L04: SQL 1

DSAN 6300/PPOL 6810: Relational Databases and SQL Programming

Irina Vayndiner

September 18 & 21, 2023

GEORGETOWN UNIVERSITY

Agenda for today's class

- Logistics
  - HW1 is published
  - If you still have issues with MySQL setup, attend OHs
    - Use today's break!
  - Schedule reminders: Mark your calendars NOW
    - Next week: Class on Thu 10/28 (only) on zoom, will be recorded
    - Midterms: Thu, 10/19 and Mon 10/23
    - Final Test: Tue, 12/5 10:30am

- Today
  - Lecture part 1: Finishing: Translating E-R diagrams
  - Lab + Break (diff today!)
  - Lecture part 2: SQL 1

AWS

- Accept invite to AWS Canvas
  - Academy Learner Lab
  - Access *to **a restricted set of AWS services***.
  - You will retain access to the AWS resources set up in this environment for the duration of this course.
  - We limit your budget **($100USD)**, so you should exercise caution to prevent charges that will deplete your budget too quickly. If you exceed your budget, you will lose access to your environment and lose all of your work.
- Each session lasts for 4 hours by default
  - can extend a session by pressing the start button to reset your session timer.
  - At the end of each session, any resources you created will persist.
    - However, we automatically shut EC2 instances down.
    - **RDS** instances will keep running
  - More info: when you log into AWS Canvas

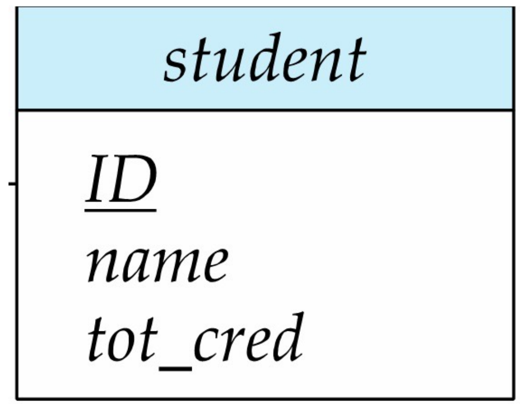
Translating E-R Diagrams to Relational Schemas

Reduction of E-R model to Relation Schemas

- Entity sets and relationship sets we saw in L02 can be expressed as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of schemas.
- For each entity set and relationship set there is a unique schema that is assigned the corresponding name
- Each schema has a number of columns (generally corresponding to attributes), which have unique names.

Representing Entity Sets

- A strong entity set translates to a schema with the same attributes
- Primary key of an entity set translates to primary keys of the schema



student(<u>ID</u>, name, tot_cred)

Representation of Entity Sets with Composite Attributes

- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - Example: given an entity set *instructor* with composite attribute *name* with component attributes *first_name, middle_initial,* and *last_name*
  - the schema corresponding to the entity set has attributes *name_first_name, name_middle_initial,* and *name_last_name*
    - Prefix omitted if there is no ambiguity (*name_first_name* could be *first_name)*
- Ignoring (for now) multivalued and derived attributes, extended instructor schema is
  - *instructor(ID,*
    first_name, middle_initial, last_name,
    street_number, street_name,
    apt_number, city, state, zip_code,
    date_of_birth)

instructor
<u>*ID*</u> *name*   *first_name*   *middle_initial*   *last_name* *address*   *street*   *street_number*   *street_name*   *apt_number*   *city*   *state*   *zip* *{ phone_number }* *date_of_birth* *age ( )*

Representation of Entity Sets with Multi-valued Attributes

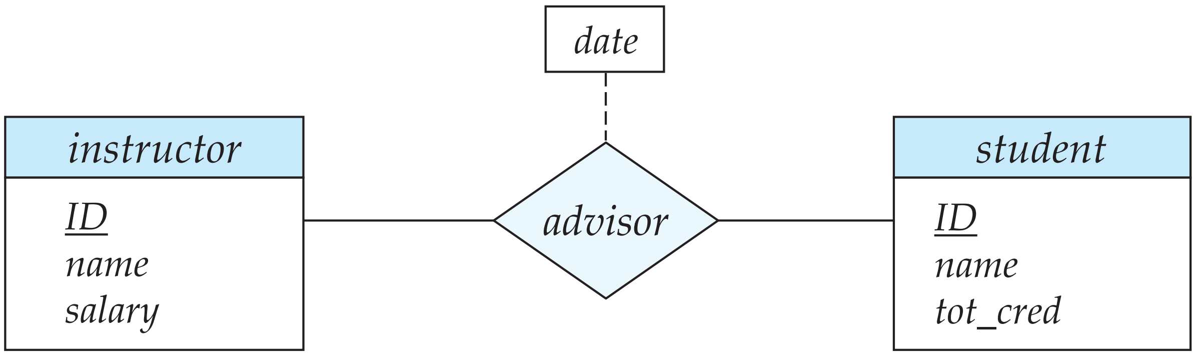
- A multivalued attribute *M* of an entity *E* is represented by a separate schema *EM*
- Schema *EM* has attributes corresponding to the primary key of *E* and an attribute corresponding to multivalued attribute *M*
- Example: Multivalued attribute *phone_number* of *instructor* is represented by a schema:

  instructor_phone= (*ID*, *phone_number*)

- Each value of the multivalued attribute maps to a separate tuple of the relation on schema *EM*
  - For example, an *instructor* entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples: (22222, 456-7890) and (22222, 123-4567)

Representing Many-to-many Relationship Sets

- A **many-to-many** relationship set is represented as a schema with attributes:
  - The primary keys of the ***all*** participating entity sets (for relationships of any degree)
  - Any descriptive attributes of the relationship set
- Foreign keys in that schema are defined to reference primary keys of relations representing participating entity sets
- Example: schema for relationship set *advisor*



instructor(<u>ID</u>, name, salary)

student(<u>ID</u>, name, tot_cred)

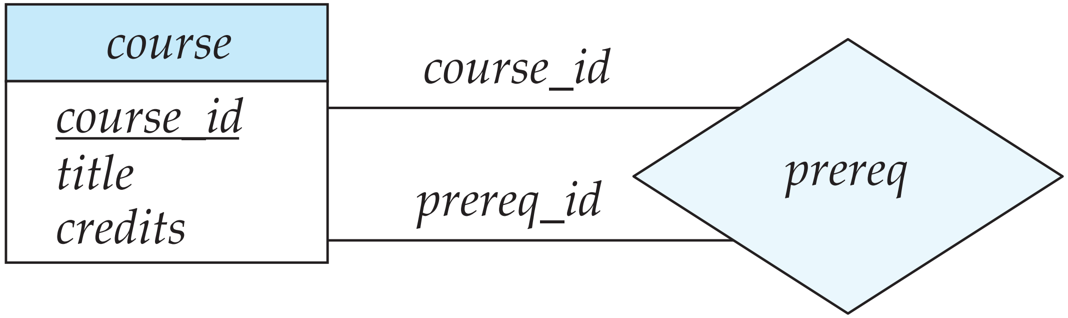
advisor = (<u>s_id, i_id,</u> date,

foreign key s_id references student(ID),

foreign key i_id references instructor(ID))

Recursive Relationships

- A relationship set is represented as a schema with attributes:
  - The primary key of the participating entity set plays two roles and thus renamed by label names
  - Any descriptive attributes of the relationship set
- Foreign keys in that schema are defined to reference primary keys of relations representing participating entity sets
- Example: schema for relationship set *prereq*



course = (<u>course_id</u>, title, credits)

prereq = (<u>course_id, prereq_id</u>,

foreign key course_id references course(course_id),

foreign key prereq_id references course(course_id))

Representing One-to-many Relationship Sets

- Many-to-one and one-to-many relationship sets can be represented by adding an extra attribute to the "many" side, containing the primary key of the "one" side
- Example: each student has at most one instructor, and an instructor can advise more than one student

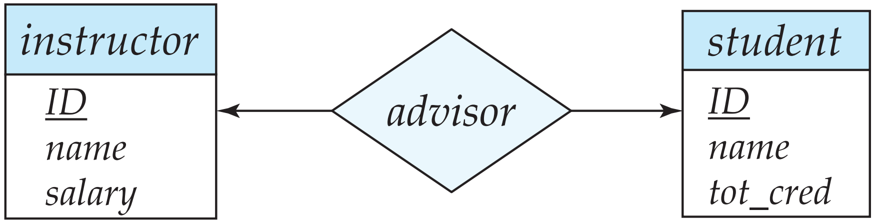
- Instead of creating a schema for relationship set *advisor*, add an attribute *i_id* to the schema for entity set *student*

instructor(<u>ID</u>, name, salary)
*student(<u>ID</u>, **i_id**, name, tot_cred,*
foreign key i_id references instructor(ID))

- Note: If participation is *partial* on the "many" side, replacing a schema by an extra attribute (corresponding to the "many" side) could result in null values
  - *Students that have no advisors*

Representing One-to-one Relationship Sets

- For one-to-one relationship sets, either side can be chosen to act as the “many” side
- Example:



- Add an attribute *i_id* to the schema arising from entity set *student*

  instructor(<u>ID</u>, name, salary)

  student(<u>ID</u>, i_id, name, tot_cred,

  foreign key i_id references instructor(ID))

OR

- Add an attribute *s_id* to the schema arising from entity set *instructor*

  instructor(<u>ID</u>, s_id, name, salary)

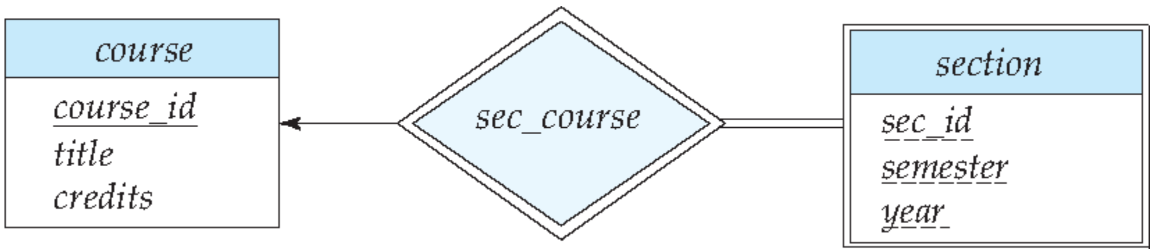
  foreign key s_id references student(ID))

  student(<u>ID</u>, name, tot_cred)

- If participation is *partial* replacing a schema by an extra attribute could result in null values

Representing Weak Entity Sets

- A weak entity set translates to a schema with the attributes for the primary key of the identifying strong entity set
- **No additional schema for relationship set:** weak entity set and identifying relationship set are translated into a single schema
- Example:

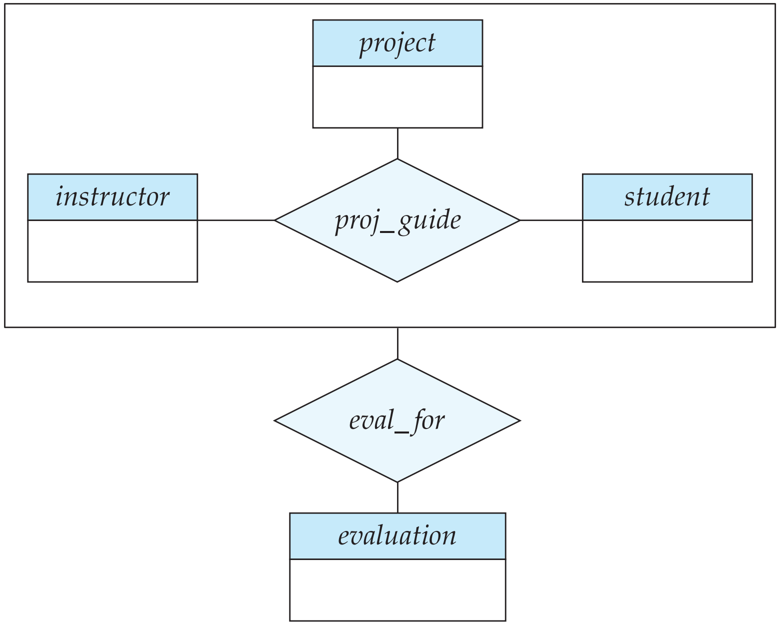


course (<u>course_id,</u> title, credits)

section (<u>course_id, sec_id, semester, year</u> ,
foreign key (course_id) references course (course_id))

Representing Aggregation

- Recall an *aggregation* example that we discussed in L02:
  - A student is guided by a particular instructor on a particular project
  - A student, instructor, project combination may have an associated evaluation



Representing Aggregation (continued)

- To represent an aggregation, one need to create a schema containing:
  - Primary key of the aggregated relationship
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval_for* is:

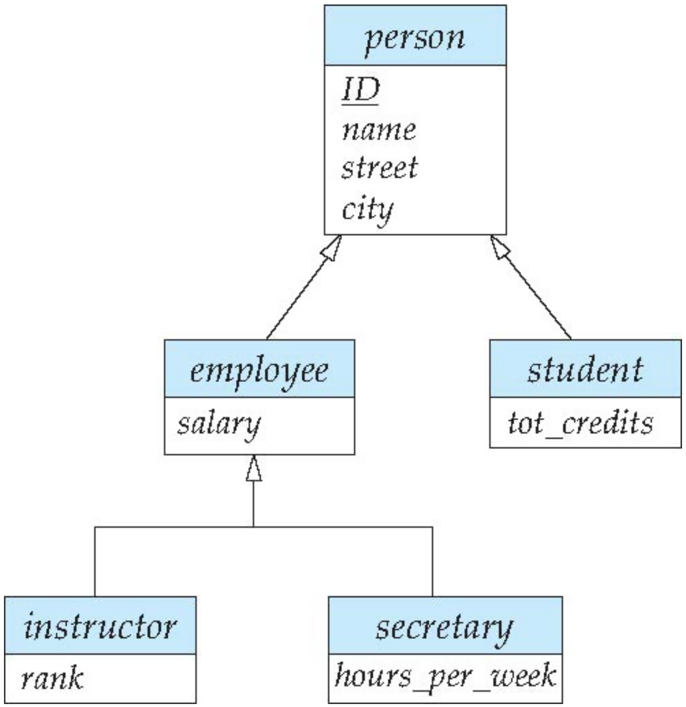
    eval_for (<u>*s_ID, project_id, i_ID, evaluation_id*</u>*, grade*)
  - The schema *proj_guide* is redundant.

Representing Specialization

- Method 1
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

  - Drawback: getting information about an *employee* might require accessing two relations, the one corresponding to the low-level schema, and the one corresponding to the high-level schema

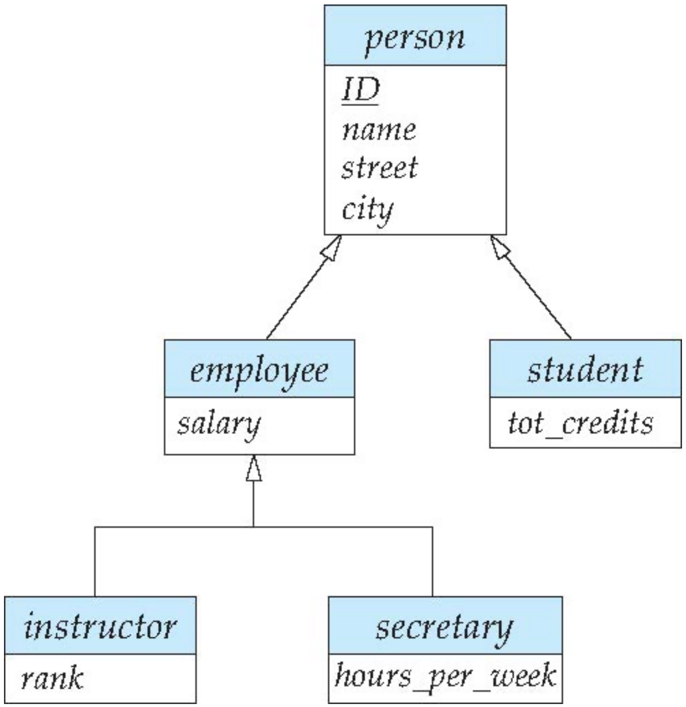


Representing Specialization (continued)

- <u>Method 2</u>
  - Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

  - Drawback: *name, street* and *city* may be stored redundantly for people who are both students and employees (TAs)



Relational Model: Summary

- A tabular representation of data (attributes and tuples).
- Integrity constraints can be specified based on requirements
  - DBMS checks for violations.
  - Two important ICs: primary and foreign keys
  - In addition, we always have domain constraints.
- Powerful and easy-to-use query languages.
- There are rules to translate E-R to relational model
- Simple and intuitive, widely used.

Lab: Translating E-R Diagrams to Relational Schemas

Parts 1 & 2

To submit: Part 2 only

Outline SQL 1

- Overview of The SQL Query Language
- SQL Data Definition
- Basic Query Structure of SQL Queries
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Modification of the Database

From Theory to Practice!

- Relational Algebra (together with Relational Calculus) provides a strong theoretical foundation for relational model, but it's success is mostly due to a powerful Query Language
- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)

History

- IBM began developing commercial products based on their System R
- In June 1979, Relational Software, Inc. introduced the first commercially available implementation of SQL, Oracle V2 (Version2) for VAX computers.
- ANSI and ISO standard SQL:
  - SQL- 86 (first!), then 89, 92, 99, 2003, 2006, 2011 and 2016
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
- Each system has its own “Lingo”, we are using MySQL v8 in class
  - <u>Your answers need to be correct in MySQL version specified</u>

SQL Parts

- **Data-definition language (DDL)**
  - The SQL DDL provides commands for defining relation schemas and view, deleting relations, and modifying relation schemas. It includes commands for specifying integrity constraints.
- **Data-manipulation language (DML)**
  - The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database. Updates that violate integrity constraints are disallowed.
- **Transaction control**. SQL includes commands for specifying the beginning and end points of transactions.
- **Embedded** SQL and **dynamic** SQL. Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C++, Java, and Python
- **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The integrity constraints
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL

The SQL standard supports a variety of built-in types, including

- **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* (overall) digits, with *d* digits to the right of decimal point. (ex., **numeric**(3,1), allows 44.5 to be stored exactly, but not 444.5 or 0.32)
- **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 Construct

- An SQL relation is defined using the **create table** command:

  create table *r*

  $(A_1\ D_1, A_2\ D_2, ..., A_n\ D_n,$
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$));

  - *r* is the name of the relation
  - each A_i is an attribute name in the schema of relation *r*
  - D_i is the data type of values in the domain of attribute A_i
- Example:

  create table *instructor* (
  ID **char**(5),
  name **varchar**(20)**,**
  dept_name **varchar**(20),
  salary **numeric**(8,2));

Integrity Constraints in Create Table

- Types of integrity constraints
  - **primary key** (A_1, ..., A_n)
  - **foreign key** (A_m, ..., A_n) **references** *r*
  - **not null**
- SQL prevents any update to the database that violates an integrity constraint.
- Example:

create table *instructor* (
ID **char**(5),
name **varchar**(20) **not null,**
dept_name **varchar**(20),
salary **numeric**(8,2),
primary key (*ID*),
foreign key *(dept_name*) **references** *department);*

And a Few More Relation Definitions

- **create table** *student* (
  ID **varchar**(5),
  name **varchar**(20) not null,
  dept_name **varchar**(20),
  tot_cred **numeric**(3,0),
  primary key *(ID),*
  foreign key *(dept_name*) **references** *department*);

- **create table** *takes* (
  ID **varchar**(5),
  course_id **varchar**(8),
  sec_id **varchar**(8),
  semester **varchar**(6),
  year **numeric**(4,0),
  grade **varchar**(2),
  primary key *(ID, course_id, sec_id, semester, year)* ,
  foreign key (*ID*) **references** *student,*
  foreign key (*course_id, sec_id, semester, year*) **references** *section*);

And still more ..

- **create table** *course* (
  course_id **varchar**(8),
  title **varchar(**50),
  dept_name **varchar**(20),
  credits **numeric**(2,0),
  primary key *(course_id),*
  foreign key *(dept_name*) **references** *department*);

Deleting and altering tables

- **Drop Table**
  - **drop table** *r*
- **Alter**
  - **alter table** *r* **add** *A D*
    - *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
    - All exiting tuples in the relation are assigned *null* as the value for the new attribute.
  - Example:
    - ALTER TABLE Customers
      ADD Email varchar(255);
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes **not** supported by many databases.

Basic Query Structure

- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i represents an attribute
  - r_i represents a relation
  - P is a predicate.
- The result of an SQL query is a relation.

The select Clause

- The **select** clause lists the attributes desired in the result of a query
- Example: find the names of all instructors:

  select *name*
  from *instructor*

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g., *Name* ≡ *NAME* ≡ *name*
  - You can use cases to improve readability of your SQL code.
    - Ex. Some people use upper case wherever we use bold font.
    - But: sometimes the case sensitivity of the underlying operating system plays a part in the case sensitivity of database and table names, in that case
      - Variable_name='lower_case_table_names'

The select Clause: Distinct and All

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, use the keyword **distinct** after select**.**
- Find the department names of all instructors, and remove duplicates

  select distinct *dept_name*
  from *instructor*

- The keyword **all** specifies that duplicates should not be removed (this is a default and thus very rarely explicitly specified in practice).

  select all *dept_name*
  from *instructor*

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

The select Clause: arithmetic operations and aliases

- The **select** clause can contain arithmetic expressions involving
  - operations: +, –, ∗, and /
  - standard function (type specific)

  operating on constants or attributes of tuples.
- The query:

  select *ID, name, salary/12*
  from *instructor*

  would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.
- Can rename "s*alary/12*" using the **as** clause:

  select *ID, name, salary/12* **as** *monthly_salary*

The select Clause: use of * and literals

- An asterisk in the select clause denotes "all attributes"

  select *
  from *instructor*

- An attribute can be a literal

  select 'A'
  from *instructor*

  - Result is a table with one column and *N* rows (number of tuples in the *instructors* table), each row with value "A"

- An attribute can be a literal with no **from** clause

  select '437'

  - Results is a table with *one* column and a single row with value "437"
  - Can give the column a name using:

    select '437' **as** *FOO*

The where Clause

- The **where** clause allows us to select only those rows in the result that satisfy a specified conditions
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.'

- SQL allows the use of the logical connectives **and, or,** and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 80000

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000

name
Katz
Brandt

The from Clause

- Queries often need to access information from multiple relations.
- Example: “Retrieve the names of all instructors, along with their department names and department building name”

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
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

- To answer the query, each tuple in the instructor relation must be matched with the tuple in the department relation with matching dept_name
- In SQL, to answer the above query, we list the relations that need to be accessed in the from clause and specify the matching condition in the where clause.

select name, instructor.dept_name, building
from instructor, department
where instructor.dept_name=department.dept_name;

Putting It Together

- A SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i represents an attribute
  - r_i represents a relation
  - P is a predicate.
- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the from clause.
- The result of an SQL query is a relation.

Semantic of the SQL Query - How It Works

Clauses operate in this order

1. **from**
2. **where**
3. **select**

Semantic of the SQL Query (continued)

Example: Given relations

instructor(ID, name, dept_name, salary)

teaches (ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

list all instructors and courses that they teach. The SQL query is:

select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;

(1) from

The from clause defines a Cartesian product of the relations listed in the clause
In our example:

(instructor.ID, instructor.name, instructor.dept_name, instructor.salary, teaches.ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

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
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018

Semantic of the SQL Query (continued)

Example: Given relations

instructor(ID, name, dept_name, salary)

teaches (ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

list all instructors and courses that they teach. The SQL query is:

select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;

(1) from

(2) where

Only tuples that satisfy condition of the predicate in where clause remains.

In our example, only tuples where **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
...	...	...	...	...	...	...	...	...

Semantic of the SQL Query (continued)

Example: Given relations

instructor(ID, name, dept_name, salary)
teaches (ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)

list all instructors and courses that they teach. The SQL query is:

select name, course_id
from instructor, teaches
where instructor.ID= teaches.ID;

(1) from

(2) where

(3) select

Finally, only expressions specifies in the select clause remains

name	*course_id*
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Example

- Find the names of all instructors in the Art department who have taught some course and the course_id

  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID* ***and*** *instructor. dept_name =* 'Art'

The Rename Operation

- The names of the attributes in the result set are derived from the names of the attribute in the relations in the from clause, however SQL allows renaming attributes and relations using the **as** clause:

  old-name **as** *new-name*

Example

select **T.**name **as instructor_name**, S.course_id

from instructor **as T**, teaches **as S**

where **T.**ID= **S.**ID;

- Keyword **as** is optional and may be omitted

  "*instructor* **as** *T" => "instructor T"*

- Motivation for attribute rename
  - Two relations in the from clause may have attributes with the same name
  - If we use an arithmetic expression in the select clause, the resultant attribute does not have a name
  - Readability

The Rename Operation (continued)

- Motivation for relation rename
  - replace a long relation name with a shortened version for readability
    - where **T.**ID= **S.**ID;
  - when comparing tuples in the same relation (**self-join**)
- Self-join example
  - "Find the names of all instructors who have a higher salary than some instructor in 'Music' "

    select distinct *T.name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary > S.salary* **and** *S.dept_name* = '*Music*'

T.ID	*T.name*	*T.dept_name*	*T.salary*	*S.ID*	*S.name*	*S.dept_name*	*S.salary*
10101	Srinivasan	Comp. Sci.	65000	12121	Wu	Finance	90000
10101	Srinivasan	Comp. Sci.	65000	15151	Mozart	Music	40000
...	...	...	...	...	...	...	...

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any one character.
- Find the names of all instructors whose name includes the substring "dar".

  select *name*
  from *instructor*
  where *name* **like** '%dar%'

- Match the string "100%"

  like '100 \%'

  in that above we use backslash (\) as the escape character.

String Operations (continued)

- Patterns are case sensitive.
- Pattern matching examples:
  - 'Intro%' matches any string beginning with “Intro”.
  - '%Comp%' matches any string containing “Comp” as a substring.
  - '_ _ _' matches any string of exactly three characters.
  - '_ _ _ %' matches any string of <u>at least </u>three characters.

- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - Converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- List the names of all instructors in the alphabetic order

  select distinct *name*
  from *instructor*
  order by *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the **default**.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*

Where Clause Predicates

- SQL includes a **between** comparison operator
  - It is inclusive for both ends of the range
- Example: Find the names of all instructors with salary between $90,000 and $100,000
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000
- Another useful feature is a tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor*.*ID*, *dept_name*) = (*teaches*.*ID*, 'Biology');

  Equivalent to
    where *instructor*.*ID*= *teaches*.*ID* **and** *dept_name* = 'Biology';
  - Also supports other comparison operators, ex <=, >, etc.

Set Operations

- SQL supports **union, intersect,** and **except** set operations
- Find names of all students and professors in the Music department

  (**select** *name* **from** *student* **where** *dept_name =* 'Music')
  union
  (**select** *name* **from** *instructor* **where** *dept_name =* 'Music')

- Resulting *schemas* must be identical in two **selects,** but *from* clauses may be different
- Find courses that ran in Fall 2017 and in Spring 2018

  (**select** *course_id* **from** *section* **where** *semester =* 'Fall' **and** *year =* 2017)
  intersect
  (**select** *course_id* **from** *section* **where** *semetser =* 'Spring' **and** *year =* 2018)

- Find courses that ran in Fall 2017 but not in Spring 2018

  (**select** *course_id* **from** *section* **where** *sem ester=* 'Fall' **and** *year =* 2017)
  except
  (**select** *course_id* **from** *section* **where** *semester =* 'Spring' **and** *year =* 2018)

Set Operations (continued)

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the
  - **R1 union all R2**
    - number of duplicates in the result = s1+s2
  - **R1 intersect all R2**
    - number of duplicates in the result = min(s1, s2)
  - **R1 except all R2**
    - number of duplicates in the result = max (s1-s2, 0)

where s1 and s2 are the numbers of duplicates in the results of the first and the second **select** respectively (R1 and R2)

Arithmetic with Null Values

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an unknown value or that a value does not exist.
- Creates complications on several levels
- Arithmetic expressions
  - The result of any arithmetic expression involving **null** is **null**
  - Example: 5 + **null** returns **null**

Comparison with Null Values

- SQL treats as **unknown** the result of any comparison involving a null value
  - Example*: 5 <* **null** or **null** <> **null** or **null** = **null**
- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
  - **and** : *(true* **and** *unknown) = unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - **or:** (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown) = unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Checking for null Values: How to deal with it

- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

    select *name*
    from *instructor*
    where *salary* **is null**
- The predicate **is not null** succeeds if the value on which it is applied is not null.
- In some databases, to test whether the result of a comparison is **unknown**, rather than true or false, use **is unknown** and **is not unknown**.
- <u>**Exception**</u>: When a query uses the **select distinct** clause to eliminate duplicate tuples, the attribute values are treated as identical if either both are non-null and equal, or both are null.
  - Ex . ('A', null) and ('A', null) are treated as being identical
  - This is an exception from a "(**null** = **null)** is **unknown"** logic

Aggregate Functions

- These functions operate on values from one column and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*) **as** *avg_salary*
    from *instructor*
    where *dept_name*= 'Comp. Sci.';

- (2) Find the total number of instructors who teach a course in the Spring 2018 semester
  - **select count** (**distinct** *ID*)
    from *teaches*
    where *semester* = 'Spring' **and** *year* = 2018;

- (3) Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

Aggregation with Grouping (group by)

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
    from *instructor*
    group by *dept_name*;
- Execution consists of two steps
  - Grouping based on the values **of group by** attribute

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
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

  - Executing query for each group

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregation with Grouping (continued)

- The query can have multiple relations in the **from** clause and a **where** clause
- Example: Find the number of instructors in each department who teach a course in the Spring 2018 semester.

  select *dept_name*, **count** (**distinct** *ID*) **as** *instr_count*

  from *instructor*, *teaches*

  where *instructor.ID*= *teaches.ID* and

  semester = 'Spring' **and** *year* = 2018

  group by *dept_name*;

dept_name	*instr_count*
Comp. Sci.	3
Finance	1
History	1
Music	1

Aggregation with Grouping (Cont.)

- All attributes in **select** clause outside of aggregate functions MUST appear in **group by** list

  - **select** *dept_name*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

  - /* erroneous query */
    select *dept_name*, *ID*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

Aggregate Functions: having Clause

- Sometimes there is a condition that applies to groups rather than to tuples
  - Find the names and average salaries of all departments whose average salary is greater than 42000

select *dept_name*, **avg** (*salary*) **as** *avg_salary*
from *instructor*
group by *dept_name*
having avg (*salary*) > 42000;

dept_name	*avg_salary*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Aggregate Functions: having Clause (continued)

- Predicates in the **having** clause are applied *after* the formation of groups whereas predicates in the **where** clause are applied *before* forming groups
- Find the average total credits (tot_cred) of all students enrolled in the section, if the section has at least 2 students.

  select course_id, semester, year, sec_id, **avg** (tot_cred)
  from student, takes
  where student.ID= takes.ID
  group by course_id, semester, year, sec_id
  having count (ID) >= 2;

Aggregate Functions: order of Execution

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.

2. If a **where** clause is present, the predicate in the where clause is applied on the result relation of the from clause.

3. Tuples satisfying the where predicate are then placed into groups by the **group by** clause if it is present. If the group by clause is absent, the entire set of tuples satisfying the where predicate is treated as being in one group.

4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the having clause predicate are removed.

5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

Aggregation with Null

- Null values, when they exist, complicate the processing of aggregate operators.
- General rule: all aggregate functions except count () ignore null values in their input collection
- In the example:

  select sum (salary)
  from instructor;

  sum operator will ignore null values in its input
- When the * is used for **count**(), all records are counted
- If **count** (*column_name*) is used, only rows where *column_name* is **not null** are counted

Issues with the null value: Summary

- **where** and **having** clause eliminates rows (groups) for which the qualification does not evaluate to true (i.e., evaluate to false or unknown).
- Aggregate functions ignore null values (except **count**(*)).
- **distinct** treats all null values as the same.
- The arithmetic operations +, -, *, / return null if one of the arguments is null.

MySQL Syntax

- End of statement: **;**
- MySQL Server supports three comment styles:
  - From a # character to the end of the line, e.g.
    - SELECT 1+1; **# This comment continues till the end of the line**
  - From a -- sequence to the end of the line. In MySQL, the -- (double-dash) comment style requires the second dash to be followed by at least one whitespace or control character (such as a space, tab, newline, and so on), e.g.
    - SELECT 1 **/* this is an in-line comment */** + 1;
  - From a /* sequence to the following */ sequence, as in the C programming language. This syntax enables a comment to extend over multiple lines because the beginning and closing sequences need not be on the same line, e.g.
    - **/* this is**
    - **a multiple-line**
    - **comment *** /