CS 348: Introduction to Database Management

University of Waterloo

Instructor: Chao Zhang

Winter 2023

Andrew Wang

Table of Contents

Database Management

This course will study data databases from three viewpoints: *database user*, *database designer*, and the *database manager*. It will teach how to use the database management system (DBMS) but treat it like a black box, focusing on its functionality and interfaces. (CS448 database system implementation)

Formal definition of *data* given by ANSI:

- Representation of *facts*, *concepts*, or *instructions* in a formal manner suitable for communication, intrepretation, or processing by humans or computers
- Any representation such as characters or analog quantities to which *meaning is or might be assigned*. Generally, we perform operations on data or data items to supply some information about an entity

Informally, we say data is any information that needs to be recorded in an application.

We primarily concern ourselves with *persistent* data (data that should not be lost with power outages) (*volatile* data is lost after a power cycle).

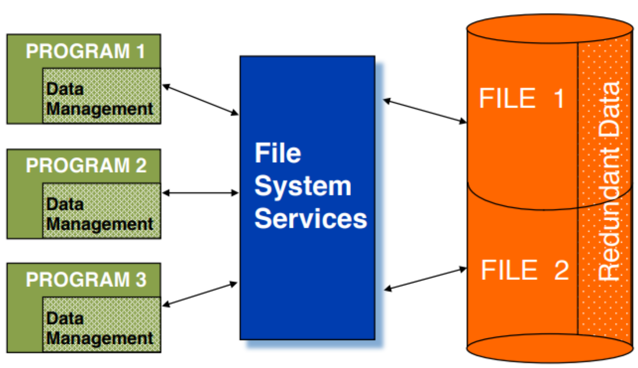
Example: a bank account

- Data: bank account belongs to a branch, have a number, owner, balance, etc.
- Persistency: balance can't disappear after power outage
- Query: what is the balance in Bob's account?
- Modification: Bob withdraws $100

Data Storage

File System

Before databases we had a file processing system, where the data is updated in manually by the program (i.e. write/read directly from some location)



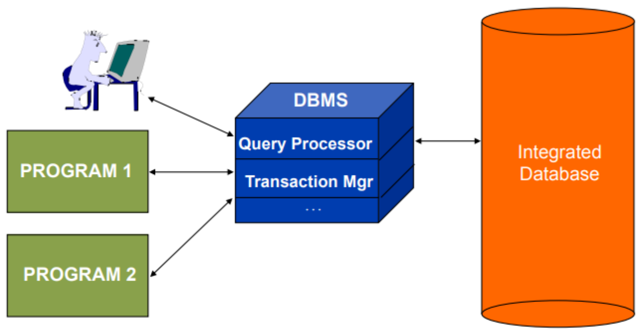
Disadvantages of file processing systems:

- Data redundancy and inconsistencies: having multiple copies of the same data leads to higher storage and inconsistencies could occur when only one copy is updated
- Difficulty in accessing or modifying data: new requests for accessing or modifying data requires writing a new application
- Integrity problems: to add constraints to restrict data entry requires changing the program

- Atomicity problems: difficult to return to state before a power failure (because it is difficult to ensure program parts are *atomic*)
- Concurrent-access anomalies: difficult to support multiple access and update to data
- Security and access control: difficult to control user access to data

Database

Instead of solving these problems anew, we can use a DBMS to handle it (*data independence*).



- **Database**: A *large* and *persistent* collection of (more-or-less similar) pieces of information organized in a way that facilitates efficient *retrieval* and *modification*
- **Database Management System (DBMS)**: A program (or set of programs) that manages details related to database storage and access for a database

DBMS Ideas (*integrated control*):

- Data Model: all data is organized in a well defined way
- Access Control: only authorized people are able to view/modify data
- Concurrency: multiple concurrent application and access/update data
- Database recovery: database can be rolled back so nothing is accidentally lost

Schema and Instance:

- **Schema**: A description of the data inferface to the database (i.e. how data is organized)
- **Database Instance**: A database (real data) that conforms to the given schema

Schema is like a class definition while the *instance* is an object created from that class.

Example: from the *schema* `PROJ(PNO, PNAME, BUDGET)` an possible *instance* could be:

PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

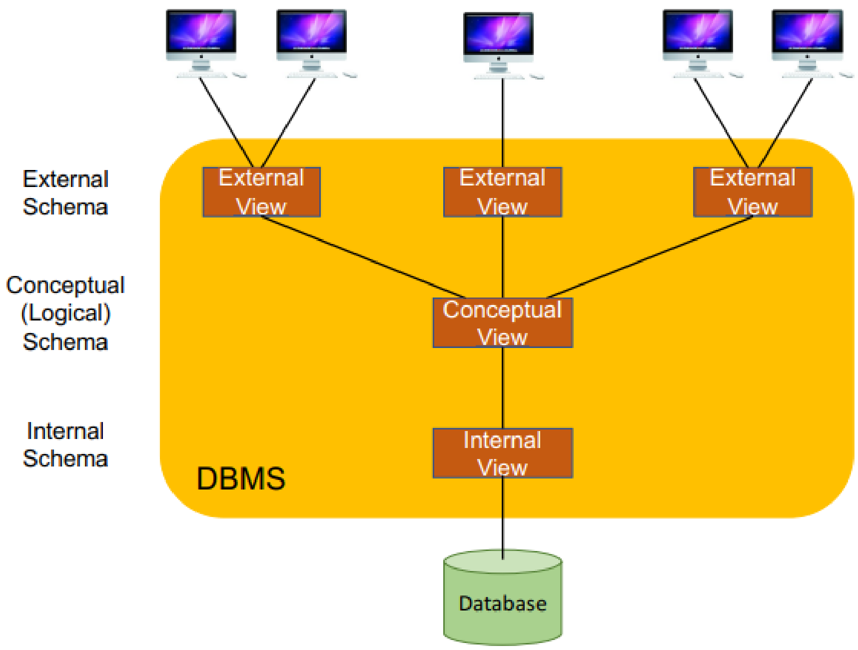
Brief History of Data Management

slides 13/27 to 16/27 and slides 24/27 to 26/27

maybe move to after next subsection?

Database Management System (DBMS)

Three Level Schema Architecture



- **External schema (view)**: what the applications and user sees (may differ for different users)
- **Conceptual schema**: description of the logical structure of *all* data in the database (*logical schema*)
- **Physical schema**: description of physical aspects of how the data is stored (e.g. storage format, low-level data structures, etc)

Data Independence

Applications *do not access data directly* but instead use an abstract data model provided by the DBMS.

There are two kinds of data independence:

- **Logical**: users of the external schema do not need to be aware of all the information at the logical schema level
- **Physical**: users of the logical level and the external schema do not need to be aware of the complexity of physical-level structure

Interfacing to the DBMS

Data Definition Lanuage (DDL): used to specify schemas

- Example: table for department can contain: dept name, building, and budget with each column being associated with a specific data type

Data Manipulation Language (DML): used for specifying queries and updates

- **Procedural DML**: requires user to specify what data is needed and how to get data
- **Declarative DML**: require user to just specify what data is needed
- Example: find instructor ID and dept name of all instructors with budget of more than $95,000

Transactions

Definition: A **transaction** is a unit of program execution that accesses and possibly updates data items.

Example: two transactions are made at the same time for the same account, if it is not handled properly the full $1500 will not be deducted.

```
Transaction T1

withdraw(AC,1000)
    Bal := getbal(AC)

    if (Bal > 1000)
        <give-money>
        setbal(AC, Bal - 1000)

```

```
Transaction T2

withdraw(AC,500)
    Bal := getbal(AC)

    if (Bal > 500)
        <give-money>
        setbal(AC, Bal - 500)

```

DBMS ensures that every application *can think* it is the sole application accessing the data at that time.

ACID properties of the transactions ensured by the DBMS:

- **Atomic**: transactions occurs entirely, or not at all
- **Consistency**: each transaction preserves the consistency of the database
- **Isolated**: concurrent transactions do not interfere with each other
- **Durable**: once completed, transaction's changes are permanent

Types of Database Users

- **End user**:
  - Accesses the database indirectly through forms or other query-generating applications
  - Generates ad-hoc queries using the DML
- **Application developer**:
  - Designs and implements applications that access the database
- **Database administrator (DBA)**:
  - Manages conceptual schema and assists with application view integration
  - Monitors and tunes DBMS performance
  - Defines internal schema
  - Is responsible for security and reliability

The Relational Model of Data

A *data model* specifies:

- the structure of the database (e.g. relations or tables)
- the operations for manipulating the data using that structure (e.g. relational algebra)
- a set of contraints that the databsea should obey (e.g. integrity constraints)

What we expect from a data model:

- Simplicity
- Ability to support data independence
- Declarative language support

The idea of the **Relational Model** is that all information is organized in relations (or tables).

Features:

- simple and clean data model
- powerful and *declarative* query/update language
- semantic integrity constraints
- data independence

attributes (or columns)

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

tuples (or rows)

Example of a Instructor Relation

Note: the order of the tuples is irrelevant (tuples may be stored in an arbitrary order)

Definition of the Relational Model

Informal Definition of the Relational Model:

- Database is a collection of *relations* (or *tables*)
- Each relation has a set of *attributes* (or *columns*)
- Each attribute has a name and a *domain* (or *type*)
  - The domain elements are required to be *atomic* (*indivisible*)
- Each relation contains a set of *tuples* (or *rows*)
  - Each tuple has a value for each attribute of the relation
  - Duplicate tuples are not allowed (two tuples with all same attributes)

Formal Definition of the Relational Model:

- **Domain**: set of allowed values of each attribute, denoted by $\text{dom}(D)$, where D is the domain name
- **Relation**:
  - *relation schema*: $R(A_1 : D_1, \ldots, A_k : D_k)$ with
    - name R
    - $A_1, \ldots, A_k$ is the set of distinct *attribute* names
    - $D_1, \ldots, D_k$ is a collection of (not necessarily distinct) domain names
  - *relation instance*: a finite relation
- **Database**:
  - *database schema*: finite set of uniquely-named relation schemas
  - *database instance*: a relation instance $\mathbf{R}_i$ for each relation schema R_i

Example: Bibliography Database

- Database schema:

```
author(aid:int, name:string)
wrote(author:int, publication:int)
publication(pubid:int, title:string)
book(pubid, publisher, year)
journal(pubid, volume, no, year)
proceedings(pubid, year)
article(pubid, crossref, startpage, endpage)
```

Note: relation schemas are sometimes abbreviated by omitting the attribute domains

- Sample database instance (tabular form on right):

$$
\begin{array}{rcll}
\textbf{author} & = & \{ & (1, \text{John}), (2, \text{Sue}) \quad \} \\
\textbf{wrote} & = & \{ & (1,1), (1,4), (2,3) \quad \} \\
\textbf{publication} & = & \{ & (1, \text{Mathematical Logic}), \\
 & & & (3, \text{Trans. Databases}), \\
 & & & (2, \text{Principles of DB Syst.}), \\
 & & & (4, \text{Query Languages}) \quad \} \\
\textbf{book} & = & \{ & (1, \text{AMS}, 1990) \quad \} \\
\textbf{journal} & = & \{ & (3, 35, 1, 1990) \quad \} \\
\textbf{proceedings} & = & \{ & (2, 1995) \quad \} \\
\textbf{article} & = & \{ & (4, 2, 30, 41) \quad \}
\end{array}
$$

author

aid	name
1	John
2	Sue

wrote

author	publication
1	1
1	4
2	3

publication

pubid	title
1	Mathematical Logic
3	Trans. Databases
2	Principles of DB Syst.
4	Query Languages

Properites of the Relational Model

Note:

- Relational schemas have named and typed attributes
- Relational instances are finite

Properties of a relation:

1. Based on (finite) set theory
   - Instance as *set semantics*:
     - No ordering among tuples
     - No duplicate tuples
2. All attribute values are atomic
3. Degree (arity) = # of attributes in schema
4. Cardinality = # of tuples in instance

Note: The standard language for interfacing with relational DBMSs is Structured Query Language (SQL). Unfortunately, there is a important difference between the Relational Model and the data model used by SQL and relational RDBMSs.

The discrepancy between relations in Relational Model and tables in RDBMSs:

- Semantics of Instances
  - Relations are sets of tuples
  - Tables are multisets (bags) of tuples

By default, SQL tables can contain duplicate elements.

Integrity Contraints

A relational schema captures only the structure of relations

Idea: Extend relational/database schema with rules called constraints. An instance is only valid if it satisfies all schema constraints.

Reasons to use constraints:

- Ensure that data entry/modification repects database design (shift s responsibility from applications to DBMS)
- Protect data from bugs in applications

Types of Integrity Constraints:

- **Tuple-level**:
  - *Domain restrictions*: restricting the domain (or type) of each attribute
    - e.g. if *student_ id* is integer then string is invalid
  - *Value comparisons*: restricting the range of values of each attribute
    - e.g. the only valid terms are {"Winter", "Summary", "Fall"}
- **Relation-level**: Key constraints
  - *Superkey*: set of attributes for which no pairs of distinct tuples in the relation will *ever* agree on the corresponding values
  - *Candidate key*: a minimal superkey (minimal set of attributes that uniquely identifies a tuple)
  - *Primary key*: a designated candidate key

  - **Example**:
    - *instructor(ID, name, dept_ name, salary)*

  Both *{ID}* and *{ID, name}* are superkeys, but only *{ID}* is a condidate key

instructor

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

- **Database-level**: Referential integrity
  - *Foreign key*: if primary key A of relation S appearing as atribute(s) B of relation R, then B is a foreign key from R, referencing S
    * S is called a *referenced* relation and R is the *referencing* relation
  - *Foreign key constraints*: a tuple in R with a non-null value for foreign key B that does not match primary key value of a tuple in the *referenced* relation S is not allowed
  - *Referential integrity constraints*: extended foreign key constraint, where referenced attribute may not be a primary key

  - **Example**:
    * *instructor(ID, name, dept_name, salary)*
    * *teaches(ID, course_id, sec_id, semester, year)*
    * Foreign-key constraint: *teaches.ID* references *instructor.ID*

    This means that on any database instance value of *ID* for each tuple in *teaches* must also be the value of a *ID* for some tuple in *instructor*.

instructor

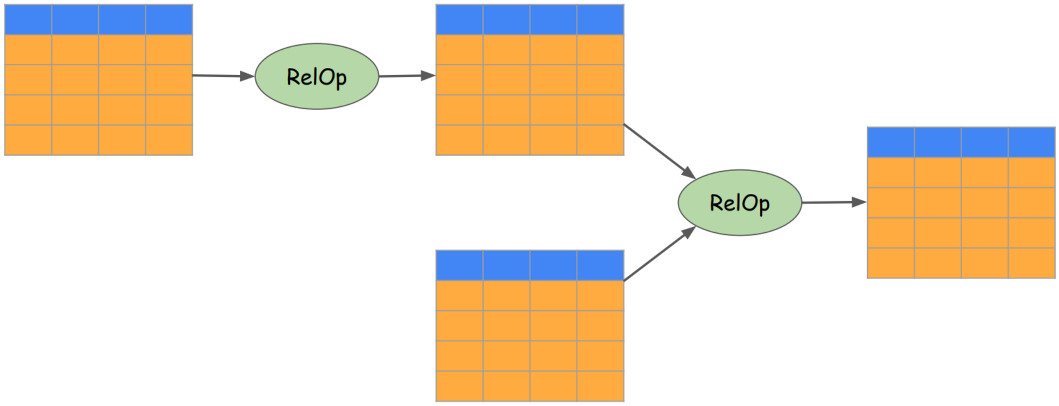
ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

The Relational Algebra

The relational algebra consists of a set of *operators*. To query relational data we use a composition of relational operators:



- Each relational operator takes one or two relations as input
- Each relational operator defines a single output/result relation in terms of its input
- Relational operators can be composed to form expression that define new relations in terms of existing relations

Selection

$$\sigma_{\text{condition}}(R)$$

- *Result schema*: same as R
- *Result instance*: subset of tuples in R that satisfies the condition

Example: find the instructors who are in the *Physics* department

instructor

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

$\sigma_{dept\_name=\text{"Physics"}}(instructor)$

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

The selection condition can include:

- any column of R or constants
- comparision $(=, \neq, >, \geq, <, \leq)$
- Boolean connectives $(\wedge, \vee, \neg)$

Note: the condition should be able to be evaluated over each *single* row of the input table

$$\text{Valid: } \sigma_{dept\_name=\text{“Physics”}\wedge salary>80000}(instructor)$$
$$\text{Invalid: } \sigma_{salary>\text{every } salary \text{ in } instructor}(instructor)$$

Projection

$$\pi_{\text{attributes}}(R)$$

- *Result schema*: includes only the specified attributes
- *Result instance*: could have as many tuples as R, except that duplicates are eliminated

Example: list all instructors' *ID*, *name*, and *salary*

instructor

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

$\pi_{ID,name,salary}(instructor)$

ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Example: we can use projection on the result of selection like so

$$\text{temp} \leftarrow \sigma_{salary>80000}(instructor)$$
$$\text{Result} \leftarrow \pi_{name,dept\_name}(\text{temp})$$

To shorten this we can directly compose the two expressions into

$$\pi_{name,dept\_name}(\sigma_{salary>80000}(instructor))$$

The result of this query is: {(Einstein, Physics), (Wu, Finance), (Brandt, Comp.Sci.), (Gold, Physics)}

Cross Product

$$R_1 \times R_2$$

- *Result schema*: has all atributes of R_1 and all attributes of R_2
- *Result instance*: includes one tuple for every pair of tpules in R_1 and R_2
- sometimes called the Cartesian product

Example:

R

A	B
a_1	b_1
a_2	b_2
a_3	b_3

S

C	D
c_1	d_1
c_2	d_2

$R \times S$

A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_2	c_1	d_1
a_3	b_3	c_1	d_1
a_1	b_1	c_2	d_2
a_2	b_2	c_2	d_2
a_3	b_3	c_2	d_2

Conditional Join

$$R_1 \bowtie_{\text{condition}} R_2$$

- equivalent to $\sigma_{\text{condition}}(R_1 \times R_2)$
- condition is a Boolean expression involving attributes from both operand relations

$$R_1.A\ \theta\ R_2.B \quad \text{where } \theta \in \{=, \neq, >, \geq, <, \leq\}$$

- sometimes called the θ-join

Example:

instructor

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

instructor $\bowtie_{instructor.ID=teaches.ID}$ *teaches*

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

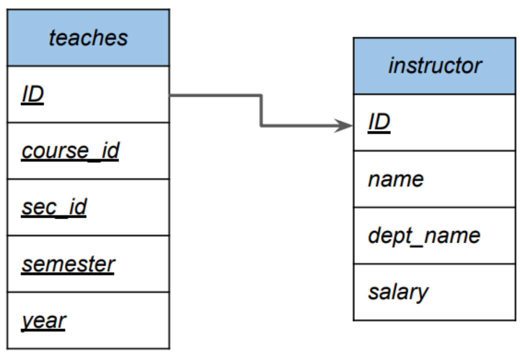
Natural Join

$$R_1 \bowtie R_2$$

1. Compute $R_1 \times R_2$ (renaming duplicate attributes)
2. Eliminate from the cross product any tuples that do have matching values for *all pairs* of attributes common in scheme R_1 and R_2
3. Project out duplicate attributes

If no attributes in common, this is just a product

Consider the natural join of the *instructor* and *teaches* tables, which have the attribute *ID* in common



- *Result(ID, name, dept_name, salary, course_id, sec_id, semester, year)*
- Resulting relation will include one tuple for each tuple in the *teaches* relation

Rename

Rename the name of the relation, the names of the attributes, or both

- Rename relation R to S

$$\rho_S(R)$$

- Rename attributes of relation R

$$\rho_{(A \to A', \ldots)}(R)$$

- Rename relation R to S and the names of its attributes

$$\rho_{S(A \to A', \ldots)}(R)$$

Output: a relation with the same rows as R, but named differently.

Example:

R

A	B
a_1	b_1
a_2	b_2
a_3	b_3

$$\rho_{S(A \to A', B \to B)}(R)$$

S

A'	B'
a_1	b_1
a_2	b_2
a_3	b_3

Set-Based Relational Operators

Definition: two schemas are *union compatible* if they both have the same number of fields with same type for corresponding fields

Example:

Courses in Fall 2017

course_id	semester	year
CS-101	Fall	2017
CS-347	Fall	2017
PHY-101	Fall	2017

Courses in Spring 2017

course_id	semester	year
CS-101	Spring	2018
CS-315	Spring	2018
CS-319	Spring	2018
CS-319	Spring	2018
FIN-201	Spring	2018
HIS-351	Spring	2018
MU-199	Spring	2018

For the following 3 set based relation operations the schemas of R and S must be *union compatible*:

- **Union**: all tuples that appear in either R or S or both (S and B must be *union compatible*)

$$R \cup S$$

  - e.g. find all course IDs taught in Fall 2017, Spring 2018, or both:

course_id	semester	year
CS-101	Fall	2017
CS-347	Fall	2017
PHY-101	Fall	2017

U

course_id	semester	year
CS-101	Spring	2018
CS-315	Spring	2018
CS-319	Spring	2018
CS-319	Spring	2018
FIN-201	Spring	2018
HIS-351	Spring	2018
MU-199	Spring	2018

=

course_id
CS-101
CS-347
PHY-101
CS-315
CS-319
FIN-201
HIS-351
MU-199

project course_ id, then union

- **Difference**: all tuples that appear in R and do not appear in S (S and B must be *union compatible*)

$$R - S$$

  - e.g. find all course IDs taught in Fall 2017 but not Spring 2018:

course_id	semester	year
CS-101	Fall	2017
CS-347	Fall	2017
PHY-101	Fall	2017

-

course_id	semester	year
CS-101	Spring	2018
CS-315	Spring	2018
CS-319	Spring	2018
CS-319	Spring	2018
FIN-201	Spring	2018
HIS-351	Spring	2018
MU-199	Spring	2018

=

course_id
CS-347
PHY-101

project course_ id, then minus

- **Intersection**: all tuples that appear in both R and S (R and S must be *union compatible*)

$$R \cap S$$

  - e.g. find all course IDs taught in both Fall 2017 and Spring 2018:

course_id	semester	year
CS-101	Fall	2017
CS-347	Fall	2017
PHY-101	Fall	2017

∩

course_id	semester	year
CS-101	Spring	2018
CS-315	Spring	2018
CS-319	Spring	2018
CS-319	Spring	2018
FIN-201	Spring	2018
HIS-351	Spring	2018
MU-199	Spring	2018

=

course_id
CS-101

project course_id, then intersect

Relational Division

$$R \div S$$

- Used to answer queries involving all (e.g. which employess work on all critical projects)
- Attributes of S must be subset of attributes of R
- $\text{attr}(R \div S) = \text{attr}(R) - \text{attr}(S)$
- Tuple t is in $(R \div S)$ iff $(t \times S)$ is a subset of R

Example: division is the inverse of product:

R

A
a_1
a_2

S

B	C
b_1	c_1
b_1	c_2
b_2	c_2

$R \times S$

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_1	b_2	c_2
a_2	b_1	c_1
a_2	b_1	c_2
a_2	b_2	c_2

$(R \times S) \div S$

A
a_1
a_2

Example: which employees work on all critical projects? `Works(Enum, Pno)` and `Critical(Pno)`

Works

Enum	Pnum
E35	P10
E45	P15
E35	P12
E52	P15
E52	P17
E45	P10
E35	P15

Critical

Pnum
P15
P10

Works ÷ Critical

Enum
E45
E35

Notice that product is not always the inverse of division (e.g. (Works ÷ Critical) × Critical)

Algebraic Equivalences

The following are all equivalent:

$$\pi_{\text{name,course\_id}}(\sigma_{\text{dept\_name=``Physics''}}(\sigma_{\text{instructor.ID=teaches.ID}}(\text{instructor} \times \text{teaches})))$$

$$\pi_{\text{name,course\_id}}(\sigma_{\text{dept\_name=``Physics''}}(\text{instructor} \bowtie_{\text{instructor.ID=teaches.ID}} \text{teaches}))$$

$$\pi_{\text{name,course\_id}}(\text{instructor} \bowtie_{\text{instructor.ID=teaches.ID}} \sigma_{\text{dept\_name=``Physics''}}(\text{teaches}))$$

$$\pi_{\text{name,course\_id}}((\pi_{\text{ID,name}}(\text{instructor})) \bowtie_{\text{instructor.ID=teaches.ID}} (\pi_{\text{ID,course\_id}}(\sigma_{\text{dept\_name=``Physics''}}(\text{teaches}))))$$

These all perform the same action but some run faster than others. More on this in database tuning topic.

Relational Completeness

Definition: a query language that is at least as expressive as relational algebra is *relationally complete*

Relational algebra and SQL are both relationally complete.

SQL even has additional expressive power because it captures aggregation, ordering, etc.

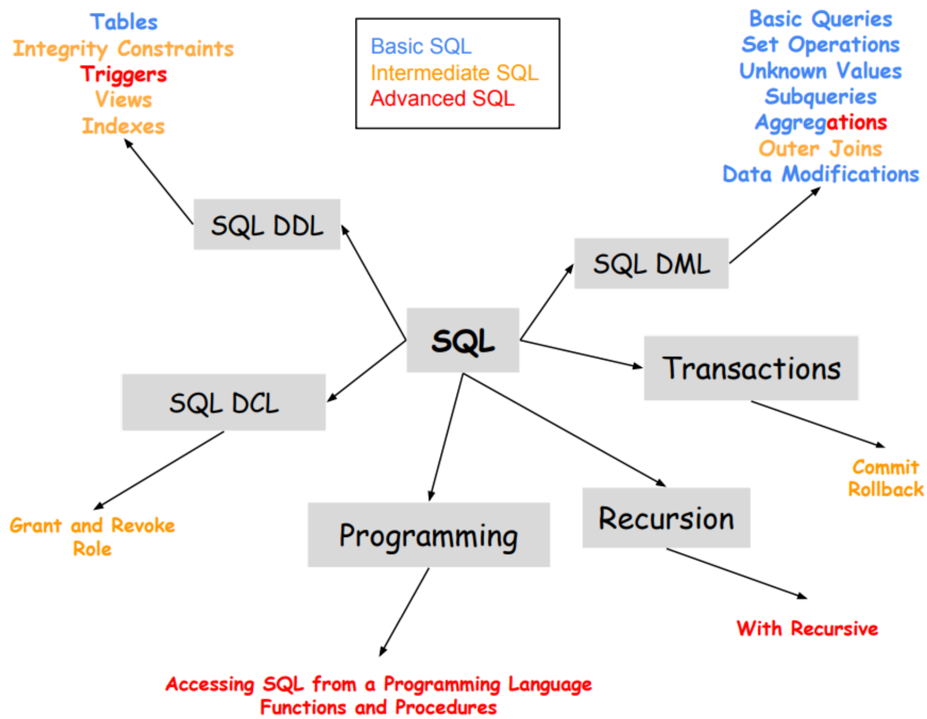
Structured Query Language (SQL)

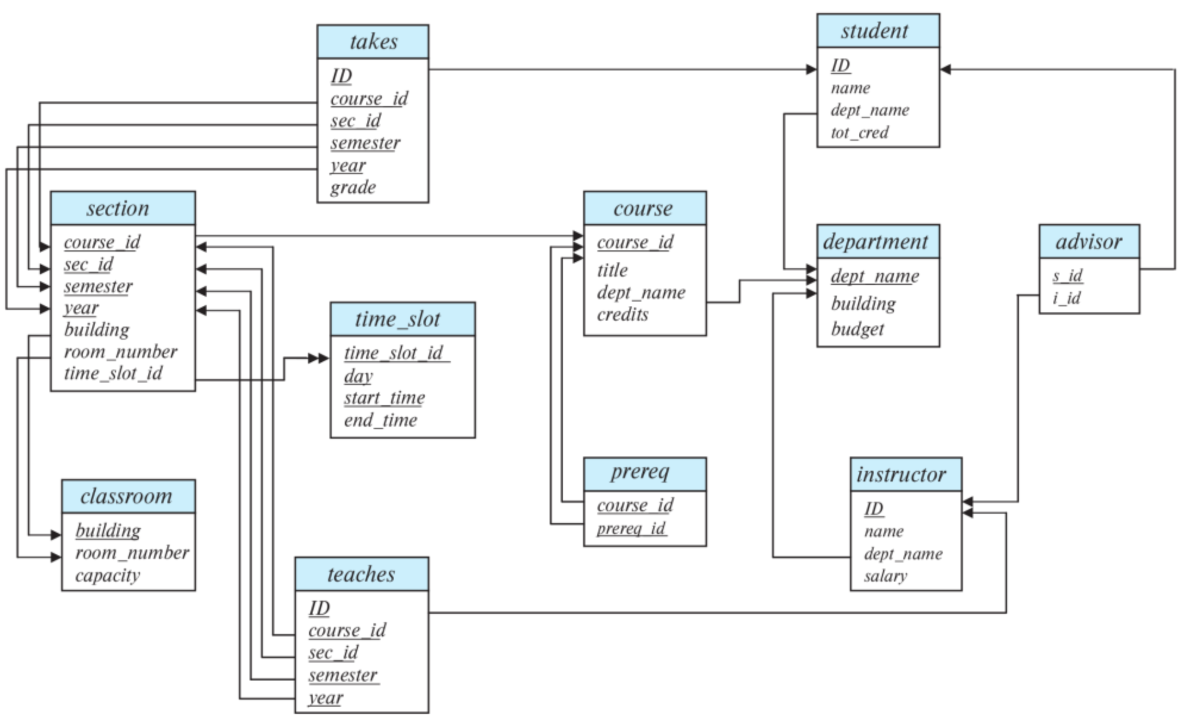
The Structured Query Language (SQL) is made up of three sub-languages:

- SQl Data Manipulation Lnaugage (DML)
  - `SELECT` statements performs queries
  - `INSERT`, `UPDATE`, `DELETE` statements modify the table instance
- SQL Data Definition Lnaguage (DDL)

  - `CREATE`, `DROP` statements modify the database schema

- SQL Data Control Language (DCL)
  - `GRANT`, `REVOKE` statements enfore the security model





Database Schema Used for Examples

Tables

SQL DDL: Data Types

Some of the attribute types (or domains) defined in SQL:

- *integer* or *int*: machine-dependent finite subset of integers
  - e.g. 4-byte integer type can store:

$$\text{signed value } \in [-2147483648, 2147483647] \qquad \text{unsigned value} \in [0, 4294967295]$$

- *smallint*: small integer (machine-dependent subset of integer domain)
  - e.g. 2-byte smallint type can store:

$$\text{signed value } \in [-32768, 32767] \qquad \text{unsigned value} \in [0, 65535]$$

- *numeric*(p, q): p-digit numbers, with q digits right of the decimal point
  - decimal point and minus sign are not counted in p
  - e.g. 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
- *char*(n): fixed length character strings, with user-specified length n
  - string shorter than n will be padded to n length by appending spaces
- *varchar*(n): variable length character strings, with user-specified maximum length n
- *date*: describes a year, month, day
- *time*: describes a hour, minute, second
- *timestamp*: describes a data and the time on that date
- *interval*: allows computation based on dates and times on interval

Create Table

A SQL relation is defined using the create table command:

```
create table r (
    A1 D1,
    ...,
    An Dn,
    integrity-constraint-1,
    ...,
    integrity-constraint-k)
```

- `r` is the name of the relation
- `Ai` is an attribute name with domain of values `Di`
- We will see more about integrity constraints later, for now we have:
  - `primary key (A1, ..., An)`

- foreign key (A1, ..., An) references r

SQL prevents any update to the database that violates an integrity constraint

Example:

```
create table instructor (
    ID          char(5),
    name        varchar(20),
    dept_name   varchar(20),
    salary      numeric(8,2),
    primary key (ID),
    foreign key (dept_name) references department)
```

Basic Structure of SQL Queries

`select` Clause

`select` clause lists attributes desired in the query result (projection operation in relational algebra)

- SQL names are case insensitive so the following queries are equivalent:
  - `select dept_name from instructor`
  - `select Dept_Name from instructor`
  - `select DEPT_NAME from instructor`
- Default is to allow duplicates in relations and query results
  - To eliminate duplicates from query result use `select distinct`
- Asterisk in select clause denotes *all attributes*
  - e.g. `select * from instructor` will return the entire `instructor` table
- `select` clause can be used with arithemetic expressions $\{+, -, *, /\}$ to modify attributes

Examples:

- `select dept_name from instructor`
  - finds the department names of all instructors

instructor

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

query result

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

- `select distinct dept_name from instructor`
  - finds department names of all instructors then removes duplicates

instructor

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

query result

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

- `select distinct dept_name, salary from instructor`
  - find department names and salary of all instructors then remove duplicate pairs

instructor

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

query result

dept_name	salary
Biology	72000
Comp. Sci.	65000
Comp. Sci.	75000
Comp. Sci.	92000
Elec. Eng.	80000
Finance	80000
Finance	90000
History	60000
History	62000
Music	40000
Physics	87000
Physics	95000

- `select * from instructor`

instructor

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

query result

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

- `select ID, name, salary/12 as monthly_salary from instructor`
  - compute monthly salary of each instructor

instructor

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

query result

ID	name	monthly_salary
10101	Srinivasan	5416
12121	Wu	7500
15151	Mozart	3333
22222	Einstein	7916
32343	El Said	5000
33456	Gold	7250
45565	Katz	6250
58583	Califieri	5166
76543	Singh	6666
76766	Crick	6000
83821	Brandt	7666
98345	Kim	6666

`where` Clause

`where` clause specifies conditions on the query result (selection operation in relational algebra)

- Comparisons can be applied to results of arithmetic expresions
- Comparison operations are {<, <=, >, >=, =, <>}
- Logical connectives are {`and`, `or`, `not`}

Examples:

- find all instructors in Comp. Sci. department

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

instructor

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

query result

name
Srinivasan
Katz
Brandt

- find all instructors in Comp. Sci. department whose monthly salary is greater than 5000

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary/12 > 5000
```

instructor

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

query result

name
Srinivasan
Katz
Brandt

`from` Clause

`from` clause lists the relations involved in the query (cartesian product operation in relational algebra)

Example: `select * from instructor, teaches`

- Result is the cross product of *instructor* and `teaches`
- Generates every possible instructor-teaches pair (common attributes are renamed with relation name)

instructor

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

teaches

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

partial result

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
...	...	...	...	...	...	...	...	...

`inner join` Clause

`inner join` is clause is equivalent using `from` and `where` (conditional join in relational algebra)

Example: find the names of all instructions who are teaching a course

```
select *
from instructor, teaches
where instructor.ID = teaches.ID
```

```
select *
from instructor inner join teaches
    on instructor.ID = teaches.ID
```

instructor

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

teaches

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

$instructor \bowtie_{instructor.ID=teaches.ID} teaches$

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

Example: name and course_id of instructors in Comp. Sci. dept who have taught a course in 2017

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID
    and instructor.dept_name = 'Comp. Sci.'
    and year = 2017
```

```
select name, course_id
from instructor inner join teaches
    on instructor.ID = teaches.ID
where instructor.dept_name = 'Comp. Sci.'
    and year = 2017
```

`natural join` Clause

`natural join` automatically removes one of the two join attributes

Example:

- Query 1: `select * from instructor, teaches`
  - *instructor.ID, name, dept_name, salary, teaches.ID, course_id, sec_id, semester, year*
- Query 2: `select * from instructor natural join teaches`
  - *ID, name, dept_name, salary, course_id, sec_id, semester, year*

Basic Query Strucutre

A typical SQL query of attributes A_i from relations r_k has the from:

```
select A1, A2, ..., An
from r1, r2, ..., rm
where condition
```

This is called an SFW (select-from-where) or SPJ (select-project-join) query.

It is equivalent to the following relational algebra expression:

$$\pi_{A_1,\ldots,A_n}(\sigma_{\text{condition}}(r_1 \times \cdots \times r_m))$$

Additional Basic Operations

`as` Clause

`as` clause changes the names of attributes and relations (rename in relational algebra)

Example:

```
select name as instructor_name, course_id
from instructor, teaches
where instructor.ID = teaches.ID
```

*query result without "**as** instructor_name"*

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

*query result with "**as** instructor_name"*

instructor_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 ID and name of insturctors who earn more than the instructor whose ID is `"12121'`

- The issue that is we need to compare tuples in the same relation for the join

```
select T.ID, T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.ID = '12121'
```

instructor

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

query results

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~
~~32343~~	~~El Said~~	~~History~~	~~60000~~
~~45565~~	~~Katz~~	~~Comp. Sci.~~	~~75000~~
~~98345~~	~~Kim~~	~~Elec. Eng.~~	~~80000~~
~~76766~~	~~Crick~~	~~Biology~~	~~72000~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~
~~58583~~	~~Califieri~~	~~History~~	~~62000~~
83821	Brandt	Comp. Sci.	92000
~~15151~~	~~Mozart~~	~~Music~~	~~40000~~
~~33456~~	~~Gold~~	~~Physics~~	~~87000~~
~~76543~~	~~Singh~~	~~Finance~~	~~80000~~

This works but the performance is pretty bad, we will see a better method to do this later.

String Operations

- SQL string are enclosed by single quotes: `'Computer'`
  - Single quotes in a string is denoted using two single quotes: `'It''s all right'`
- SQL standard specifies that string comparison is case sensitive
  - `'database' = 'DataBase'` should be *false* but not all DBMS follow this (e.g. MySQL)
- SQL permits a variety of string functions:
  - concatenation
  - upper and lower case conversions
  - string length, substrings, etc (check DBMS manual for more)

`like` Operation

`like` is a string-matching operation for string pattern matching.

- Two special characters are used for describing patterns
  - percent %: matches any substring
  - underscore _: matches any character
- Patterns are case sensitive
- Use `escape` to define an escape character
  - e.g. to match the string "100%" we use: `like '100\%' escape '\'`
- Some pattern matching examples:
  - `'Intro%'` matches any string that begins with "Intro"
  - `'%Comp%'` matches any string containing "Comp" as a substring
  - `'___'` matches any string of exactly three characters
  - `'___%'` matches any string of at least three characters

Example: find the names of all instructors whose name includes the substring “in”

```
select name
from instructor
where name like '%in%'
```

instructor

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

query result

name
Srinivasan
Einstein
Singh

`between` Operation

Find the names of all instructors with salary between $\$90,000$ and $\$100,000$

```
select name
from instructor
where salary >= 90000 and salary <= 100000
```

```
select name
from instructor
where salary between 90000 and 100000
```

Tuple Comparision

Find the names of instructors in the department of Biology and the IDs of courses taught by them

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID
    and dept_name = 'Biology'
```

```
select name, course id
from instructor, teaches
where (instructor.ID, dept_name)
    = (teaches.ID, 'Biology')
```

Ordering Operations

`order by attr` is followed by `asc` for ascending or `desc` for descending

- By default `order by` assumes `asc`
- Can sort by multiple attributes

List all instructor names alphabetically (ascending)

```
select name
from instructor
order by name asc
```

query result

name
Brandt
Califieri
Crick
Einstein
El Said
Gold
Katz
Kim
Mozart
Singh
Srinivasan
Wu

List all instructor names alphabetically (descending)

```
select name
from instructor
order by name desc
```

query result

name
Wu
Srinivasan
Singh
Mozart
Kim
Katz
Gold
El Said
Einstein
Crick
Califieri
Brandt

Sort two attributes in ascending order

```
select dept_name, name
from instructor
order by dept_name, name
```

query result

dept_name	name
Biology	Crick
Comp. Sci.	Brandt
Comp. Sci.	Katz
Comp. Sci.	Srinivasan
Elec. Eng.	Kim
Finance	Singh
Finance	Wu
History	Califieri
History	El Said
Music	Mozart
Physics	Einstein
Physics	Gold

Sort two attributes in descending order

```
select dept_name, name
from instructor
order by dept_name, name desc
```

query result

dept_name	name
Biology	Crick
Comp. Sci.	Srinivasan
Comp. Sci.	Katz
Comp. Sci.	Brandt
Elec. Eng.	Kim
Finance	Wu
Finance	Singh
History	El Said
History	Califieri
Music	Mozart
Physics	Gold
Physics	Einstein

Set Operations

Set operations in SQL and Relational Algebra (RA):

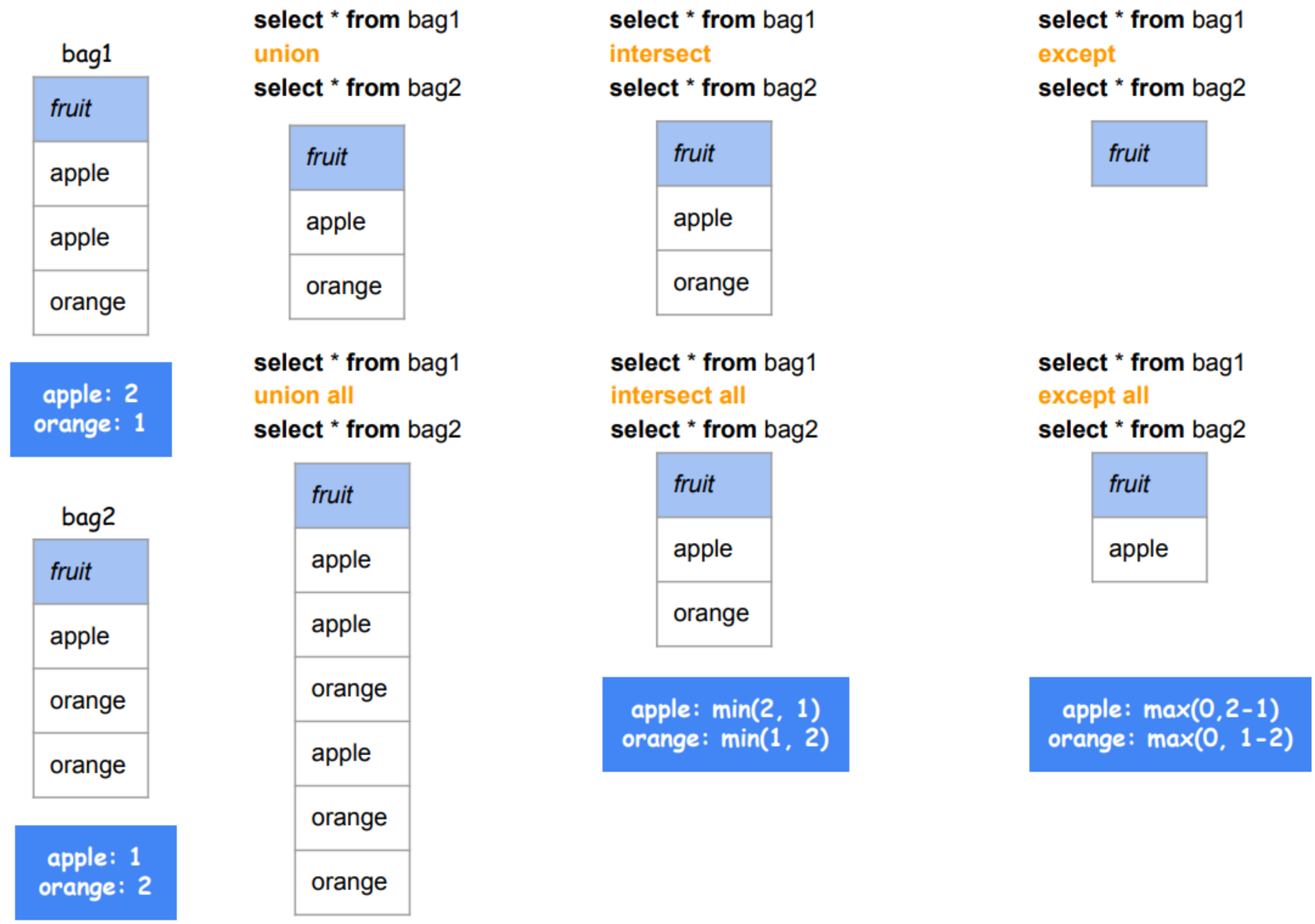
SQL	RA
`union`	$\cup$
`intersect`	$\cap$
`except`	$-$

- When using `union`, `intersect`, `except` the duplicates will be removed
  - To retain duplicates use `union all`, `intersect all`, `except all`

- Not every DBMS supports `intersect all` and `except all`, e.g. SQLite

- The requirements to use set operations is that the two schemas be *union compatible*
  - Same number of attributes
  - Same type for corresponding attribute

Example: SQL set operations



Aggregate Functions

Definition: an *Aggregate Function* takes the values of a table column and outputs a single scalar value

SQL standard aggregate functions:

- `avg` finds average value
- `min` finds minimum value
- `max` finds maximum value
- `sum` gets sum of values
- `count` gets number of values

Examples:

- find the average salary of instructors in Comp. Sci. department

```
select avg(salary) as avg_salary
from instructor
where dept_name = 'Comp. Sci.'
```

instructor

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

query results

avg_salary
77333.33333333333

- count the number of tuples in the instructor table

```
select count(*)
from instructor
```

instructor

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

query results

count(*)
12

- count the number of distinct department names in the instructor table

```
select count(distinct dept_name)
from instructor
```

instructor

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

query results

count(distinct dept_name)
7

`group by` Clause

```
select ...
from ...
where ...
group by list_of_columns
```

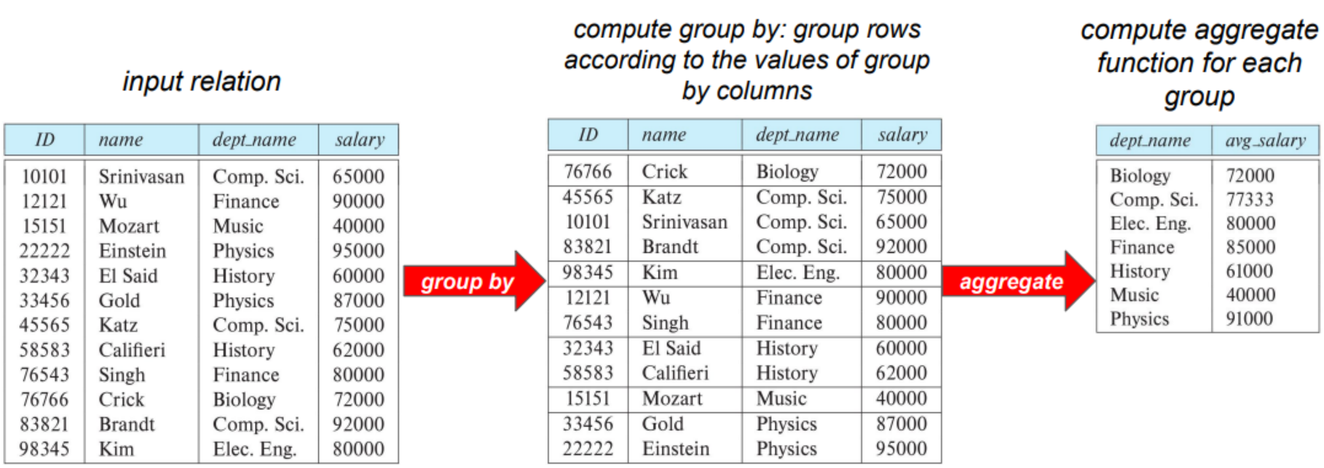
1. Compute `from` ($\times$)
2. Compute `where` (σ)
3. Compute `group by`
   - group rows accround to the values of `group by` columns
4. Compute aggregate functions for each group
   - for aggregation functions with `distinct` inputs, first eliminate duplicates within the group

Notes:

- Number of rows in final output = number of groups
- Aggregate query with no `group by` clause treats all the rows as a single group
- Multiple aggregate functions can be used in `select`
- If a query uses aggregation/group by, then every column referenced in `select` must appear either
  - in aggregate functions
  - in group by list

Example: find the average salary of instructors in each department

```
select dept_name, avg(salary) as avg_salary
from instructor
group by dept_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

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

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

Example: every column referenced in `select` must appear in an aggregate function or the group by list

WRONG!!!

```
select dept_name, ID, avg (salary)
from instructor
group by dept_name
```

WRONG!!!

```
select dept_name, avg (salary)
from instructor
```

The syntax is correct, but the dept_name value in the result is a random one!

`having` Clause

```
select ...
from ...
where ...
group by ...
having condition
```

This clause is used to filter gorups based on group properties (e.g. aggregate values, `group by` column values)

1. Compute `from` ($\times$)
2. Compute `where` (σ)
3. Compute `group by`
   - group rows accround to the values of `group by` columns
4. Compute aggregate functions for each group
   - for aggregation functions with `distinct` inputs, first eliminate duplicates within the group
5. Compute `having` (another σ over the resulting relation in step 4)

Example: find the names and average salaries of all departments whose average salary is over 42000

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

input relation

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

group by →

compute group by: group rows according to the values of group by columns

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

aggregate →

compute aggregate function for each group, and then the having condition

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

Unknown Values

Null Values

In every domain the special value `null` indicates unknown or missing data.

Example: *user(uid, name, age)*

- Value unknown: we do not know Bob's age
- Value missing: Bob did not fill in his name so his name is missing

Three-Valued Logic

When we compare a `null` with another value (including another `null`) the result is `unknown`

If the expression contains connectives like `and`, `or`, `not` we evalute them using *three-valued logic*:

$$\text{true} = 1, \text{false} = 0, \text{unknown} = 0.5$$
$$x \texttt{ and } y = \min(x, y)$$
$$x \texttt{ or } y = \max(x, y)$$
$$\texttt{not } x = 1 - x$$

- The `where` and `having` clauses only select rows for output if condition is *true*.
- Aggregate functions ignore `null` except `count(*)`.

Truth tables:

NOT(A)

A	¬A
F	T
U	U
T	F

AND(A, B)

A ∧ B		B: F	B: U	B: T
A	F	F	F	F
A	U	F	U	U
A	T	F	U	T

OR(A, B)

A ∨ B		B: F	B: U	B: T
A	F	F	U	T
A	U	U	U	T
A	T	T	T	T

Example: the following is NOT equivalent

```
select name
from instructor
where salary = null
```

```
select name
from instructor
where salary is null
```

- `salary = null` will *always* produce an `unknown` as we are comparing `null` to another value with =
- `salary is null` is the proper way to check if the is null

Joins

Outer Joins

Outer joins are an extension of the join operation that avoid loss of information

- *Dangling tuples*: tuples from one relation that do not match tuples in other relation in join result

- Compute the (inner) join then add *dangling* tuples padded with `null`

Example:

student

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

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*

```
select *
from student natural join takes
```

Expected Query Result

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-
70557	Snow	Physics	0	*null*	*null*	*null*	*null*	*null*
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*

Snow is not included in the result of natural join!

Tuples from one relation that do not match tuples in the other relation are excluded in a join. In this example the *student* "Snow" does not match any tuple in *takes*.

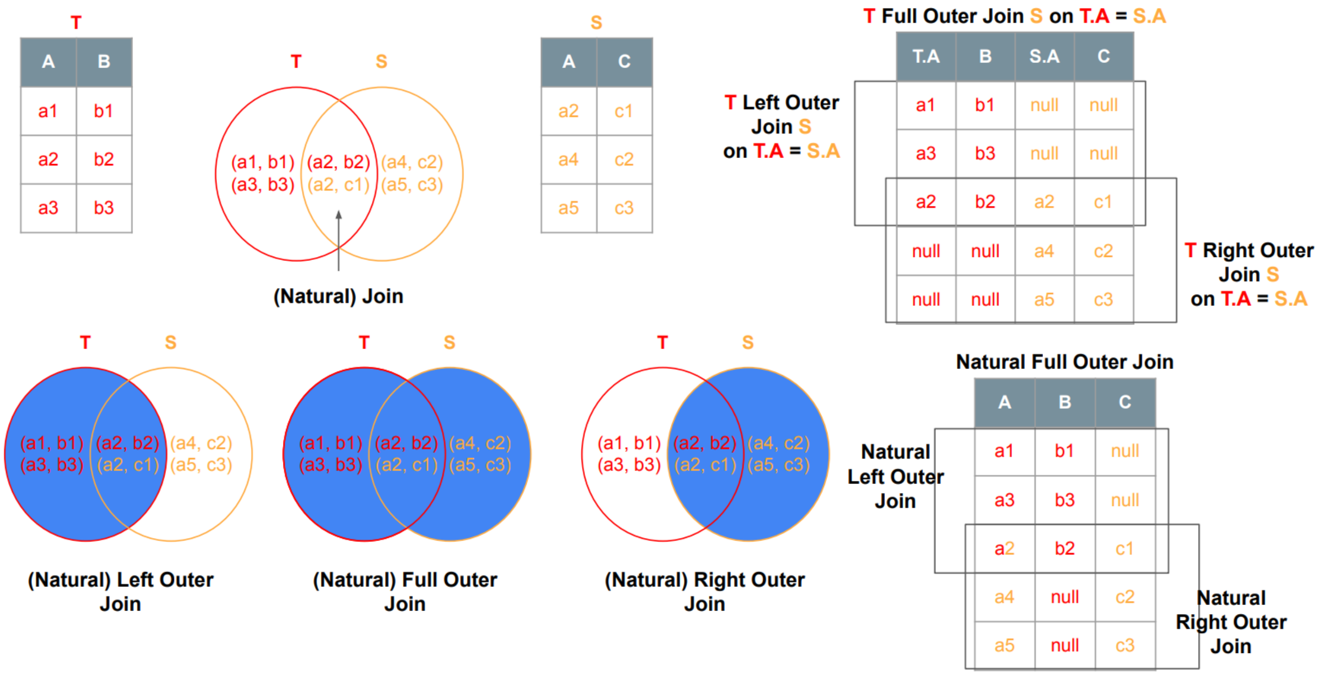
To produce the *Expected Query Result* we need to perform:

```
select *
from student left outer join takes on student.ID = takes.ID
```

Note: even `select * from student, takes on student.ID = takes.ID` does not result in expected

For T join S we choose which *dangling* tuples to retain using:

- `left outer join` or `left join`: keep all tuples in T (left side)
- `right outer join` or `right join`: keep all tuples in S (right side)
- `full outer join` or `full join`: keep all tuples in T plus all tuples in S



Somewhat confusing graphic

Highly suggest going to db-book.com to try out some examples there.

Join Expressions

Given the relations $T(A, B)$ and $S(A, C)$

- Comparison operators $<$ *comp* $>$ are: $<, \leq, >, \geq, =, <>$
- Inner join expressions:

```
... from T, S where T.A <comp> S.A ...
... from T join S on T.A <comp> S.A ...
... from T inner join S on T.A <comp> S.A ...
... from T natural join S ...
```

- Outer join expressions:

```
... from T full outer join S on T.A <comp> S.A ...
... from T left outer join S on T.A <comp> S.A ...
... from T right outer join S on T.A <comp> S.A ...
... from T full join S on T.A <comp> S.A ...
... from T left join S on T.A <comp> S.A ...
... from T right join S on T.A <op> S.A ...
... from T natural full outer join S ...
```

```
... from T natural left outer join S ...
... from T natural right outer join S ...
... from T natural full join S ...
... from T natural left join S ...
... from T natural right join S ...
```

`using` Clause

```
... join S using (list_of_attributes)
```

- Specifies which columns should be equated and removes duplicate attributes in the result relation
- `using` clause can be usd with full, left, or right outer joins

Example: given relations $T(A, B)$ and $S(A, C)$ the following are equivalent

```
select T.A, B, C
from T join S on T.A = S.A
```

```
select *
from T natural join S
```

```
select *
from T join S using(A)
```

Natural Join Pitfalls

Given $T(A, B, D)$ and $S(A, C, D)$

- For any tuple (a, b, d) in T and any tuple (a', c', d') in S check if $a = a'$ and $d = d'$

  ```
  select A, B, C from T natural join S
  ```

- For any tuple (a, b, d) in T and any tuple (a', c', d') in S check only that $a = a'$

  ```
  select T.A, B.C from T inner join S on T.A = S.A
  ```

  ```
  select A, B, C from T join S using (A)
  ```

Given the relations

```
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
course(course_id, title, dept_name, credits)
```

To list the name of students along with the titles of courses they have taken

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

The following is incorrect because natural join will require that both *course_id* and *dept_name* match

```
select name, title
from student natural join takes natural join course
```

Subqueries

Nested Subqueries

Recall that a *select-from-where* (SFW) query is an expression of the form:

```
select A1, ..., An
from r1, ..., rm
where condition
```

A *subquery* is a SFW expression that is nested within another query.

A subquery even be nested within an SFW query:

- `select`: A_i can be replaced by a subquery that generates a single value
- `from`: r_i can be replaced by any valid subquery
- `where`: condition can be replaced with an expression of the form

  ```
  B <op> (subquery)
  ```

  where B is an attribute and $< op >$ will be seen later

Example:

- Find average instructor's salaries of departments where the average salary is greater than \$42,000

```
select dept_name, avg_salary
from (select dept_name, avg(salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 4200
```

inner query result

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333.33333333333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

query result

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333.33333333333
Elec. Eng.	80000
Finance	85000
History	61000
~~Music~~	~~40000~~
Physics	91000

- Find maximum across all departments of the total of all instructors' salaries in each department

```
select max (tot_salary)
from (select dept_name, sum (salary) as tot_salary
      from instructor
      group by dept_name)
```

inner query result

dept_name	tot_salary
Biology	72000
Comp. Sci.	232000
Elec. Eng.	80000
Finance	170000
History	122000
Music	40000
Physics	182000

max

query result

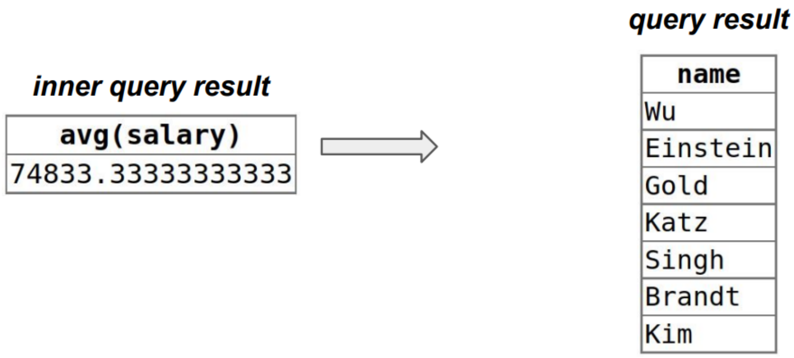
max (tot_salary)
232000

Scalar Subqueries

Subqueries that only returns a single tuple can be used as a value in `where` and `select` clauses

Example: find all instructors whose salary is above average

```
select name
from instructor
where salary > (select avg(salary)
                from instructor)
```



If the subquery returns more than one tuple then things will go wrong.

Set Membership

- Check if x is in the result of *subquery* (corresponds to *intersect* clause)

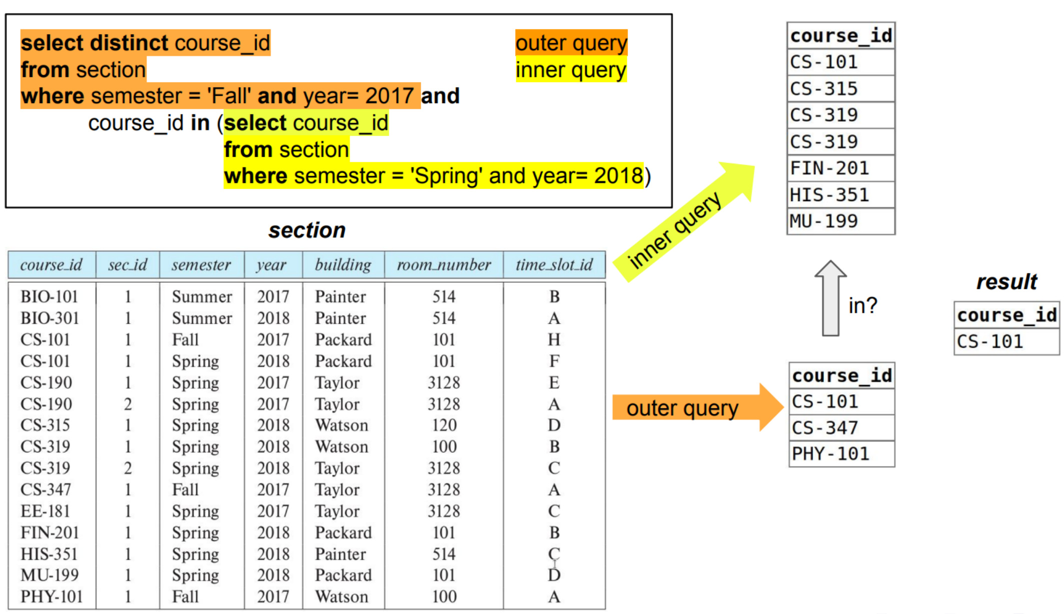
  `x in (subquery)`

- Check if x is *not* in the result of *subquery* (corresponds to *except* clause)

  `x not in (subquery)`

Example:

- Find all courses taught in both Fall 2017 and Spring 2018 semesters



- Find all courses taught in the Fall 2017 semester but not in the Spring 2018 semester

```
select distinct course_id
from section
where semester = 'Fall' and year = 2017 and
   course_id not in (select course_id
                     from section
                     where semester = 'Spring' and year = 2018)
```

outer query
inner query

section

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

inner query →

course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

in?

outer query →

course_id
CS-101
CS-347
PHY-101

result

course_id
CS-347
PHY-101

- `in` and `not in` operators can be used on enumerated sets

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein')
```

- `in` and `not in` operators can be used on *attribute relations*
  - Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 110011

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year)
    in (select course_id, sec_id, semester, year
        from teaches
        where teaches.ID='10101')
```

Set Comparison

- *for each one* (universal quantification)

`x <comp> all (subquery)`

(5 < **all** {0, 5, 6}) = false | (5 < **all** {6, 10}) = true | (5 = **all** {4, 5}) = false | (5 ≠ **all** {4, 6}) = true

- *at least one* (existential quantification)

`x <comp> some (subquery)`

(5 < **some** {0, 5}) = false | (5 < **some** {0, 5, 6}) = true | (5 = **some** {0, 5}) = true | (5 ≠ **some** {0, 5}) = true

Example:

- Names of instructors whose salary is greater than salary of *all* instructors in the Comp. Sci.
  - Notice that we can use *max* instead of *all*

```
select name                                  inner query
from instructor
where salary > all (select salary
                    from instructor
                    where dept_name = 'Comp. Sci.')
```

instructor

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

inner query →

salary
65000
75000
92000

for each salary x in instructor, evaluate x > ***all*** *(65000, 72000, 92000)*

result

name
Einstein

- Names of instructors whose salary is greater than salary of *some* instructor in the Comp. Sci.
  - Notice that we can use *min* instead of *some*

```
select name                                  inner query
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = 'Comp. Sci.')
```

instructor

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

inner query →

salary
65000
75000
92000

for each salary x in instructor, evaluate x > ***some*** *(65000, 72000, 92000)*

result

name
Wu
Einstein
Gold
Katz
Singh
Crick
Brandt
Kim

Empty Relations Testing

- If the reuslt of the *subquery* is non-empty then return *true*

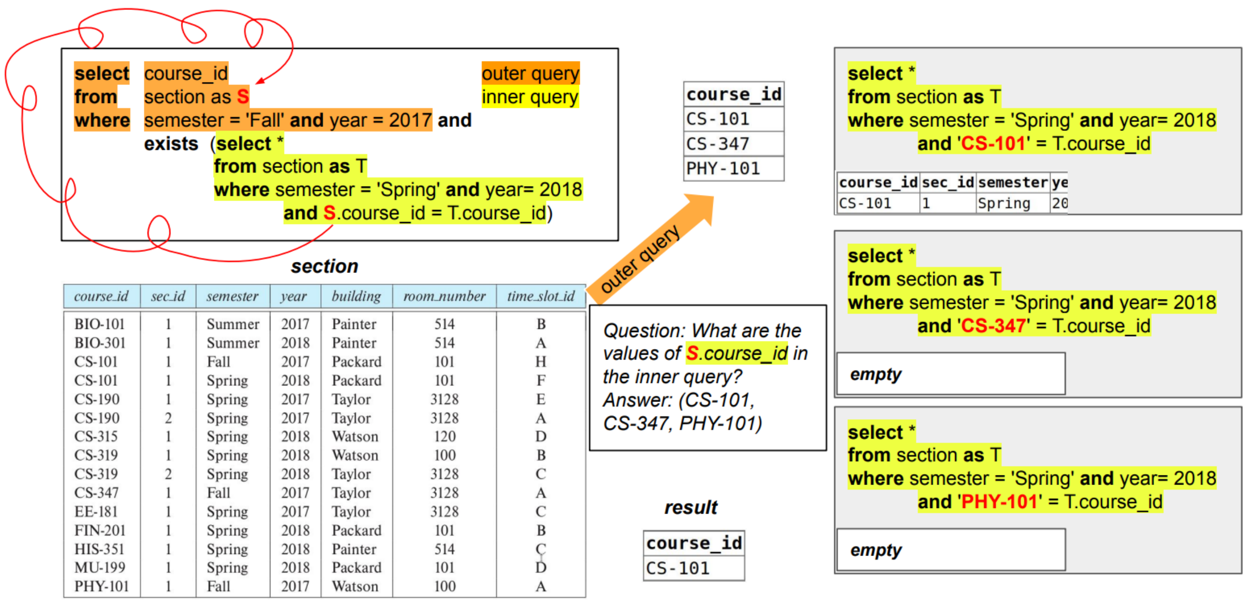
  `exists (subquery)`

- If the results of the *subquery* is empty then return *false*

  `not exists (subquery)`

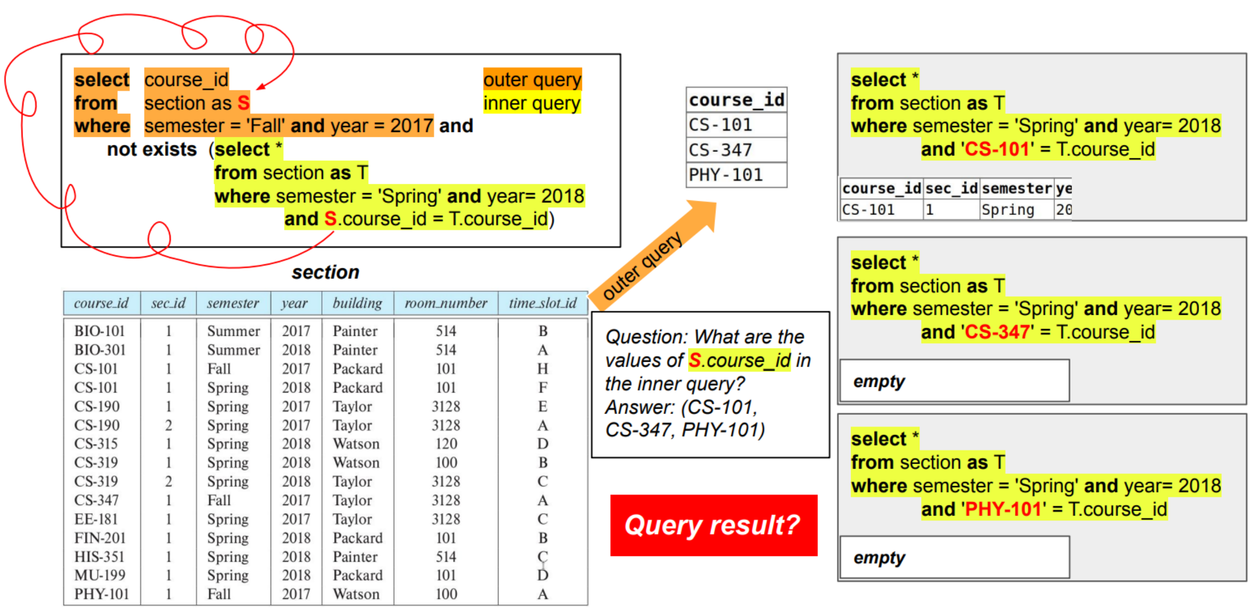
Example:

- Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester



course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

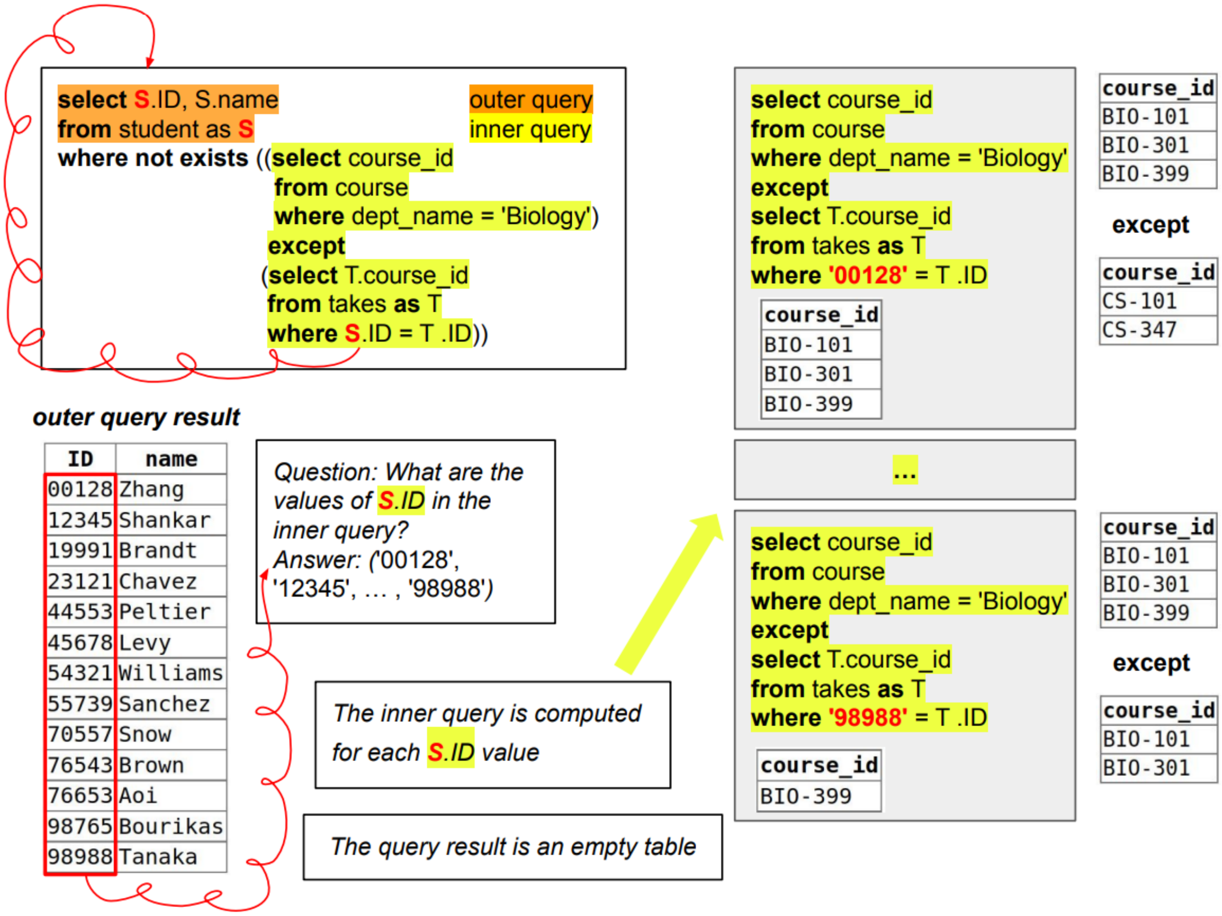
- Find all courses taught in the Fall 2017 semester but not in the Spring 2018



course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

  – Query result is: `CS-315, CS-319, CS-319, FIN-201, HIS-351, MU-199`

- Find all students who have taken all courses offered by in the Biology department



Duplicate Tuples Testing

- If the result of the *subquery* contains no duplicate tuples then return *true*

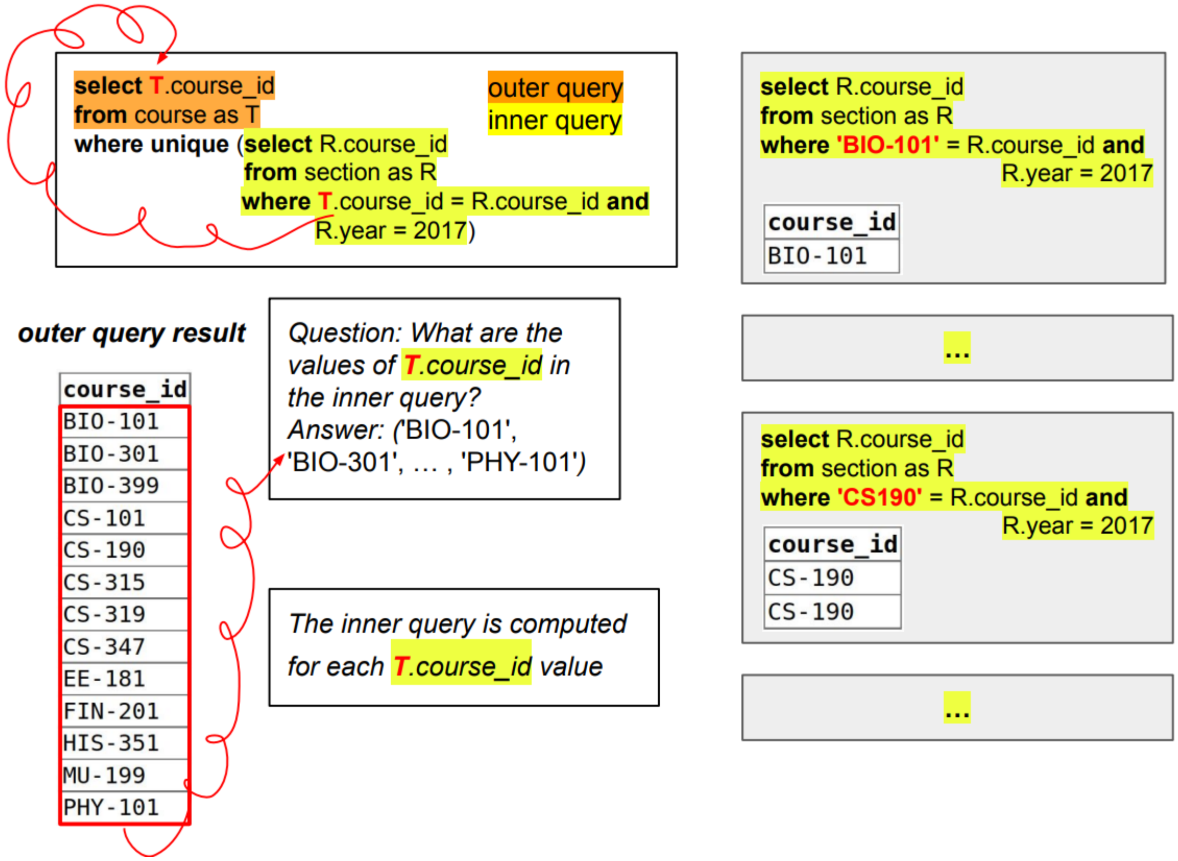
```
unique (subquery)
```

- If the result of the *subquery* contains duplicate tuples then return *true*

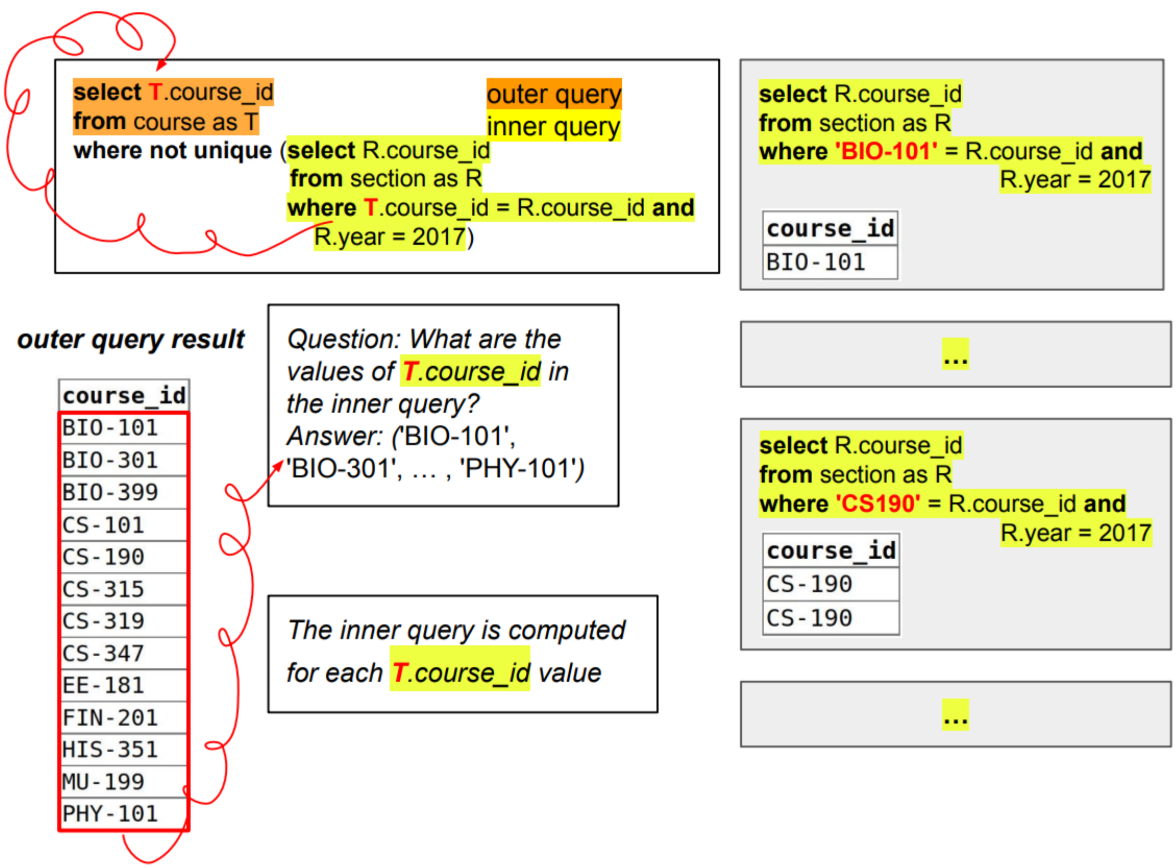
```
not unique (subquery)
```

Example:

- Find all courses that were offered *at most once* in 2017



- Find all courses that were offered *at least twice* in 2017



Correlated Subqueries

- A subquery that uses a correlation attribute from an outer query
- Semantic: for each tuple obtained from the outer query, compute the inner query
- Correlated subqueries can be used in the `select` and `where` clauses of SFW queries
- Nested subqueries in the `from` clause cannot use correlation variables from other relations in the same `from` clause, unless the subqueries are prefixed by the `lateral` keyword

Example:

- Find all instructors whose salary is above average for their department

```
select name
from instructor as S
where salary > (select avg(salary)
                from instructor
                where dept_name = S.dept_name)
```

instructor

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

query result

name
Wu
Einstein
Califieri
Brandt

- Print the names of each instructor, along with their salary and the average salary in their department

```
select name, salary,
    (select avg(salary)
    from instructor
    where dept_name = S.dept_name) as dept_avg
from instructor as S
```

instructor

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

query result

name	salary	dept_avg
Crick	72000	72000
Srinivasan	65000	77333.33333333333
Katz	75000	77333.33333333333
Brandt	92000	77333.33333333333
Kim	80000	80000
Wu	90000	85000
Singh	80000	85000
El Said	60000	61000
Califieri	62000	61000
Mozart	40000	40000
Einstein	95000	91000
Gold	87000	91000

– An alternative way to do the same thing:

```
select name, salary, dept_avg
from instructor T,
    lateral (select avg(salary) as dept_avg
             from instructor S
             where T.dept_name = S.dept_name)
```

`with` Clause

To define a temporary relation:

```
with temp_r (list_of_attributes) as (subquery)
```

Example: find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
    from instructor
    group by dept_name),
dept_total_avg (value) as
    (select avg(value)
    from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value
```

dept_total

dept_name	sum(salary)
Biology	72000
Comp. Sci.	232000
Elec. Eng.	80000
Finance	170000
History	122000
Music	40000
Physics	182000

dept_total_avg

avg(value)
128285.71428571429

query result

dept_name
Comp. Sci.
Finance
Physics

By default just use the `with` clause, it is almost powerful enough to do almost anything you need.

Data Modification

Updating Table Schema

`alter table r add A D`

- Add attribute A of type D to table r
- For existing tuples in r, the values of A are assigned `null`

`alter table r drop A`

- Drop attribute A in table r
- Dropping of attributes is not supported by many databases (e.g. SQLite)

```
alter table r rename column old_name to new_name
```

- Rename column *old_ name* to column *new_ name* in table *r*

```
alter table r modify A data_type
```

- Change the type of attribute *A* to *data_ type* in table *r*

Deletion

```
drop table instructor
```

- Delete the instructor relation (instance + schema)

```
delete from instructor
```

- Delete all instructors (instance)

```
delete from instructor
where dept_name = 'Finance'
```

- Delete all instructors from the Finance department

```
delete from instructor
where dept_name in (select dept_name
                    from department
                    where building = 'Watson')
```

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

```
delete from instructor
where salary < (select avg (salary)
                from instructor)
```

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

Insertion

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

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

- Add a new tuple to *course*

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

- Add a new tuple to *student* to tot_creds set to `null`
- When inserting partial rows, the values of omitted attributes are set to `null`

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

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

```
insert into student
    select *
    from student
```

- The insertion will insert infinite tuples if the primary key constraint on student is absent

Note: SQL evaluates the select statement fully before it performs any insertions

Update

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

- Give a 5% salary raise to all instructors

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

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

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

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

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

- Increase salaries of instructors whose salary is over $\$100,000$ by 3% and all others by 5%
- Note that the order is important

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

- `case` can be used in any statement or clause that allows a valid expression

Integrity Constraints

Declared as part of the schema and enforced by the DBMS

- Restrictions on allowable data in a database
  - In addition to the simple structure and type restrictions imposed by the table definitions
- Example:
  - An instructor name cannot be null
  - No two instructors can be have the same insturctor ID
  - Budgest of a department must be greater than $0.00
  - Every dept name in course relation must have a matching dept name in the dept relation

Type of SQL Constraints:

- not null
- primary key
- unique
- check (<cond>)
- foreign key
- assertion
  - specifying general constraints but not supported by *any* DBMS

not null Constraint

not null prohibits the insertion of a null value for an attribute

Example:

Null is not allowed for instructor name.

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

instructor

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000
00007	**null**	**Comp. Sci.**	**10000**

Reject

`primary key` Constraint

`primary key` prohibits the insertion of values that already exist for attributes

- Only one `primary key` constraint per table
- Does not permit null values

Example:

Another form: ID varchar(5) primary key,

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

At most one primary key per table

instructor

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000
22222	James	Comp. Sci.	10000

Reject

```
create table section
   (course_id    varchar (8),
    sec_id       varchar (8),
    semester     varchar (6),
    year         numeric (4,0),
    building     varchar (15),
    room_number  varchar (7),
    time_slot_id varchar (4),
    primary key (course_id, sec_id, semester, year),
    foreign key (course_id) references course);
```

Primary key of multiple attributes

`unique` Constraint

`unique` specifies that no duplicate tuples are allowed for attributes

- Any number of `unique` constraints per table
- Permits null values (if specified without `not null`)

Example:

```
create table user
   (uid        varchar(10)  primary key,
    name       varchar(30),
    twitter_id varchar(15)  not null unique,
    phone_no   varchar(15)  unique)
```

uid	*name*	*twitter_id*	*phone_no*	
101	James	tw_1	2268990000	
102	Lee	tw_3	null	
103	Mark	tw_6	2268990000	Reject
104	Henry	tw_3	2268990001	Reject
105	Long	tw_7	null	Accept

Modern DBMSs allow multiple null values for a unique constraint, e.g., PostgreSQL, MySQL, etc.

`check` Constraint

`check (<cond>)` ensures that the check condition is not *false*

- Only checked when the tuple/attribute is inserted/updated
- Accepted if the condition returns *true* or *unknown*

Example:

```
create table department
    (dept_name    varchar (20),
     building     varchar (15),
     budget       numeric (12,2) check (budget > 0),
     primary key (dept_name));
```

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000
Law	Painter	0
Chemistry	Watson	null

Law, Painter, 0: Reject

Chemistry, Watson, null: Accept

`foreign key` Constraint

`foreign key attr references T` ensures that the value of *attr* exists the in the table *T*

Example: if dept_name appears in *instructor*, it must appear in *department*

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

referencing relation

```
create table department
    (dept_name    varchar (20),
     building     varchar (15),
     budget       numeric (12,2),
     primary key (dept_name));
```

referenced relation

instructor

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

department

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

- `foreign key` specifies that:
  - a value that appears for a given set of attributes in one relation (*referencing relation*)
  - also must appear for a certain set of attributes in another relation (*refereneced* relation)
- *Referenced* column(s) must be either `primary key` or have explicit `unique` constraints
- *Referencing* column(s) form a foreign key

Remarks:

- If the referenced attributes are omitted, then the foreign key references the primary key
  - e.g. `foreign key (dept_name) references department`
- Attributes of `foreign key` are allowed to be `null` (if not declared to be `not null`)
  - In this case the foreign key constraint is said to be satisfied

Foreign Key Constraint Enforcement

- Modification to the *referencing* relation
  - Foreign key does not impose constraints on *deletion*
  - *insertion* and *updates* require verifying the foreign key constraints
- Modifications on *referenced* relation
  - Foreign key does not impose constraints on *insertion*
  - *deletion* and *updates* require verifying the foreign key constraints
    - Option 1: throw an error (default)
    - Option 2: set referencing attributes to `null`
    - Option 3: cascade the changes to referencing attributes

Example: insertion in referencing relation

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

referencing relation

```
create table department
   (dept_name     varchar (20),
    building      varchar (15),
    budget        numeric (12,2),
    primary key (dept_name));
```

referenced relation

instructor

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
98449	James	Law	89000
98500	Bob	null	10000
98510	Alice	Comp. Sci.	100000

98449 James Law 89000: Reject

98500 Bob null 10000: Accept

98510 Alice Comp. Sci. 100000: Accept

department

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

Example: deletion in referenced relation

- Option 1: throw an error

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

referencing relation

```
create table department
   (dept_name     varchar (20),
    building      varchar (15),
    budget        numeric (12,2),
    primary key (dept_name));
```

referenced relation

instructor

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

department

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

Reject

Why?
Comp. Sci. is referenced by multiple tuples in instructor!

- Option 2: set referencing attributes to `null`

```
create table instructor
   (ID          varchar (5),
    name        varchar (20) not null,
    dept_name   varchar (20),
    salary      numeric (8,2),
    primary key (ID),
    foreign key (dept_name) references department
    on delete set null
```

referencing relation

```
create table department
   (dept_name   varchar (20),
    building    varchar (15),
    budget      numeric (12,2),
    primary key (dept_name));
```

referenced relation

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	**null**	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	**null**	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	**null**	92000
98345	Kim	Elec. Eng.	80000

department

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

Accept

Why?
The definition of the foreign key specifies **set null.**

- Option 3: cascade the chagnes to referencing attributes

```
create table instructor
   (ID          varchar (5),
    name        varchar (20) not null,
    dept_name   varchar (20),
    salary      numeric (8,2),
    primary key (ID),
    foreign key (dept_name) references department
    on delete cascade
```

referencing relation

```
create table department
   (dept_name   varchar (20),
    building    varchar (15),
    budget      numeric (12,2),
    primary key (dept_name));
```

referenced relation

instructor

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

department

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

Accept

Why?
The definition of the foreign key specifies **cascade.**

Deferred Constraint Checking

In some cases due to constraint requirements the first insertion will always violate a constraint:

```
create table dept
    (name char(20) primary key,
    chair char(30) not null,
    foreign key (chair) references prof(name));

create table prof
    (name char(30) primary key,
    dept char(20) not null,
    foreign key (dept) references dept(name));
```

Since dept and prof reference each other we cannot insert either

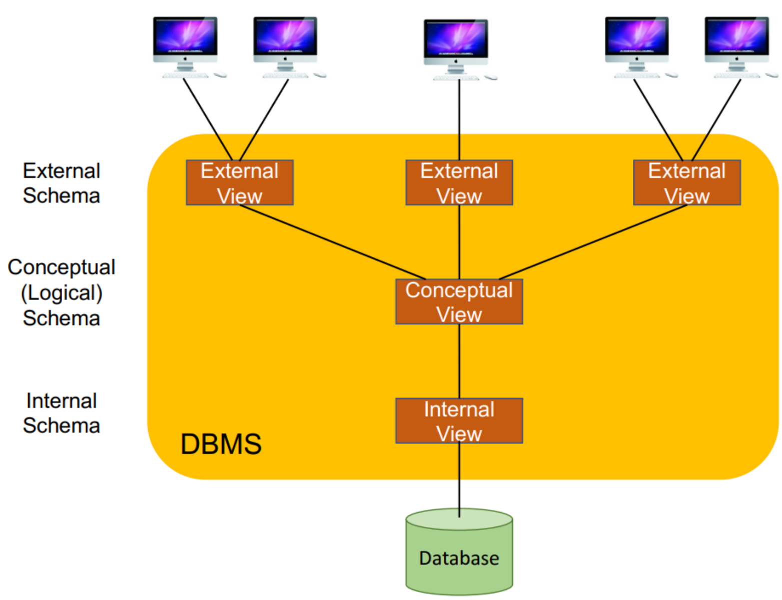
In such a case *deferred constraint checking* is necessary

- Check only at the end of a transaction (of multiple insertions)
- Allowed in SQL as an option (go read a manual)

Views

Recall the three-level schema architecture:

- External schema
- Conceptual schema
- Physical schema



A view is like a *virtual* table:

- Defined by a query, which describes how to compute the view contents
- Can be used in queries just like a regular table

- Stored as a query expression by a DBMS, instead of in actual tables

Views are used to hide complexity and data from users for logical data independence.
(we can change the schema of actual data, and person still calls same view)

Example:

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

```
create view physics_fall_2017 as
  select course.course_id, sec_id, building, room_number
  from course, section
  where course.course_id = section.course_id
        and course.dept_name = 'Physics'
        and section.semester = 'Fall'
        and section.year = '2017';

create view physics_fall_2017_watson as
   select course_id, room_number
   from physics_fall_2017
   where building= 'Watson';
```

Updating Views

Requirements for a view to be updateable:

- The `from` clause can have only one database relation
- `select` clause contains only attribute names of the relation
  - Does not have any expressions, aggregates, or `distinct` specification
- any atribute not listed in the `select` clause can be set to `null`
- The query does not have a `group by` or `having` clause

i.e. an SFW query on a single relation without arithmetic expressions

Note that even if we follow all these there are still some cases where things go wrong:

```
create view history_instructors as
    select *
    from instructor
    where dept_name='History'
    with check option;
```

This will reject insertions that do not satisfy the `where` clause condition

Example: notice that the tuple created in the actual table has `null` values

```
create view instructor_info as
    select ID, name, building
    from instructor, department
    where instructor.dept_name = department.dept_name;
insert into instructor_info
    values ('69987', 'White', 'Taylor');
```

Will the tuple be included in the result of the following query?

```
select * from instructor_info
```

instructor

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
69987	White	*null*	*null*

department

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

Access Control

Authorize users a combination of privileges (select, insert, update, delete) on relations, views, etc.

- e.g. student can't see other students' grades
- e.g. the instructor can assign/update grades to only their own students

Granting and Revoking Privileges

Privilege list: `select`, `insert`, `update`, `delete`, `all`

- `grant <privilege list> on <relation or view> to <user list>`
- `revoke <privilege list> on <relation or view> to <user list>`

Note: the grantors must hold the privilege they are granting

Example:

- `grant select on department to Amit, Satoshi`
- `revoke select on department from Satoshi`

Roles

Instead of managing privileges on the individual level, we can grant privileges to roles.

Example:

```
create role instructor;
grant select on takes to instructor;
create role dean;
grant instructor to dean;
grant dean to Satoshi;
```

Transfer of Privileges

Examples:

- `grant select on department to Amit with grant option`
  - Give Amit select privilege on *department* and allow Amit to grat the privilege to others
- `revoke select on department from Amit restrict`
  - Revoke select privilege on *department* from Amit
- `revoke select on department from Amit cascade`
  - Revoke select privilege on *department* from Amit and others granted by Amit

Indexes

An *index* is an auxiliary *persistent* data structure to speed up operations.

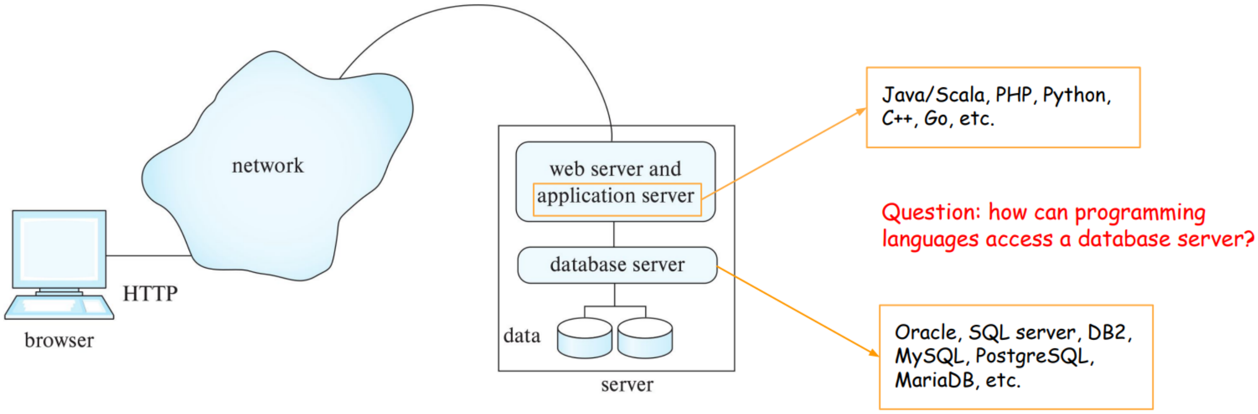
- Search tree (e.g. B+-tree), lookup table (e.g. hash table), etc.
- Typically created automatically by the DBMS for `primary key` and `unique` attributes
- An index on $R.A$ can speed up accesses of the form:
  - `R.A = value`
  - `R.A > value` (depending on the index type)

Example:

```
create index ins_name_index on instructor (name);
create unique index ins_name_index on instructor (name);
drop index ins_name_index
```

An error will occur on the second command if *name* is not a candidate key

SQL from a Programming Language



Websites	Popularity (unique visitors per month)[1]	Front-end (Client-side)	Back-end (Server-side)	Database	Notes
YouTube	1,100,000,000	JavaScript, TypeScript	C, C++, Java,[11] Go[12]	Vitess, BigTable, MariaDB[5][13]	The most popular video sharing site.
Facebook	1,120,000,000	JavaScript, Typescript, Flow	Hack, PHP (HHVM), Python, C++, Java, Erlang, D,[6] XHP,[7] Haskell[8]	MariaDB, MySQL,[9] HBase, Cassandra[10]	The most visited social networking site.
Amazon	2,400,000,000[19]	JavaScript	Java, C++, Perl[20]	DynamoDB, RDS/Aurora, Redshift[21]	The most used e-commerce site in the world.
Google[2]	2,500,000,000	JavaScript, TypeScript	C, C++, Go,[3] Java, Python, Node, PHP	Bigtable,[4] MariaDB[5]	The most used search engine in the world.
Netflix	223.090.000 (Subscribers, not visitors)	JavaScript	Python, Java[39]	NMDB,[40] PostgreSQL	The biggest video streaming service in the world.
WordPress.com	240,000,000 [36]	JavaScript	PHP[37]	MariaDB[38]	Website manager software.
Pinterest	250,000,000	JavaScript	Python (Django),[33] Erlang, Elixir[34]	MySQL, Redis[35]	Search engine for ideas.
LinkedIn	260,000,000	JavaScript	Java, JavaScript,[30] Scala	Venice[31][32]	World's largest professional network.
MSN	280,000,000	JavaScript	C# (ASP.NET)	Microsoft SQL Server	An email client, for simple use. Previously known as "messenger", not to be confused with Facebook's messaging platform.
Bing	285,000,000	JavaScript	C++, C#	Microsoft SQL Server, Cosmos DB	Search engine from Microsoft.
eBay	285,000,000	JavaScript, HTML	Java,[27] JavaScript,[28] Scala[29]	Oracle Database	Online auction house.
Twitter	290,000,000	JavaScript	C++, Java,[24] Scala,[25] Ruby (Ruby On Rails)	MySQL[26]	Popular social network.

We have two method to write SQL in a programming language:

- Embedded SQL: example in C

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main() {
   EXEC SQL WHENEVER SQLERROR GOTO error;
   EXEC SQL CONNECT TO sample;
   EXEC SQL UPDATE Employee
            SET salary = 1.1*salary
            WHERE empno = '000370';
   EXEC SQL COMMIT WORK;
   EXEC SQL CONNECT RESET;
   return(0);
error:
   printf("update failed, sqlcode = %ld\n",SQLCODE );
   EXEC SQL ROLLBACK WORK
   return(-1);
}
```

SQL statement

 - SQL queries are hard coded into the program
 - Translated into function calls at compile time by preprocessors

- Dynamic SQL: example in Java

```
public static void JDBCexample(String userid, String passwd)
{
    try (
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
                userid, passwd);
        Statement stmt = conn.createStatement();
    ) {
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987','Kim','Physics',98000)");
        }
        catch (SQLException sqle) {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select dept_name, avg (salary) "+
                " from instructor "+
                " group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept_name") + " " +
                    rset.getFloat(2));
        }
    }
    catch (Exception sqle)
    {
        System.out.println("Exception : " + sqle);
    }
}
```

SQL statement

Main Focus

 - SQL query as a string
 - String is submitted as the query and retrieve the result into program varaibles a tuple at a time

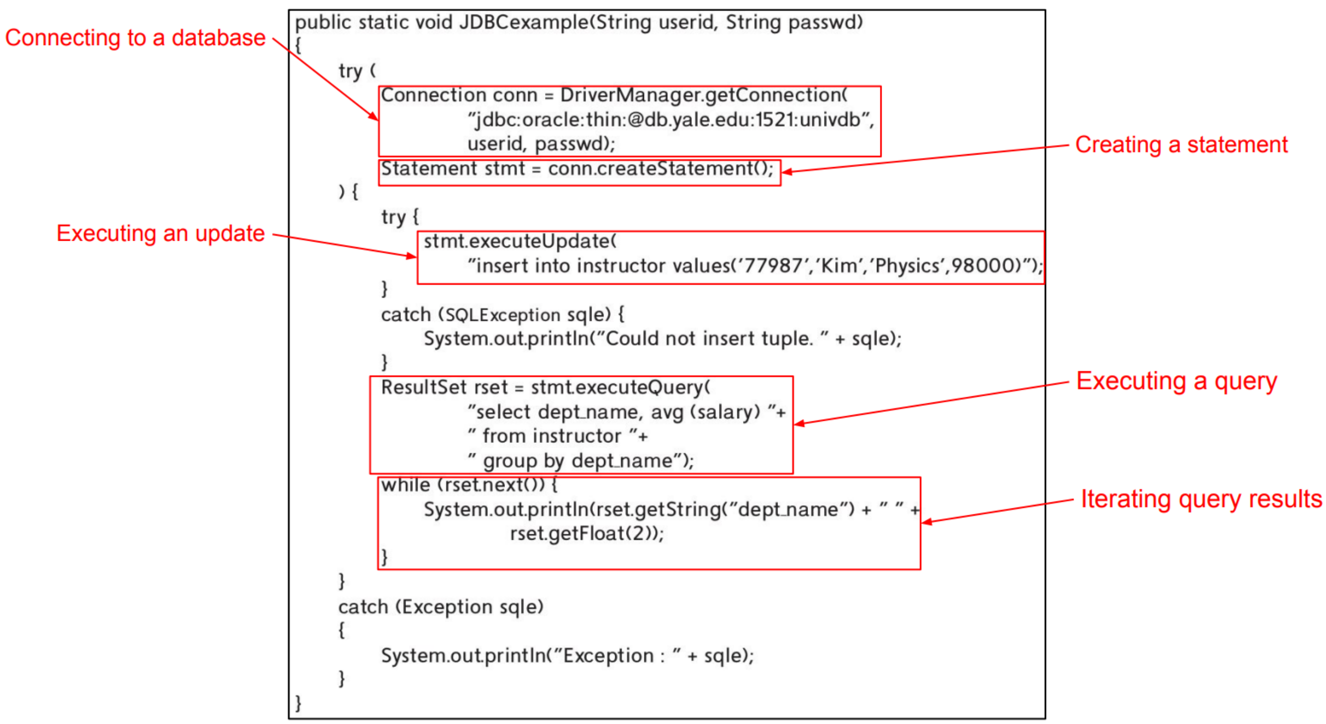
- JDBC (Java), ODBC (C/C++/VB), Python (psycopg2), etc.
 - All based on the SQL/CLI (Call-Level Interface) standard
- Application program sends SQL commands to the DBMS at runtime
- Responses/results are converted to objects in the application program

JDBC

JDBC is the Java API for communicating with database systems supporting SQL

- Variety of features for querying and updating data along with retrieving query results
- Able to perform metadata retrieval such as names of tables and names and types of table attributes
- Model for communicating with the database:
  1. Open a connection
  2. Create a *statement* object
  3. Execute queries using the statement object to send queries and fetch results

Example: a breakdown of the JDBC example we saw earlier



- Executing statements:
  - `executeQuery()` for `select`
    * e.g. `stmt.executeQuery("select ...")`
  - `executeUpdate()` for `update`, `insert`, `delete`, and `create table`
    * e.g. `stmt.executeUpdate("insert ..")`
- Geting result fields:
  - The following is equivalent if dept_name is the first attribute of the resulting relation
    * e.g. `resultSet.getString("dept_name")`
    * e.g. `resultSet.getString(1)`

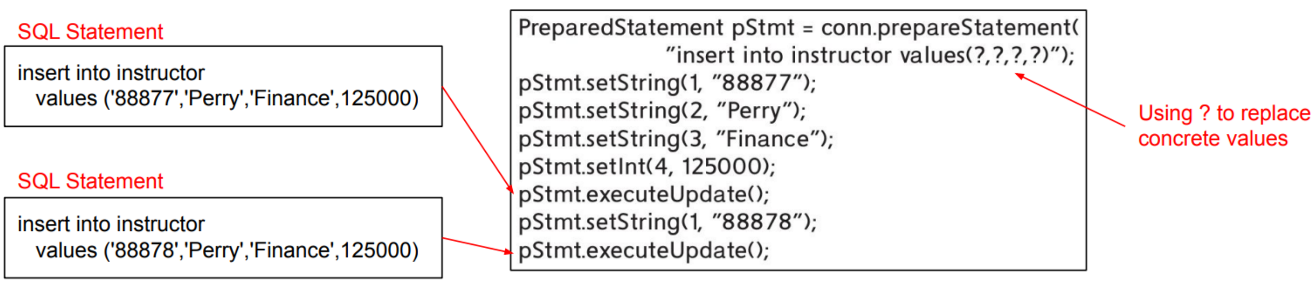
- Dealing with `null` values:
  - Call `get()` to get a value of an attribute then to check if `null` use `resultSet.wasNull()`
  - e.g.

```
int a = rs.getInt("a");
if (rs.wasNull())
    Systems.out.println("Got null value");
```

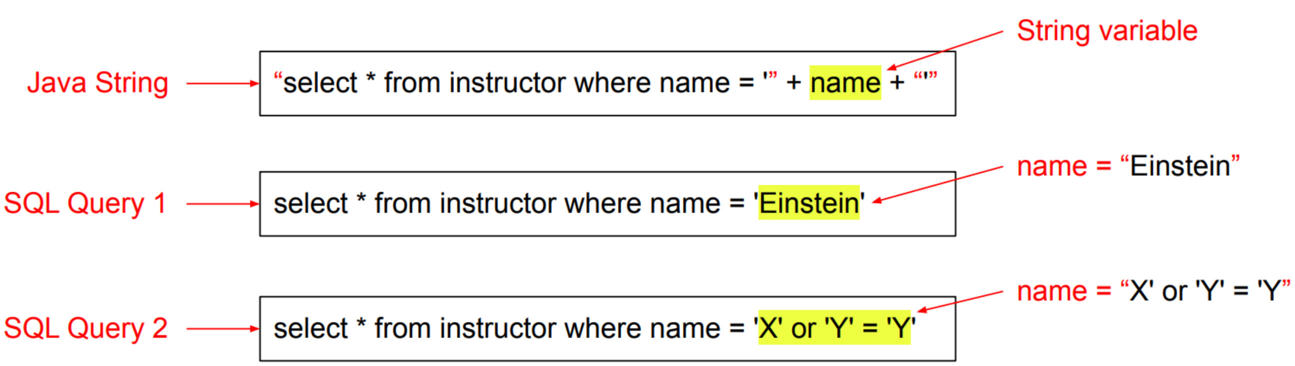
Prepared Statements

A *prepared statement* is a precompiled SQL statement (precompile once then use many times)

Example:



Example: SQL Query 2 is an examples of a SQL injection where the quote is escaped with another quote:



Using prepared statements would prevent this attack because the input string would have escaped.

Metadata Features

Example: print out the names of types of all columns of a result set (resulting relation)

```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

Functions and Procedures

Persistent Storage Module (PSM): provides constructs that give SQL almost all the power of a general-purpose programming language.

```
create function func_name(param_decls)
returns return_type
    local_decls
    func_body;
```

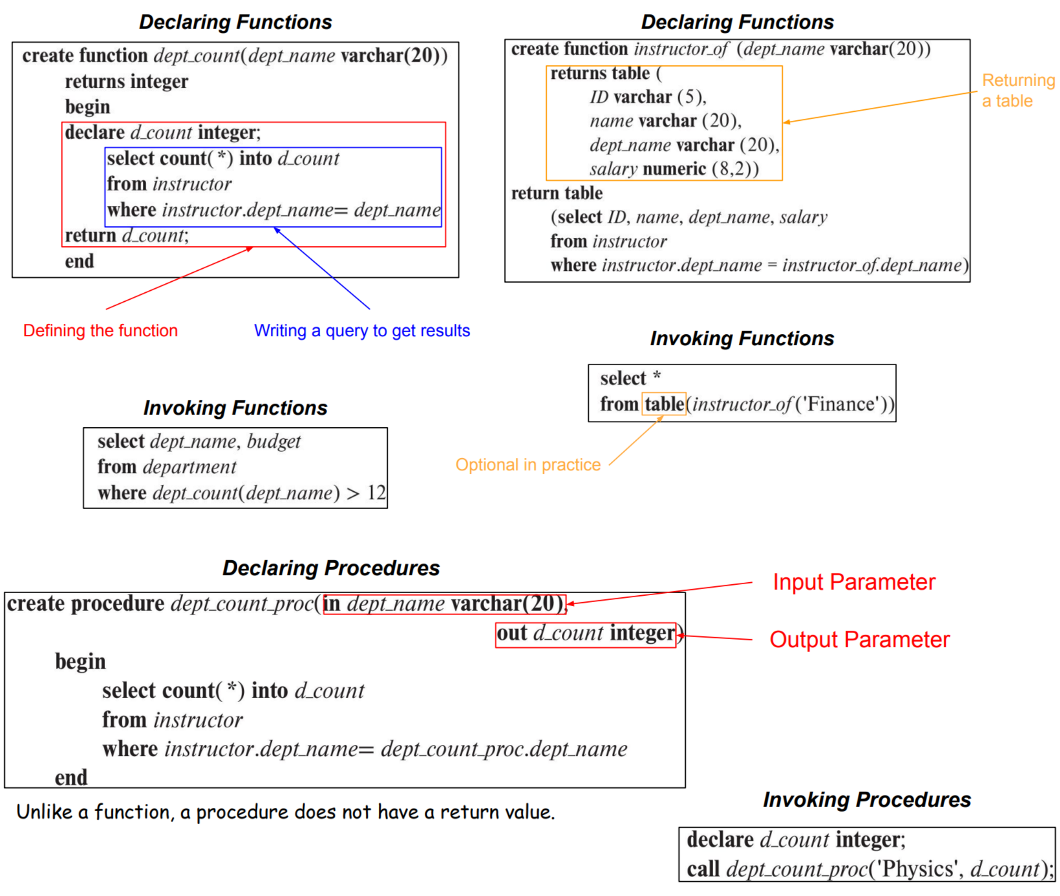
```
create procedure proc_name(param_decls)
    local_decls
    proc_body;
```

```
call proc_name(params);
```

Inside the function/procedure body we have:

- Variables: `set variable = call func_name(params);`
- Assignment using scalar query results: `select ... into ...`
- Loop constructs: `for`, `repeat until`, `loop`
- Flow control: `if-else-then`
- Exception handlers (check your DBMS manual)

Examples:



While Statements

while *boolean expression* **do**
  sequence of statements;
end while

If-Then-Else Statements

if *boolean expression*
  then *statement or compound statement*
elseif *boolean expression*
  then *statement or compound statement*
else *statement or compound statement*
end if

An Example of a For Loop

declare *n* **integer default** 0;
for *r* **as**
  select *budget* **from** *department*
  where *dept_name* = 'Music'
do
  set *n* = *n*− *r.budget*
end for

r is a tuple of one attribute, i.e., *budget*

Remark: DBMS implementations have non-standard versions of the SQL standard syntax (e.g. Oracle, Microsoft SQL Server, PostgreSQL all differ from standard)

Example: PostgresSQL function and procedure:

```
create or replace function get_film (
        p_pattern varchar,
        p_year int
)
returns table (
        film_title varchar,
        film_release_year int
)
language plpgsql
as $$
declare
    var_r record;
begin
        for var_r in(
            select title, release_year
            from film
             where title ilike p_pattern and
                    release_year = p_year
        ) loop  film_title := upper(var_r.title) ;
                film_release_year := var_r.release_year;
           return next;
        end loop;
end; $$
```

A record is a tuple

A for loop over the resulting relation

The computation for each tuple

Adding a row to the returned table

```
create or replace procedure transfer(
   sender int,
   receiver int,
   amount dec
)
language plpgsql
as $$
begin
    -- subtracting the amount from the sender's account
    update accounts
    set balance = balance - amount
    where id = sender;

    -- adding the amount to the receiver's account
    update accounts
    set balance = balance + amount
    where id = receiver;

    commit;
end;$$
```

```
call transfer(1,2,1000);
```

Transferring $1000 from account 1 to account 2

Triggers

A *trigger* is an *event-condition-action* (ECA) rule

- When *event* occurs, test *condition*, if condition s satisfied, execute *action*
- This is a generalization of the integrity condition constraints we saw earlier

Example:

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
          select time_slot_id
          from time_slot)) /* time_slot_id not present in time_slot */
begin
  rollback
end;
```

Transition variable: stores the values of the inserted row (nrow)

Event: after insert on section

Condition: when (...) clause

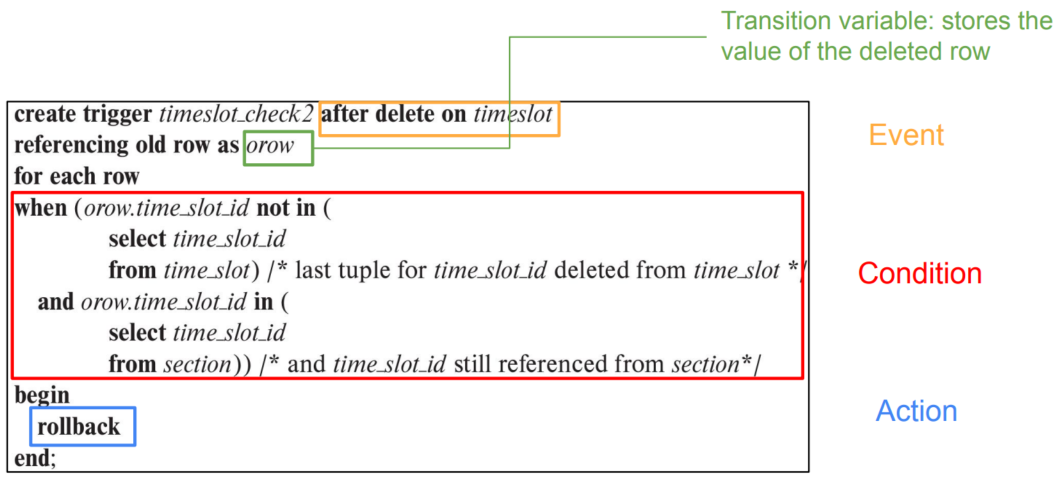
Action: rollback

Trigger Events

- The types of events include:
  - `insert` on table
  - `delete` on table
  - `update` on table

- The action can be executed:
  - `after` the data is modified
  - `before` the data is modified

Example:



Granularity

The triggers can be activated:

- *for each row*:
  - Fires once for each row affected by the triggering event
    - If no rows where modified then trigger does not fire
  - `referencing new row` or `referencing old row`
- *for each statement*:
  - Fires once per triggering event
    - Regardless of whether any rows are modified
  - `referencing new table` or `referencing old table`
    - Refers to temporary tables (aka transition tables) containing affected rows
  - Can only be used with `after` triggers

Statement-level triggers can be more efficient when dealing with SQL statements that update many rows.

SQLite only implements *for each row*.

Advanced Aggregations

Ranking

Finds a ranking value for each row.

- Multiple values can have same rank but the next rank will be number of elements before it

- e.g. 1: P8, 2: P2, 2: P5, 4: P9, etc

- Rank of `null` values can be controlled by:
  - `nulls first` (default)
  - `nulls last` can be added after `desc`
- Use `partition by` to perform ranking within partitions of data

Example:

select *ID*, **rank**() **over** (**order by** (*GPA*) **desc**) **as** *s_rank*
from *student_grades*;

student_grades

ID	GPA
003	83
005	83
002	98
001	79
004	75

ID	s_rank	GPA
003	2	83
005	2	83
002	1	98
001	4	79
004	5	75

Computing rankings in descending order based on the GPA column

select *ID*, **rank** () **over** (**order by** (*GPA*) **desc**) **as** *s_rank*
from *student_grades*
order by *s_rank*;

ID	s_rank
002	1
005	2
003	2
001	4
004	5

select *ID*, *dept_name*,
rank () **over** (**partition by** *dept_name* **order by** *GPA* **desc**) **as** *dept_rank*
from *dept_grades*
order by *dept_name*, *dept_rank*;

dept_grades

ID	dept_nam	GPA
003	Comp. Sci.	83
005	Elec. Eng.	83
002	Comp. Sci.	98
001	Comp. Sci.	79
004	Elec. Eng.	75
006	Elec. Eng.	92

ID	dept_nam	dept_rank	GPA
002	Comp. Sci.	1	98
003	Comp. Sci.	2	83
001	Comp. Sci.	3	79
006	Elec. Eng.	1	92
005	Elec. Eng.	2	83
004	Elec. Eng.	3	75

Computing rankings in descending order based on the GPA column within each department

- Multiple `rank` clauses can occurs in a single `select` clause
- Ranking is done after applying `group by` clause or aggregation
- Ranking can be used to find top n results
  - e.g. find the top 5 ranking students based on GPA

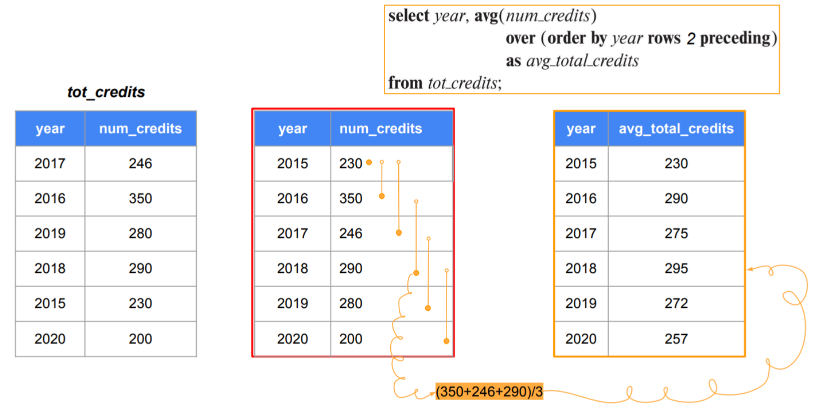
```
select *
from (select ID, rank() over (order by (GPA) desc) as s_rank
      from student_grades)
where s_rank <= 5
```

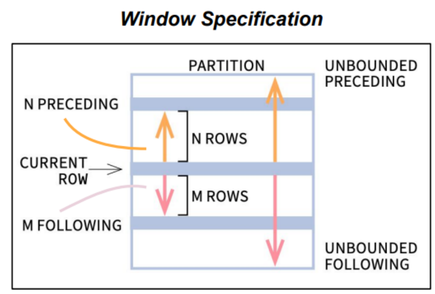
  - More general than `limit` n clause, since it allows top n within each partition

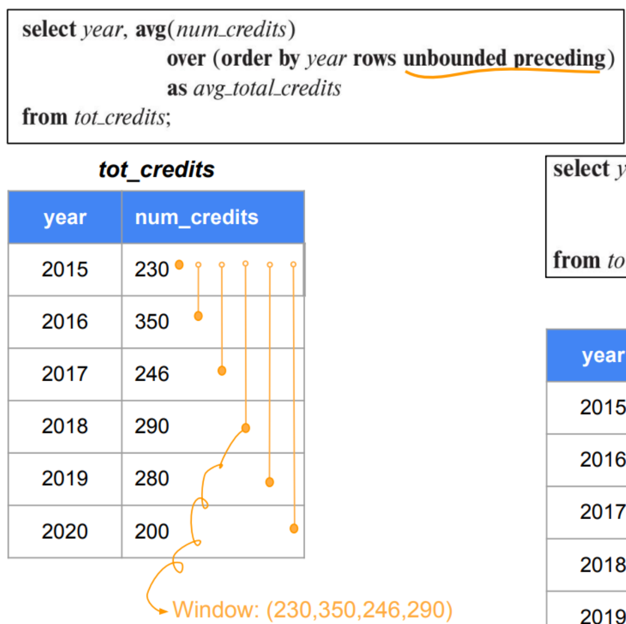
Windowing

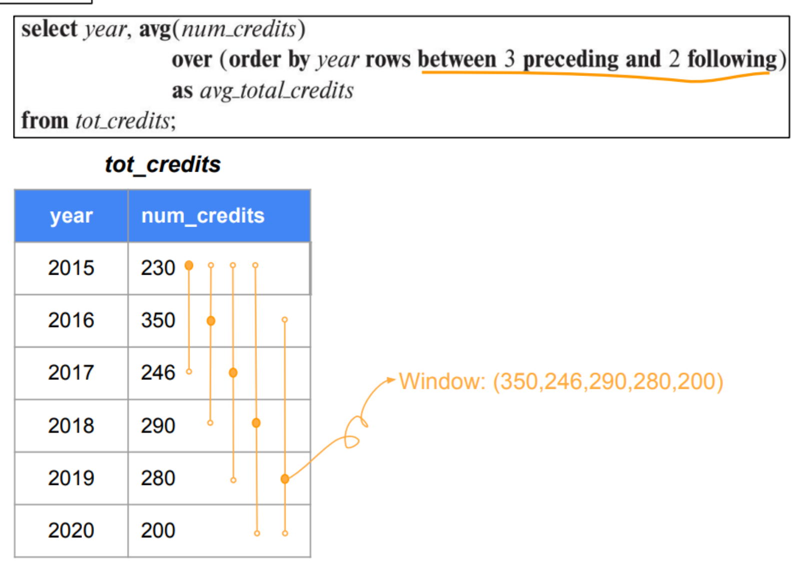
Compute an aggregation function over a range of tuples

Examples:









Recursion

We perform recursion in SQL in a simular way to regular programming:

- **Fixed point**: there is no further changes in the result of the recursive query evaluation
- Reaching the fixed point indicates we can terminate the recursive query

Example: union our current query with the past one until union no longer adds anything

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id
        from rec_prereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select *
from rec_prereq;
```

Base query

Recursive query

Example: remove courses we have already found

```
create function findAllPrereqs(cid varchar(8))
    -- Finds all courses that are prerequisite (directly or indirectly) for cid
returns table (course_id varchar(8))
    -- The relation prereq(course_id, prereq_id) specifies which course is
    -- directly a prerequisite for another course.
begin
    create temporary table c_prereq (course_id varchar(8));
        -- table c_prereq stores the set of courses to be returned
    create temporary table new_c_prereq (course_id varchar(8));
        -- table new_c_prereq contains courses found in the previous iteration
    create temporary table temp (course_id varchar(8));
        -- table temp is used to store intermediate results
    insert into new_c_prereq
        select prereq_id
        from prereq
        where course_id = cid;
    repeat
        insert into c_prereq
            select course_id
            from new_c_prereq;

        insert into temp
            (select prereq.prereq_id
                from new_c_prereq, prereq
                where new_c_prereq.course_id = prereq.course_id
            )
            except (
                select course_id
                from c_prereq
            );
        delete from new_c_prereq;
        insert into new_c_prereq
            select *
            from temp;
        delete from temp;

    until not exists (select * from new_c_prereq)
    end repeat;
    return table c_prereq;
end
```

Iterations

Join the result of the previous iteration with the prereq relation

Remove courses that have been found

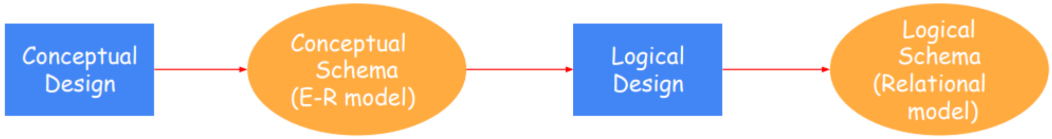
Prerequisites of CS-347 in iterations of function findAllPrereqs

Iteration number	Tuples in c_prereq
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)

Data Modeling

The goal is to be able to convert a written specification into a database schema

- Step 1: understand the real-world domain being modeled
  - Specify these considerations into an entity-relationshiop (E-R) model
- Step 2: translate this into the data model of the DBMS
  - Create the relational model



There are two major pitfalls that should be avoided

- **Redundancy**: storing multiple copies of the same data
  - Inconsistencies will occur if all the copies of the data is not updated at the same time
- **Incompleteness**: unable to store all valid data
  - We want our database to be complete with all the data it should have

Entity-Relationship Model (E-R Model)

E-R diagrams were proposed to help with designing database schema and described the world in terms of:

- Entities
- Relationships
- Attributes on entities and relationships

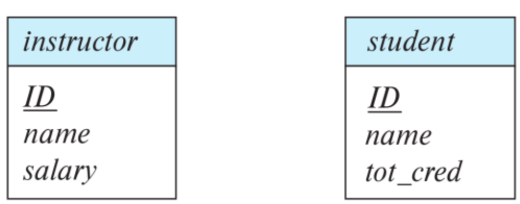
Entity Set

- *Entity*: is an object that exists and is distinquishable from other objects (an instance)
- *Entity set*: set of entities of the same type that share the same properties (or attributes)
- An entity is represented its *attributes* which are properties all members of the entity set possess
- Subset of the attributes form a *primary key* of the entity set

An entity set is represented using:

- Each rectangle is its own entity set
- Attributes are listed inside the rectangle
- Underline indicates the primary key attributes

Example:



A specific entity of the *instructor* entity set could be (ID: 1, name: Joe, salary: $8)

Relationship Set

- *Relationship*: an association among multiple entities
- *Relationship Set*: mathematical irelation among $n \geq 2$ entities
  - Let $E_1, \ldots, E_n$ be entity sets, then a relationship set is a subset of

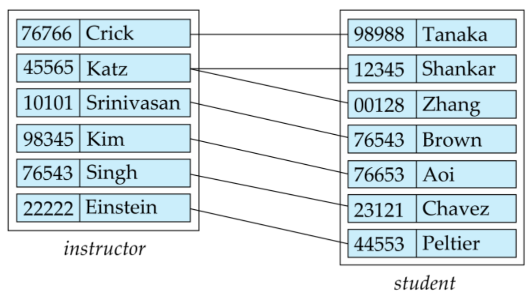
$$\{(e_1, \ldots, e_n) : e_1 \in E_1, \ldots, e_n \in E_n\}$$

    where $(e_1, \ldots, e_n)$ is a relationship instance

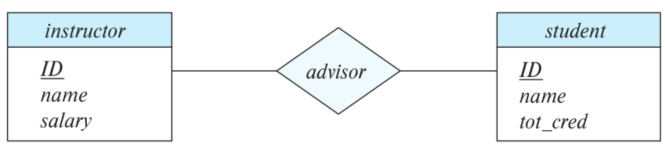
Diamonds are used to represent the relationship set and an attribute can also be associated.

Examples:

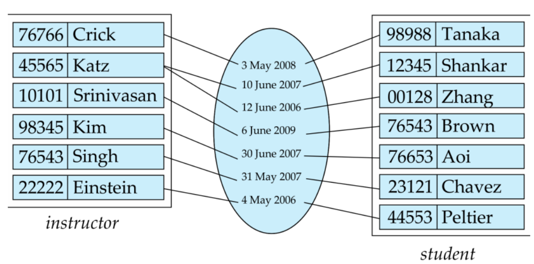
- Relationship set advisor denotes associations between students and instructors



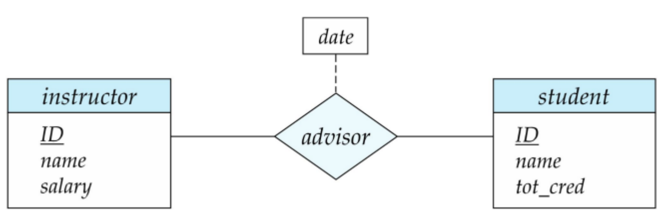
  - In E-R notation the relationship set is denoted:



- Relationship set advisor can have an attribute



  - In E-R notation the relationship set is denoted:

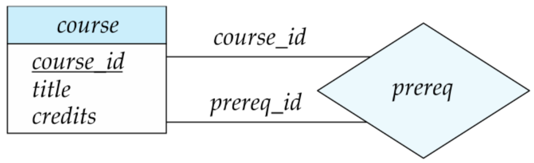


Roles

An entity set may participate more than once in a relationship set

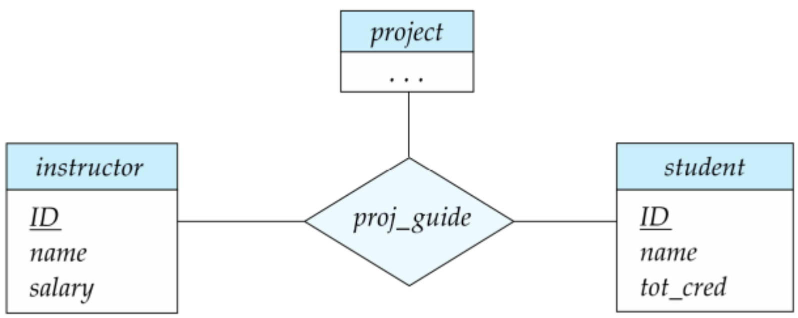
- Each occurrence fo an entity set plays a *role* in the relationship

Example: the labels `course_id` and `prereq_id` are called *roles*



Relationship Set Degree

- *Binary Relationship*:
  - Involves exactly two entity sets (degree two)
  - Most relationship sets in a database system are binary
- *Non-binary Relationship*:
  - On occasion it is more convenient to represent relationships as non-binary
  - e.g. *students* work on research *projects* under the guidance of an *instructor*



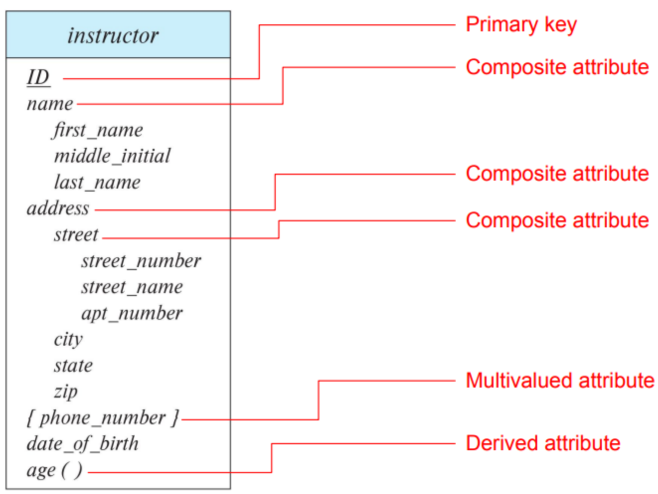
proj_ guide is a ternary relationship (degree three) between *instructor*, *student*, and *project*

Attributes

The types of attributes:

- *simple* and *composite* attributes
  - Whether the attribute can be divided into other attributes
  - e.g. *address* is a composite attribute because it could consist of the attributes *street*, *city*, *state*, and *postal code*
- *single-valued* and *multivalued* attributes
  - e.g. if an instructor can be multiple phone numbers then *phone_ numbers* is a multivalued attribute
- *derived* attributes
  - Attributes that can be computed from other attributes
  - e.g. *age* attribute can be derivated when required from *date_ of_ birth*

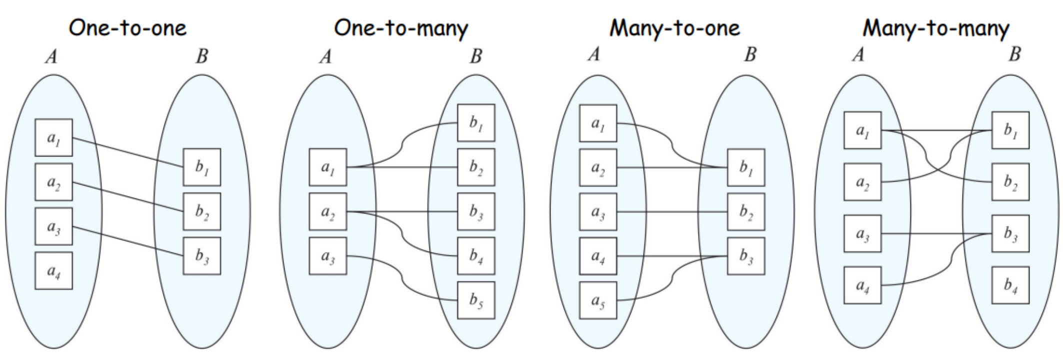
Example:



Mapping Cardinality Constraints

Express the number of entities that can be assocated from one set to another via a relationship set.

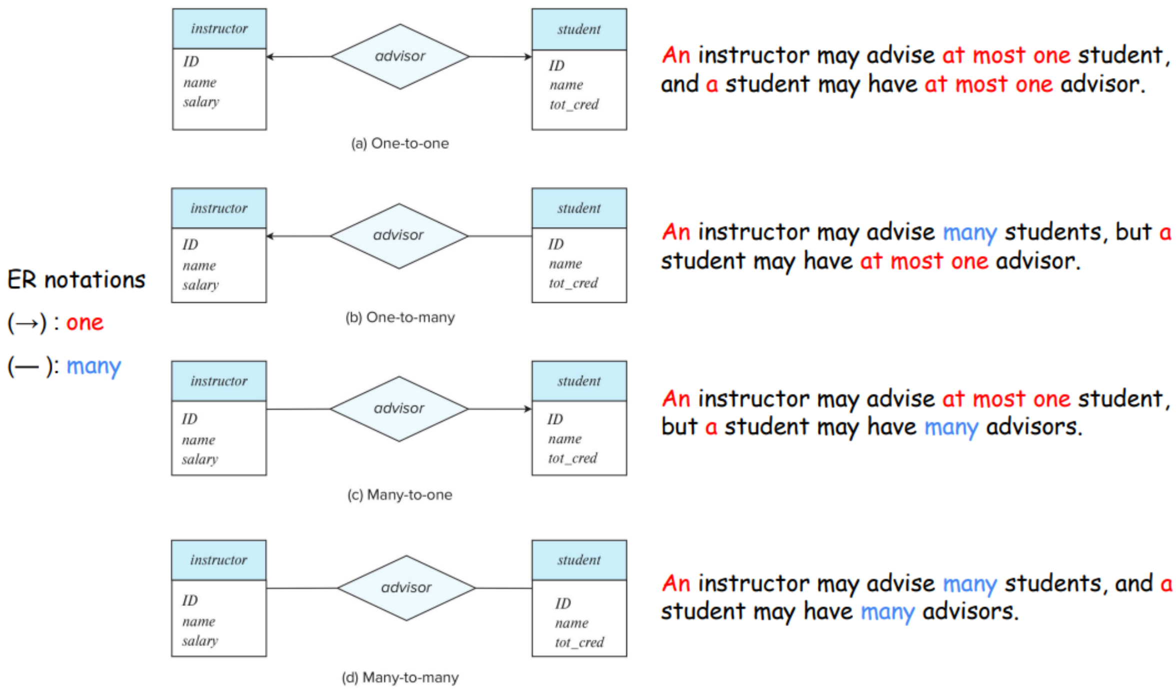
For binary relationship sets we have the mapping cardinality can be:



Note: some elements in A or B could be not mapped to any elements in other set

Example: in E-R notation

- Arrow: *at most one entity* from this entity set in the relationship
- Line: *any number of entities* from this entity set in the relationship

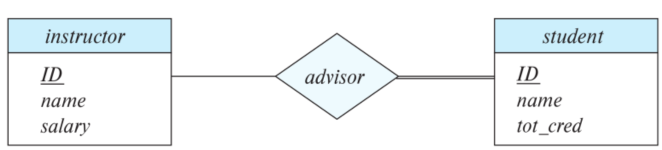


Total and Partial Participation

- *Total participation* (double line)
  - Every entity in this entity set must participate in at least one relationship
- *Partial participation* (single line)
  - Some entities may not participate in any relationship in the relationship set

Example:

- Every student must ahve an associated instructor (participation of *student* in *advisor* is total)
- Some instructors may not advise students (participation of *instructor* in *advisor* is partial)



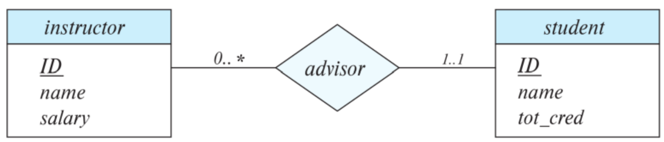
General Cardinality Constraints

We denote a general cardinality constraint as: $\ell ... h$

- This denotes the minimum ℓ and maximum h number of relationships a entity participates in
  - $\ell = 0$: partial participation
  - $\ell = 1$: total participation
  - $h = 1$: at most one relationship
  - $h = *$: not limit of relationships

Example:

- An instructor can advise zero or more students
- Each student must have exactly one advisor



Primary Keys

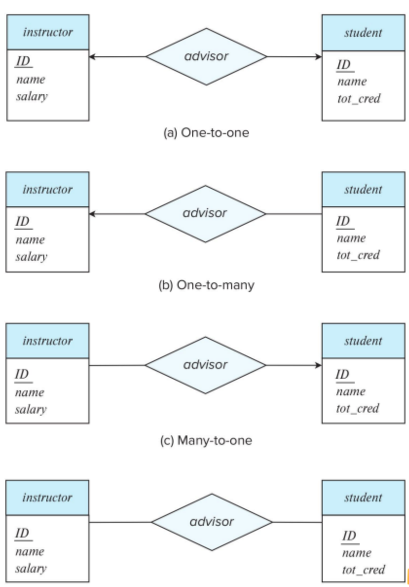
- Primary key for entity sets is a set of attributes that suffice to distinguish entities from each other
- Primary key for binary relationship depends on the mapping cardinality of relationship set
  - one-to-one: primary key of either one of the participating entity sets
  - one-to-many or many-to-one: primary key of the *many* side
  - many-to-many: union of primary keys of the participating entity sets

Example:

ER notations

(→) : one

(—): many



(a) One-to-one

An instructor may advise at most one student, and a student may have at most one advisor.

Primary key of advisor: either instructor.ID or student.ID

(b) One-to-many

An instructor may advise many students, but a student may have at most one advisor.

Primary key of advisor: student.ID

(c) Many-to-one

An instructor may advise at most one student, but a student may have many advisors.

Primary key of advisor: instructor.ID

(d) Many-to-many

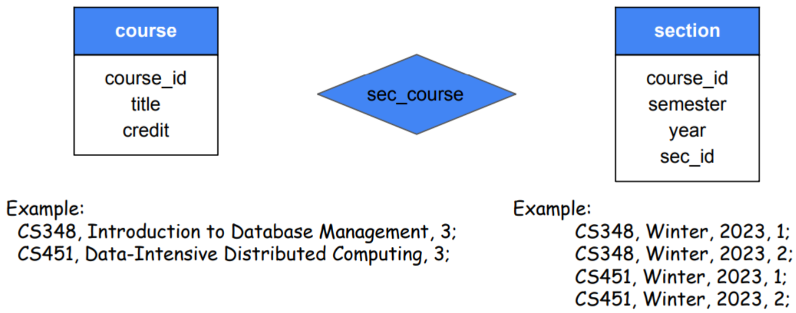
An instructor may advise many students, and a student may have many advisors.

Primary key of advisor: (instructor.ID, student.ID)

Weak Entity Sets and Identifying Relationships

Definition: a *weak entity set* is an entity set whose existence is dependent on some entity called its *identifying entity* (an entity set that is not a weak entity set is called a *strong entity set*).

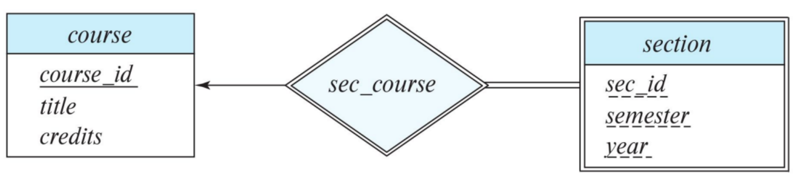
Example: *course* is the identifying entity of the *section* weak entity set:



The *identifying relationship* associates the weak entity set with its identifying entity set

- Identifying relationship is a many-to-one from the weak entity set to the identify entity set
- Participation of the weak entity set in the identifying relationship is total
- Identifying relationship set should not have any attributes
  - Instead the attributes should be associated with the weak entity set

Example: E-R diagram of a weak entity set (double outline for identifying relation and weak entity set)



The attribute `course_id` is not stored in the *section* but is provided by `sec_course`

Redundant Attributes in Entity Sets

General steps to design an E-R diagram:

1. Identify entity sets
2. Add attributes to entity sets
3. Form relationship sets between the various entity sets
4. Remove redundant attributes from entity sets

Example:

1. Identify entity sets: *instructor* and *department*
2. Added attributes:

- *instructor*: <u>*ID*</u>, *name, dept_name, salary*
- *department*: <u>*dept_name*</u>, *building, budget*

3. Formed relationship: *inst_dept* relating *instructor* and *department*
4. The attribute *dept_name* in *instructor* is redundant due to the *stud_dept* relationship so we would remove it to get the entity sets:
    - *instructor*: <u>*ID*</u>, *name, salary*
    - *department*: <u>*dept_name*</u>, *building, budget*

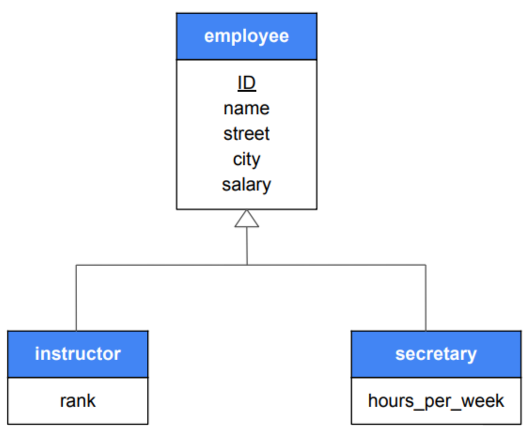
Due to the relationship this is basically the same as before.

Specialization and Generalization

- *Specialization*: general groups into more specific groups
    - Create general purpose entity sets, then provide specializations of them
- *Generalization*: specific groups into more general groups
    - Begin with specific entities, then find common attributes and generalize them

Example: a university employee could be either a instructor or a secretary

- Instructors and secretaries both have: ID, name, street, city, salary
- Instructors also have: rank (e.g. associate, full, etc)
- Secretaries also have: hours_per_week

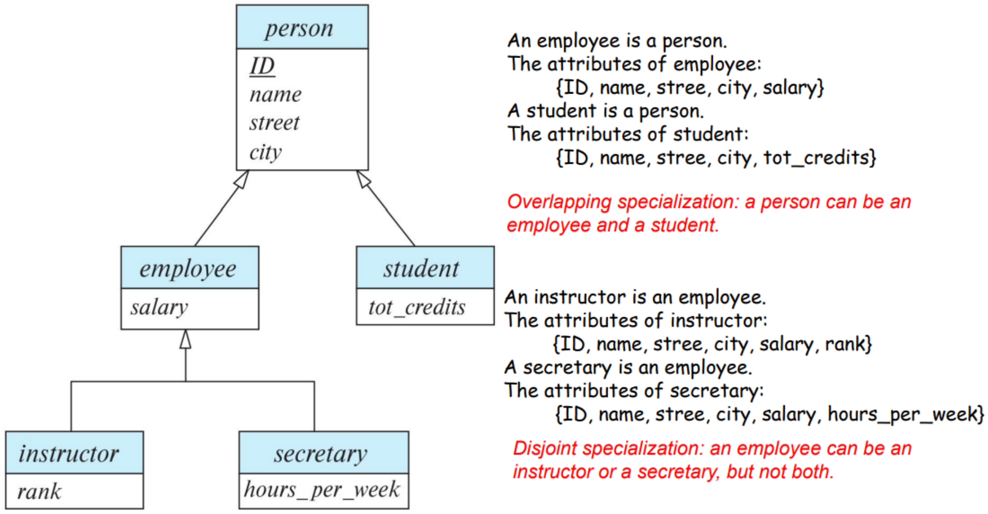


- *Superclass*: the higher-level entity sets used to represent common attributes
- *Subclass*: lower-level entity sets used to represent specialized attributes
- Superclass is connected to subclasses using a hollow-headed arrow
- Lower-level entity sets *inherit attributes and relationships* of higher-level entity sets

Specialization constraints:

- *Disjoin specialization*: entity can be member of *at most one* lower-level entity sets
- *Overlapping specialization*: entity can be member of *many* lower-level entity sets

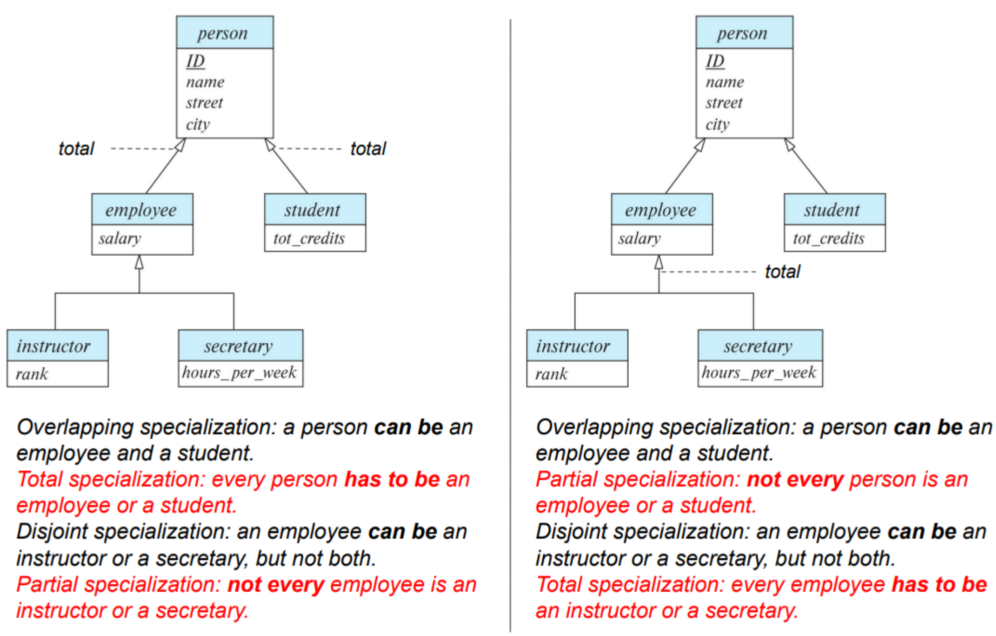
Example: if one hollow arrow has many lower-level entity sets then the specialization is disjoint



Completeness constraints:

- *Total specialization*:
  - Every higher-level entity **must** be a member of at least one lower-level entity set
  - e.g. a person must be either an employee or a student (not allowed to be neither)
- *Partial specialization*:
  - Every higher-level entity is **not required** to a member of some lower-level entity set (default)
  - e.g. not every employee is an instructor or a secretary (allowed to be neither)

Example: total specialization constraint is added by annotating specialization arrow(s)

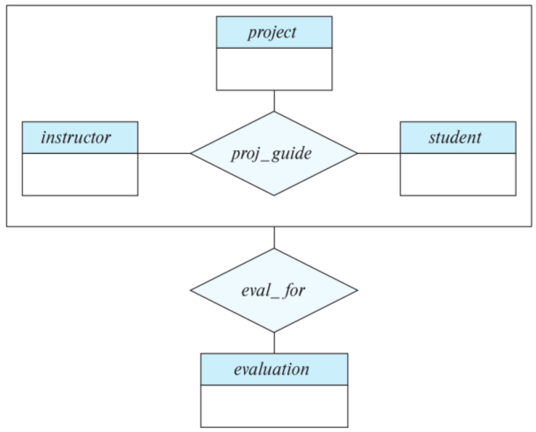


Aggregation

Relationships can be viewed as a higher-level entity set

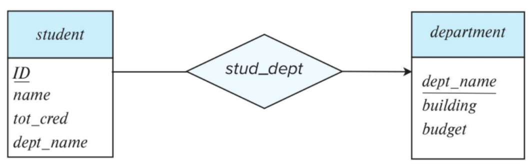
- Expressing an relationship where another relationship acts as a component entity set

Example: each instructor guiding a studnet on a project is required to fill a monthly evaluation report



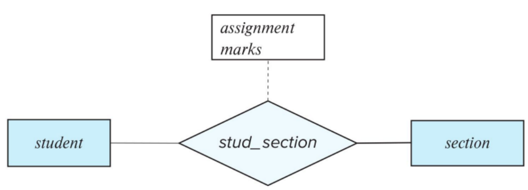
Entity-Relationship Design Issues

- Common mistake 1:

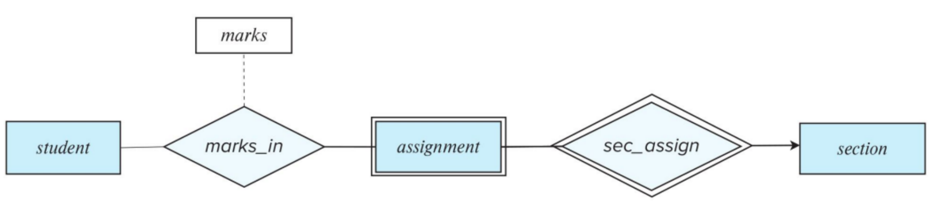


  - Using a primary key of an entity set as an attribute of another entity set, instead of using a relationship
  - attribute `dept_name` is redundant in *student*

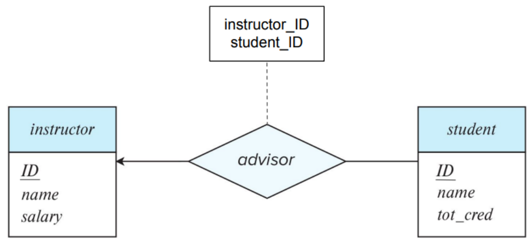
- Common mistake 2:



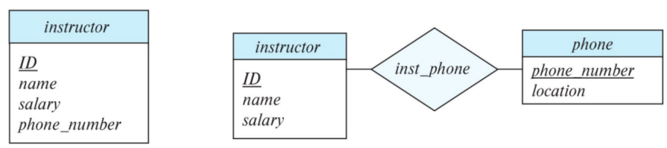
  - Using a relationship with a single-valued attribute when we require a multivalued attribute
  - In the given E-R diagram each *student-section* pair can only have one assignement
  - The corrected version allows for a section to have multiple assignments:



- Common mistake 3:



  - Using the primary-key attributes of the related entity sets as attributes of the relationship set
  - The corrected version could be either of the following two:

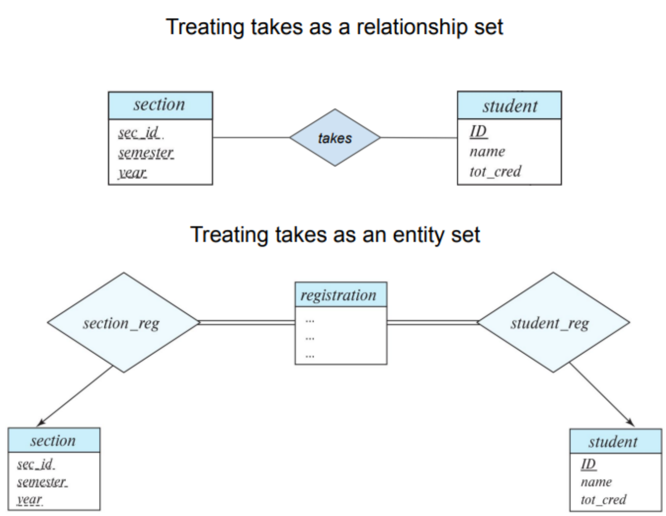


Entity Set vs Relationship Set

It is not always clear when an object is best expressed by an entity set or a relationship set.

- One guideline is thinking about whether it describes an action that occurs between two entities

Example: *takes* can be viewed as either a relationship set or entity set



E-R Diagrams Summary

To create an E-R diagram we have 6 steps

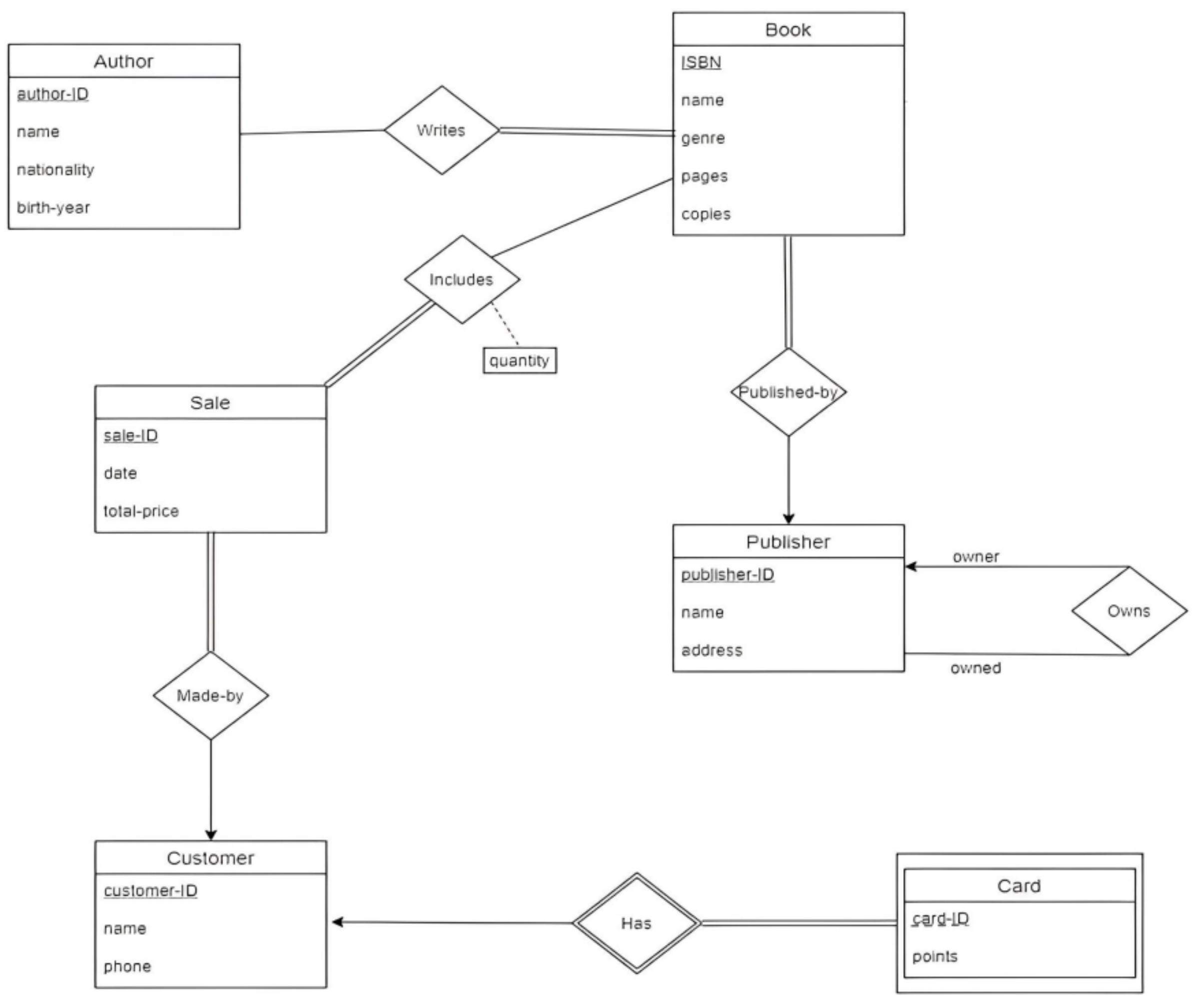
1. Recognize entity sets
2. Recognize relationship sets and participating entity sets
3. Recognize attributes of entity and relationship sets
4. Define relationship types and existence dependencies
5. Define general cardinality constraints, keys, and discriminators

6. Draw diagram

For each step, maintain a log of assumptions motivating the choices and of restrictions imposed.

Example: a bookstore's system

- Each book in this system has a unique ISBN in addition to the attributes name, genre, the number of pages, and the number of copies in the bookstore.
- Each book is written by one or more authors and published by a publisher.
- Authors have unique author ID, name, nationality, and birth year. An author can write more than one book.
- Each publisher is identified by a publisher ID, and also has name and address. Some publishers can be owned by another publisher. And, a publisher can own more than one publisher.
- Each customer has unique customer ID, name, and phone number. Customers may have customer cards, each card having a card ID and points attributes. Customer cards cannot be identified uniquely and they can exist in the system only with customers.
- Each sale in the bookstore is made by a customer, and identified by a sale ID, and also has the date and total price. Each sale is associated with one or more books, specifying the number of copies sold for each book.



E-R Diagram to Relational Tables

Intuitively, we perform the following translation:

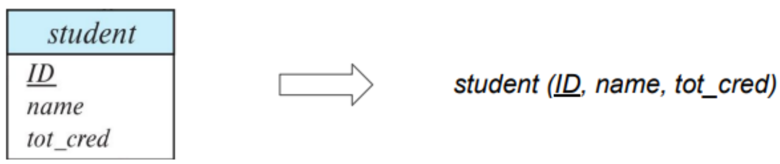
- Each entity set maps to a new table
- Each relationship set maps to a new table
- Each simple and single-valued attribute maps to a new table column

Representation of Strong Entity Sets

Method: the entity set E with attributes $a_1, \ldots, a_n$ translates to table E with attributes $a_1, \ldots, a_n$

- Entities of E correspond to rows in table E
- Primary key of an entity set is the primary key of the table

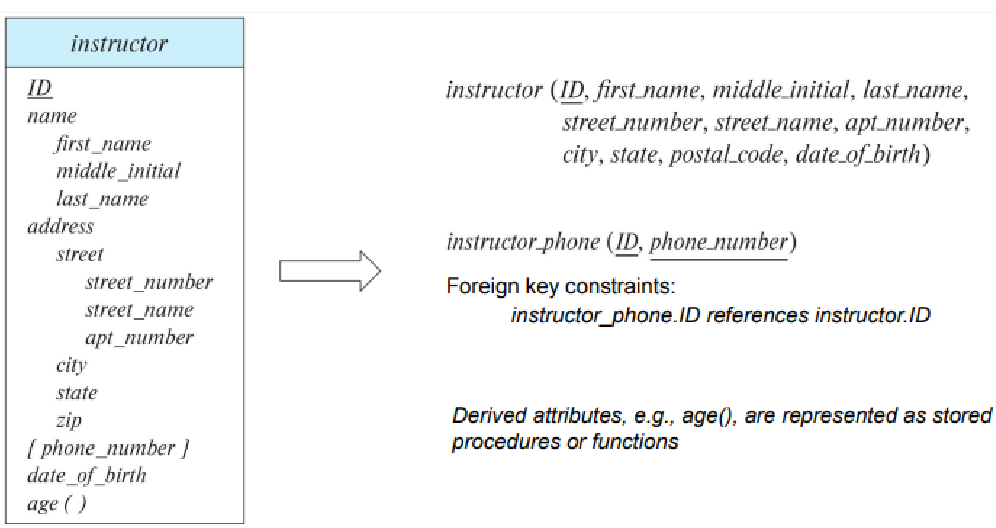
Example:



For complex attributes:

- **Composite** attributes are flattened out by creating a separate attribute for each component
- **Multivalued** attribute M of an entity E is represnted by a separate table EM consisting of the primary key A of E and M, then foreign key constraint of $EM.A$ references $E.A$ is added to EM

Example:



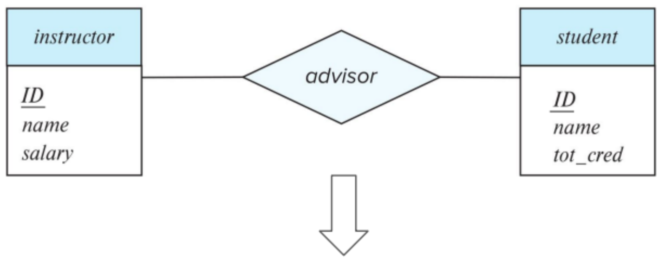
The derived attributes (e.g. `age()`) are represented as stored procedures or functions

Representation of Relationship Sets

Method: A many-to-many relationship set R translates to table R

- Columns of table R should include:
  - Primary keys of the two participating entity sets
  - Descriptive attributes of the relationship set R
- Primary key and foreign key constraints:
  - PK: primary keys of table R is the primary keys of relationship set R
    - primary keys of the two participating entity sets
  - FK: 2 foreign key constraints referencing the primary keys of the 2 participating entity sets

Example:



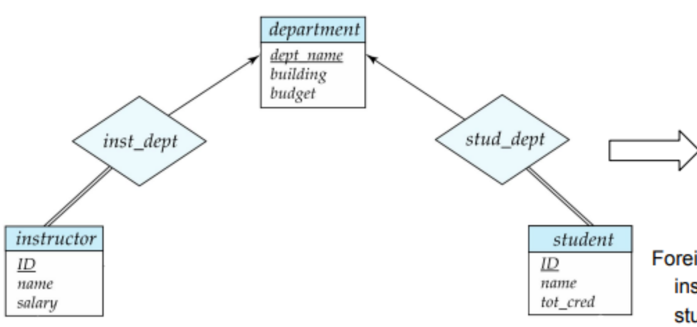
advisor = (student_ID, instructor_ID)

Foreign key constraints:
advisor.student_ID **references** student.ID
advisor.instructor_ID **references** instructor.ID

Method: many-to-one or one-to-many relationship sets are represented through adding the primary key A of the *one* side to the *many* side

- If participation is total on the *many* then the added attribute(s) A is `not null`
- Foreign key constraint: added attribute(s) A of *many* side references primary key A of *one* side

Example:



instructor(ID, name, salary, dept_name)

student(ID, name, tot cred, dept_name)

department(dept_name, building, budget)

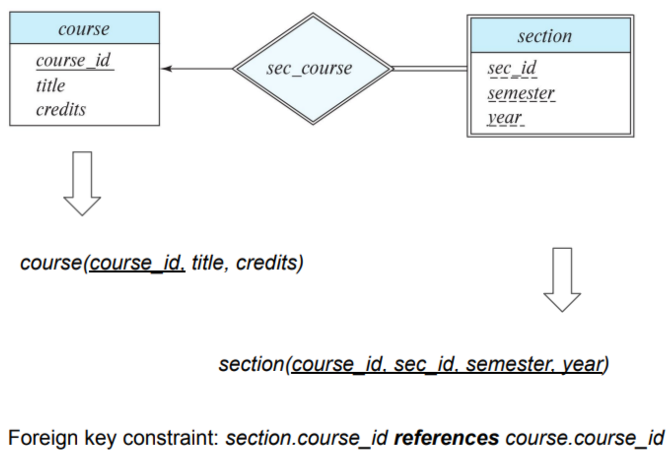
Foreign key constraints:
instructor.dept_name **references** department.dept_name
student.dept_name **references** department.dept_name

Representation of Weak Entity and Relationship Sets

Method: the weak entity set WE translates to table WE

- Columns of table WE should include:
  - All attributes of the weak entity set
  - Primary key A of the identifying (strong) entity set SE
- Primary key and foreign key constraints:
  - PK: discriminator attributes of WE plus primary key A of identifying entity set SE
  - FK: $WE.A$ references $SE.A$

Example:



Note: the identifying relationship for a weak entity does not require any translation (i.e. *sec_ course* does not need any translation)

Representation of Specialization and Generalization

Higher-level entity set:

- Person with attributes ID, name, street, and city

Lower-level entity set:

- Person can be an employee with additional attribute salary
- Person can be student with addition attribute tot_credits

Then the specialization is represented by the following tables:

- Person(<u>*ID*</u>, *name, street, city*)
- Employee(<u>*ID*</u>, *salary*)
- Student(<u>*ID*</u>, *tot_ cred*)

Key constraints:

- FK: Employee.ID references Person.ID
- FK: Student.ID references Person.ID

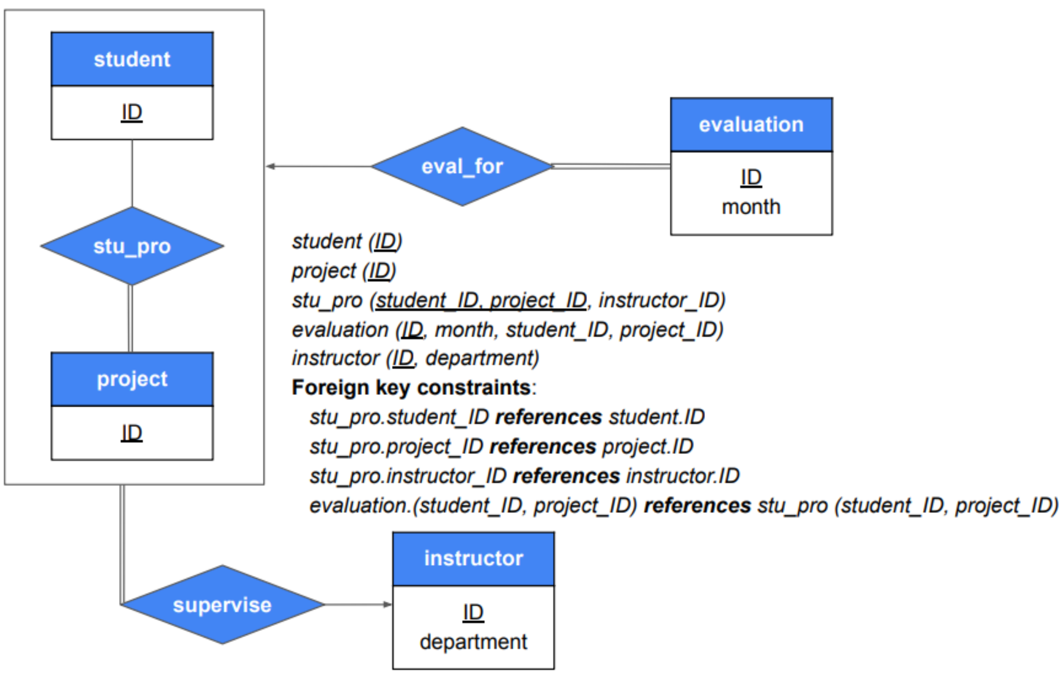
If the speicalization is *disjoint* and *total* then the tables would be

- Person(*<u>ID</u>, name, street, city, salary*)
- Student(*<u>ID</u>, name, street city, tot_ cred*)

Aggregation

Method: treat the aggregation like an entity set whose primary key is the primary key of the aggregated relationship set

Example:



Function Dependencies (FD)

Definition: let R be a relational schema and $X, Y \subseteq R$ be a set of attributes. The *functional dependency*

$$X \to Y$$

holds on R if whenever an instance of R contains two tuples t and u such that $t[X] = u[X]$ then $t[Y] = u[Y]$.

- We say that that X *functionally determines* Y in R
- Note both $t.X$ and $t[X]$ hold the same meaning

$t[A_1, \ldots, A_k]$ is a projection of r onto attributes $A_1, \ldots, A_k$ i.e. the tuple $(t.A_1, \ldots, t.A_k)$

Example: for the relational schema EmpProj(*<u>SIN</u>*, *<u>PNum</u>*, *Hours, EName, PLoc, Allowance*)

- If *SIN* determines employee name then: *SIN* $\rightarrow$ *EName*
- If project number determines project name and location then: *PNum* $\rightarrow$ *PName,PLoc*
- If allowance is the same for number of hours at the same location: *PLoc, Hours* $\rightarrow$ *Allowance*
- Trivial FD: *SIN, EName* $\rightarrow$ *EName*

Functional Dependencies and Keys

Recall the defined keys:

- *superkey*: set of attributes such that no two tuples have the same values for the attributes
- *candidate key*: minimal superkey
- *primary key*: candidate key chosen by the database designer

Functional dependencies and keys:

- If $K \subseteq R$ is a *superkey* for R then dependency $K \rightarrow R$ holds on R
- If dependency $K \rightarrow R$ holds on R then $K \subseteq R$ is a *superkey* for relation schema R
  - Needs assumption that R does not contain duplicate tuples

Functional Dependencies Implication

Armstrong's Axioms:

- *Reflexivity*:

$$X \subseteq X \quad \Longrightarrow \quad X \rightarrow Y$$

- *Augmentation*:

$$X \rightarrow Y \quad \Longrightarrow \quad XZ \rightarrow YZ$$

- *Transitivit*:

$$X \rightarrow Y,\ Y \rightarrow Z \quad \Longrightarrow \quad X \rightarrow Z$$

These axioms are:

- Sound (anything derived from F is in F^+)
- Complete (anything in F^+ can be derived from F)

Additional rules can also be derived:

- *Union*:

$$X \rightarrow Y,\ X \rightarrow Z \quad \Longrightarrow \quad X \rightarrow YZ$$

- *Decomposition*:

$$X \rightarrow YZ \quad \Longrightarrow \quad X \rightarrow Y,\ X \rightarrow Z$$

Example: prove SIN, PNum → Allowance using the following set of FDs

$$\begin{aligned} F = \{&\text{SIN, PNum} \rightarrow \text{Hours} \\ &\text{SIN} \rightarrow \text{EName} \\ &\text{PNum} \rightarrow \text{PName, PLoc} \\ &\text{PLoc, Hours} \rightarrow \text{Allowance}\} \end{aligned}$$

1. SIN, PNum → Hours ($\in F$)
2. PNum → PName, PLoc ($\in F$)
3. PLoc, Hours → Allowance ($\in F$)
4. SIN, PNum → PNum (reflexivity)
5. SIN, PNum → PName, PLoc (transitivity, 4 and 2)
6. SIN, PNum → PLoc (decomposition, 5)
7. SIN, PNum → PLoc, Hours (union, 6 and 1)
8. SIN, PNum → Allowance (transitivity, 7 and 3)

Functional Dependencies Attribute Closure

Definition: closure Z^+ of attributes Z in the relation R with respect to the set of FDs F is the set of all attributes $\{A_1, \dots, A_t\}$ functionally determined by Z (i.e. $Z \rightarrow A_1 \cdots A_t$)

Algorithm to compute the closure: Compute$Z^+(Z, F)$:

- Start by setting $Z^+ \leftarrow Z$
- If $X \in Z^+$ and $X \rightarrow Y \in F$ then update $Z^+ \leftarrow Z^+ \cup Y$
- Repeat until no new attributes can be added

Example: ComputeZ^+({PNum,Hours},F) with

$$\begin{aligned} F = \{&\text{SIN} \rightarrow \text{EName} \\ &\text{PNum} \rightarrow \text{PName, PLoc} \\ &\text{PLoc, Hours} \rightarrow \text{Allowance}\} \end{aligned}$$

- {PNum, Hours}
- {PNum, Hours, PName, PLoc} (PNum → PName, PLoc)
- {PNum, Hours, PName, PLoc, Allowance} (PLoc, Hours → Allowance)

Given a relation R and set of FDs F we Compute$X^+(X, F)$

- $Y \subseteq X^+ \iff X \rightarrow Y \in F$
- $R \subseteq X^+ \iff X$ is a superkey
  - To verify that X is a minimal superkey we need to check attribute closure of its proper subset

Schema Refinement

After designing a E-R diagram and converting that into a relational schema we need to determine that schema has any issues.

A *good* relational database schema should have independent facts in separate tables:

> "Each relation schema should consist of a *primary key*
> and *set of mutually independent attributes*"

This is achieved by transforming a schema into a *normal form.*

Lossless-Join Decomposition

Definition (Schema Decomposition): Let R be a relation schema (set of attributes). The collection $\{R_1, \ldots, R_n\}$ of relation schemas is a *decomposition* of R if

$$R = R_1 \cup \cdots \cup R_n$$

Example: consider the following decomposing of *Marks* into *SGM* and *AM*

Marks

Student	Assignment	Group	Mark
Ann	A1	G1	80
Ann	A2	G3	60
Bob	A1	G2	60

SGM

Student	Group	Mark
Ann	G1	80
Ann	G3	60
Bob	G2	60

AM

Assignment	Mark
A1	80
A2	60
A1	60

While this is a valid decomposition notice that the natural join of *SGM* and *AM* has *spurious tuples*:

Student	Assignment	Group	Mark
Ann	A1	G1	80
Ann	A2	G3	60
Ann	A1	G3	60
Bob	A2	G2	60
Bob	A1	G2	60

We are therefore losing information if we replace *Marks* with *SGM* and *AM* (*lossy decomposition*)

Definition (Lossless Decomposition): A decomposition $\{R_1, R_2\}$ of R is *lossless* if and only if the common attributes of R_1 and R_2 form a superkey for either schema

$$R_1 \cap R_2 \to R_1 \quad \text{or} \quad R_1 \cap R_2 \to R_2$$

(also called a lossless-join decomposition)

Example: we can losslessly decompose R into R_1, R_2, R_3

Supplied Items (R)

Sno	Sname	City	Pno	Pname	Price
S1	Magna	Ajax	P1	Bolt	0.50
S1	Magna	Ajax	P2	Nut	0.25
S1	Magna	Ajax	P3	Screw	0.30
S2	Budd	Hull	P3	Screw	0.40

Suppliers (R1)

Sno	Sname	City
S1	Magna	Ajax
S2	Budd	Hull

Parts (R2)

Pno	Pname
P1	Bolt
P2	Nut
P3	Screw

Supplies (R3)

Sno	Pno	Price
S1	P1	0.50
S1	P2	0.25
S1	P3	0.30
S2	P3	0.40

Dependency Preservation

Definition (Dependency-Preserving Decomposition): Given a schema R and a set of functional dependencies $\mathcal{F}$, a decomposition:

$$D = \{R_1, \ldots, R_n\}$$

of R is *dependency preserving* if there is an equivalent set of functional dependencies $\mathcal{F}'$, none of which is interrelational in D.

Example: a table for a company database could be

R

Proj	Dept	Div

FD1: Proj $\rightarrow$ Dept
FD2: Dept $\rightarrow$ Div
FD3: Proj $\rightarrow$ Div

We are given two decompositions:

- $D_1 = \{R1\{\text{Proj, Dept}\},\ R2\{\text{Dept, Div}\}\}$
- $D_2 = \{R1\{\text{Proj, Dept}\},\ R3\{\text{Proj, Div}\}\}$

Of these two decompositions they are both lossless but it is actually D_1 that is better

- D_1 lets us test FD1 on table $R1$ and FD2 on table $R2$. If both are satisfied then FD3 is satisfied
- D_2 lets us test FD1 on table $R1$ and FD3 on table $R3$. However FD2 is an *interrelational constraint* as testing it requires joining tables $R1$ and $R3$

Boyce-Codd Normal Form (BCNF)

Definition (BCNF Informal): relation schema R is in BCNF if and only if any group of attributes in R that functionally determines any other attributes in R, functionally determines all attributes in R.

Schema R is in *BCNF* (w.r.t. $\mathcal{F}$) if and only if whenever $(X \rightarrow Y) \in \mathcal{F}^+$ and $XY \subseteq R$ then either

- $(X \rightarrow Y)$ is trivial (i.e. $Y \subseteq X$), or
- X is a superkey of R

Database schema $\{R_1, \ldots, R_n\}$ is in BCNF if each relation schema R_i is in BCNF.

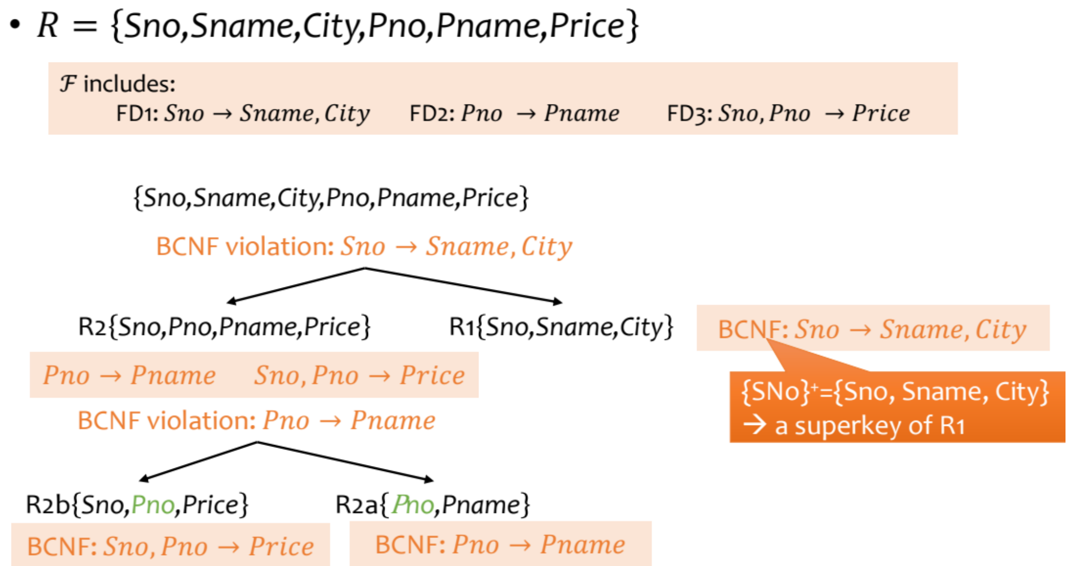
To convert to BCNF:

- Find a BCNF violation: non-trivial FD $X \rightarrow Y$ in $\mathcal{F}^*$ of R where X is *not* a super key of R
- Decompose R into R_1 and R_2 where

  $$R_1 = X \cup Y \quad \text{and} \quad R_2 = X \cup Z$$

  where Z contains all attributes of R that are neither in X nor Y
- Repeat until there are no more BCNF violations

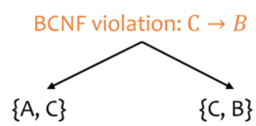
Example:



BCNF guarantees:

- Lossless join decomposition
- No redundancy

Not necessarily dependency preserving. Take $R = \{A, B, C\}$ and $\mathcal{F} = \{AB \to C,\ C \to B\}$



Notice that $AB \to C$ is interrelational and cannot be tested directly.

3NF produces lossless join decomposition and is dependency preserving, but may have redundancy.

Third Normal Form (3NF)

Definition: schema R is in *3NF* (w.r.t. $\mathcal{F}$) if and only if whenever $(X \to Y) \in \mathcal{F}^+$ and $XY \subseteq R$ then

- $(X \to Y)$ is trivial (i.e. $Y \subseteq X$), or
- X is a superkey of R, or
- each attribute in $Y - X$ is part of a candidate key of R

Database schema $\{R_1, \ldots, R_n\}$ is in 3NF if each relation schema R_i is in 3NF.

The first two conditions are the same as BCNF while the third condition is new (3NF is looser than BCNF)

To convert to 3NF:

- Initialize the decomposition with an empty set
- Find a minimal cover for F, let it be F^*
- For every $(X \to Y) \in F^*$, add relation $\{XY\}$ to the decomposition

- If no relaion contains a candidate key for R, then compute a candidate key K for R and add $\{K\}$ to the decomposition

Example:

- $R = \{$Sno, Sname, City, Pno, Pname, Price$\}$
- Functional dependencies:

$$\begin{array}{ll} \text{Sno} \rightarrow \text{Sname, City} & \text{Pno} \rightarrow \text{Pname} \\ \text{Sno, Pno} \rightarrow \text{Price} & \text{Sno, Pname} \rightarrow \text{Price} \end{array}$$

- Minimal cover:

$$\begin{array}{ll} \text{Sno} \rightarrow \text{Sname} & \text{R1} = \{\text{Sno, Sname}\} \\ \text{Sno} \rightarrow \text{City} & \text{R2} = \{\text{Sno, City}\} \\ \text{Pno} \rightarrow \text{Pname} & \text{R3} = \{\text{Pno, Pname}\} \\ \text{Sno, Pname} \rightarrow \text{Price} & \text{R4} = \{\text{Sno, Pname, Price}\} \end{array}$$

- Add relation for candidate key $R_5 = \{$Sno, Pno$\}$

Minimal Cover

Definition: a set of dependencies F is *minimal* if

- Every right-hand side of an dependency in F is a single attribute
- There does not exist $X \rightarrow A$ in F, such that the set

$$F - \{X \rightarrow A\}$$

  is equivalent to F (no redundant FD in F)
- There does not exist $X \rightarrow A$ and $Z \subset X$, such that set

$$(F - \{X \rightarrow A\}) \cup \{Z \rightarrow A\}$$

  is equivalent to F (no extra attributes on left hand side of FD in F)

To compute the minimal cover for F we have three steps (repeat *each* step until F is no longer updated)

- Replace $X \rightarrow YZ$ with $X \rightarrow Y$ and $X \rightarrow Z$
- Remove $X \rightarrow A$ from F if $A \in \text{Compute}X^+(X, F - \{X \rightarrow A\})$
- Remove A from left hand size of $X \rightarrow B$ in F if $B \in \text{Compute}X^+(X - \{A\}, F)$

Example: $R = \{\text{Sno, Sname, City, Pno, Pname, Price, Ptype}\}$ F includes

$$\text{FD1: Sno} \rightarrow \text{Sname, City}$$
$$\text{FD2: Pno} \rightarrow \text{Pname}$$
$$\text{FD3: Sno, Pno} \rightarrow \text{Price}$$
$$\text{FD4: Sno, Pname} \rightarrow \text{Price}$$
$$\text{FD5: Pno, Pname} \rightarrow \text{Pty}$$

- Fail condition 1: FD1
- Fail condition 2: FD2 and FD4 implies FD3 (remove FD3)
- Fail condition 3: FD5 can be replaced by FD5′ Pno → Ptype

Then the minimum cover is:

$$\text{Sno} \rightarrow \text{Sname}$$
$$\text{Sno} \rightarrow \text{City}$$
$$\text{Pno} \rightarrow \text{Pname}$$
$$\text{Sno, Pname} \rightarrow \text{Price}$$
$$\text{Pno} \rightarrow \text{Pty}$$

Transactions

Concurrency and Power Failure

The database is a *shared* resource that the accessed by many users and processes *concurrently*.

Due to the database being a shared resource there can be problems due to *concurrency* or *power failure*.

Problems caused by *concurrency*:

- *Inconsistent reads*: if two applications read and write concurrently then it is possible to read halfway through an update operation
- *Lost updates*: if two applications write to the time place concurrently then it is possible to *lose* one of the updates
- *Non-repeatable reads*: if two applications read and write concurrently then it is possible to read same updated values and some old values

Overall we run into concurrently problems when between two applications:

- one *reads* and another *writes* to the database
- both *write* to the database

Don't need to worry about when two applications *only* read from the database.

Problems caused by *failure*:

- If system crashes *while* processing update then only some tuples are updated, but not all
- If system crashes *after* update but before the they are made permanent (e.g. written to disk) then the changes may not survive

- If system fails between two updates, then only one may complete while the other disappears

We need to worry about *partial* results of application on the database when a crash occurs

Need to make sure that when applications are completed the changes to the database can survive crashes.

SQL Transaction

A transaction is automatically started when a user executes a SQL statement:

- Subsequent statements in the same session are executed as part of the same transaction
- Statements see changes made by earlier ones *in the same transaction*
- Statements in other concurrently running transactions do not

There are two SQL commands to terminate a transaction:

- *commit*: make its effects final and visible to subsequent transactions
- *rollback*: abort the transaction by undoing its effects

A new transaction then begins with the application's next SQL command after the commit or rollback.

The fine print:

- Performing any schema operation (e.g. create table) will commit the current transaction
  - schema is usually fixed and it is extremely difficult to undo a schema operation
- Most DBMS support an autocommit feature, which automatically commits every single statement
  - can be turned off/on through the API (but may be on by default)
- Statements can be enclosed with `begin transaction` and `commit transaction` to explicitly specify a transaction

ACID

A *transaction* is a sequence of database operations that is ACID:

- *Atomic*: operations of this transaction are exected all-or-nothing (never half done)
  - Achieved using logging (to support undo)
- *Consistency*: assume all database constraints are satisfied at start of transaction and are satisfied at the end of the transaction
  - Onus on the user to define that is consistent
  - If inconsistency does arise either abort or attempt to fix before commit
- *Isolation*: transactions must behave as if they are exected in complete isolation from each other
  - Achieved using locking, multi-version concurreny control, etc.
  - DBMS executes transaction using a *serializable schedule* for extra performance
    * Operations from different transaction can interweave and execute concurrently
    * However schedule guarantees the same effects as if transactions where executed serially

- *Durability*: if the DBMS crashes after a transaction commits, all effects of the transaction must remain in the database when the DBMS comes back up
  - Achieved using logging (to support redo)

We will not study these in detail but they will be fully covered in CS448

Constraint Conflicts in SQLite

SQLite has `on conflict` which is a non-standard clause that allows us to specify how to handle constraint conflicts:

- Can be applied to constraints: unique, non-null, check, primary key (but not foreign key)
- Options for actions to perform when constraint violation occurs: abort, fail, ignore, replace, rollback

Example: the following produces the same results

```
CREATE TABLE Products(
   ProductId INTEGER PRIMARY KEY,
   ProductName VARCHAR(15) NOT NULL ON CONFLICT IGNORE,
   Price NUMERIC(5,2)
);

INSERT INTO Products VALUES
  (1, 'Hammer', 9.99),
  (2, NULL, 1.49),
  (3, 'Saw', 11.34);
```

```
CREATE TABLE Products(
   ProductId INTEGER PRIMARY KEY,
   ProductName VARCHAR(15) NOT NULL,
   Price NUMERIC(5,2)
);

INSERT OR IGNORE INTO Products VALUES
  (1, 'Hammer', 9.99),
  (2, NULL, 1.49),
  (3, 'Saw', 11.34);
```

Examples:

- ignore: skips the row that violates the constraint, and continues processing subsequent rows

```
sqlite> CREATE TABLE Products(
    ProductId INTEGER PRIMARY KEY,
    ProductName VARCHAR(15) NOT NULL,
    Price NUMERIC(5,2)
);
sqlite> INSERT OR IGNORE INTO Products VALUES
  (1, 'Hammer', 9.99),
  (2, NULL, 1.49),
  (3, 'Saw', 11.34),
  (4, 'Wrench', 37.00),
  (5, 'Chisel', 23.00),
  (6, 'Bandage', 120.00);
sqlite> SELECT * FROM Products;
1|Hammer|9.99
3|Saw|11.34
4|Wrench|37
5|Chisel|23
6|Bandage|120
```

The row violating the constraint was skipped.

- fail: aborts the current SQL statement with an error
  - Does not undo prior change of the statement that failed
  - Does not end the transaction

```
sqlite> CREATE TABLE Products(
    ProductId INTEGER PRIMARY KEY,
    ProductName VARCHAR(15) NOT NULL,
    Price NUMERIC(5,2)
);
sqlite> INSERT OR FAIL INTO Products VALUES
  (1, 'Hammer', 9.99),
  (2, NULL, 1.49),
  (3, 'Saw', 11.34),
  (4, 'Wrench', 37.00),
  (5, 'Chisel', 23.00),
  (6, 'Bandage', 120.00);
Runtime error: NOT NULL constraint failed: Products.ProductName (19)
sqlite> SELECT * FROM Products;
1|Hammer|9.99
```

Insertion stopped here.

An error raised.

```
sqlite> DELETE FROM Products;
sqlite> BEGIN TRANSACTION;
sqlite> INSERT OR FAIL INTO Products VALUES (1, 'Hammer', 9.99);
sqlite> INSERT OR FAIL INTO Products VALUES (2, NULL, 1.49);
Runtime error: NOT NULL constraint failed: Products.ProductName (19)
sqlite> INSERT OR FAIL INTO Products VALUES (3, 'Saw', 11.34);
sqlite> INSERT OR FAIL INTO Products VALUES (4, 'Wrench', 37.00);
sqlite> INSERT OR FAIL INTO Products VALUES (5, 'Chisel', 23.00);
sqlite> INSERT OR FAIL INTO Products VALUES (6, 'Bandage', 120.00);
sqlite> COMMIT;
sqlite> SELECT * FROM Products;
1|Hammer|9.99
3|Saw|11.34
4|Wrench|37
5|Chisel|23
6|Bandage|120
```

An error raised, but the transaction was still alive

- abort: (the default option) aborts the current SQL with an error
  - Undo the statement that failed but keeps the statements before
  - Does not end the transaction

```
sqlite> DELETE FROM Products;
sqlite> INSERT OR ABORT INTO Products VALUES
  (1, 'Hammer', 9.99),
  (2, NULL, 1.49),
  (3, 'Saw', 11.34),
  (4, 'Wrench', 37.00),
  (5, 'Chisel', 23.00),
  (6, 'Bandage', 120.00);
Runtime error: NOT NULL constraint failed: Products.ProductName (19)
sqlite> SELECT * FROM Products;
sqlite>
```

An error raised, and none of the rows got inserted.

```
sqlite> DELETE FROM Products;
sqlite> BEGIN TRANSACTION;
sqlite> INSERT OR ABORT INTO Products VALUES (1, 'Hammer', 9.99);
sqlite> INSERT OR ABORT INTO Products VALUES (2, NULL, 1.49);
Runtime error: NOT NULL constraint failed: Products.ProductName (19)
sqlite> INSERT OR ABORT INTO Products VALUES (3, 'Saw', 11.34);
sqlite> INSERT OR ABORT INTO Products VALUES (4, 'Wrench', 37.00);
sqlite> INSERT OR ABORT INTO Products VALUES (5, 'Chisel', 23.00);
sqlite> INSERT OR ABORT INTO Products VALUES (6, 'Bandage', 120.00);
sqlite> COMMIT;
sqlite> SELECT * FROM Products;
1|Hammer|9.99
3|Saw|11.34
4|Wrench|37
5|Chisel|23
6|Bandage|120
```

An error raised, but the transaction was still alive

- replace:
  - unique or primary key: delete pre-existing rows causing the violation (before the current row) then continue continue normally
  - not null: replace null values with the default one (if no default value then abort)
  - check: does same as abort

```
sqlite> DELETE FROM Products;
sqlite> INSERT OR REPLACE INTO Products VALUES
  (1, 'Hammer', 9.99),
  (2, 'Nails', 1.49),
  (3, 'Saw', 11.34),
  (1, 'Wrench', 37.00),
  (5, 'Chisel', 23.00),
  (6, 'Bandage', 120.00);
sqlite> SELECT * FROM Products;
1|Wrench|37
2|Nails|1.49
3|Saw|11.34
5|Chisel|23
6|Bandage|120
```

The second replaced the first.

- rollback: aborts current SQL statement with an error then rolls back the current transaction

```
sqlite> DELETE FROM Products;
sqlite> BEGIN TRANSACTION;
sqlite> INSERT OR ROLLBACK INTO Products VALUES (1, 'Hammer', 9.99);
sqlite> INSERT OR ROLLBACK INTO Products VALUES (2, NULL, 1.49);
Runtime error: NOT NULL constraint failed: Products.ProductName (19)
sqlite> COMMIT;
Runtime error: cannot commit - no transaction is active
sqlite> SELECT * FROM Products;
sqlite>
```

An error raised, and the transaction was rolled back.

The transaction had been terminated.