UMD DATA605 - Big Data Systems

Relational DBs SQL Intro SQL tutorial

Instructor: Dr. GP Saggese - gsaggese@umd.edu**
TAs: Krishna Pratardan Taduri, kptaduri@umd.edu Prahar Kaushikbhai Modi, pmodi08@umd.edu
v1.1

UMD DATA605 - Big Data SystemsRelational DBs SQL Intro SQL tutorial Silbershatz: Chap 2

SCIENCE ACADEMY

Relational Model: Overview

- Introduced by Ted Codd (late 60's, early 70's)
- **First prototypes**
  - Ingres Project at Berkeley (1970-1985)
    - Ingres (INteractive Graphics REtrieval System) → PostgreSQL (=Post Ingres)
  - IBM System R (1970) → Oracle, IBM DB2
- **Contributions from relational data model**
  - Formal semantics for data operations
  - Data independence: separation of logical and physical data models
  - Declarative query languages (e.g., SQL)
  - Query optimization
- **Key to commercial success**

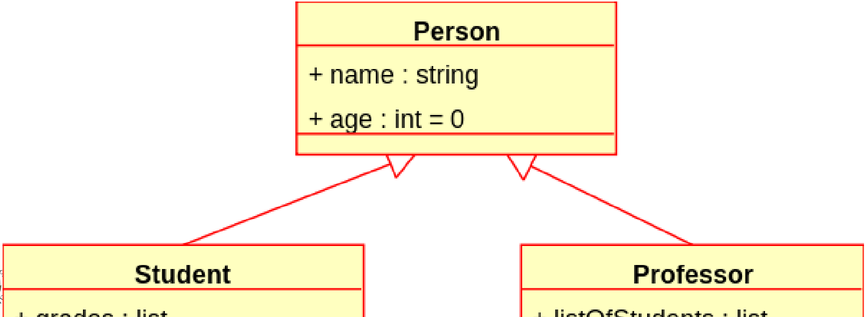
Relational Model: Key Definitions

- A relational DB consists of a collection of `text tables / relations`
  - Each table has a unique name and a schema
- Each `text row / tuple / record` in a table represents a relationship among a set of values
- Each **element** of a row corresponds to a `text column / field / attribute`
  - Each element in a column is atomic (e.g., a phone number is a single object and not a sequence of numbers)
  - `text NULL` represents a value that is unknown or doesn't exist (e.g., someone not having a phone number)
- E.g., `text instructor` and `text course` relations
- **Schema of a relation**
  - A list of attributes and their domains
  - It's like type definition in programming languages
  - E.g., the domain of `text salary` is integers $>= 0$
- **Instance of relation**
  - A particular instantiation of a relation with actual values
  - Will change over time

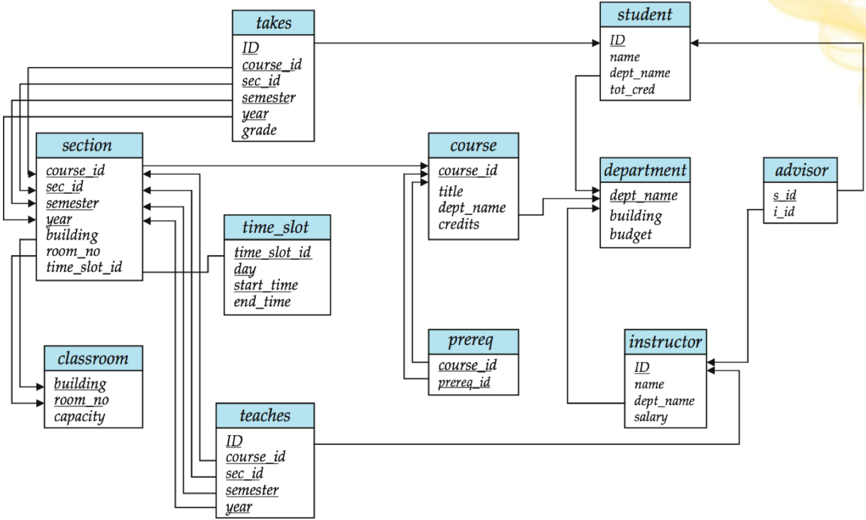
ID	*name*	*dept_name*	*salary*

UML Class Diagram

- UML class diagram
  - UML = Unified Modeling Language
  - Used in OOP and DB design
- **In OOP design**
  - Diagram showing classes, attributes, methods, and relationships
- **In DB design**
  - Each box is a table / relation
  - Columns / fields / attributes are listed inside the box
  - Primary keys are underlined
  - Foreign key constraints are arrows



Example: University DB



- **UML diagram of a DB and schemas representing a University**
- Each box is a table / relation

Primary Key

- *R* is the set of attributes of a relation *r*
  - E.g., **ID, name, dept_name, salary** are attributes of relat

- *K* is a superkey of *R* if values for *K* are sufficient to ide
  - E.g., **(ID)** and **(ID, name)** are both superkeys of **inst
  - ```text (name)``` is not a superkey of ```text instructor```

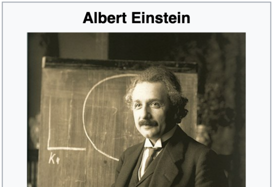
- **Superkey** *K*` is a candidate key** if *K* is minimal
  - E.g., (ID) is a candidate key for **instructor**

- One of the candidate keys is selected to be the **primary key**
  - Typically one that is small and immutable (or at least it does
  - Would **SSN** be a primary key? Yes and no
- A primary key is a minimal set of attributes that identify *uniq
- **Primary key constraint**: rows in the relation can’t have the

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000

Question: What are Primary Keys?

- Marital status
  - Married(person1_ssn, person2_ssn, date_married, date_divorced)
- Bank account
  - Account(cust_ssn, account_number, cust_name, balance, cust_address)
- Research assistantship at UMD
  - RA(student_id, project_id, supervisor_id, appt_time, appt_start_date, appt_end_date)
- Information typically found on Wikipedia
  - Person(Name, Born, Died, Citizenship, Education, . . .)
- Info about US President on Wikipedia
  - President(name, start_date, end_date, vice_president, preceded_by, succeeded_by)
- Tour de France: historical rider participation information
  - Rider(Name, Born, Team-name, Coach, Sponsor, Year)



Answer: What are Primary Keys?

- Marital status
  - text Married(**person1_ssn, person2_ssn, date_married**, date_divorced)
- Bank account
  - Account(cust_ssn, **account_number**, cust_name, balance, cust_address)
- Research assistantship at UMD
  - RA(**student_id, project_id**, supervisor_id, appt_time, **appt_start_date**, appt_end_date)
- Information typically found on Wikipedia
  - Person(**Name, Born, Died, Citizenship, Education, ...**)
- Info about US President on Wikipedia
  - President(**name, start_date**, end_date, vice_president, preceded_by, succeeded_by)
- Tour de France: historical rider participation information
  - Rider(**Name, Born, Team-name,** Coach, Sponsor, **Year**)

![](data605/lectures_source/images/lecture_4_1/lec_4_1_slide_9_ima

SCIENCE ACADEMY

Foreign Key

- **Foreign key** = primary key of a relation that appears in another relation
  - E.g., **(ID)** from *student* appears in the relations *takes*, *advisor*
  - *takes* is the "referencing relation", has the foreign key
  - *student* is the "referenced relation", has the primary key
  - Typically shown by an arrow from referencing → referenced
- **Foreign key constraint**: for each row, the tuple corresponding to a primary key must exist
  - Aka referential integrity constraint
  - If there is a **(student101, DATA605)** in *takes*, there must be a tuple with **student101** in *student*
- In words, the key referenced as foreign key needs to exist as primary key

Relational Algebra: 1/4

- **Relation**: set of tuples
- **Relational algebra**: operations that take one or more relations as input and produce a new relation, e.g.,
  - Unary relation: selection, projection, rename
  - Binary relation: union, set difference, intersection, Cartesian product, join
- **Selection (Σ)**: select tuples that satisfy a given predicate
  - E.g., select tuples of *instructor* where **dept_name = "Physics"**
- **Projection (π)**: return tuples with a subset of attributes
  - E.g., project tuples of *instructor* with only **(name, salary)**
- **Set operations**: union, intersection, set_difference of relations
  - Need to be compatible (i.e., have the same attributes)

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000

Relational Algebra: 2/4

- **Cartesian product**: combine information from two relations into a new one
  - *instructor* = **(ID, name, dept_name, salary)**
  - *teaches* = **(ID, course_id, sec_id, semester, year)**
- E.g., *instructor* x *teaches* gives **(instructor.ID, instructor.name, instructor.dept_name, teaches.ID, ...)**

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000

Relational Algebra: 3/4

- **Join**: composition of two operations
  - Cartesian-product
  - A selection based on equality between two fields
- E.g., *instructor* x *teaches* when **instructor.ID = teaches.ID**

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017

Relational Algebra: 4/4

- **Query**: combination of relational algebra operations
  - E.g., "find the **course_id** from the rows of the table section for the fall semester of 2017"
- **Assignment**: assign parts of relational algebra to temporary relation variables
  - A query can be written as a sequential program
  - E.g., "find the **course_id** for the classes that are run in both fall 2017 and spring 2018"
- **Equivalent queries**: two queries that give the same result on any DB instance
  - Some formulation can be more efficient than others

$$courses\_fall\_2017 \leftarrow \Pi_{course\_id}(\sigma_{semester = \text{“Fall”} \land year=2017}(section))$$
$$courses\_spring\_2018 \leftarrow \Pi_{course\_id}(\sigma_{semester = \text{“Spring”} \land year=2018}(section))$$
$$courses\_fall\_2017 \cap courses\_spring\_2018$$

$$\Pi_{course\_id}(\sigma_{semester = \text{“Fall”} \land year=2017}(section))$$

SCIENCE ACADEMY

SQL Overview

- **Relational algebra**: mathematical description of a language to manipulate *relations*
- **SQL**: programming language to describe and transform data in a relational DB
  - Originally called Sequel
  - Name was changed to Structured Query Language
- **SQL statements can be grouped based on their goal**
  - Data definition language (DDL)
    - Define schema of the data (e.g., tables, attributes, indices)
    - Specify integrity constraints (e.g., primary key, foreign key, not null)
  - Data modification language (DML)
    - Modify the data in tables
    - E.g., Insert, Update, Delete
  - Query data (DQL)
  - Control transactions
    - E.g., specify beginning and end, control isolation level
  - Define views
  - Authorization
    - Specify access and security constraints

SQL Overview

- Data description language (DDL) `text CREATE TABLE <name> (<field> <domain>, ... )`
- Data modification language (DML) `text INSERT INTO <name> (<field names>) VALUES (<field values>) text DELETE FROM <name> WHERE <condition> text UPDATE <name> SET <field name> = <value> WHERE <condition>`
- Query language `text SELECT <fields> FROM <name> WHERE <condition>`

Create Table

`text CREATE TABLE r (A_1 D_1, A_2 D_2, ..., A_n D_n, IntegrityConstraint_1, IntegrityConstraint_n);` - **Constraints** - SQL will prevent changes to the DB that violate any integrity constraint - Primary key - Need to be all non-null and unique - `text PRIMARY KEY (A_j1, A_j2, ..., A_jn)` - Foreign key - Values of attributes for any tuple in current relation must correspond to values of the primary key attributes of some tuple in relation s - `text FOREIGN KEY (A_k1, A_k2, ..., A_kn) REFERENCES s` - Not null - Specify that null value is not allowed for that attribute - `text A_i D_i NOT NULL`

where: - r is name of *table* (aka *relation*) - A_i name of *attribute* (aka *field*, *column*) - D_i domain of attribute A_i

Select

`text SELECT A_1, A_2, ..., A_n FROM r_1, r_2, ..., r_m WHERE P;` - **SELECT**: select the attributes to list (i.e., projection) - **FROM**: list of tables to be accessed - Define a Cartesian product of the tables - The query is going to be optimized to avoid to enumerate tuples that will be eliminated - **WHERE**: predicate involving attributes of the relations in the FROM clause (i.e., selection) - In SELECT or WHERE clauses, might need to use the table names as prefix to qualify the attribute name - E.g., instructor.ID vs teaches.ID - A SELECT statement can be expressed in terms of relational algebra - Cartesian product $\rightarrow$ selection $\rightarrow$ projection - Difference: SQL allows duplicate values, relational algebra works with mathematical sets

Null values

- An arithmetic operation with NULL yields NULL
- Comparison with NULL
  - $1 < \text{NULL}$?
  - What about $\text{NOT}(1 < \text{NULL})$?
  - SQL yields UNKNOWN when comparing with NULL value
  - There are 3 logical values: True, False, Unknown
- Boolean operators
  - Can be extended according to common sense, e.g.,
  - True AND Unknown = Unknown
  - False AND Unknown = False
- In a WHERE clause, if the result is Unknown it's not included

Group by Query

- The attributes in GROUP BY are used to form groups
  - Tuples with the same value on all attributes are placed in one group
- Any attribute that is not in the GROUP BY can appear in the SELECT clause only as argument of aggregate function text `SELECT dept_name, AVG(salary) FROM instructor GROUP BY dept_name;` text `-- Error.` text `SELECT dept_name, salary FROM instructor GROUP BY dept_name;`
- salary is not in GROUP BY so it must be in an aggregate function

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000

Having

- State a condition that applies to groups instead of tuples (like WHERE)
- Any attribute in the HAVING clause must appear in the GROUP BY clause
- E.g., find departments with avg salary of instructors $>$ 42k `text SELECT dept_name, AVG(salary) AS avg_salary FROM instructor GROUP BY dept_name HAVING AVG(salary) > 42000;`
- How does it work
  - FROM is evaluated to create a relation
  - (optional) WHERE is used to filter
  - GROUP BY collects tuples into groups
  - (optional) HAVING is applied to each group and groups are filtered
  - SELECT generates tuples of the results, applying aggregate functions to get a single result for each group

UNIVERSITY OF MARYLAND 18 56 SCIENCE ACADEMY

Nested subqueries

- SQL allows to use the result of a query in another query
  - E.g., one can use a subquery returning only one attribute (aka scalar subquery) in any place a value is used
  - E.g., use the result of a query for set membership in the WHERE clause
  - E.g., use the result of a query in a FROM clause text `SELECT tmp.dept_name, tmp.avg_salary FROM (SELECT dept_name, AVG(salary) AS avg_salary FROM instructor GROUP BY dept_name) AS tmp WHERE avg_salary > 42000`

dept_name	avg_salary
Finance	85000.00000000000
History	61000.00000000000
Physics	91000.00000000000

With

- WITH clause allows to define a temporary relation containing the results of a subquery
- It can be equivalent to a nested subqueries, but clearer
- Find department with the maximum budget `text WITH max_budget(value) as (SELECT MAX(budget) FROM department) SELECT department.dept_name, budget FROM department, max_budget WHERE department.budget = max_budget.value`

dept_name	budget
Finance	120000.00

Insert

To insert data into a relation we can specify tuples to insert - Tuples text `INSERT INTO course VALUES ('DATA-605', 'Big data systems', 'Comp. Sci.', 3)` text `INSERT INTO course(course_id, title, dept_name, credits) VALUES ('DATA-605', 'Big data systems', 'Comp. Sci.', 3)` - Query whose results is a set of tuples text `INSERT INTO instructor (SELECT ID, name, dept_name, 18000 FROM student WHERE dept_name = 'Music' AND tot_cred > 144)` - Nested queries are evaluated and then inserted so this doesn't create infinite loops text `INSERT INTO student (SELECT * FROM student)` - Many DB have bulk loader utilities to insert a large set of tuples into a relation, reading from formatted text files This is much faster than INSERT statements

SCIENCE ACADEMY

Update

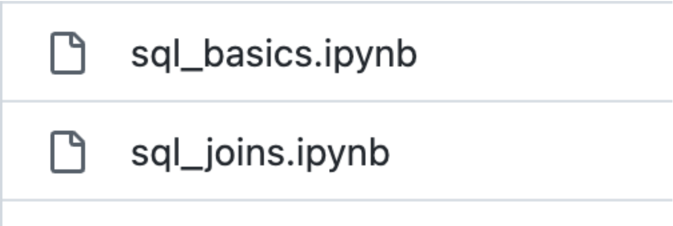
- SQL can change a value in a tuple without changing all the other values
- E.g., increase salary of all instructors by 5% `text UPDATE instructor SET salary = salary * 1.05`
- E.g., conditionally `text UPDATE instructor SET salary = salary * 1.05 WHERE salary < 70000`
- Nesting is allowed `text UPDATE instructor SET salary = salary * 1.05 WHERE salary < (SELECT AVG(salary) FROM instructor)`

Delete

- One can delete tuples using a query returning entire rows of a table **DELETE FROM r WHERE p** where:
- r is a relation
- P is a predicate
- Remove all tuples (but not the table) **DELETE FROM instructor**

SQL Tutorial

- SQL tutorial dir
- Readme
  - Explains how to run the tutorial
- Three notebooks in tutorial_university
- **How to learn from a tutorial**
  - Reset the notebook
  - Execute each cell one at the time
  - Ideally create a new file and retype (!) everything
  - Understand what each cell does
  - Look at the output
  - Change the code
  - Play with it
  - Build your mental model



Example Schema for SQL Queries

Movie(title, year, length, inColor, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName) MovieStar(name, address, gender, birthdate) MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)

title	year	length	inColor	studioName	producerC#
Plane Crazy	1927	6	no	Disney	WD100
Casablanca	1942	102	no	Warner Bros.	HW101
Star Wars	1977	121	yes	LucasFilm	GL102
Star Wars: ESB	1980	120	yes	LucasFilm	GL102
Star Wars: TPM	1999	133	yes	LucasFilm	GL102
Raiders of the Lost Ark	1981	115	yes	Paramount	FM103

Movies

name	address	gender	birthdate
H. Ford	...	M	7/13/1942
M. Hamill	...	M	9/25/1951
C. Fisher	...	F	10/21/1956

MovieStar

movieTitle	movieYear	starName
Star Wars	1977	H. Ford
Star Wars	1977	M. Hamill
Star Wars	1977	C. Fisher
Star Wars: ESB	1980	H. Ford
Star Wars: ESB	1980	M. Hamill
Star Wars: ESB	1980	C. Fisher
Raiders of the Lost Ark	1981	H. Ford

StarsIn

name	address	cert#	netWorth

SQL: Data Definition

- CREATE TABLE
- Must define movieExec before movie. Why?

CREATE TABLE movieExec (name char(30), address char(100), cert# integer primary key, networth integer); CREATE TABLE movie (title char(100), year integer, length integer, inColor smallint, studioName char(20), producerC# integer references movieExec(cert#));

SQL: Data Manipulation

- INSERT `text INSERT INTO StarsIn values('King Kong', 2005, 'Naomi Watts'); text INSERT INTO StarsIn(starName, movieTitle, movieYear) values('Naomi Watts', 'King Kong', 2005);`
- DELETE `text DELETE FROM movies WHERE movieYear < 1980;`
  - Syntax is fine, but this command will be rejected. Why? `text DELETE FROM movies WHERE length < (SELECT avg(length) FROM movies);`
  - Problem: as we delete tuples, the average length changes
  - Solution:
    - First, compute avg length and find all tuples to delete
    - Next, delete all tuples found above (without recomputing avg or retesting the tuples)

SQL: Data Manipulation

- UPDATE
  - Increase all movieExec netWorth's over 100,000 USD by 6%, all other accounts receive 5%
  - Write two update statements: `text UPDATE movieExec SET netWorth = netWorth * 1.06 WHERE netWorth > 100000; text UPDATE movieExec SET netWorth = netWorth * 1.05 WHERE netWorth <= 100000;`
  - The order is important
  - Can be done better using the case statement `text UPDATE movieExec SET netWorth = CASE WHEN netWorth > 100000 THEN netWorth * 1.06 WHEN netWorth <= 100000 THEN netWorth * 1.05 END;`

SQL Single Table Queries

- Movies produced by Disney in 1990: note the *rename* `text SELECT m.title, m.year FROM movie m WHERE m.studioname = 'disney' AND m.year = 1990;`
  - The SELECT clause can contain expressions `text SELECT title || ' (' || to_char(year) || ')' AS titleyear` `text SELECT 2014 - year`
  - The WHERE clause support a large number of different predicates and combinations thereof `text year BETWEEN 1990 and 1995` `text title LIKE 'star wars%'` `text title LIKE 'star wars _'`

Single Table Queries

- Find distinct movies sorted by title **SELECT DISTINCT title FROM movie WHERE studioname = 'disney' AND year = 1990 ORDER by title;**
- Average length of a movie **SELECT year, avg(length) FROM movie GROUP BY year;**
- **GROUP BY:** is a very important concept that shows up in many data processing platforms
  - What it does:
    - Partition the tuples by the group attributes (*year* in this case)
    - Do something (*compute avg* in this case) for each group
    - Number of resulting tuples == number of groups

Single Table Queries

- Find movie with the maximum length **SELECT title, year FROM movie where movie.length = (select max(length) from movie);**
  - The smaller "subquery" is called a "nested subquery"
- Find movies with at most 5 stars: an example of a correlated subquery **SELECT * FROM movies m WHERE 5 >= (SELECT count(*) FROM starsIn si WHERE si.title = m.title AND si.year = m.year);**
  - The "inner" subquery counts the number of actors for that movie.

Single Table Queries

- Rank movies by their length **SELECT title, year, (SELECT count(*) FROM movies m2 WHERE m1.length <= m2.length) AS rank FROM movies m1;**
  - Key insight: A movie is ranked 5th if there are exactly 4 movies with longer length.
  - Most database systems support some sort of a *rank* keyword for doing this
  - The above query doesn't work in presence of ties, etc.
- Set operations **SELECT name FROM movieExec union/intersect/minus SELECT name FROM movieStar**

Single Table Queries

- Set Comparisons **SELECT * FROM movies WHERE year IN [1990, 1995, 2000];

SELECT * FROM movies WHERE year NOT IN (SELECT EXTRACT(year from birthdate) FROM MovieStar);**

Multi-table Queries

- Key:
  - Do a join to get an appropriate table
  - Use the constructs for single-table queries
  - You will get used to doing all at once
- Examples: **SELECT title, year, me.name AS producerName FROM movies m, movieexec me WHERE m.producerC# = me.cert#;**

Multi-table Queries

- Consider the query: **SELECT title, year, producerC#, count(starName) FROM movies, starsIn WHERE title = starsIn.movieTitle AND year = starsIn.movieYear GROUP BY title, year, producerC#**
  - What about movies with no stars?
  - Need to use **outer joins** **SELECT title, year, producerC#, count(starName) FROM movies LEFT OUTER JOIN starsIn ON title = starsIn.movieTitle AND year = starsIn.movieYear GROUP BY title, year, producerC#**
  - All tuples from 'movies' that have no matches in starsIn are included with NULLs
  - So if a tuple (m1, 1990) has no match in starsIn, we get (m1, 1990, NULL) in the result
  - The count(starName) works correctly then
  - Note: count(*) would not work correctly (NULLs can have unintuitive behavior)

Other SQL Constructs

- Views **CREATE VIEW DisneyMovies SELECT * FROM movie m WHERE m.studioname = 'disney'; **
  - Can use it in any place where a table name is used
  - Views are used quite extensively to: (1) simplify queries, (2) hide data (by giving users access only to specific views)
  - Views may be *materialized* or not

Other SQL Constructs

- NULLs
  - Value of any attribute can be NULL
  - Because: value is unknown, or it is not applicable, or hidden, etc.
  - Can lead to counterintuitive behavior
  - For example, the following query does not return movies where length = NULL **SELECT * FROM movies WHERE length >= 120 OR length <= 120**
  - Aggregate operations can be especially tricky
- Transactions
  - A transaction is a sequence of queries and update statements executed as a single unit
  - For example, transferring money from one account to another
    - Both the *deduction* from one account and *credit* to the other account should happen, or neither should
- Triggers
  - A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database

Other SQL Constructs

- Integrity Constraints
  - Predicates on the database that must always hold
  - Key Constraints: Specifying something is a primary key or unique `text` `CREATE TABLE customer (ssn CHAR(9) PRIMARY KEY, cname CHAR(15), address CHAR(30), city CHAR(10), UNIQUE (cname, address, city));`
  - Attribute constraints: Constraints on the values of attributes `text bname char(15) not null` `text balance int not null, check (balance>= 0)`

SCIENCE ACADEMY

Integrity Constraints

- Referential integrity: prevent dangling tuples `text CREATE TABLE branch(bname CHAR(15) PRIMARY KEY, ...); text CREATE TABLE loan(..., FOREIGN KEY bname REFERENCES branch);`
- Can tell the system what to do if a referenced tuple is being deleted

Integrity Constraints

- Global Constraints
  - Single-table text `CREATE TABLE branch (..., bcity CHAR(15), assets INT, CHECK (NOT(bcity = 'Bkln') OR assets>5M))`
  - Multi-table text `CREATE ASSERTION loan-constraint CHECK (NOT EXISTS (SELECT* FROM loan AS L WHERE NOT EXISTS(SELECT* FROM borrower B, depositor D, account A WHERE B.cname = D.cname AND D.acct_no = A.acct_no AND L.lno= B.lno)))`

Additional SQL Constructs

- **Select** subquery factoring
  - To allow assigning a name to a subquery, then use its result by referencing that name **WITH temp AS (SELECT title, avg(length) FROM movies GROUP BY year)SELECT COUNT(*) FROM temp; **
  - Can have multiple subqueries (multiple with clauses)
  - Real advantage is when subquery needs to be referenced multiple times in main select
  - Helps with complex queries, both for readability and maybe performance (can cache subquery results)

Another SQL Construct

- **SELECT HAVING** clause
  - Used in combination with **GROUP BY** to restrict the groups of returned rows to only those where condition evaluates to true **SELECT year, count(*) FROM movies WHERE year > 1980 GROUP BY year HAVING COUNT(*) > 10;**
  - Difference from **WHERE** clause is that it applies to summarized group records, and where applies to individual records