Transactions: start with **begin**, end with **commit** or **rollback**. Can contain multiple sql stmt.Consider a transaction T: (ACID)

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

**Relational Model:**

**Foreign key**: a subset of attributes of relation R1 that refers to the primary key of R2. (R1: referencing relation; R2: referenced relation)

**Requirement**: each foreign key in R1 must satisfy one of the following: a. Appear as primary key in R2; b. NULL or a tuple containing at least one NULL value. (even if the value never appears in R2, as long as one of other irrelevant attributes in that tuple is NULL)

**Note**: The referencing relation and the referenced relation can be the same relation. (e.g., Employees.manager->Employs.id)

**Relational Algebra:(Relation: Set of tuples)**

1.**Three-valued logic**: False ^ NULL = False; False V NULL = NULL; True ^ NULL = NULL; True V NULL = True;﹃NULL = NULL; Null = Null: Null; Null <> Null: Null; Null ≡ Null: True; v1 ≡ Null: False; v1 ≡ v2: same as v1 = v2

2.**Selection**:**σ**[c](R). Result have same schema. Usually number of rows are smaller. Note: c must specify only attributes in R.

3.**Projection**:**π**[**l**](R), **l** is an ordered list of attributes. Not allow operations and duplicate in **l**. The number of rows may be smaller: **relation is defined as set of tuples**. (**Remove duplicates**). Note: l must specify only attributes in R

4.**Renaming**:**ρ**[r](R), r is a unordered collection of Bi <- Ai (new <- old). Renaming will be relevant for set and join operations. Column orders and number of row remain the same.

**Note:** **No two different attributes may be renamed to the same name; No attributes may be renamed twice in a single operation.**

5.R and S are **Union-compatible**: R and S have same number of attributes AND corresponding attributes have same or compatible domains. (INT and TEXT are not compatible)

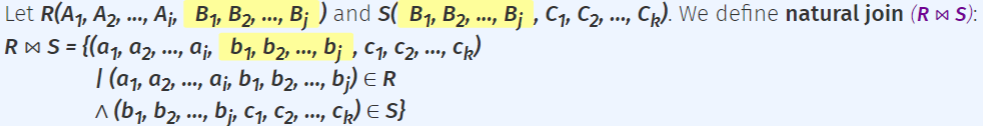
6.**Cross product**: R×S= {(a1, a2, ..., an, b1, b2, ..., bn) | (a1, a2, ..., an)∈R ∧ (b1, b2, ..., bn)∈S}. **Note:** The set of attributes in R and S must be disjoint (*Attr*(R) ∩ *Attr*(S) = ∅)

7.**Inner joins**:

i. The**θ-join** (R ⋈[θ] S) of two relations R and S is defined as: R ⋈[θ] S =σ[θ](R × S)

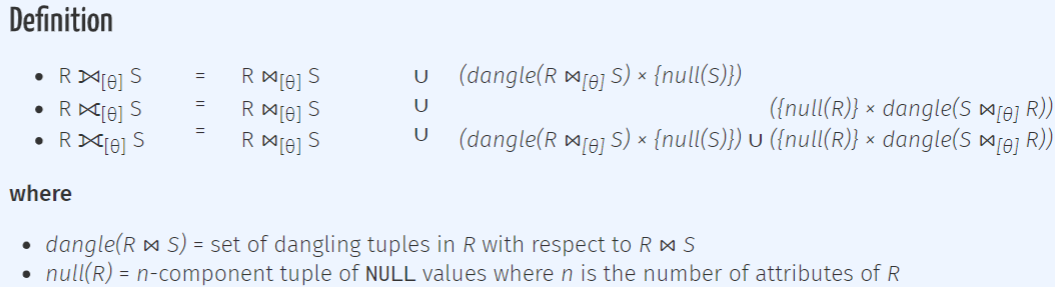
ii. **Equi join** (R ⋈= S) is a specialθ-join where the only relational operator that can be used is equality (e.g., = or≡)

iii. **Natural join: (R NATURAL JOIN 空 = 空)**



Alternatively, R ⋈ S = π[ℓ](R ⋈[θ] S), where:  
ℓ = Attr(R) ∪ (Attr(S) - Attr(R))  
θ = ∀ Ai ∈ Attr(R) ∩ Attr(S): RAi = SAi

**8.Outer joins**:



9.**Natural outer join**: θ is equality operations

10.**Strong equivalence:**

πℓ1(πℓ2(R)) ≢ πℓ1(R) (unless ℓ1 ⊆ ℓ2)

π[ℓ](σ[θ](R)) ≢ σ[θ](π[ℓ](R)) (unless θ uses only attributes in ℓ)

σ[θ](R × S) ≢ σ[θ](R) × S (unless θ uses only attributes in R)

R × S ≢ S × R (different column order)

R ⋈ S ≢ S ⋈ R (i.e., non-commutative)

R ⋈[θ1] (S ⋈[θ2] T) ≢ (R ⋈[θ1] S) ⋈[θ2] T (if θ1 uses T and θ2 uses R)

**SQL Part 1:**

1.**CREATE** table Employees (

id INT DEFAULT 0, name VARCHAR(50), time DATE, role TEXT);

2.**INSERT** into Employees values (100, ‘Sarah’, ...) / insert into Employees (name, id) values (‘Sarah’, 100); (Missing values are replaced with Null)

3.**DELETE** from Employees where role <> ‘dev’; (Perform the operation if the condition in WHERE clause evaluates to true)

4.**UPDATE** Employees set id = 1, role = ‘work’ where name = ‘Sarah’;

5.**DROP** table is exists Employees;

6.x IS NULL: x ≡ NULL; x IS NOT NULL: x ≡/ NULL;

7.x IS DISTINCT FROM y: x ≡/ y; x IS NOT DISTINCT FROM y: x ≡ y;

**Integrity Constraints:** **Reject** the insertion if condition **evaluates to False (accept evaluates to True, NULL)**

8.**NOT NULL** constraint: id INT not null; id INT CONSTRAINT nn\_id NOT NULL. Condition: IS NOT NULL

9.**UNIQUE** constraint: id INT unique; unique(id, name) -- at bottom. Condition: x.Ai <> y.Ai

(100, NULL), (100, NULL) are consider UNIQUE: NULL<>NULL = NULL

10.**Primary key** constraint: id INT primary key; primary key(id, name) -- at bottom. Condition: UNIQUE and NOT NULL.

11.**Foreign key constraint**: id INT references Workers (id); foreign key (id, name) references Workers(id, name) -- at bottom. Satisfy if: tuple in Employees are also in Workers, or one of referencing columns is NULL.

ON DELETE NO ACTION ON UPDATE CASCADE. action:

No action: Reject delete/update if it violates constraint (default)

Restrict: similar to “no action” except that check of constraint cannot be defer

Cascade: propagates delete/update to the referencing tuples

Set default: updates the foreign key of the referencing tuples to some default value (default value must be a primary key in the referencing table)

Set null: updates the foreign key of the referencing tuples to NULL. (corresponding column must be allowed to contain NULL values)

12.**Check** constraint: id INT check (id > 0): scope is one row;

check (start\_year <= end\_year) -- at bottom: scope is one table

Condition: Check(c), c does not evaluate to False

13.**ALTER** table Employees alter column id drop default;

alter table Employees alter column name type TEXT;

alter table Employees add column salary NUMERIC DEFAULT 0.0;

alter table Employees add constraint table\_pk primary key(id, name);

14.**Deferrable** constraint: unique, primary key, foreign key constraint can be deferred using “deferrable initially deferred”. Or “deferrable initially immediate”, with “set constraints fkey deferred” in transactions

**SQL Part 2:**

1.SELECT DISTINCT a1, a2,..., am FROM r1, r2,...,rn WHERE c

=> Relational Algebra: π[a1, a2, ..., am](σ[c](r1 × r2 × ... × rn))

2.**No distinct (in default)**. To check: IS DISTINCT FROM

3.**SELECT** Clause:

mathematical: +, \*, -, %, /, ^, |/, etc; **String**: || (concatenate), LOWER(s),  
UPPER(s), etc; Date time: +, NOW()  
Renaming: column **AS** alias

Select rname, pizza, ‘S$’ || (price \* 1.36) AS sgd From sells;

4.**WHERE** Clause:

Where area IS NULL: area ≡ NULL; area = NULL: NULL;

**[NOT] LIKE**: \_ matches any single character; % matches any sequence of 0 or more characters; Where pizza LIKE ‘Ma%a’;

5.**Set operations**: (UNION, INTERSECT, EXCEPT [ALL])

union, intersect, except are same as ∪∩－. Eliminates duplicates.

union all, intersect all, except all preserve duplicate records. (each element treat as distinct element)

7.**Outer Join**: from Customers C LEFT JOIN Likes L ON C.cname = L.cname WHERE L.pizza **IS NULL**;

8.**Scalar subqueries**: a query that returns at most a single value. (i.e., a table with 1 row and 1 column)

If the result returns 0 row, it is treated as NULL.

9.**[NOT] IN**: where <expr> IN <subquery>;

IN returns True if ∃row ∈ <subquery> : row = <expr>

10.**ANY**: where <expr> <op> ANY <subquery>;

∃row ∈ subquery : (<expr> <op> row) = TRUE

11.**ALL**: where <expr> <op> ALL <subquer>;

∀ row ∈ subquery : (<expr> <op> row) = TRUE

12.**[NOT] EXISTS**: where [NOT] EXISTS <subquery>

∃row ∈ subquery : TRUE

13.**Scoping rule**: a table alias declared in a (sub)query Q can only be used in Q, or subqueries nested within Q.

If the same table alias is declared in both subquery Q1 and outer query Q0, the declaration in Q1 is applied.

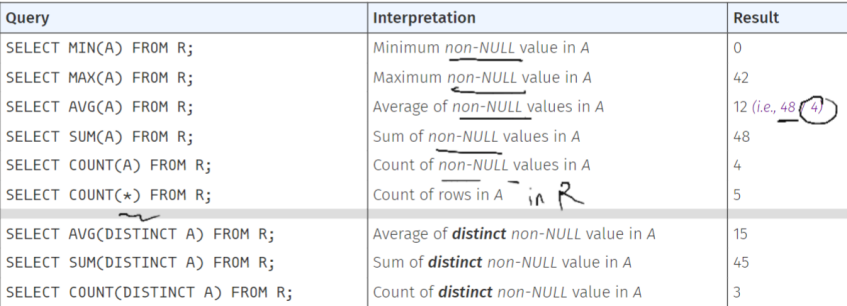
14.**Order by**: order by <attribute> ASC/DESC

15.**Limit**: limit 3; show top 3

16.**Offset**: offset 3, show 4th and onwards

**SQL Part 3:**

1.**Aggregate function**: cannot be used in WHERE



min, max, avg, sum(NULL) = NULL

2.**GROUP BY**: use “is not distinct from” to check. column Ai of table R appears in SELECT, one of following **conditions** must hold:

1. Ai appears in the GROUP BY clause
2. Ai appears as input of an aggregation function in SELECT
3. The primary key of R appears in the GROUP BY

3.**HAVING**: column Ai appears in HAVING, **one condition** must hold:

i. Ai appears in GROUP BY

ii. Ai appears as input of an aggregation function in HAVING

iii. The primary key of R appears in GROUP BY

4.**Conceptional evaluation of query**:

i. Compute cross-product of tables in from-list. ii. Select tuples that evaluate to true for where-condition. iii. Partition selected tuples into groups using groupby-list. iv. Select the groups that evaluate to true for the having-condition. v.For each selected group, generate an output tuple based on select-list. vi. Remove duplicate output tuples because of distinct. vii. Sort the output tuples based on orderby-list. viii. Remove the appropriate output tuples based on offset-specification and limit-specification.

5.**CASE** expression:

case case grade

when marks >= 70 then ‘A’ when ‘A’ then 70

else ‘D’ else 0

end as grade end as marks

6.**COALESCE**(v1, v2, v3): Returns the **first** non-null value in the list; Returns Null if all values in the list are Null.

7.**NULLIF**(v1, v2): Returns Null if v1 = v2, otherwise returns v1.

NULLIF(1, 1): NULL; NULLIF(2, 1): 2

8.**Table expressions**: with CTE\_1 AS (Q\_1), CTE\_2 AS (Q\_2) SELECT...

each CTE\_i can reference any other CTE\_j that declared before CTE\_i

WITH PizzaWithCheese(pizza) AS (select pizza from Recipes) SELECT...

9.**Views**: virtual relation that be used for querying.

create or replace view v3 (rname1, rname2) as SELECT... ;

This creates an interface, where users/applications query using the schema of the view, thus safe from changes to the actual logical/physical schema.

**Functions in SQL:**

1. CREATE OR REPLACE **FUNCTION** convert (mark INT) RETURNS char(1) AS $$

SELECT ...

$$ LANGUAGE sql;

2.IN: input a value; OUT: return a value; INOUT: pass in an initial value, return updated value back.

3.RETURNS: i. **tables**: return only one tuple of that table; ii. **SETOF tables**: return multiple tuples of that table; iii. **RECORD**: return one tuple (composite values, not in the table) iv. **SETOF RECORD**; v. **TABLE(Mark INT, id INT)**: same as using SETOF RECORD & OUT parameters.

4.If no return value/tuple is needed, use **SQL procedures**:

CREATE OR REPLACE **PROCEDURE** transfer(fromAcct TEXT, toAcct TEXT, amount INT) AS $$

To execute procedure: **CALL** transfer(‘Alice’, ‘Bob’, 100);

5.**Variables and Control Structures**:

as $$ IF condition1 THEN

DECLARE stmt1;

temp\_val integer; ELSEIF condition2 THEN

BEGIN stmt2;

-- function body ELSE

END; else-stmt;

$$ LANGUAGE plpgsql; END IF;

LOOP

EXIT WHEN NOT FOUND (condition)

stmts;

END LOOP;

6.**Cursor:**

DELCARE

curs CURSOR FOR (SELECT \* FROM Scores ORDER BY A DESC);

r RECORD;

BEGIN

OPEN curs;

LOOP

FETCH curs INTO r; //read the next tuple from curs, put it into r

EXIT WHEN NOT FOUND; //**Update values before exit if last tuple has also to be considered or updated.**

-- assign output table column names to values

**RETURN NEXT**; //inserts a tuple (name, mark) to the output of the function.

prv\_mark **:=** r.Mark; //Update value. **Use := to assign value**

END LOOP;

CLOSE curs; //Releases the resources allocated to curs.

END;

FETCH PRIOR FROM cur INTO r; FETCH FIRST (LAST) FROM cur INTO r; FETCH ABSOLUTE 3 FROM cur INTO r //Fetch the 3rd tuple.

FETCH ... FROM curs INTO r;

**RETURN NEXT r**; //return the curr Record r (不连续的curr, return the record r)

**Triggers in SQL:**

1.create or replace function scores\_func() RETURNS **TRIGGER** AS $$

BEGIN

END; $$ LANGUAGE plpgsql;

create **TRIGGER** trigger\_scores [before/after] [insert/update/delete] on R for each [row/statement] execute function scores\_func();

2.**TG\_OP:** operations: “INSERT”, “UPDATE”, “DELETE”

**TG\_TABLE\_NAME**: name of table that caused the trigger invocation.

**NEW:** the NEW tuple being **INSERT or UPDATE**. **NULL for deletion.**

Update: NEW:将要update的值; OLD:要被update的tuple

**OLD:** the OLD tuple being **UPDATE or DELETE. NULL for insertion.**

3.**Return Values for Row-Level Triggers:**

**before insert:** non-null t: inserted; null: no insertion.

**before update**: non-null t: updated; null: no update.

**before delete**: non-null t: deletion; null: no deletion.

**after** insert/update/delete: **return value doesn’t matter**.

4.**INSTEAD OF Trigger**: (**Only** **Row-Level**)

Defined on views only. Instead of doing smth on a view, do it on a table. When ppl want to update view, we update the corresponding tuple in the table.

-- non-null t: proceed; null: ignore operations on current row.

5.If a **before row-level** trigger returns null, all subsequent triggers on the same row are omitted.

6.**Return Values for Statement-Level Triggers**:

i.One statement might change multiple rows.

ii.Return values are **ignored**. (RETURN NULL doesn’t affect statement-level trigger. Operations still done well)

iii.RAISE NOTICE. Raise EXCEPTION to ignore all operation.

7.**Trigger condition:**

for each row when (NEW.name = ‘Hello’) execute function f();

No ‘select’ in ‘when’; No OLD in ‘when’ for INSERT; No NEW in ‘when’ for DELETE; No ‘when’ for INSTEAD OF.

**8.Deferred Triggers:**

i. create **constraint** trigger trigger\_name **after**... on R

ii. **deferrable initially [deferred/immediate] for each row**...

iii. Only works for **after** and **for each row**

iv. If initially immediate, we need to change on the fly.

begin transaction; **set constraints** trigger\_name **deferred**; commit; //trigger will be activated at “COMMIT”

**FD: (only know instance of table, cannot say any FD definitely holds**

1.Anomalies: Update/Delete/Insert

2.We say that A1A2...Am -> B1B2...Bn, A decides B, if:  
Whenever two objects have the same values on A1, A2,..., and Am, they always have the same values on B1, B2, ..., Bn

3.**Requirement to FD implied**: A -> B

i. Determine attributes

ii. find counter-examples, two tuples

iii. 两个一样的放左边, 不一样的放右边

4.**Armstrong’s Axioms:**

i. Reflexivity: ABC -> A (set to subset)

ii. Augmentation: If A -> B, then AC -> BC for any C

iii. Transitivity: If A -> B and B -> C, then A -> C

Extended Axioms:

1. Decomposition: If A -> BC, then A -> B and A -> C
2. Union: If A -> B and A -> C, then A -> BC
3. AC -> BC **cannot get** A -> B

5.Trivial: {A, B} -> {A}

Non-Trivial: {A, B} -> {A, C} //B不是A的子集

Completely non-trivial: {A} -> {B} //A∩B=**∅**, B非空

1. (1) {A, B, C} → {E} [Given]  
   (2) {B, D} → {A} [Given]  
   (3) {C, F } → {B} [Given]  
   (4) {C, D, F } → {B, D} [Augmentation of (3) with {D}]  
   (5) {C, D, F } → {A} [Transitivity of (4) and (2)]  
   (6) {C, D, F } → {C, F } [Reflexivity]  
   (7) {C, D, F } → {B} [Transitivity of (6) and (3)]  
   (8) {C, D, F } → {B, C, D, F} [Augmentation of (7) with {C, D, F }]  
   (9) {B, C, D, F } → {A, B} [Augmentation of (5) with {B}]  
   (10) {C, D, F } → {A, B} [Transitivity of (8) and (9)]  
   (11) {C, D, F } → {A, B, C} [Augmentation of (10) with {C}]
2. {C, D, F } → {E} [Transitivity of (11) and (1)]

**7.Closures:**

{A}+ denotes the set of attributes that can be directly or indirectly decided by A, called closure of A.

To **compute closure** of A:

1. Initialize the closure to {A1,A2...An,B}. ii. If there is an FD: Ai,Aj...Am->B, such that Ai,Aj...Am are all in the closure, then put B into closure. iii. repeat step 2 until cannot find any new attribute to put into.

To prove X -> Y, just need to show {X}+ contains Y. Opposite is true.

8.SuperKey: a set of attributes in a table that decides all other attribute. Key: a superkey that is minimal. **Prime attributes**: attributes that appears in a key.

9.**Find keys**: compute closures which contain all the attributes; select minimal closure.

Trick1: Always check small attribute sets first

Trick2: If an attribute not appear in RHS of any FD, then it must in every key.

**BCNF: (lossless join; no FD preservation)**

1.Decomposed FD: an FD whose RHS has only one attribute

2.**Non-trivial and decomposed FD**: a decomposed FD whose RHS doesn’t appear in LHS.

3.**BCNF def**: every non-trivial and decomposed FD has a superkey as LHS.

4.**BCNF Check:** violate iff a closure satisfies “**more but not all**”

a closure {A1,A2,...,Ak}+ contains some attributes not in {A1,A2,...Ak}, but does not contain all attributes in the table.

**5.BCNF Decompose Algo: (Binary Split)**

i.Find a subset X satisfy “more but not all”

ii.Decompose R into R1, R2 such that: R1 contains all attributes in {X}+; R2 contains all X and all attributes not in {X}+.

iii.If R1 or R2 not in BCNF, further decompose it.

Note: if a table has only two attributes, then it must be in BCNF

**6.Project FD of R to R1 or R2:**

i.Derive closures of subset of attributes in **R2** on **table R**

ii.Remove irrelevant attributes in each closure

7.**Lossless Join Decomposition: (aslongas one way of compos LJ)**

Decomposition guarantees lossless join, whenever the common attributes in R1 and R2 constitute a superkey of R1 or R2.

eg. R(A,B,C,D) decomposed into R1(A,B,C) and R2(B,C,D) with BC being a superkey of R1.

**3NF: (lossless join + preserve all FDs) (3NF contains BCNF)**

1.**Decomposition** **preserves** all FDs, iff S and S’ are **equivalent:**

i.Every FD in S’ can be derived from S;

ii. Every FD in S can be derived from S’

**Equivalent Check Algo**: compute each dependency’s LHS attributes’ closure based on the other set, see if that dependency’s RHS attributes are in the closure.

2.**3NF def:** every non-trivial and decomposed FD, LHS is a superkey or RHS is a prime attribute. (BCNF + RHS prime attribute)

3.**3NF check:** For each given FD, check if: LHS is a superkey, or, each attribute on RHS is a prime attribute. **(Check on original FDs. Don’t check on minimal basis!)**

**4.Minimal basis (cover) conditions: (Not Unique)**

i.Every FD in the minimal basis can be derived from S, vice versa.(Equivalent)

ii.Every FD in the minimal basis is a non-trival and decomposed FD.

iii.No FD in the minimal basis is redundant. (No FD can be derived from other FDs in the minimal basis) (All FDs in S **except** curr potential redundant FD)

iv.For each FD, none of attributes on LHS is redundant. (If remove an attribute from LHS, resulting FD is a new FD that cannot be derived from **original set, including potential redundant attribute**)

5.**Minimal basis Algo:**

i.Transfer FDs such that each RHS contains only one attribute.

ii.Remove redundant attributes on LHS of each FD.

iii.Remove redundant FDs.

6.**3NF Decomposition Algo: (Only one split)**

i.Find a minimal basis of FDs

ii.Combine FDs whose LHS are the same.

iii.For each FD, construct a table that contains all attributes in the FD

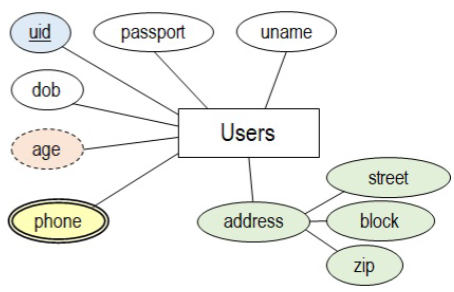
iv.Check if any of tables contain a key for R; if not, create a table that contains a key for R. (**to ensure lossless join**)

v.Remove redundant tables (remove R’, if all attributes of R’ are in another R’’)

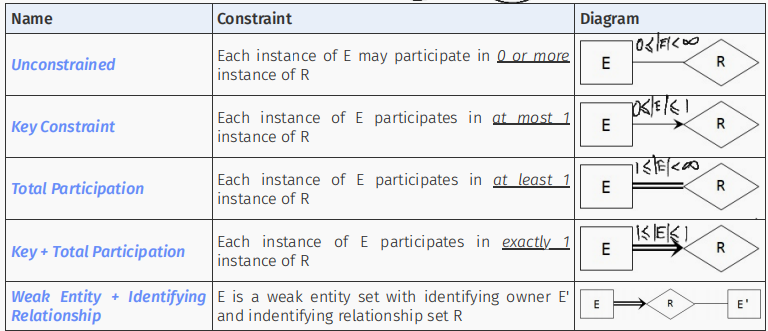
**ER Diagram:**

**1.Entities:** rectangle; **Relationships:** diamond

2.**Attributes: Key Attributes (underlined):** uniquely identifies each identity; **Composite Attributes (composed of other ovals)**; **Multiple Attributes (double-lined):** one or more values for a given entity; **Derived Attributes (dashed line)**: derived from other attributes.



3.Degree of Relationship Sets: number of entity sets connected. n-ary = n entity.



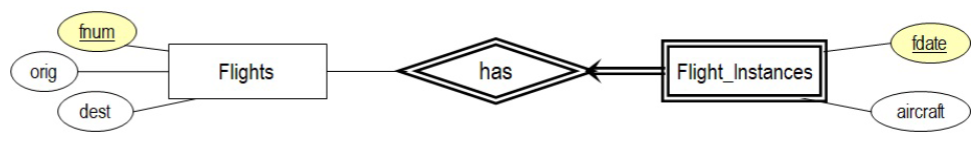
4.Weak Entity Set:

i.Entity set that does **NOT** have its own key. Its own key is called **partial key.** It uniquely identify a weak entity for a **given owner entity.**

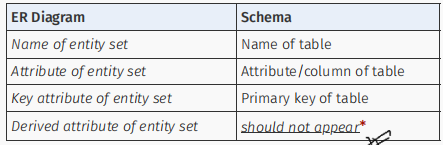
ii.cannot uniquely identify entity; can only uniquely identify with the help of primary key from **owner entity.**

iii.Its existence depends on the existence of its owner entity.

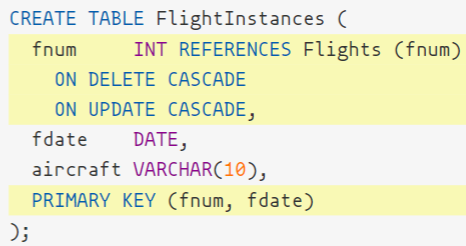
iv.Weak entity set and identifying relation set are represented via double-lined rectangles / diamonds.

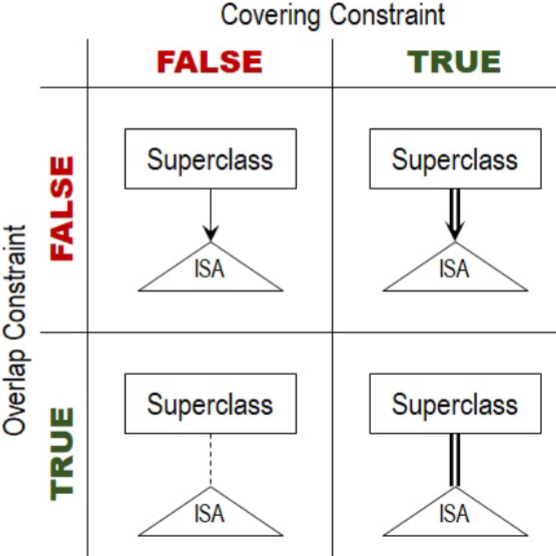


5.Basic Mapping:



CREATE TABLE Flights(fnum, orig, dest);





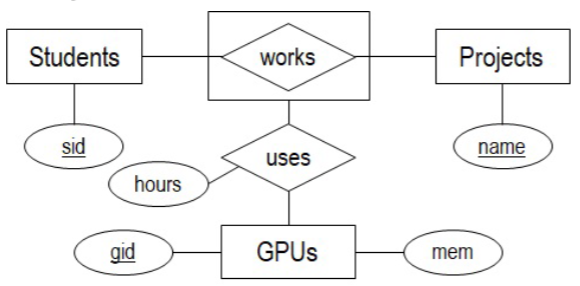
6.ISA:

i.Subclass key should NOT be shown in the ER diagram.

ii.Covering: Must a superclass entity belong to at least one subclass?

iii.Overlap: Can a superclass entity belong to multiple subclass?

7.Aggregation:



(Work.sid) -> (Students.sid)

(Work.name) -> (Projects.name)

