CSE 132A Midterm Review

Resources

Week two discussion slide:
https://drive.google.com/file/d/15t7ad3K48IyCiXHENEpRDSlS6a0cbRo9/view?usp=sharing

Week three discussion slide:
https://docs.google.com/presentation/d/1CjQ64_DdJUoRT0oHsWxuVKax2xF_TtJ9jXD_py4zHT8/edit?usp=sharing

Yilin’s Review Doc
cooperative_cheat_sheet

LEC 2 - Relational Model

- Relational Model
    - Single structure as tables (**relations)**
    - Columns as **attributes** (each has a **domain**)
    - Table consists of a set of rows (**tuples**) providing values for attributes

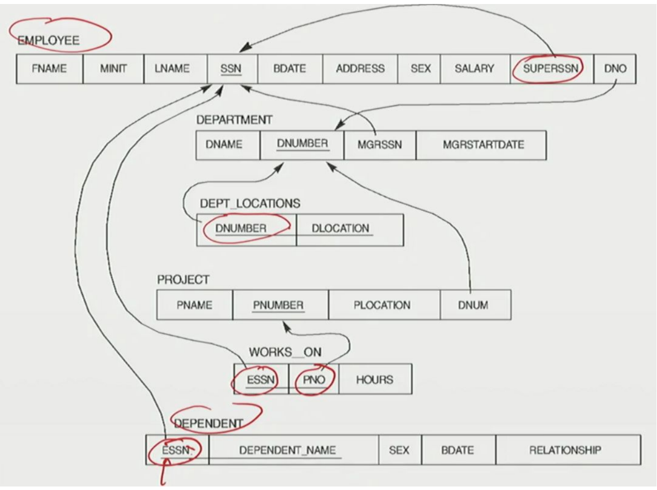
- Relation Schema (type declaration)
    - Relation name
    - Set of attributes
    - Domain of each attribute: must be **atomic**
    - **Integrity constraints**
    - E.g. CUSTOMER (cust-id, cust-name, phone_num)

- Relation Instance
    - Current content as a set of **tuples**

- Notes
    - The value of attribute A_i for tuple t: $t(A_i) = v_i$
    - Attributes are generally assumed to be **ordered**
    - Tuples are **not** considered to be ordered (equal as long as if set of tuples are the same)

- Database
    - Consists of one or several relations
    - Storing all information as a single relation is possible but not desirable
        - Repetition
        - Null values

- **Relational Integrity Constraints**
    - Constraints are **conditions** that must hold on **all** valid relation instances of a database
    - **Key Constraints**
        - Superkey: a set of attributes such uniquely defines a distinct tuple
            - Always have one superkey
        - Key: a "minimal" superkey
        - One relation has several **candidate keys**, one is chosen as **primary key**
        - Ordered generally based on the primary key and primary keys are underlined
    - Example relations
        - Multiple employees work on the same projects and one employee can work on multiple projects, multiple dependents and names etc.
    - **Entity Integrity**
        - The primary key attributes of each relation schema **cannot have null** values in any tuple.
    - **Referential Integrity**
        - Connects different values from same or different relations
            - Referencing relation should point to the **primary key** of target
            - Foreign key (don't need to be the primary key) references the **primary key** of the target or **null**.
            - Can go from a relation to itself



- **Other types of constraints**
    - Semantic integrity constraints: based on semantics
        - Specification language like assertions and triggers

        - Assert the requirements, trigger to take actions

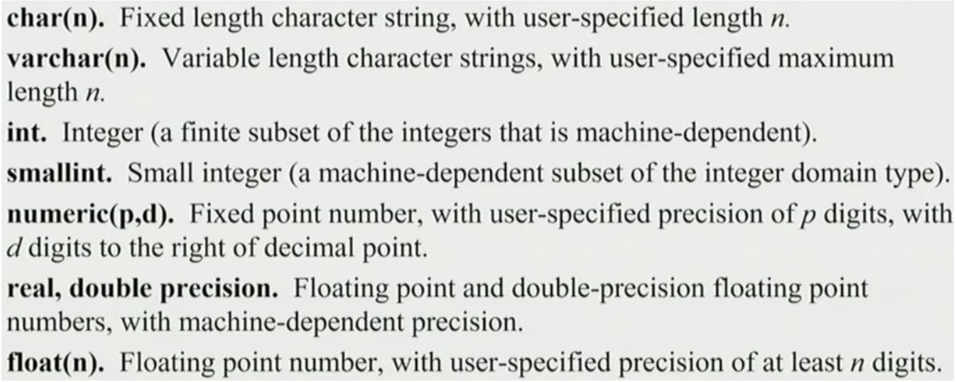
- Update Operations on Relations
    - Operations: INSERT, DELETE, MODIFY
    - Integrity constraints should not be violated
        - Cancel (REJECT) the update
        - Perform the operation but inform user
        - Trigger additional updates so violation is corrected
        - Execute user defined correction procedure
    - Group update operations may be grouped together (constraints can be violated in the middle, but not the result).

LEC 3 - Structured Query Language

- Standard for relational DB systems, but they differ.

- **Data Definition Language**
    - Name, attributes and domain
    - Integrity constraints
    - Others
        - Indice, security, physical storage

    - Types (no list or array)



char(n). Fixed length character string, with user-specified length *n*.

varchar(n). Variable length character strings, with user-specified maximum length *n*.

int. Integer (a finite subset of the integers that is machine-dependent).

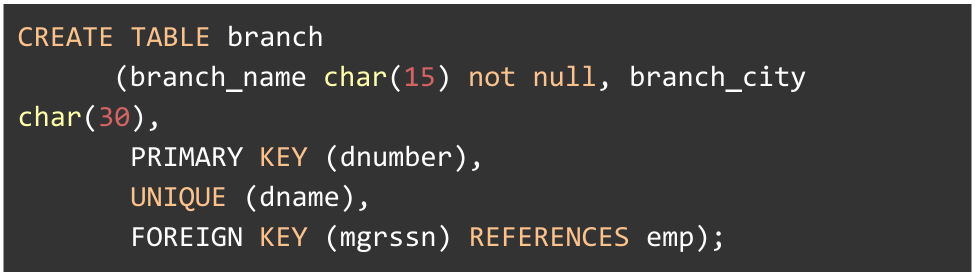
smallint. Small integer (a machine-dependent subset of the integer domain type).

numeric(p,d). Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point.

real, double precision. Floating point and double-precision floating point numbers, with machine-dependent precision.

float(n). Floating point number, with user-specified precision of at least *n* digits.

    - CREATE TABLE



```
CREATE TABLE branch
      (branch_name char(15) not null, branch_city
char(30),
       PRIMARY KEY (dnumber),
       UNIQUE (dname),
       FOREIGN KEY (mgrssn) REFERENCES emp);
```

        - Primary key, unique, foreign key, check (P) where P as predicate on attribute values only by tuple

    - DROP TABLE
        - Used to remove a relation and its definition

```
DROP TABLE dependent;
```

    - ALTER TABLE
        - Add attributes to an existing relation and all tuples are assigned **null** as default
        - Drop attributes of a relation and many database doesn't support dropping

```
ALTER TABLE r ADD att domain
ALTER TABLE r DROP att
```

LEC 4 - SQL

Data Manipulation Language (Query)

- Primarily **declarative** query language, starting with **relational calculus** as first-order logic
- Corresponding procedural language as **relational algebra**

- Basic Queries Example
  Find the titles and directors of all currently playing movies

```
SELECT movie.title, director
FROM movie, schedule
WHERE movie.title = schedule.title
```

    - May have a nested loop in the background: for each movie in movies; check each schedule in schedules

  Tuple variable: Find the actors who are also directors

```
SELECT t.actor
FROM movie t, movie s
WHERE t.actor = s.director
```

    - Needed the same relation more than once in the FROM clause.

- Features
    - Select all attributes: *
    - Pattern matching conditions: **<att> LIKE <pattern>**

        - %: any string
        - _: single character
    - Duplicate elimination: `SELECT DISTINCT attribute FROM relation`
    - Uniqueness to test multiset: UNIQUE, NOT UNIQUE
    - Order the display of tuples and descending order
    - Renaming attribute as construct: `SELECT title AS berto-title`

- **Aggregate functions**
    - Input are the **columns (multiset)** of relations: min, max, sum, avg, count
        - E.g. Count number of depositors in the bank

```
SELECT COUNT (DISTINCT customer_name)
FROM depositor
```

        - E.g. Find max, min, avg salaries of employees who work for the research department

```
SELECT MAX(salary), MIN(salary), AVG(salary)
FROM employee, department
WHERE dno=dnumber AND dname = 'research'
```

            - No repetition of employees?
                - SSN correspondings to only one DNUMBER

    - **GROUP BY**: allow the aggregate functions to separately apply to groups, in order to get information about the group

        - E.g. for each department, find the department number, number of employees and their average salary

```
SELECT dno, count(*) AS numEmployees,
       AVG(salary) as AVGSAL
FROM employee
GROUP BY dno
```

        - E.g. select the project number and name, and number of employees work on the project (assume duplicates of essn)

```
SELECT pnumber, pname, COUNT(DISTINCT essn)
FROM project, work_on
WHERE pnumber = pno
GROUP BY pnumber, pname
```

        - In the select clause, list all the **group by attributes**; to know about other attributes, must apply **the aggregate functions**.

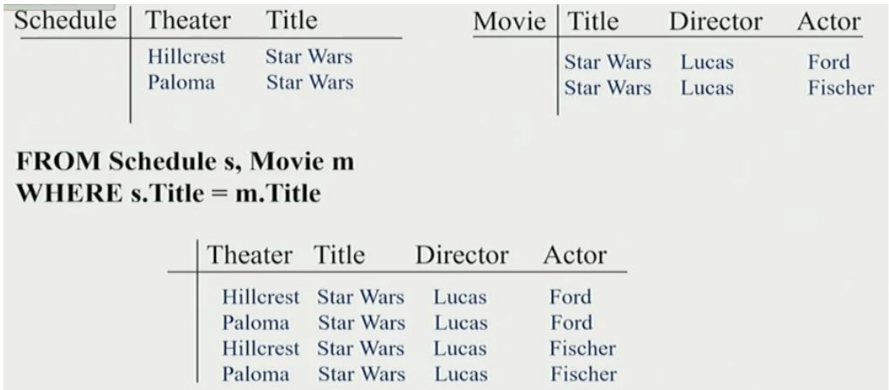
- **HAVING:** retrieve the values of aggregate functions for only **those groups satisfy certain condition**

    - E.g. Find the names of all branches where the average account balance is more than 1200

```
SELECT branch_name, AVG(balance)
FROM account
GROUP BY branch_name
HAVING AVG(balance) > 1200
```

    - E.g. for each movie having more than 100 actors, find the number of theaters showing the movie.

```
SELECT m.title, COUNT(DISTINCT s.theater)
FROM movie m, schedule s
GROUP BY m.title
HAVING COUNT(DISTINCT m.actor) > 100
```

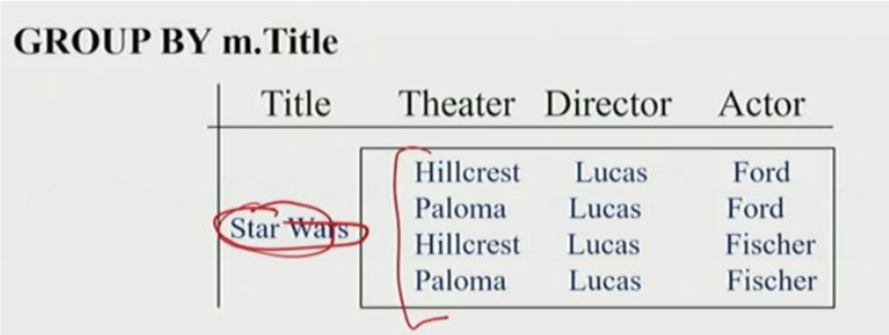


Schedule	Theater	Title
	Hillcrest	Star Wars
	Paloma	Star Wars

Movie	Title	Director	Actor
	Star Wars	Lucas	Ford
	Star Wars	Lucas	Fischer

FROM Schedule s, Movie m
WHERE s.Title = m.Title

Theater	Title	Director	Actor
Hillcrest	Star Wars	Lucas	Ford
Paloma	Star Wars	Lucas	Ford
Hillcrest	Star Wars	Lucas	Fischer
Paloma	Star Wars	Lucas	Fischer



GROUP BY m.Title

Title	Theater	Director	Actor
Star Wars	Hillcrest	Lucas	Ford
	Paloma	Lucas	Ford
	Hillcrest	Lucas	Fischer
	Paloma	Lucas	Fischer

- **Nested Queries**
    - **IN and NOT IN:** Allow the query to have WHERE clause of the form
        - E.g. find the actors in movies directed by Bertolucci.

```sql
SELECT actor FROM movie
WHERE title IN (
      SELECT title
      FROM movie
      WHERE director = 'bertolucci'
)
```

  - E.g. find the name of employees with the maximum salary: not among the salaries for which I can find larger salary.

```sql
SELECT name FROM employee
WHERE salary NOT IN (
      SELECT e.salary
      FROM employee e, employee o
      WHERE e.salary < o.salary
)
```

- Queries involving nesting but **no negation** can always be flattened, but using NOT IN increases the expressive power
  - Basic queries with no nesting are **monotonic**
    - Find the theaters showing some movie by Fellini
    - Find the actors who are also directors
    - Find the actors playing in some movie showing at Paloma
  - Queries using NOT IN are usually **not monotonic**
    - *IF the relations **INCREASE***, the answers may **DECREASE**
    - Find the theaters showing only movies by Fellini
    - Find the actors playing in every movie by Bertolucci
  - E.g. find the actors playing in every movie by “berto”

```sql
SELECT actor FROM movie
WHERE actor NOT IN (
    SELECT m1.actor
    FROM movie m1, movie m2
    WHERE m2.director = 'berto'
    AND m1.actor NOT IN (
          SELECT actor
          FROM movie
          WHERE title = m2.title
    )
)
```

- **Correlated Nested Queries**
  - The condition of a nested query references an attribute of a relation

declared in the outer query

- E.g. find the name of each employee who has a dependent with the same first name as employee

```
SELECT e.fname, e.lname
FROM employee e
WHERE e.ssn IN (
      SELECT essn
      FROM dependent
      WHERE essn = e.ssn
      AND e.fname = dependent_name
)
```

- **EXISTS (NOT EXISTS)**: the query is not empty (empty)
    - E.g. Find the titles of currently playing movies by “Berto”

```
SELECT s.title
FROM schedule s
WHERE EXISTS (
      SELECT * FROM movie
      WHERE movie.director = 'berto'
      AND movie.title = s.title
)
```

- Any: A **op ANY** <query>
    - If any X of the result of query satisfies A op X.
    - E.g. find directors currently playing movies

```
SELECT director
FROM movie
WHERE title = ANY (
      SELECT title FROM schedule
)
```

- All: A **op ALL** <query>
    - If all X of the result of query satisfies A op X.
    - E.g. find max salary employees

```
SELECT name
FROM employee
WHERE salary >= ALL
      SELECT salary FROM employee
```

- Set comparison (not declarative)

- CONTAINS: <query> CONTAINS <query>
- **UNION**: <query> UNION <query>
  - E.g. for each title in movie, find number of theaters showing that title: titles in schedule UNION titles not in the schedule
    SELECT title, COUNT(DISTINCT theater)

- INTERSECTION: intersection between two sets.
- EXCEPT: take the difference Q_1 - Q_2

- FROM: nested query in the from clause
  - E.g. Find directors of movies showing in Hillcrest:

```
SELECT m.director
FROM movie m,
      (SELECT title FROM schedule WHERE theater =
'Hillcrest') t,
WHERE m.title = t.title
```

LEC5 - SQL Queries Examples

- Find the theaters that show > 1 titles
  - Basic

```
SELECT s.theater
FROM schedule s, schedule t
WHERE s.theater = t.theater AND s.title <> t.title
```

  - Nested with counts

```
SELECT s.theater
FROM schedule s
WHERE (SELECT count(title) FROM schedule WHERE theater =
s.theater) > 1
```

  - Group by

```
SELECT s.theater
FROM schedule s
GROUP BY s.theater
HAVING count(title) > 1
```

  - Exists

```
SELECT s.theater
FROM schedule s
WHERE EXISTS (
      SELECT * FROM schedule
      WHERE theater = s.theater AND title <> s.title
)
```

    - Can also use the unique keyword

- Find theaters that showing only movies by Berto
    - NOT IN (not assumptions)

```
SELECT theater FROM schedule
WHERE theater NOT IN (
      SELECT theater FROM schedule
      WHERE title NOT IN (
            SELECT title FROM movie
            WHERE director = 'berto'
      )
)
```

    - NOT EXISTS (counter examples

```
SELECT s.theater FROM schedule s
WHERE NOT EXISTS (
      SELECT * FROM schedule x
      WHERE x.theater = s.theater AND
      NOT EXISTS (
            SELECT * FROM movie
            WHERE title = x.title
            AND dir = 'Berto'
      )
)
```

    - If a unique director assumption

```
SELECt theater FROM schedule
WHERE theater NOT IN (
      SELECT s.theater FROM schedule s, movie m
      WHERE s.theater = theater AND s.title = m.title AND
m.director <> 'berto'
)
```

- Close world assumption
    - If a tuple is missing in database, then it’s not true.

LEC 6 - Relational Calculus

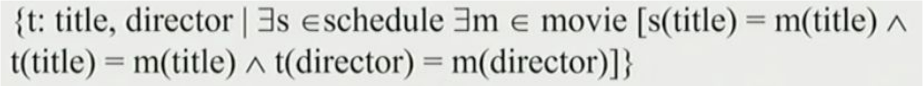
- Atoms

- m ∈ R: refer to tuple variable, m is in relation R.
- x(A): references to the attributes of x (boolean combination)
- Equality, inequality, etc

- Boolean operations
  - And, Not, Or (implication)
- Quantifiers
  - ∃ x ∈ R φ(t): existential quantification
  - ∀ x ∈ R φ(t): universal quantification
  - Scope is **φ**
  - If no quantifier, then the variable is free. We want answer variable to be the only **free** variable

- Query: **{t: <att> | φ(t)}**

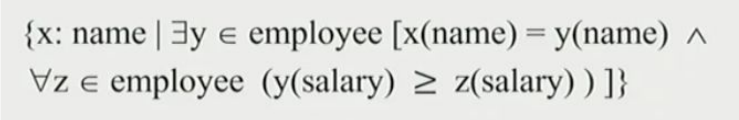
  - Find all values of t that makes **φ(t)** true
    - E.g. Find the title and director of currently playing movies



$\{t\text{: title, director} \mid \exists s \in \text{schedule } \exists m \in \text{movie } [s(\text{title}) = m(\text{title}) \wedge t(\text{title}) = m(\text{title}) \wedge t(\text{director}) = m(\text{director})]\}$

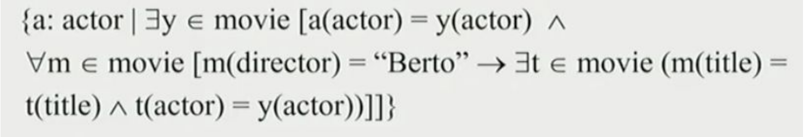
  - Active Domain: restrict the answers in the range of database, or explicitly defined in the query
  - Steps
    - What's the answer variable
    - Use existential or universal quantifiers
    - The attributes and properties

  - Examples
    - E.g. Find the employees with the highest salary

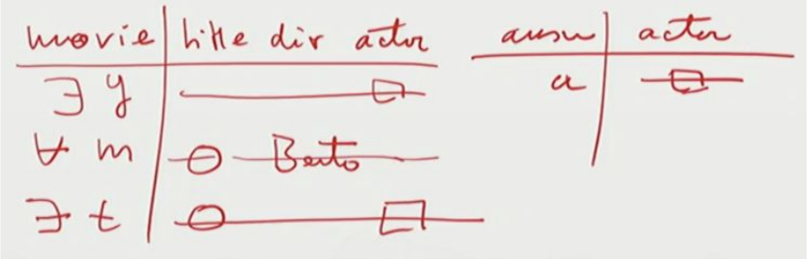


$\{x\text{: name} \mid \exists y \in \text{employee } [x(\text{name}) = y(\text{name}) \wedge \forall z \in \text{employee } (y(\text{salary}) \geq z(\text{salary}))\ ]\}$

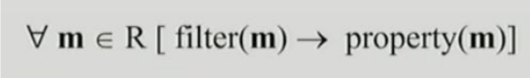
    - E.g. Find actors playing in **every** movie by Berto



$\{a\text{: actor} \mid \exists y \in \text{movie } [a(\text{actor}) = y(\text{actor}) \wedge \forall m \in \text{movie } [m(\text{director}) = \text{“Berto”} \rightarrow \exists t \in \text{movie } (m(\text{title}) = t(\text{title}) \wedge t(\text{actor}) = y(\text{actor}))]]\}$



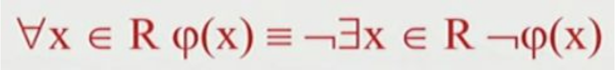
- Typical use of universal quantification



$\forall \mathbf{m} \in R \, [\, \text{filter}(\mathbf{m}) \rightarrow \text{property}(\mathbf{m})]$

    - Check property(m) for all those m that satisfy filter(m) and we don't care about the m's that do not satisfy filter(m)

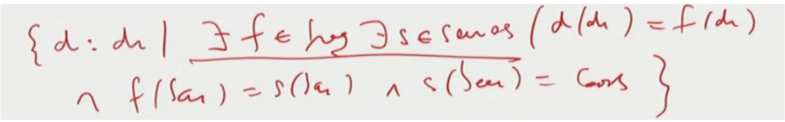
- Tuple Calculus and SQL
    - Simple and basic SQL uses only existential quantifier
    - **Eliminate universal quantifier**:



$\forall x \in R \; \varphi(x) \equiv \neg\exists x \in R \; \neg\varphi(x)$

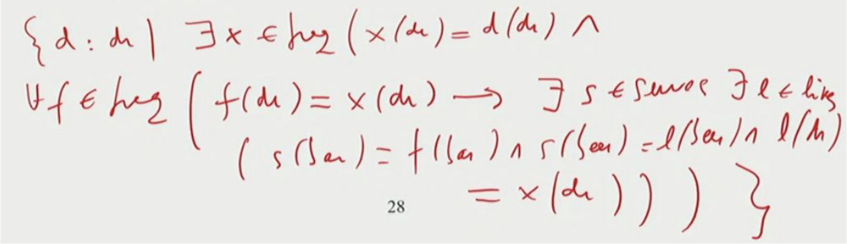
    - Negation of implication
      ! (P → Z) == P ^ ! Z
    - Calculus is more flexible than SQL because of the uses of universal and existential quantifiers

- Examples
    - E.g. Find the drinkers who frequent some bar serving Coors



$\{ d : dr \mid \underline{\exists f \in freq \; \exists s \in serves} \; (d(dr) = f(dr) \wedge f(bar) = s(bar) \wedge s(beer) = Coors \}$

    - E.g. Find the drinkers who frequent ONLY bars serving a beer they like



$\{ d : dr \mid \exists x \in freq \, (x(dr) = d(dr) \wedge \forall f \in freq \, (f(dr) = x(dr) \rightarrow \exists s \in serves \; \exists l \in likes \; (s(bar) = f(bar) \wedge s(beer) = l(beer) \wedge l(dr) = x(dr)))) \}$

28

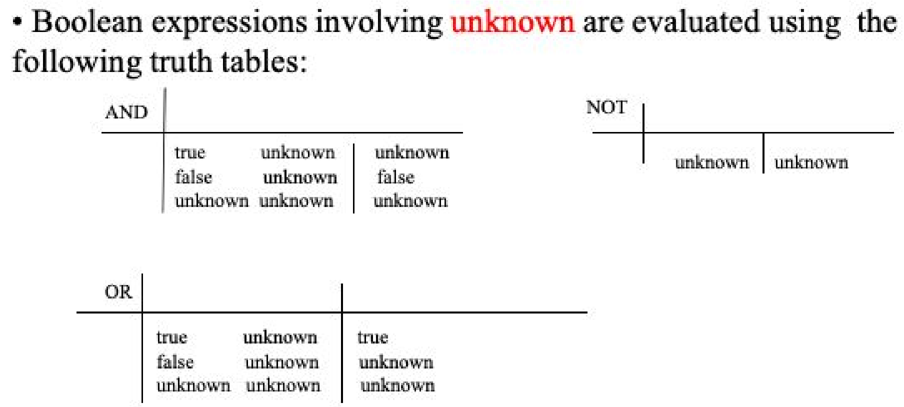
LEC 7

Null Values

- Basic
    - Testing if an attribute A is null: **IS null, IS NOT null**
    - Arithmetic operations: involves any null return null
    - Comparison: involves any null return ***unknown***

- Truth tables involving ***unknown***

- AND: false then must false, else ***unknown***
- OR: true then must true, else ***unknown***



• Boolean expressions involving unknown are evaluated using the following truth tables:

AND		
true	unknown	unknown
false	unknown	false
unknown	unknown	unknown

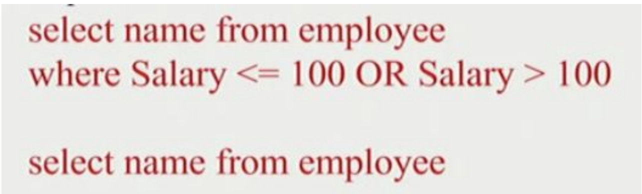
NOT	
unknown	unknown

OR		
true	unknown	true
false	unknown	unknown
unknown	unknown	unknown

-

- Where clause
  - Any WHERE clause involving unknown are **false**

- Anomalies
  - Null * 0 = Null
  - Null > 0 evaluates to unknown even if the domain restricts positive integers
  - E.g. not equivalent if some salaries are null



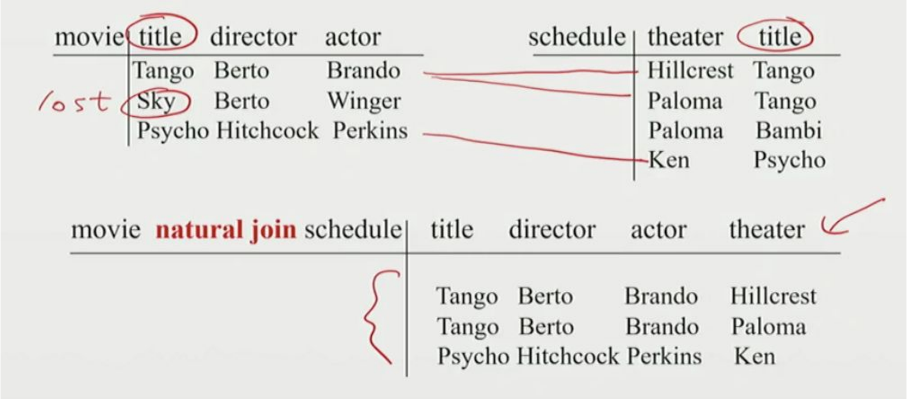
```
select name from employee
where Salary <= 100 OR Salary > 100

select name from employee
```

- Aggregate functions
  - All except the COUNT(*) ignore tuples with null values on the aggregate attributes

- GROUP BY
  - Null group-by values are treated like any other value

Join

- Natural Join
  - Combine tuples from two tables by matching on **common attributes**



lost

movie	title	director	actor
	Tango	Berto	Brando
	Sky	Berto	Winger
	Psycho	Hitchcock	Perkins

schedule	theater	title
	Hillcrest	Tango
	Paloma	Tango
	Paloma	Bambi
	Ken	Psycho

movie **natural join** schedule	title	director	actor	theater
	Tango	Berto	Brando	Hillcrest
	Tango	Berto	Brando	Paloma
	Psycho	Hitchcock	Perkins	Ken

- Often used in the **FROM** clause

```
SELECT director
FROM movie NATURAL JOIN schedule
WHERE theater = 'Hillcrest'
```

- Outer Join
    - Allow the results to have ***null***
        - *Left, full, right* outer joins
    - E.g. find the theaters showing only movies by Berto

```
SELECT theater FROM schedule
WHERE theater NOT IN (
      SELECT theater
      FROM schedule NATURAL LEFT OUTER JOIN (
            SELECT title, director FROM movie
            WHERE director = 'Berto'
      )
      WHERE director IS NULL
)
```

SQL Update Language

- Insertion
    - Some values may be left NULL
    - Inserting tuples:

```
INSERT INTO r(attr, att) VALUES (v1, v2, v3, ...)
```

- Inserting the result of queries:

```
INSERT INTO bertoMovie
            SELECT * FROM movie WHERE director = 'berto'
```

- Deletions
    - Delete from relations where condition is satisfied
    - E.g. delete all theaters showing more than one title

```
DELETE FROM schedule s
WHERE EXISTS (
      SELECT * FROM schedule
      WHERE theater = s.theater AND title <> s.title
)
```

        - Delete from relation takes sequential order, don't break it.
        - Must first find all theaters showing more than one title and then delete all from the tables

- Update
    - Update every tuple in R that satisfies <*cond*> in the way specified by the SET clause: UPDATE r SET a <expression>
    - E.g. change all "Berto" to "Bertolucci"

```
UPDATE movie
SET director = 'bertolucci'
WHERE director = 'berto'
```

LEC 8 - Views

- Customize the logical views and create temporary virtual tables
    - **Hide or restructure** data from users
    - Simply the information users should handle

- Create view statement

```
CREATE VIEW v AS <query expression>
```

- Features
    - Once defined, the view can be used in database
    - Only **limited updates** can be applied to the view
    - View definition is not the same as creating a new relation by evaluating the query expression: view content is **refreshed automatically** when the database is updated
- Dependence
    - If V_1 is used directly in V_2, then it's directly depend on.
    - If an acyclic graph has a path from V_2 to V_1
    - Recursions…

- Simplify complex queries

```
CREATE VIEW berto-movie AS
SELECT title FROM movie WHERE director = 'bertolucci'

CREATE VIEW not-all-berto AS
SELECT m.actor FROM movies m, berto-movies
WHERE berto-movies.title NOT IN
      (SELECT title FROM movies
       WHERE actor = m.actor)

SELECT actor FROM movies WHERE actor NOT IN
      (SELECT * FROM not-all-berto)
```

- **WITH** clause
    - Defines a temporary variable similar to view, but used in one command
    - With name AS (query)

- Implementation
    - Materialized views
        - Physically create and maintain a view table
        - Pros: Expect many queries, fast
        - Cons: cost of space; refresh every time database is updated
        - Strategy: incremental update (find the update without computing again)
    - Virtual views
        - Never physically created
        - Answer query on the view b reformulating it as a query
        - Pro: no need to maintain correspondence with base
        - Cons: inefficient for views defined via complex queries
        - Strategy: view unfolding (Note: no conflicting variables)

- Views Update
    - Database has to change to reflect the changes in the views.
    - Allow update on views **without** aggregates, nesting, group-by or tuple alias, defined on a **single** base table, maps naturally to an update of the underlying base table