions:

```
SELECT <function name> (<arguments>);
```

Common data types:

Type	Alias	Remarks
boolean	bool	
character(n)	char(n)	n: length of string
integer	int	
numeric(t, d)	decimal(t, d)	t: total number of digits d: max decimal places (rounding to occur)
text	-	
timestamp	timestamptz	
uuid	-	Use with DEFAULT gen_random_uuid()
serial	-	Autoincrementing integer

Variable declaration

- Goes inside DECLARE block

```
DECLARE
  <variable name> <data type> [NOT NULL] [:= <initial value> | <subquery>];
BEGIN
```

- Reassignment in BEGIN must be done with walrus operator (**:=**)
- SELECT <column> INTO <variable> used to assignment variables from within SELECT statements

Control flow

- Goes inside BEGIN block

If logic: ELSEIF can be used as many times as you wish

```
BEGIN
  IF <condition1> THEN
    <truthy behavior>
  [ELSEIF <condition2> THEN]
    <truthy behavior>
  [ELSE]
    <falsey behavior>
  END IF;
END;
```

Loops:

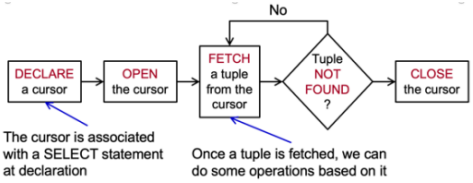
```
BEGIN
  LOOP
    EXIT WHEN <exit condition>;
    <loop behavior>
  END LOOP;
END;
```

<exit condition>: supports multiple conditions with AND and OR

- NOT FOUND – if CURSOR reaches the end of the table
- <boolean predicate> – when predicate is true, stop the loop

Cursors

- Access individual rows of a SELECT statement



- Declared in DECLARE block (along with RECORD) and used in BEGIN block

```
DECLARE
  <cursor> CURSOR FOR (<SELECT>);
  <record> RECORD;
BEGIN
  OPEN <cursor>;
  LOOP
    FETCH [<movement>] <cursor> INTO <record>;
    EXIT WHEN NOT FOUND;
    <loop body>
  END LOOP;
  CLOSE <cursor>;
END;
```

Record: represents a row of data from the SELECT table

Alternate cursor movements:

- PRIOR FROM – previous row
- FIRST FROM – first row
- LAST FROM – last row
- ABSOLUTE <n> FROM – n^{th} row

Triggers

- Type of functions (where return type is TRIGGER)
- Enforces behavior before/after INSERT and/or UPDATE and/or DELETE

Trigger

```
CREATE TRIGGER <trigger name>
<timing> <action> ON <table name>
<trigger level>
[WHEN (<trigger condition>)]
EXECUTE FUNCTION <trigger function name>();
```

<action> types: can be chained with OR

- INSERT
- UPDATE
- DELETE

<timing> types:

- AFTER – trigger executes after operation is done (cannot override behavior of operation)
- BEFORE – trigger executes before operation is performed (can override behavior of operation)
- INSTEAD OF – trigger executes in place of operation (only for views)
  - Only on row-level trigger

<trigger level> types: primary difference seen in bulk INSERT

- FOR EACH ROW – row-level trigger that executes trigger for every tuple
- FOR EACH STATEMENT – statement-level trigger that executes trigger once per statement
  - Ignores values returned by trigger function
  - OLD/NEW not available*

<trigger condition>: only executes trigger if condition is satisfied

- Has access to OLD/NEW depending on <action>
- Cannot use SELECT
- Cannot be used with INSTEAD OF

Trigger function: note the lack of input parameters*

```
CREATE OR REPLACE FUNCTION <trigger function name>()
RETURNS TRIGGER AS $$
DECLARE
  <variable declaration>
BEGIN
  <trigger body>
END;
$$ LANGUAGE plpgsql;
```

- Trigger functions can access:
  - TG_OP – operation (INSERT/UPDATE/DELETE) that activated the trigger
  - TG_TABLE_NAME – table name that caused trigger activation
  - OLD – previous tuple (available only when UPDATE/DELETE)
  - NEW – new tuple (available only when INSERT/UPDATE)

Return values: depending on <timing> of trigger can influence behavior of operation (assuming *t* is a non-null tuple returned)

- If row-level trigger returns NULL for a single row, subsequent triggers are omitted

Operation	BEFORE		AFTER	
	NULL	*t*	*NULL*	*t*
INSERT	No insertion	*t* inserted	No effect	No effect
UPDATE	No update	*t* for update	No effect	No effect
DELETE	No delete	Delete	No effect	No effect

INSTEAD OF trigger: behavior on view is executed on table instead

- Return NULL: ignore operation
- Return non-NULL: proceed as normal

Deferred triggers: avoids execution of trigger until end of transaction (upon COMMIT)

```
CREATE CONSTRAINT TRIGGER <trigger name>
AFTER <action> ON <table name>
DEFERRABLE <defer timing>
FOR EACH ROW
EXECUTE FUNCTION <trigger function name>();
```

- Must include CONSTRAINT and DEFERRABLE keywords
- Only works with AFTER and FOR EACH ROW

<defer timing>:

- INITIALLY DEFERRED – by default, trigger is deferred
- INITIALLY IMMEDIATE – by default, trigger is not deferred
  - Used with SET CONSTRAINTS <trigger name> DEFERRED at start of transaction to defer transaction

```
BEGIN TRANSACTION
  SET CONSTRAINTS <trigger name> DEFERRED;
COMMIT;
```

Order of trigger activation: within each category, triggers activate alphabetically

5. BEFORE statement-level
6. BEFORE row-level
7. AFTER row-level
8. AFTER statement-level

Raising messages

RAISE NOTICE: prompts with message

RAISE EXCEPTION: prompts with message and terminates operation early

- Used to terminate statement-level triggers

SQL injection

- Type of attacks on dynamic SQL
- SQL injected into concatenated user-provided parameters and executed

Solutions:

3. Function/procedures: strictly treats the parameters as the given type
4. Prepared statements: SQL statement is compiled when prepared and treats any inputs as constants