W4111 – Introduction to Databases
Section 002, Spring 2026
Lecture 2: Foundation – ER, Relational, SQL (I)

Contents

- Show HW tips and guidelines on Ed.

Contents

- ER Model/Modeling:
  - More details
  - An example.
- A deeper dive on *keys.*
  - Formal model for keys.
  - Some practical concepts for keys.
- Relational Algebra:
  - π, σ examples
  - X (Cartesian Product)
  - $\bowtie$ (JOIN) operator
- Continuing SQL
  - More details on SELECT
  - Cartesian Product
  - Join

Entity Relationship
Entity Relationship Models

Modeling (Reminder/Review)

Modeling According to Wikipedia

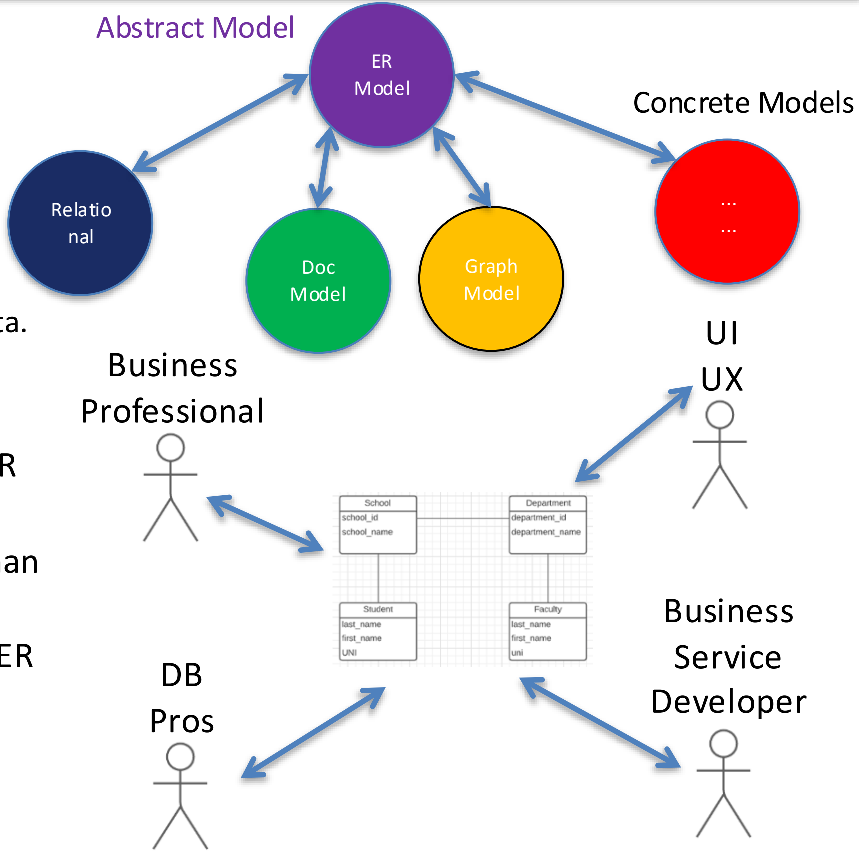
- Model-driven engineering (MDE) is a software development methodology that focuses on creating and exploiting domain models, which are conceptual models of all the topics related to a specific problem. Hence, it highlights and aims at abstract representations of the knowledge and activities that govern a particular application domain, rather than the computing (i.e. algorithmic) concepts.

- In software engineering, a domain model is a conceptual model of the domain that incorporates both behavior and data.

- A modeling language is any artificial language that can be used
express data, information or knowledge or systems in a structure that is defined by a consistent set of rules. A modeling language can be graphical or textual.
  - Graphical modeling languages use a diagram technique with named symbols that represent concepts and lines that connect the symbols and represent relationships and various other graphical notation to represent constraints.
  - Textual modeling languages may use standardized keywords accompanied by parameters or natural language terms and phrases to make computer-interpretable expressions.

ER Model and ER Modeling

- ER Model: Agility, Separation of Concerns
  - ER model is a generalization that most DB models implement in some form.
  - Using the ER model enables:
    - Thinking about and collaborating on design with getting bogged down in details.
    - Enable flexible choices about how to realize/Implement data.
- ER Diagrams: Communication, Quality, Precision
  - With a little experience, everyone can understand and ER diagram.
  - Easier to discuss and collaborate on application's data than showing SQL table definitions, JSON, ...
  - People think visually. That is why we have whiteboards. ER diagrams are precise and unambiguous.
  - Guides you to think about relationships, keys, ... And prevents "re-dos" later in the process. It is easier to fix a diagram than a database schema.



ER Modeling – Reasonably Good Summary

Advantages of ER Model

Conceptually it is very simple: ER model is very simple because if we know relationship between entities and attributes, then we can easily draw an ER diagram.

Better visual representation: ER model is a diagrammatic representation of any logical structure of database. By seeing ER diagram, we can easily understand relationship among entities and relationship.

Effective communication tool: It is an effective communication tool for database designer.

Highly integrated with relational model: ER model can be easily converted into relational model by simply converting ER model into tables.

Easy conversion to any data model: ER model can be easily converted into another data model like hierarchical data model, network data model and so on.

Disadvantages of ER Model

Limited constraints and specification

Loss of information content: Some information be lost or hidden in ER model

Limited relationship representation: ER model represents limited relationship as compared to another data models like relational model etc.

No representation of data manipulation: It is difficult to show data manipulation in ER model.

Popular for high level design: ER model is very popular for designing high level design

No industry standard for notation

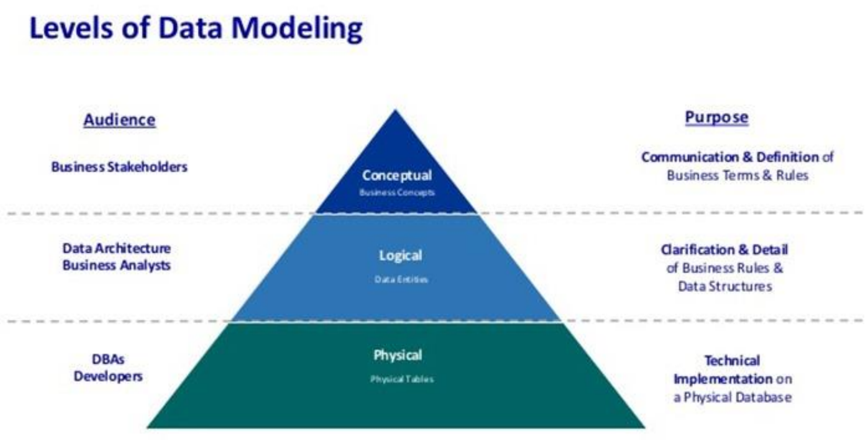
https://pctechnicalpro.blogspot.com/2017/04/advantages-disadvantages-er-model-dbms.html

Note:

- If you get to use Google to help with take home exams, HW, etc.
- I get to use Google to help with slides.

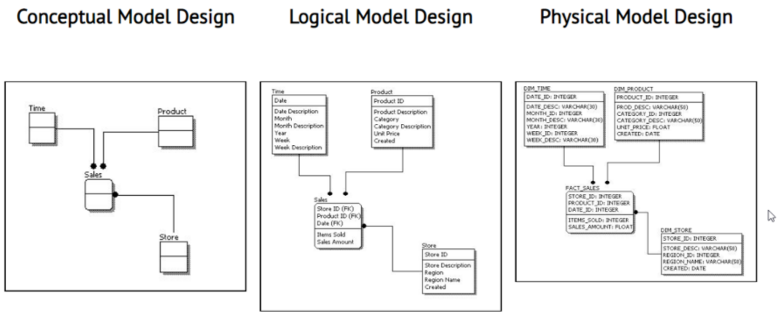
A Common and my Approach: Conceptual → Logical → Physical

https://ehikioya.com/conceptual-logical-physical-database-modeling/



https://www.1keydata.com/datawarehousing/data-modeling-levels.html

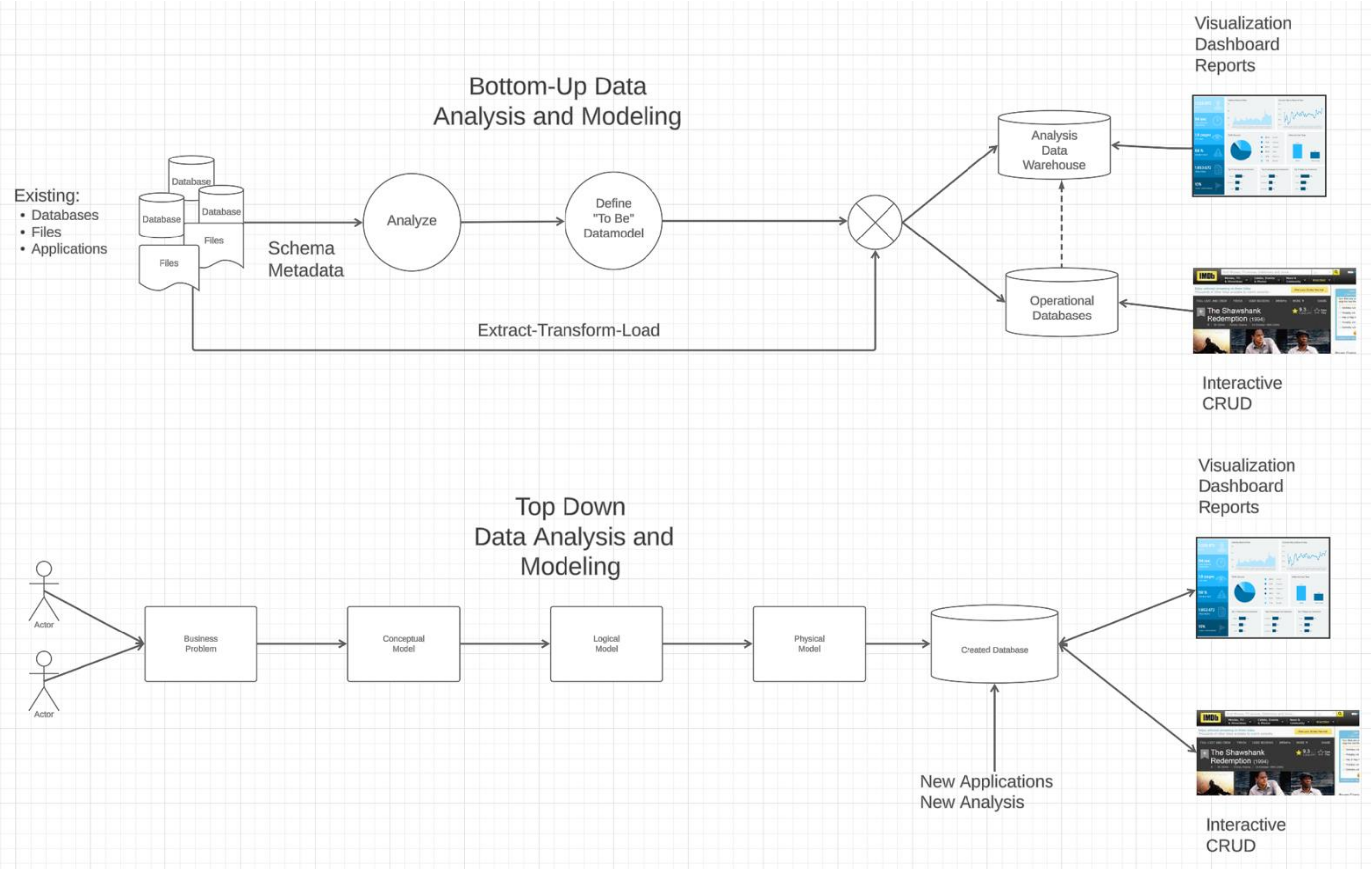
Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓



- It is easy to get carried away with modeling. You can spend all your time modeling and not actually build the schema.
- We will use the approaches in class.
- Mostly to understand concepts and patterns.

https://www.1keydata.com/datawarehousing/data-modeling-levels.html

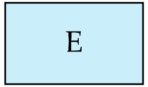
Modeling



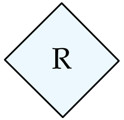
Most of the time there is a mix → Meet-in-the-Middle

There are Many Languages/Notations

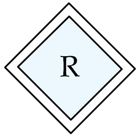
Summary of Symbols Used in E-R Notation



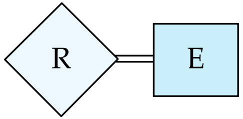
entity set



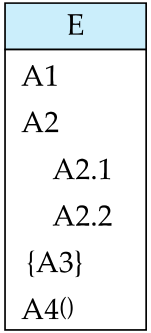
relationship set



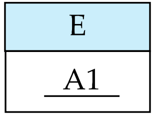
identifying
relationship set
for weak entity set



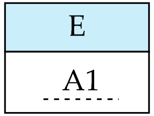
total participation
of entity set in
relationship



attributes:
simple (A1),
composite (A2) and
multivalued (A3)
derived (A4)

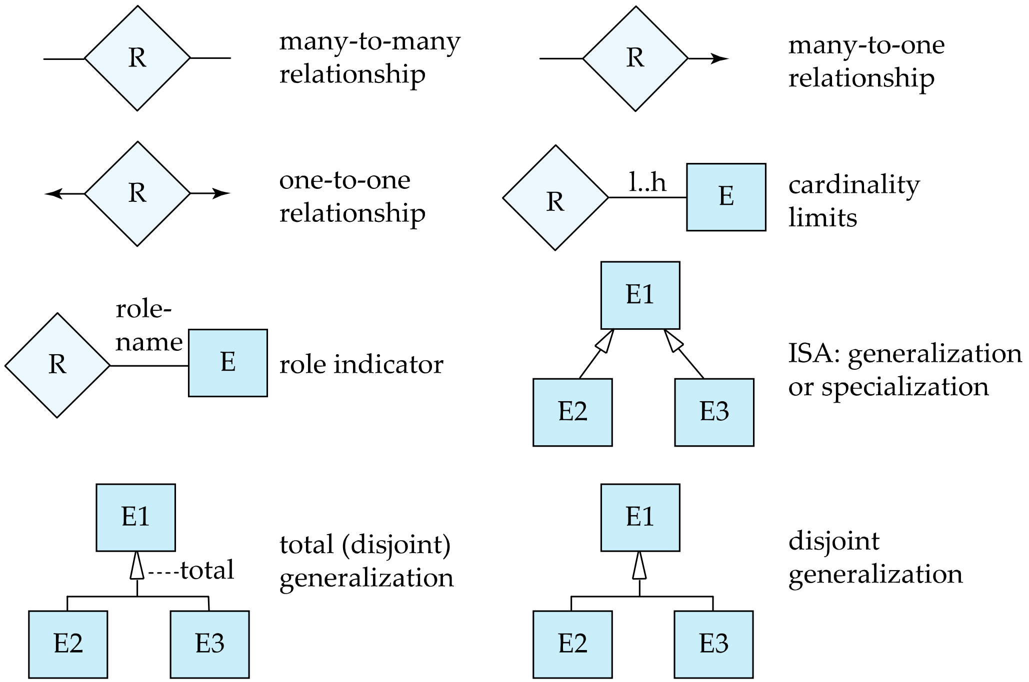


primary key

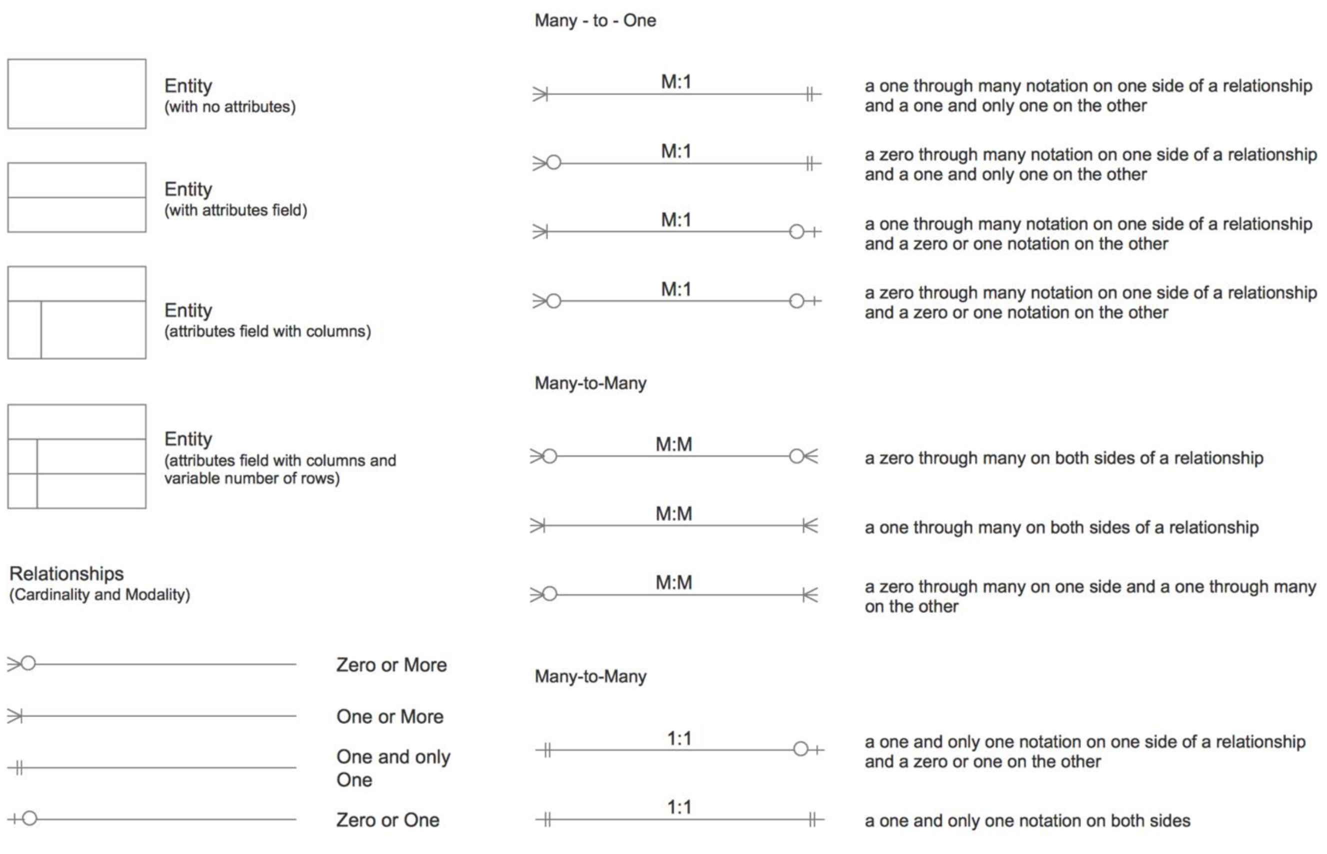


discriminating
attribute of
weak entity set

Symbols Used in E-R Notation (Cont.)



Crow’s Foot Notation

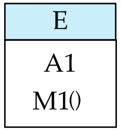


UML

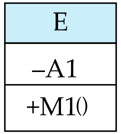
- **UML**: Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.

ER vs. UML Class Diagrams

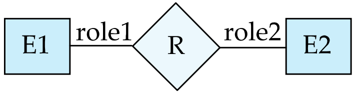
ER Diagram Notation | **Equivalent in UML**



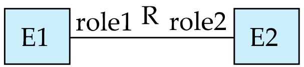
entity with attributes (simple, composite, multivalued, derived)

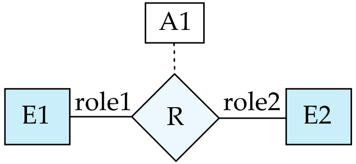


class with simple attributes and methods (attribute prefixes: + = public, – = private, # = protected)

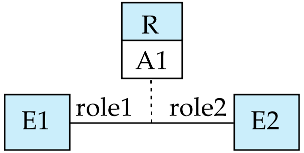


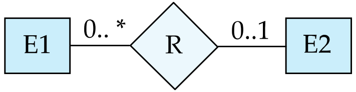
binary relationship



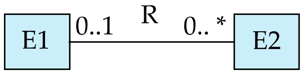


relationship attributes





cardinality constraints

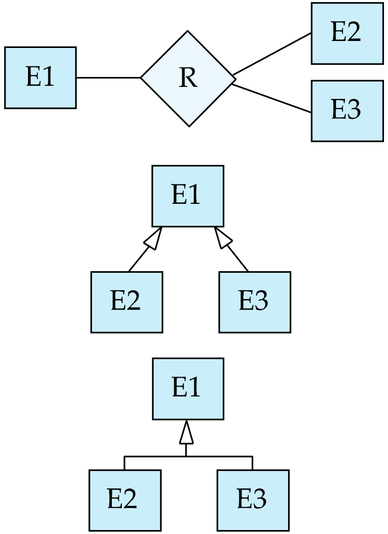


* Note reversal of position in cardinality constraint depiction

ER vs. UML Class Diagrams

ER Diagram Notation

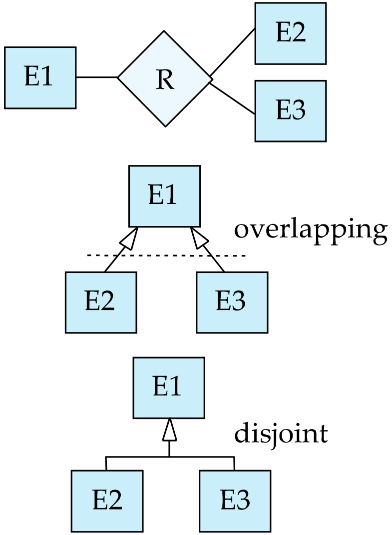
Equivalent in UML



n-ary relationships

overlapping generalization

disjoint generalization



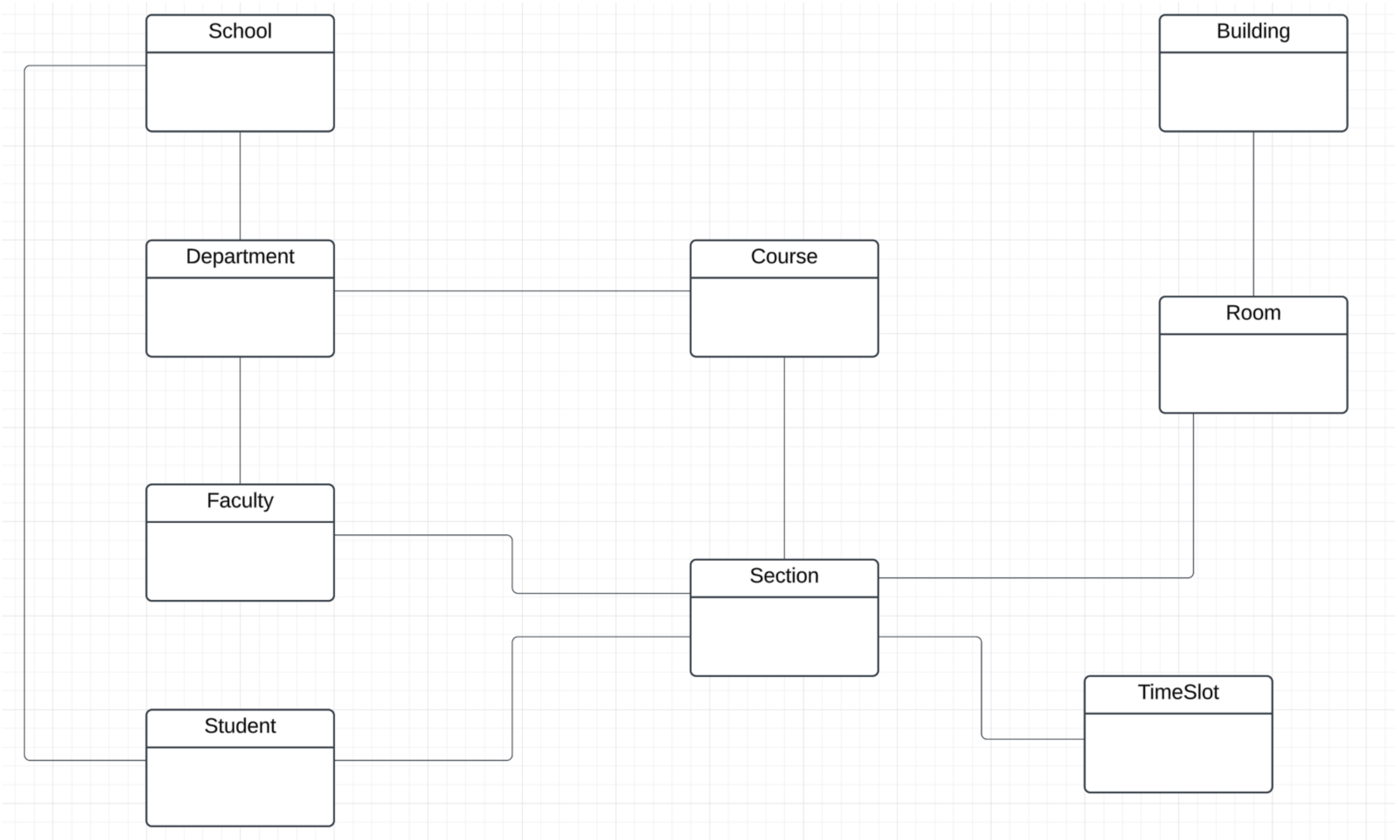
* Generalization can use merged or separate arrows independent of disjoint/overlapping

Modeling Summary

- If you are in a project/team meeting and draw a diagram on the whiteboard/chalk board, you are probably “visual modeling.”
- A visual notation is simply an agreement on precisely defining what the symbols, lines, colors, … … “mean.”
- Even within a well-defined notation, people are often lazy or class but use slightly different interpretations/notations.
- The two key concepts are:
  - Everyone in the process agrees on the meanings and adheres to them.
  - You do “just enough” modeling:
    - No modeling ➔ You are randomly, frantically coding like a slam poet.
    - Too much modeling ➔ You spend all of your time arguing and never deliver the project.
- We will take an approach to Crow’s Foot Notation in the class, but will also examine the modeling language that comes with the book. *This is a good fit with relational databases and relational model concepts.*

Our First Example

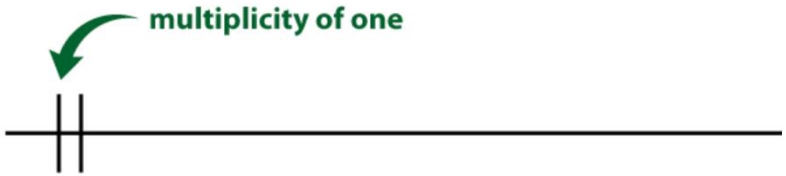
Modeling Columbia Courses – Conceptual

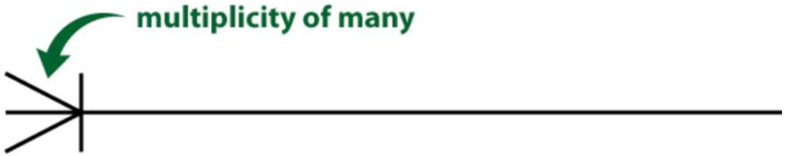


Crow’s Foot Notation

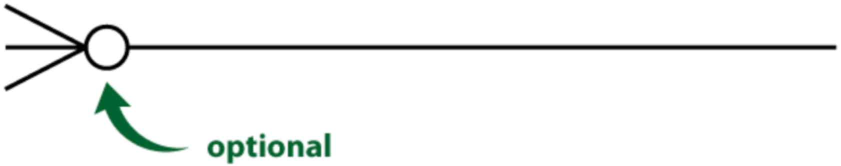
Relationships have two indicators. These are shown on both sides of the line.

- The first one (often called multiplicity) refers to the *maximum* number of times that an instance of one entity can be associated with instances in the related entity. It can be one or many.



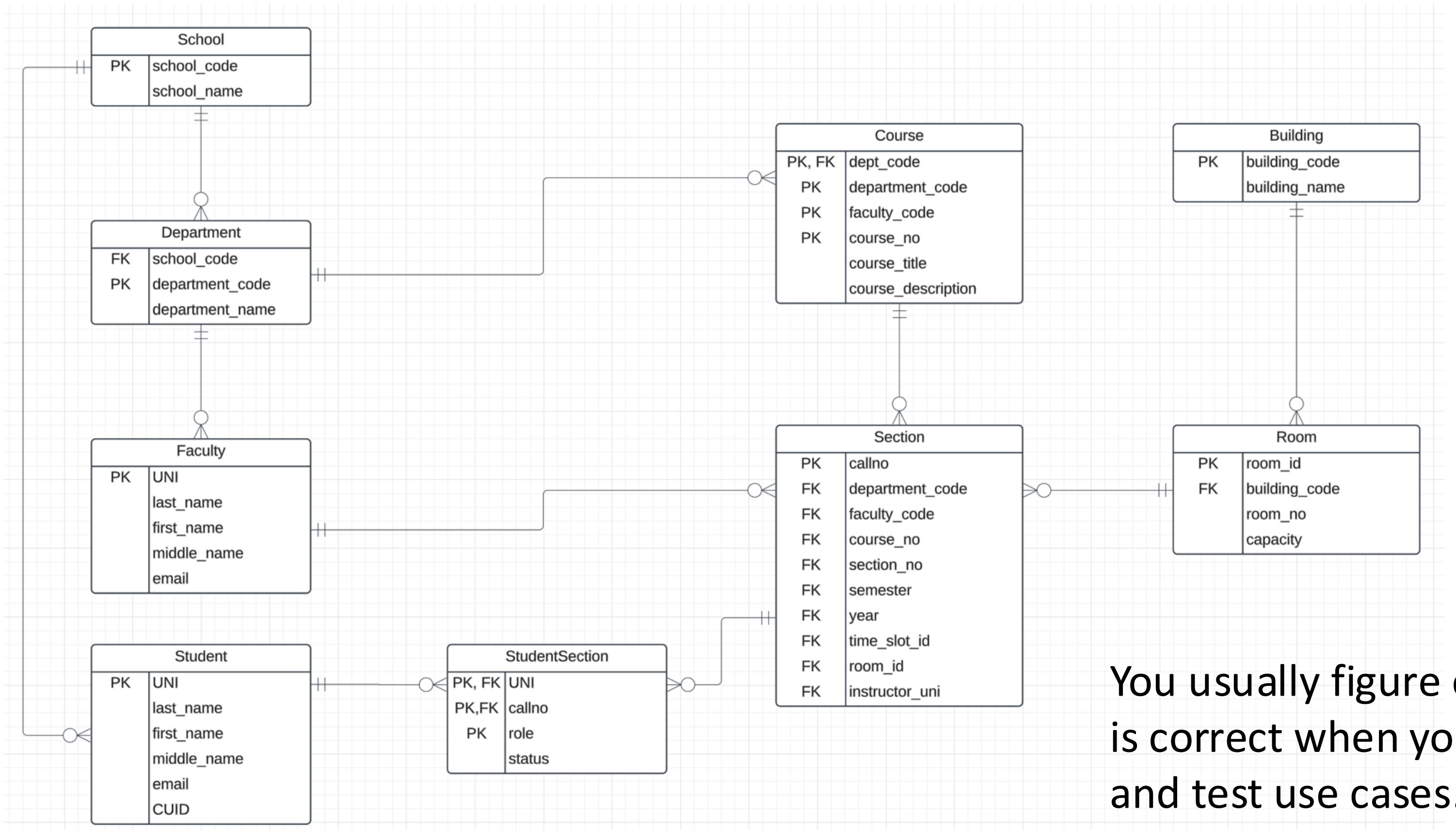


- The second describes the *minimum* number of times one instance can be related to others. It can be zero or one, and accordingly describes the relationship as optional or mandatory.



https://vertabelo.com/blog/crow-s-foot-notation/

Modeling Columbia Courses – Logical

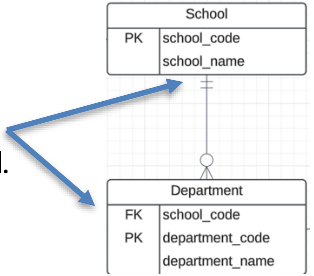


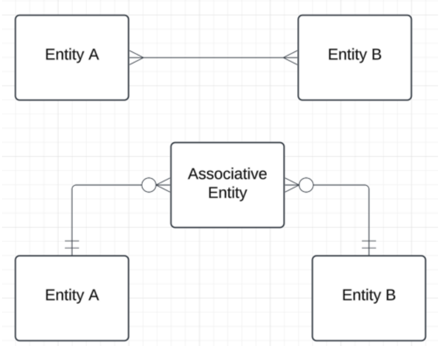
You usually figure out if this is correct when you define in SQL and test use cases.

Some Observations

- Foreign keys can be confusing and seem to sometimes be on the wrong end.
- Mandatory on both ends may require two new entities. The SQL INSERT statement operates on one table making INSERT complex. The data is temporarily inconsistent/invalid.
- Many-to-Many in SQL requires a 3rd entity called an Associative Entity.

For Example
Mandatory on this end
implies
Not-NULL, FK on this end.





Keys

What are these Keys of which You Speak?

Keys

- Let $K \subseteq R$
- *K* is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
  - Example: {*ID*} and {ID,name} are both superkeys of *instructor.*
- Superkey *K* is a **candidate key** if *K* is minimal
  Example: {*ID*} is a candidate key for *Instructor*
- One of the candidate keys is selected to be the **primary key**.
  - which one?
- **Foreign key** constraint: Value in one relation must appear in another
  - **Referencing** relation
  - **Referenced** relation
  - Example: *dept_name* in i*nstructor* is a foreign key from *instructor* referencing *department*

Our Model

- Student has three candidate keys:
  - UNI
  - CUID
  - email
- Faculty has two candidate keys:
  - UNI
  - email
- Section has two candidate keys:
  - callno
  - The combination of (dept_code, faculty_code, course_no, section_no, semester, year)
- Some types of key:
  - UNI, CUID, callno are *surrogate keys* as opposed to *natural keys.*
  - email and (dept_code, faculty_code, course_no, section_no, semester, year) are natural keys,

Types of/Patterns for Keys

Key Type	Definition	Example
Primary Key	Uniquely identifies a record, no duplicates.	Employee ID, Student Roll Number
Super Key	A combination of attributes that can uniquely identify records.	Employee ID + Name
Candidate Key	A minimal super key that can act as a primary key.	Employee ID, Email Address
Alternate Key	A candidate key is not chosen as the primary key.	Email Address (if Employee ID is primary)
Foreign Key	Links tables by referencing a primary key from another table.	Department ID in Employee Table
Composite Key	Multiple columns are used together to identify records uniquely.	Course ID + Student ID
Unique Key	Ensures uniqueness in a column but allows one null value.	Email Address, Phone Number

Primary Keys are Recommended to be immutable.

You may also here the term *compound key* ➔ Each of the attributes is a foreign key.

Relational Model
Relational Algebra

Relational Algebra

- A procedural (? Declarative) language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- Six basic operators
  - select: σ
  - project: Π
  - union: $\cup$
  - set difference: –
  - Cartesian product: x
  - ~~JOIN ($\bowtie$)~~
  - rename: ρ

Covered last week.

- Let's do some practice with the dreaded RelaX calculator.
- Note that the RelaX calculator has an extended set of noncore operators.

Let’s Watch Professor Ferguson Do Examples

- Let’s do some practice with the dreaded RelaX calculator.
- Pay attention to see how I use the “out of data” dataset associated with the recommended textbook.
- Note that the RelaX calculator has an extended set of noncore operators.
- Get the names of the *departments* with a budget >= 100000 or in the building ‘Painter’

  π dept_name

  (σ budget ≥ 100000 ∨ building = 'Painter' (department))
- Rename attributes and operate on attributes in π

  π the_course_id←course_id,

  sec_id, year, something_weird←sec_id+year

  (section)

Notes:

- Select comparators from the menu, e.g. ≥.
- Single quote matters.
- Unlike SQL, strings are case sensitive.
- Formally, relational algebra use subscript but this is hard to type.

The Dreaded Relax Calculator

- Let's look at an online tool that you will use.
- RelaX (https://dbis-uibk.github.io/relax/calc/local/uibk/local/0)
- The calculator:
  - Has an older version of the data from the recommended textbook. (https://dbis-uibk.github.io/relax/calc/gist/4f7866c17624ca9dfa85ed2482078be8/relax-silberschatz-english.txt/0)
  - You can also upload new data.
- Some queries:
  - σ dept_name='Comp. Sci.' ∨ dept_name='History' (department)
  - π name, dept_name (instructor)
  - π ID, name (
    σ dept_name='Comp. Sci.' (instructor)
)

Cartesian-Product Operation

- The Cartesian-product operation (denoted by X) allows us to combine information from any two relations.
- Example: the Cartesian product of the relations *instructor* and *teaches* is written as:

  instructor X *teaches*

- We construct a tuple of the result out of each possible pair of tuples: one from the *instructor* relation and one from the *teaches* relation (see next slide)
- Since the instructor *ID* appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
  - *instructor.ID*
  - *teaches.ID*

R x R -> R   f(x,y)->z

Composition of Relational Operations

- The result of a relational-algebra operation is relation and therefore of relational-algebra operations can be composed together into a **relational-algebra expression**.
- Consider the query -- Find the names of all instructors in the Physics department.

$$\Pi_{name}(\sigma_{dept\_name = \text{“Physics”}}\ (instructor))$$

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

The *instructor* x *teaches* table

- This only sort of makes sense. The result is:
  - Every possible combination of the form
  - (instructor, teaches)
  - Even if the instructor is NOT the instructor in the teaches row.
- Examining in SQL makes a little clearer.
  - We will see in later in this lecture.
  - Confusingly, in SQL cartesian Product is **JOIN but there is also a relational JOIN operator.**

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
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Simpler Example

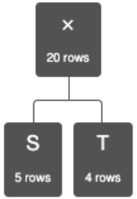
- Assume we have two tables
  - S has two columns, 5 rows.
  - T has two columns, 4 rows.
- S x T has
  - 4 columns.
  - 20 rows.
- Cartesian product does not come up a lot in applications.
- There are cases in optimization and other scenarios in which:
  - You want to generate all possible combinations.
  - Score, rate, rank etc. to determine best choices.

T

T.b	T.d
'a'	100
'd'	200
'f'	400
'g'	120

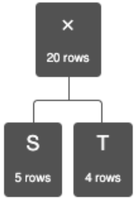
S

S.b	S.d
'a'	100
'b'	300
'c'	400
'd'	200
'e'	150



S × T

S.b	S.d	T.b	T.d
'a'	100	'a'	100
'a'	100	'd'	200
'a'	100	'f'	400
'a'	100	'g'	120
'b'	300	'a'	100
'b'	300	'd'	200
'b'	300	'f'	400
'b'	300	'g'	120
'c'	400	'a'	100
'c'	400	'd'	200



S × T

S.b	S.d	T.b	T.d
'c'	400	'f'	400
'c'	400	'g'	120
'd'	200	'a'	100
'd'	200	'd'	200
'd'	200	'f'	400
'd'	200	'g'	120
'e'	150	'a'	100
'e'	150	'd'	200
'e'	150	'f'	400
'e'	150	'g'	120

Join Operation

- The Cartesian-Product

  instructor X *teaches*

  associates every tuple of instructor with every tuple of teaches.

  - Most of the resulting rows have information about instructors who did NOT teach a particular course.

- To get only those tuples of "*instructor* X *teaches* " that pertain to instructors and the courses that they taught, we write:

  $\sigma_{\,instructor.id\,=\,teaches.id}\ (instructor\ \times\ teaches\ ))$

  - We get only those tuples of "*instructor* X *teaches*" that pertain to instructors and the courses that they taught.

- The result of this expression, shown in the next slide

A fundamental definition:

- $\sigma_{\text{instructor.ID=teaches.ID}}$ (instructor × teaches) = instructor ⋈ teaches
- ⋈ is the JOIN operations.

Join Operation (Cont.)

- The **join** operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.
- Consider relations *r* (*R*) and *s* (*S*)
- Let "theta" be a predicate on attributes in the schema R "union" S. The join operation r $\bowtie_\theta$ s is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta \, (r \times s)$$

- Thus

$$\sigma_{\, instructor.id = \, teaches.id} \, (instructor \;\; x \; teaches \,))$$

- Can equivalently be written as

$$instructor \bowtie_{\, Instructor.id \, = \, teaches.id} teaches.$$

Let’s Watch Professor Ferguson Do Examples

- More fun with RelaX
- Validate the definition with an example
  - σ instructor.dept_name=department.dept_name (instructor × department)
  - instructor ⋈ department
  - instructor ⋈ instructor.dept_name=department.dept_name department
  - Note: You sometimes have to fully qualify attribute names.
- Let’s pull it all together … …

```
σ instructor_department='Comp. Sci.'
(
    π
        instructor_id←instructor.ID, instructor_name←instructor.name,
            instructor_department←instructor.dept_name, s_id
        (instructor ⋈ instructor.ID=i_id advisor)
    ⋈ student_id=s_id
        (π
            student_id←student.ID, student_name←student.name,
            student_department←student.dept_name (student))
    )
```

Notes:

- Show them how to break this down Barney Style.
- Show them approaches to testing and verifying.
- Being able to verify is CRITICAL if you use generative AI.

SQL

SQL Parts

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- integrity – the DDL includes commands for specifying integrity constraints.
- View definition -- The DDL includes commands for defining views.
- Transaction control –includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes commands for specifying access rights to relations and views.

SQL Language Statements

The core SQL language statements are:

- SELECT: Implements both σ, π
- INSERT
- UPDATE
- DELETE
- CREATE TABLE
- ALTER TABLE
- JOIN, which is an operator within SELECT.

$$\pi_{\text{ID, name}}\left(\sigma_{\text{dept\_name='Comp. Sci.'}}(\text{instructor})\right)$$

=

```
SELECT ID, name FROM instructor
    WHERE
    dept_name='Comp. Sci.'
```

- Many, if not most, SQL statements:
  - Implement multiple relational algebra expressions.
  - Cannot easily (or at all) be represented in relational algebra.

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.

Multiple tables and cause confusion and errors because you are doing a cross=product. Be careful

Note:

- The SELECT … FROM … WHERE … Combines two relational operators, σ and $\prod$.
- Actually, it also combines other operators, e.g. x

The select Clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- 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*
  - Some people use upper case wherever we use bold font.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert 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.

  select all *dept_name*
  from *instructor*

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

  select *
  from *instructor*

- 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*

- An attribute can be a literal with **from** clause

  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”

The select Clause (Cont.)

- The **select** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and 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 "*salary/12"* using the **as** clause:

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

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - 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

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  select $*$
  from *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

Examples

- Find the names of all instructors who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID* = *teaches.ID*

- 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'

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.

Create Table Construct

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

  create table *r*

  $(A_1\ D_1,\ A_2\ D_2,\ \ldots,\ A_n\ D_n,$
  $(\text{integrity-constraint}_1),$
  $\ldots,$
  $(\text{integrity-constraint}_k))$

  - *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))

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

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

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.
  - **select distinct** *T.name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary* > *S.salary* **and** *S.dept_name* = *'Comp. Sci.'*

This is an example of using Cartesian Product.

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

  instructor **as** *T* ≡ *instructor T*

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.
- The result of any arithmetic expression involving **null** is **null**
  - Example: 5 + **null** returns **null**
- 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.

Note:

- NULL is an extremely important concept.
- You will find it hard to understand for a while.

Null Values (Cont.)

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**).
  - 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*

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

- *Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.*

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = 'Watson');

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors

  delete from *instructor*
  where *salary* < (**select avg** (*salary*)
  from *instructor*);

  - Problem: as we delete tuples from *instructor*, the average salary changes
  - Solution used in SQL:
    1. First, compute **avg** (salary) and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Insertion

- Add a new tuple to *course*

  insert into *course*
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
  values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values ('3003', 'Green', 'Finance', *null*);

Insertion (Cont.)

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

  insert into *instructor*
  select *ID, name, dept_name, 18000*
  from *student*
  where *dept_name* = 'Music' **and** *total_cred* > 144;

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.

  Otherwise queries like

  insert into *table*1 **select** * **from** *table*1

  would cause problem

Updates

- Give a 5% salary raise to all instructors

```
update instructor
  set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor
   set salary = salary * 1.05
  where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor
 set salary = salary * 1.05
 where salary <  (select avg (salary)
                   from instructor);
```

Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

    update *instructor*
       set *salary* = *salary* * 1.03
       where *salary* > 100000;
    update *instructor*
       set *salary* = *salary* * 1.05
       where *salary* <= 100000;

  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

  update *instructor*
  set *salary* = **case**
  when *salary* <= 100000 **then** *salary* * 1.05
  else *salary* * 1.03
  end

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause
- Three types of joins:
  - Natural join
  - Inner join
  - Outer join

Notes:

- You will also hear terms like equi-join, non-equi-join, theta join, semi-join, ...
- I ask for definitions on exams, but you can just look them up.

Natural Join in SQL

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.
- List the names of instructors along with the course ID of the courses that they taught
  - **select** *name*, *course_id*
    from *students, takes,*
    where *student.ID* = *takes.ID*;
- Same query in SQL with "natural join" construct
  - **select** *name*, *course_id*
    from *student* **natural join** *takes*;

Natural Join in SQL (Cont.)

- The **from** clause can have multiple relations combined using natural join:

  select $A_1, A_2, \ldots A_n$
  from r_1 **natural join** r_2 **natural join .. natural join** r_n
  where P ;

Student Relation

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Takes Relation

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

student natural join *takes*

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly
- Example -- List the names of students instructors along with the titles of courses that they have taken
  - Correct version

    select *name*, *title*
    from *student* **natural join** *takes*, *course*
    where *takes*.*course_id* = *course*.*course_id*;

  - Incorrect version

    select *name*, *title*
    from *student* **natural join** *takes* **natural join** *course*;

    - This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.
    - The correct version (above), correctly outputs such pairs.

Natural Join with Using Clause

- To avoid the danger of equating attributes erroneously, we can use the "**using**" construct that allows us to specify exactly which columns should be equated.
- Query example

  select *name*, *title*
  from (*student* **natural join** *takes*) **join** *course* **using** (*course_id*)

Join Condition

- The **on** condition allows a general predicate over the relations being joined
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**
- Query example

  select *
  from *student* **join** *takes* **on** *student_ID = takes_ID*

  - The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.

- Equivalent to:

  select *
  from *student , takes*
  where *student_ID = takes_ID*

Join Condition (Cont.)

- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**.
- Query example

  select *
  from *student* **join** *takes* **on** *student_ID = takes_ID*

  - The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.
- Equivalent to:

  select *
  from *student , takes*
  where *student_ID = takes_ID*