DATA BASE STUFF - slide 30

Classes:

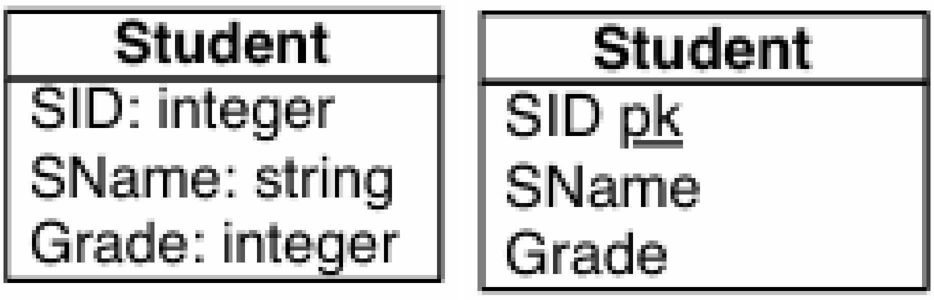
- Descriptor of a set of objects that share the same properties (semantics, attributes, and relationships)
- Characterized by name, attributes, and restraints
- Class name is usually written in the singular instead of plural and with the first letter being uppercase

Attributes:

- Defined in terms of class, while values of the attributes are defined at the instance level. For example:
    - A **student** has the attributes: **identifier**, **name** and **admission grade** (defined in terms of class)
    - **John** is a student with the **id 123**, name **John Smith** and an admission grade of **18** (each instance will/might have different values)

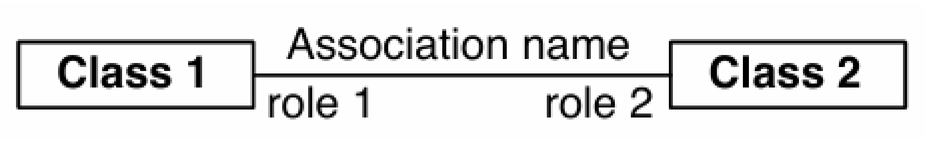
Some Attribute-related rules:

- A class **should not** have two attributes with the same name
- Attributes can be associated with types (Integer, string, …)
- For each class, we can specify primary and foreign keys



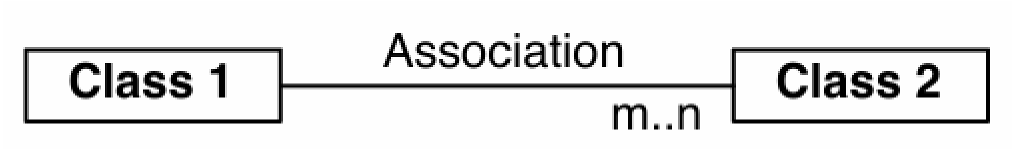
Associations:

Relationships between objects of two classes



- As an object is an instance of a class, a link is an instance of an association (The name is optional)
- There may be more than one association between the same pair of classes (Having different names)

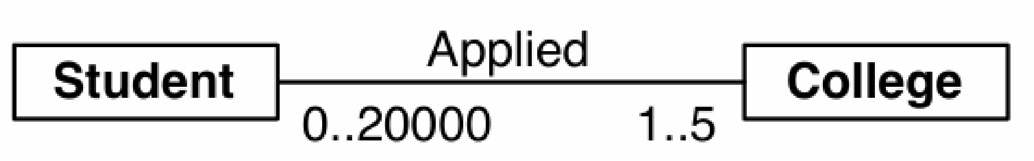
Multiplicity of Associations:



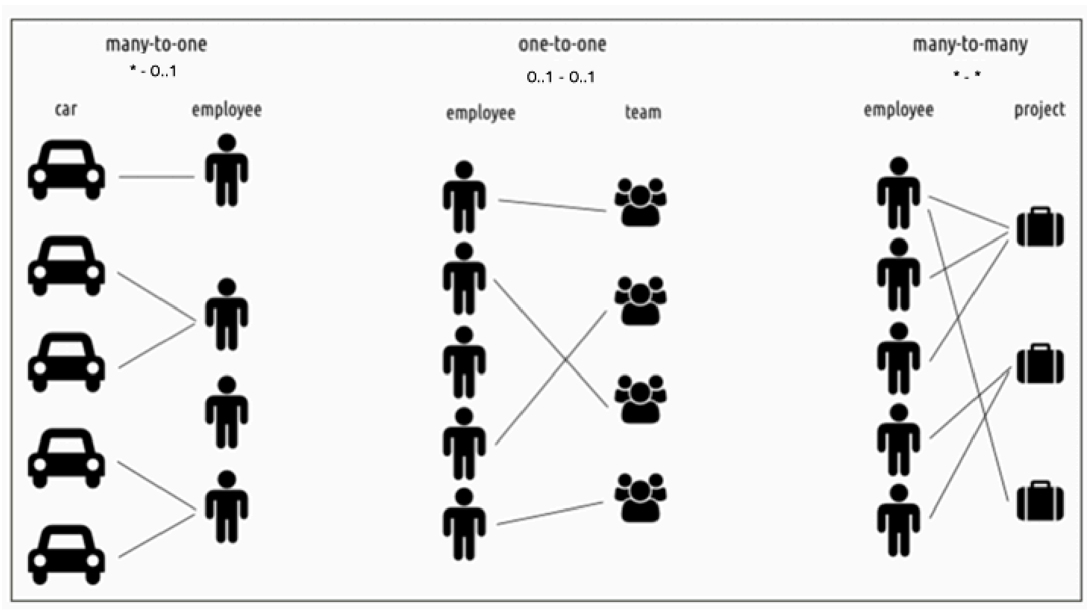
Each object of Class 1 is related to at least **m** and at most **n** objects of Class 2.

- A * in place of n stands for *no upper limit*
- * stands for 0..* which means no restrictions
- 1 stands for 1..1 (the default when nothing is shown)

Example:
Students must apply somewhere and may not apply to more than 5 colleges. No college takes more than 20,000 applications.



Common Multiplicities:



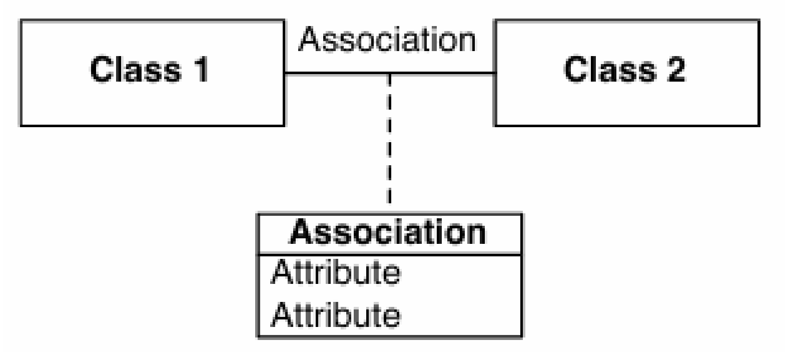
Complete Associations:

Every object participates in the association

- Complete many-to-one: 1..* - 1..1
- Complete one-to-one: 1..1 - 1..1
- Complete many-to-many: 1..* - 1..*

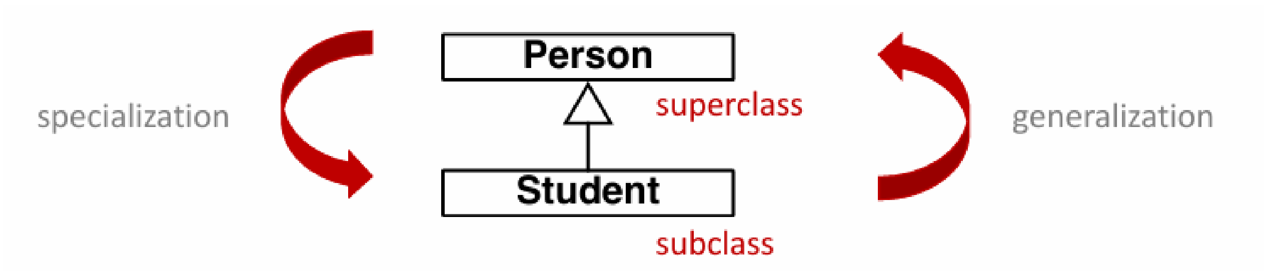
Association Classes:

Relationships between objects of two classes, with attributes *on relationships*



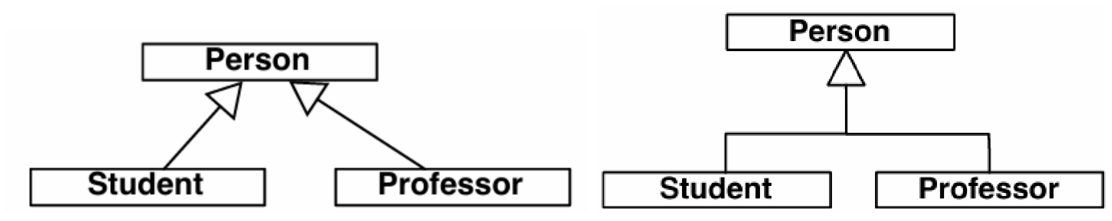
- The name may be placed in the association, in the class or in both
- It only captures one relationship between the two specific objects across the two classes

Generalizations:



- “is a “ semantic relationship (for example, A student “is a “ person)
- The subclass inherits the properties (attributes and relationships) of the superclass and may add others.

Alternative Notations:



Properties of a Generalization:

Complete (Total Participation)

- Every instance of the parent class **must** belong to **at least one** subclass.
- If every "Employee" must be either a "Manager" or a "Technician," the generalization is complete.

Incomplete (Partial Participation)

- Some instances of the parent class **may not** belong to **any** subclass.
- If some "Employees" do not fit into "Manager" or "Technician," the generalization is incomplete.

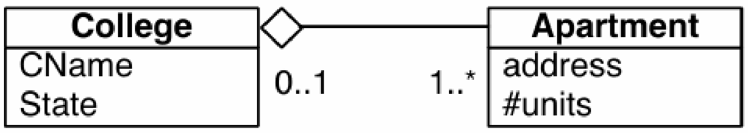
Disjoint (Mutually Exclusive Subclasses)

- Each instance belongs to **at most one** subclass—no overlap is allowed.
- An "Employee" can be either a "Manager" or a "Technician," but not both.

Overlapping (Non-Exclusive Subclasses)
- An instance can belong to **multiple** subclasses **at the same time**.
- A "Person" can be both a "Student" and a "Teacher" simultaneously.

Aggregation:

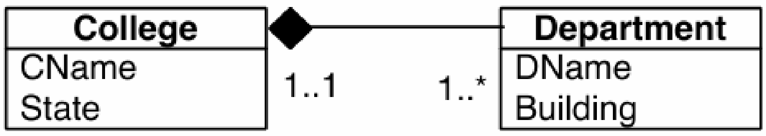
It's a special type of association. It models a "whole/part" relationship, and represents a "has-a" relationship. **0..1 is implicit**
- A "part" may be associated with more than one "whole"
    - A segment can be part of two different triangles.
- **Does not link** the lifetimes of the whole and its parts



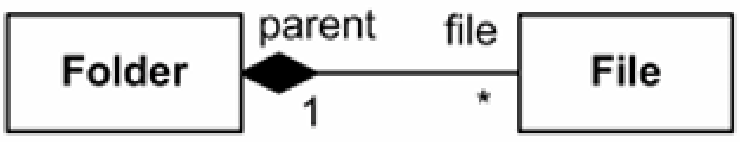
College would still exist without the Apartments

Composition:

Strongest form of aggregation. The whole is responsible for managing the creation and destruction of its parts. **1..1 is implicit**
- Strong ownership and coincident lifetime as part of the whole
- An object may be part of only one composite at a time



College has 1 or more Departments and each Department belongs to exactly one College. If College is closed, so are all of its Departments.



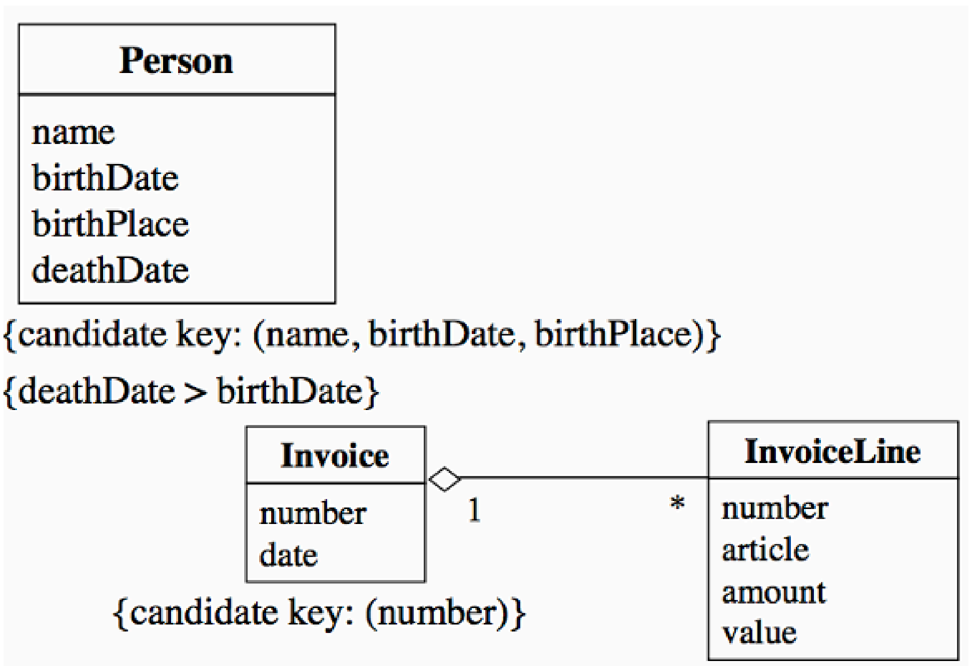
Folder can contain many files, while each File has exactly one Folder parent. If Folder is deleted, all contained Files are deleted as well.

Constraints:

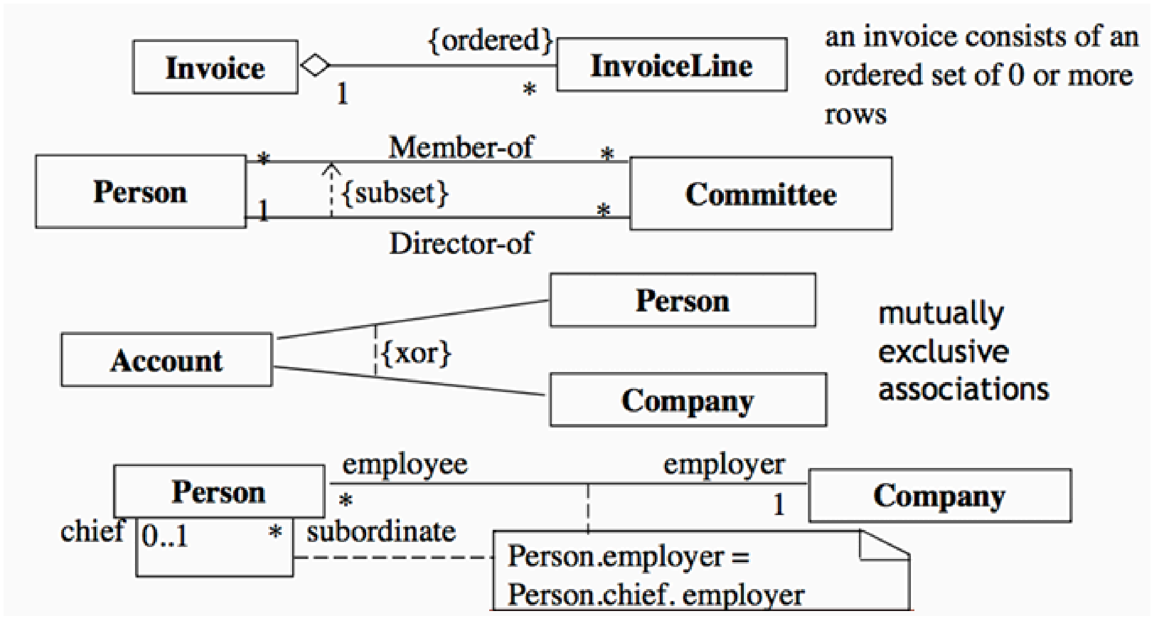
Specifies a condition that has to be present in the system.

- It is indicated by an expression or text between brackets
- Or by a note placed near (or connected by dotted lines to) the elements to which it relates

Constraints in classes:



Constraints in associations:

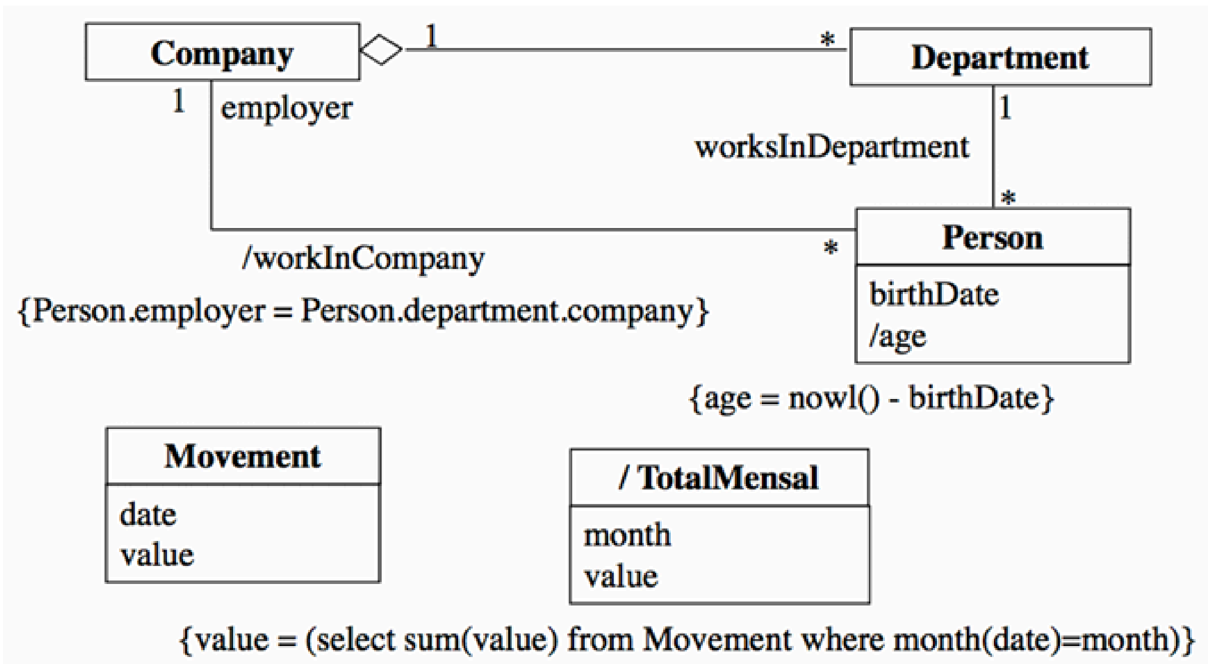


Derived Elements:

A derived elements is an element (class, attribute or association) computed using other elements in the model

- Notation: '/' before the name of the derived element

Usually have an associated constraint that relates them with other elements.

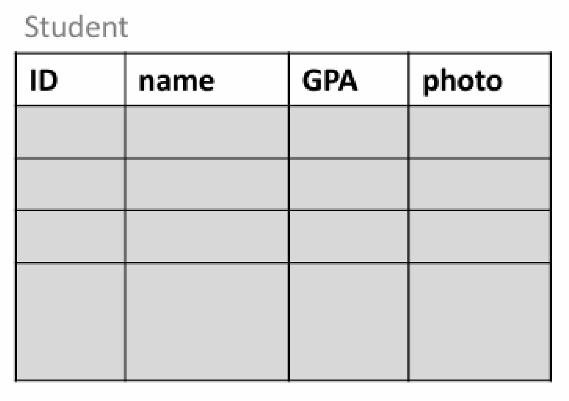

Company
1
*
Department
1
employer
1
worksInDepartment
*
Person
birthDate
/age
/workInCompany
*
{Person.employer = Person.department.company}
{age = nowl() - birthDate}
Movement
date
value
/ TotalMensal
month
value
{value = (select sum(value) from Movement where month(date)=month)}

For example:

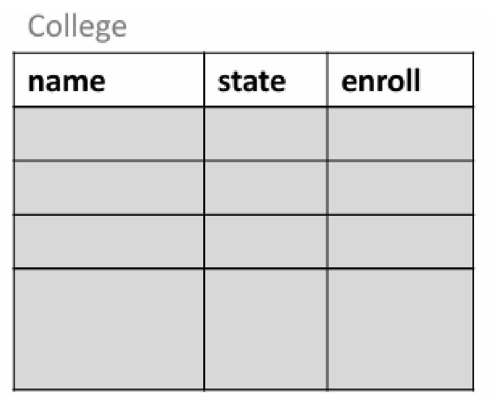
- Notice how age can be calculated with the current date minus the Person's birth date.

Relations - slide 77

Database = set of named **relations** (or **tables**)
Each relation has a set of named **attributes** (or **columns**)
The number of attributes is the arity of the relation


Student

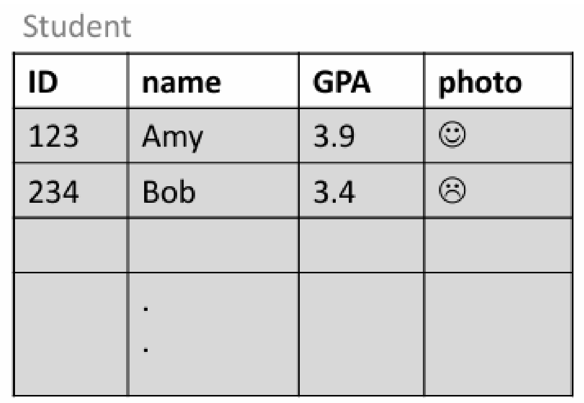
ID	name	GPA	photo


College

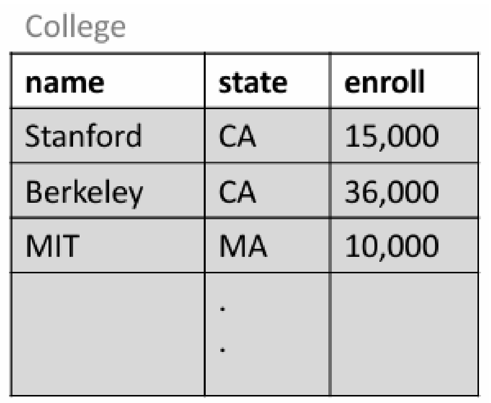
name	state	enroll

Tuples:
Each **tuple** (or **row**) has a value for each attribute.
The number of tuples is the cardinality of the relation.
Each relation/table has a **type** (or **domain**).

- Set of possible values. Examples: integer, string, …


Student

ID	name	GPA	photo
123	Amy	3.9	☺
234	Bob	3.4	☹
	. .		


College

name	state	enroll
Stanford	CA	15,000
Berkeley	CA	36,000
MIT	MA	10,000
	. .	

Schema:

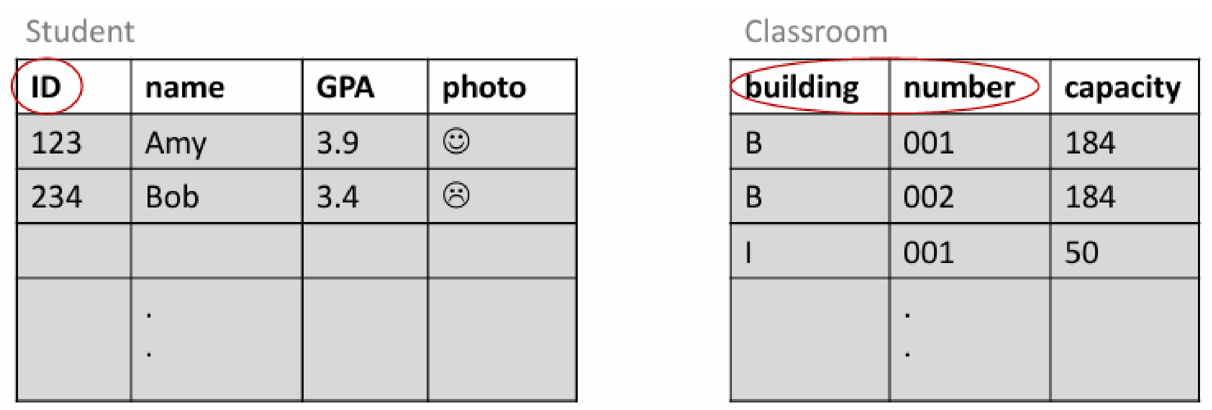
- structural description of relations in a database
- name, attributes, and types of those attributes
- typically set up in advance

Instance:

- actual contents at a given point in time
- they may change over time

NULL Value:
Special value for “unknown” or “undefined”

Key:
Set of one or more attributes whose combined values are unique within a relation.
Often denoted by underlining the set of key attributes.



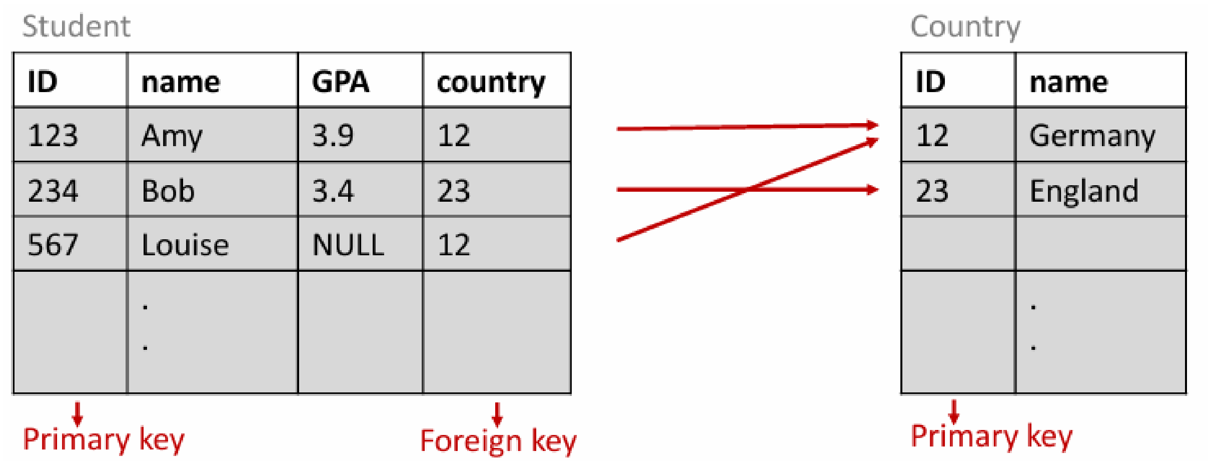
Student

ID	name	GPA	photo
123	Amy	3.9	☺
234	Bob	3.4	☹
	:		

Classroom

building	number	capacity
B	001	184
B	002	184
I	001	50
	:	

Foreign Key:
An attribute (or set of attributes) that always matches a key attribute in another relation.
Often denoted by an arrow pointing to the name of the relation being referenced.



Student

ID	name	GPA	country
123	Amy	3.9	12
234	Bob	3.4	23
567	Louise	NULL	12
	:		

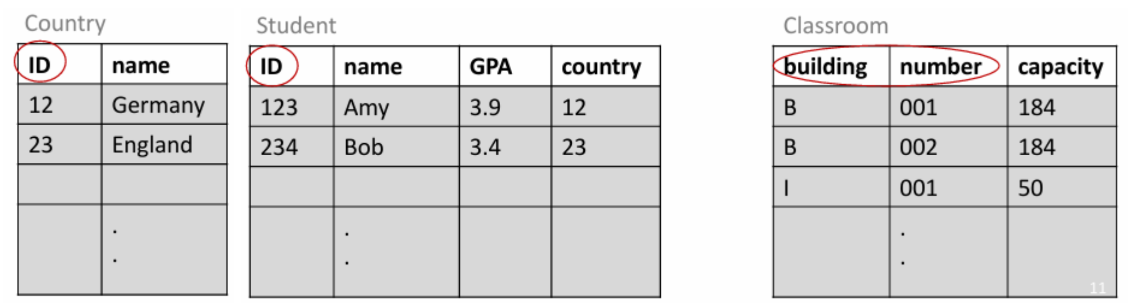
Country

ID	name
12	Germany
23	England
	:

Relational Notation:

- Student (ID, name, GPA, country → Country)
- Classroom (building, number, capacity)
- Country (ID, name)

In foreign keys, the name of the attribute can be different from the referenced attribute/relation (country doesn't have to match Country, could be any other name).



Country

ID	name
12	Germany
23	England
	. .

Student

ID	name	GPA	country
123	Amy	3.9	12
234	Bob	3.4	23
	. .		

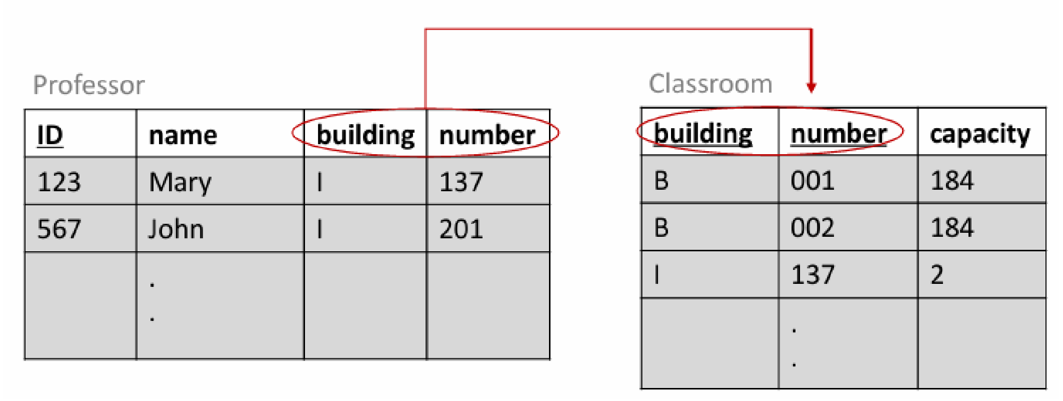
Classroom

building	number	capacity
B	001	184
B	002	184
I	001	50
	. .	

Composite Key:

A multi-column primary key or foreign key.

- Classroom (building, number, capacity)
- Professor (ID, name, building → Classroom.building, number → Classroom.number)



Professor

ID	name	building	number
123	Mary	I	137
567	John	I	201
	. .		

Classroom

building	number	capacity
B	001	184
B	002	184
I	137	2
	. .	

UML to Relations - slide 86

Classes:

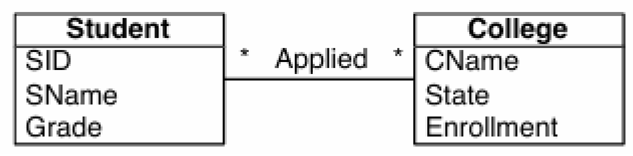
Every class becomes a relation.

Student
SID
SName
Grade
College
CName
State
Enrollment

Student (<u>SID</u> , SName, Grade)
College (<u>CName</u>, State, Enrollment)

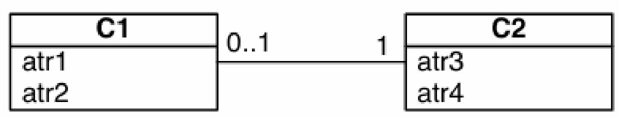
Associations:

Many-to-many associations:
Add a new relation with a key from each side



- Student (<u>SID</u>, SName, Grade)
- College (<u>CName</u>, State, Enrollment)
- Applied (<u>SID</u>->Student, <u>Cname</u>->College)

One-to-one associations:
Add a foreign key from one of the relations to the other



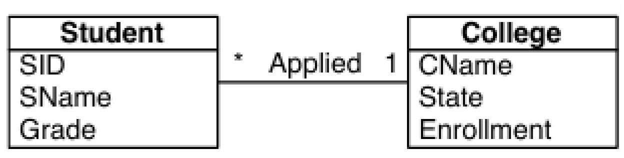
- C1 (atr1 , atr2, c2_id->C2)
- C2 (atr3 , atr4)

Add the foreign key with the unique key constraint to the relation that is expected to have less tuples.

Many-to-one associations:

1.
Add a foreign key to the **many** side of the relationship directing to the relation in the **one** side.

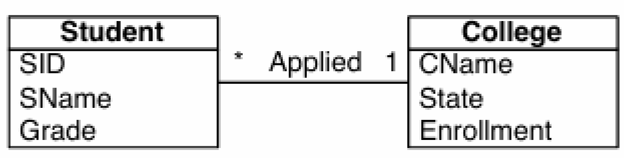
- Most common
- Less Relations in Schema
- Increased performance due to smaller number of relations



- Student (<u>SID</u>, SName, Grade, College->College)
- College (<u>CName</u>, State, Enrollment)

2.
Add a relation with key from the many side

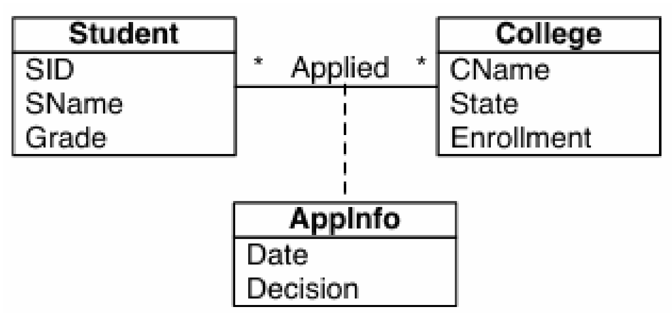
- Increased rigour of the schema
- Increased extensibility



- Student (<u>SID</u>, SName, Grade)
- College (<u>CName</u>, State, Enrollment)
- Applied (<u>SID</u>->Student, Cname->College)

Association Classes:

Add attributes to a relation for association

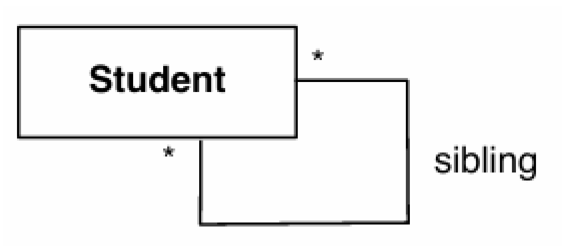


- Student (<u>SID</u> , SName, Grade)
- College (<u>CName</u> , State, Enrollment)
- Applied (<u>SID</u>->Student, <u>Cname</u>->College, Date, Decision)

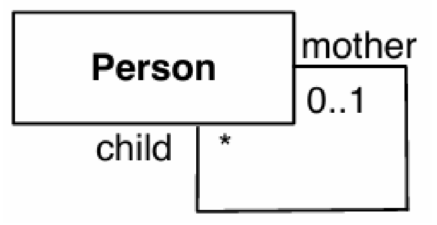
OR

- Student (<u>SID</u> , SName, Grade)
- College (<u>CName</u> , State, Enrollment)
- Applied (<u>SID</u>->Student, Cname->College, Date, Decision)

Self-associations:



- Student (id , …)
- Sibling (sid1->Student, sid2->Student)

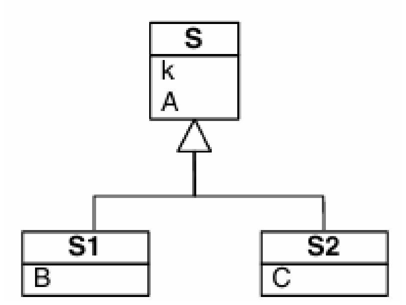


- Person (id , …)
- Relationship (mother->Person, child->Person)

Generalizations:

There are 3 conversion strategies:

- E/R style
- Object-oriented
- Use nulls



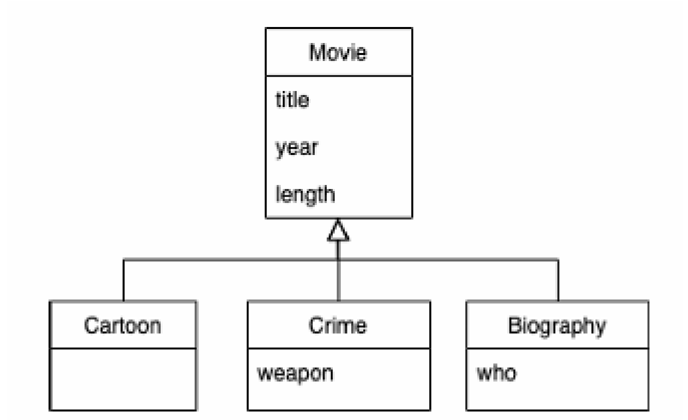
! Best translation may depend on the properties of the generalization !

1. E/R Style:
A relation per class. Subclass relations contain superclass key (in this case k).

- S(<u>k</u>, A)
- S1(<u>k</u> → S, B)
- S2(<u>k</u> → S, C)

Subclasses S1 and S2 reference the superclass and each one may have its own specialized attributes (like B and C).

Example:

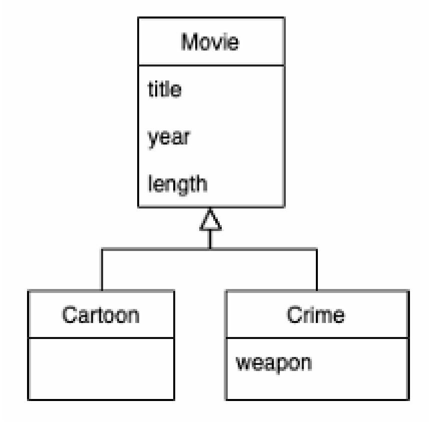


- Movie (<u>title</u>, <u>year</u>, length)
- Cartoon (<u>title</u> → Movie.title, <u>year</u> → Movie.year)
- Crime (<u>title</u> → Movie.title, <u>year</u> → Movie.year, weapon)
- Biography (<u>title</u> → Movie.title, <u>year</u> → Movie.year, who)

2. Object-oriented:

- Create a relation for all possible subtrees of the hierarchy
    - Schema has all the possible attributes in that subtree
- In complete generalizations, the relation for the subtree with only the superclass may be eliminated
- Object-oriented because each object belongs to one and only one subtree

In overlapping generalizations:

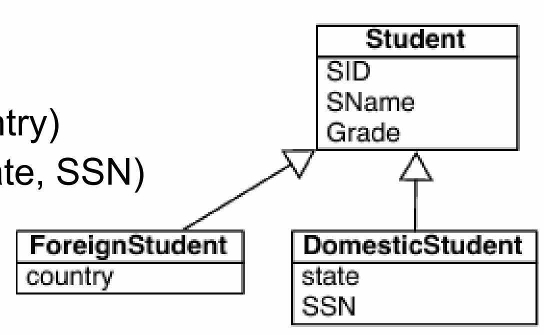


- Movie (<u>title</u>, <u>year</u>, length)
- MovieCartoon (<u>title</u>, <u>year</u>, length)
- MovieCrime (<u>title</u>, <u>year</u>, length, weapon)
- MovieCartoonCrime (<u>title</u>, <u>year</u>, length, weapon)

If the generalization is complete, then < Movie (<u>title</u>, <u>year</u>, length) > is not required.

In disjoint generalizations:

- Student (<u>SID</u>, SName, Grade)
- ForeignStudent (<u>SID</u>, SName, Grade, country)
- DomesticStudent (<u>SID</u>, SName, Grade, state, SSN)



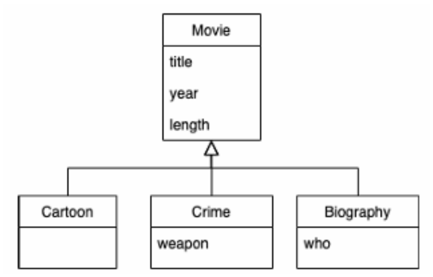
If this generalization is complete, then:

- ForeignStudent (<u>SID</u>, SName, Grade, country)
- DomesticStudent (<u>SID</u>, SName, Grade, state, SSN)

3. Use Nulls:
One relation with all the attributes of all the classes.
Use NULL values on non-existing attributes for a specific object.

- S(<u>k</u>, A, B, C)

Example:



- Movie (<u>title</u>, <u>year</u>, length, weapon, who)

Comparison of approaches in space:
Object-oriented

- Only one tuple per object with components for only the attributes that makes sense
- Minimum possible space usage
- Good for:
    - overlapping generalizations with a large number of subclasses

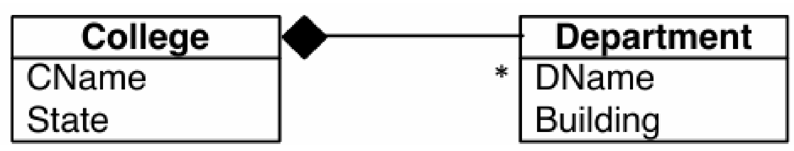
Use nulls

- Only one tuple per object but these tuples are “long”, they have components for **all** attributes
- Used space depends on the attributes not being used
- Good for:
    - disjoint generalizations. Superclass has few attributes and subclasses many attributes

E/R approach

- Several tuples for each object but only the key attributes are repeated
- Can use more or less space than the nulls method
- Good for:
    - heavily overlapping generalizations with a small number of subclasses

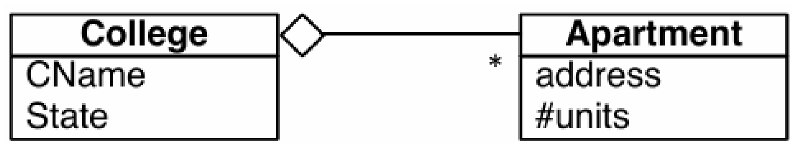
Composition:



Treat it as a regular association

- College (<u>CName</u>, State)
- Department (<u>DName</u>, Building, CName->College)

Aggregation:



Treat it as a regular association

- College (<u>CName</u>, State)
- Department (<u>DName</u>, Building, CName->College)

CName can be NULL

Constraints:

NOT NULL
UNIQUE
PRIMARY KEY
FOREIGN KEY
CHECK

- Ensures that the value in a column meets a specific condition

DEFAULT

- Specifies a default value for a column

Derived Elements:

Treat them as regular elements

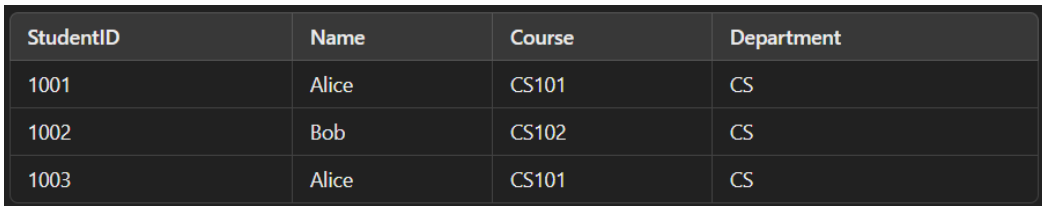
Relational Design - slide 123

The following parts are only relevant to learn Normal Forms like BCNF.

Design by decomposition:
Start with “mega” relations containing everything.
Decompose into smaller, better relations with the same information.

Functional Dependencies: (slide 136)

Imagine you're managing a student database at a university.



StudentID	Name	Course	Department
1001	Alice	CS101	CS
1002	Bob	CS102	CS
1003	Alice	CS101	CS

- "If I know a **StudentID**, I always get the same **Name**."
- "If I know a **Course**, I always get the same **Department**."

This naturally leads to the idea of **functional dependencies**—certain attributes in a table determine others.

A Functional Dependency (FD) is a relationship between attributes in a table, where one attribute (or set of attributes) uniquely determines another.

Using the example:

- **StudentID → Name** (If we know the StudentID, we can determine the Name.)
- **Course → Department** (If we know the Course, we can determine the Department.)

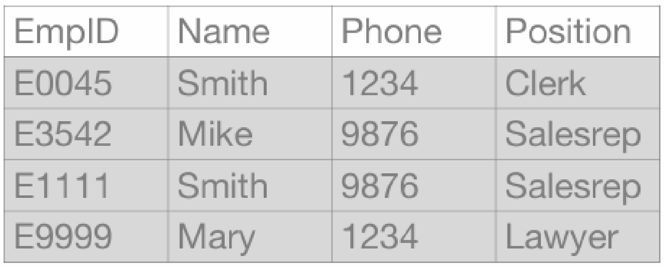
General Notation:
If **X** → **Y**, then, knowing **X** we know **Y**: <u>**X** functionally determines **Y**</u>

FDs are **crucial** for:

- **Avoiding redundancy** → Ensuring data isn't unnecessarily repeated.
- **Ensuring data consistency** → Preventing conflicting information.
- **Database normalization** → Designing better-structured databases.

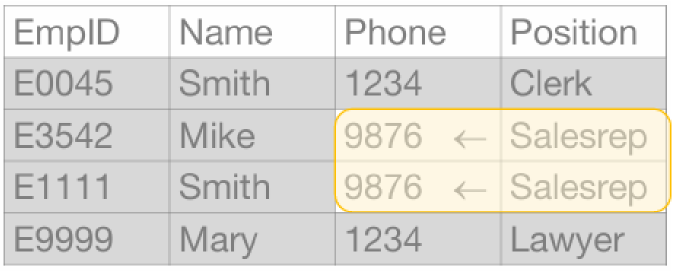
For example, if "CS101" were mistakenly linked to two different departments, we'd have data inconsistency.

Example 1:



EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

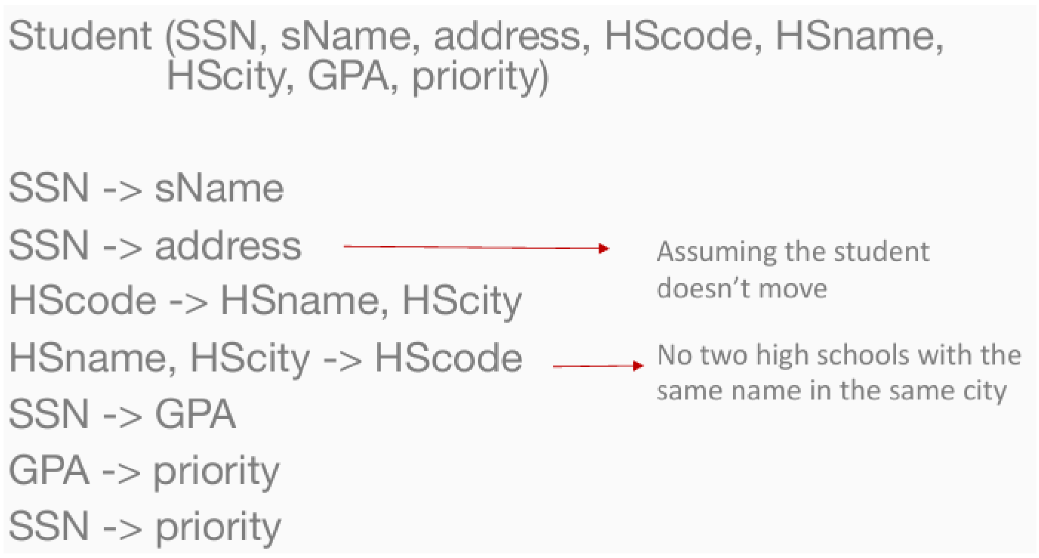
In this table we can find a FD:



EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876 ←	Salesrep
E1111	Smith	9876 ←	Salesrep
E9999	Mary	1234	Lawyer

{Position} → {Phone}

Example 2:



Student (SSN, sName, address, HScode, HSname, HScity, GPA, priority)

SSN -> sName

SSN -> address → Assuming the student doesn't move

HScode -> HSname, HScity

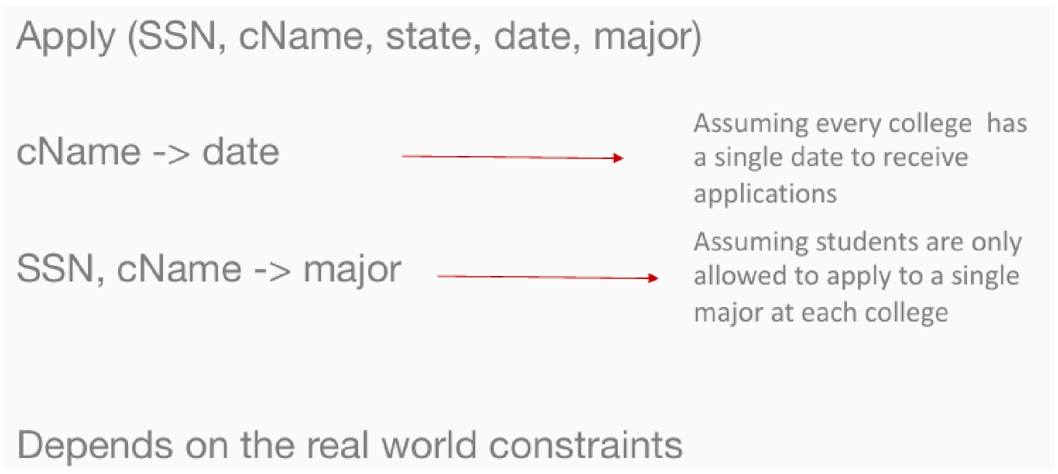
HSname, HScity -> HScode → No two high schools with the same name in the same city

SSN -> GPA

GPA -> priority

SSN -> priority

Example 3:



Apply (SSN, cName, state, date, major)

cName -> date → Assuming every college has a single date to receive applications

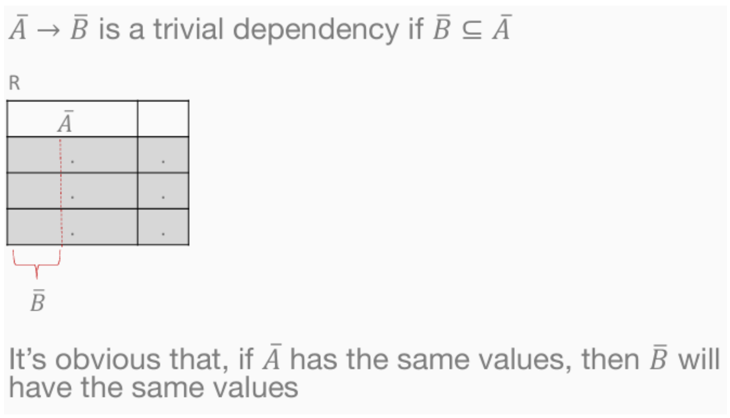
SSN, cName -> major → Assuming students are only allowed to apply to a single major at each college

Depends on the real world constraints

Keys:
Relation with no duplicates: R(A,B)
If A can determine all attributes, then A is a key to this relation.
This means two separate tuples won't have the same A values, since A is a key.

Types of Functional Dependencies:

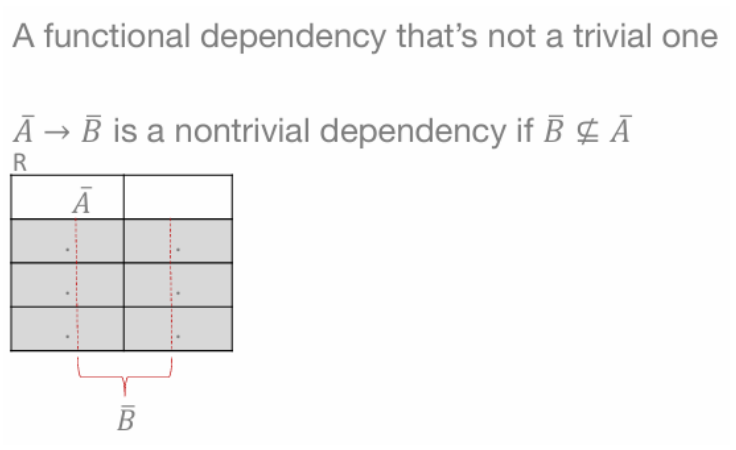
- **Trivial FD**: If Y is a subset of X, then X → Y is always true.
    - Example: {StudentID, Name} → Name (Name is already in the left side.)



$\bar{A} \rightarrow \bar{B}$ is a trivial dependency if $\bar{B} \subseteq \bar{A}$

It's obvious that, if $\bar{A}$ has the same values, then $\bar{B}$ will have the same values

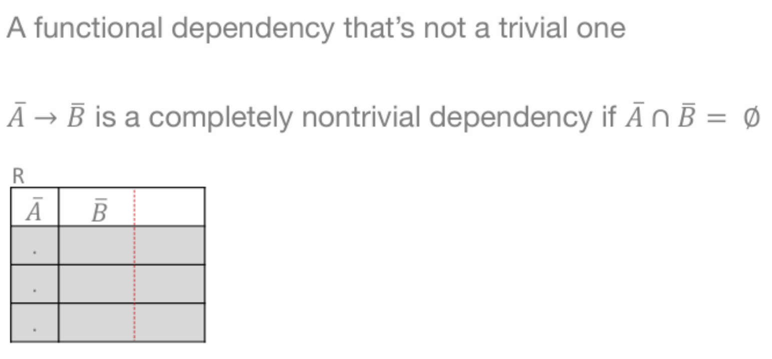
- **Non-Trivial FD**: When Y is not part of X.
    - Example: StudentID → Name (We use StudentID to determine Name.)



A functional dependency that's not a trivial one

$\bar{A} \rightarrow \bar{B}$ is a nontrivial dependency if $\bar{B} \not\subseteq \bar{A}$

    - Completely non-trivial dependency:



A functional dependency that's not a trivial one

$\bar{A} \rightarrow \bar{B}$ is a completely nontrivial dependency if $\bar{A} \cap \bar{B} = \emptyset$

How to find FDs?
Using rules, such as splitting/combining, trivial, and transitivity.

Splitting:

It refers to the ability to break down a functional dependency into **multiple smaller dependencies** or, conversely, to **combine (combining)** multiple dependencies into a single one—without changing the meaning of the dependencies.For example, suppose we have a functional dependency:

- X → Y1, Y2

Then, we can split it into two separate dependencies:

- X → Y1
- X → Y2

This means that if knowing **X** determines both **Y1** and **Y2**, then it also independently determines each one.

Combining:

Essentially, this is the inverse of splitting.
Suppose we have:

- X → Y1
- X → Y2

We can combine them into a single functional dependency:

- X → Y1, Y2

This means that if X determines Y1 and X determines Y2, we can express it as X determining both together.

Why is Splitting/Combining important?

- Normalization: Helps in breaking FDs into simpler ones, which aids in database design.
- Checking for Redundancies: Ensures dependencies are well-defined.
- Minimal Cover: Helps find the smallest and most efficient set of functional dependencies.

Trivial Dependency Rules:

A **trivial functional dependency** is a dependency where the right-hand side (RHS) is a **subset** of the left-hand side (LHS). In other words,
A → B is a trivial dependency if B ⊆ A

General Rule:
A functional dependency X → Y is trivial if:

- Y ⊆ X

In simple terms:

- If the right side (Y) is already part of the left side (X), then it's **trivial**.
- Trivial dependencies **always hold** in any relation.

Example 1: Trivial FD

- {StudentID, Name} → Name
  - Here, Name is already part of the left-hand side.
  - This is trivial because knowing {StudentID, Name} obviously determines Name.

Example 2: Trivial FD

- {A,B,C} → B
  - Since B is already in {A, B, C}, this is trivial.

Example 3: Non-trivial FD

- {StudentID} → Name
  - This is non-trivial because Name is not already part of the left-hand side.
  - This must be logically derived from the data.

Why are Trivial Dependency Rules important?

- They always hold true: No need to check them in normalization.
- Used in decomposition: Helps in breaking large FDs into smaller ones.
- Help define minimal cover: When reducing a set of FDs, we ignore trivial ones.

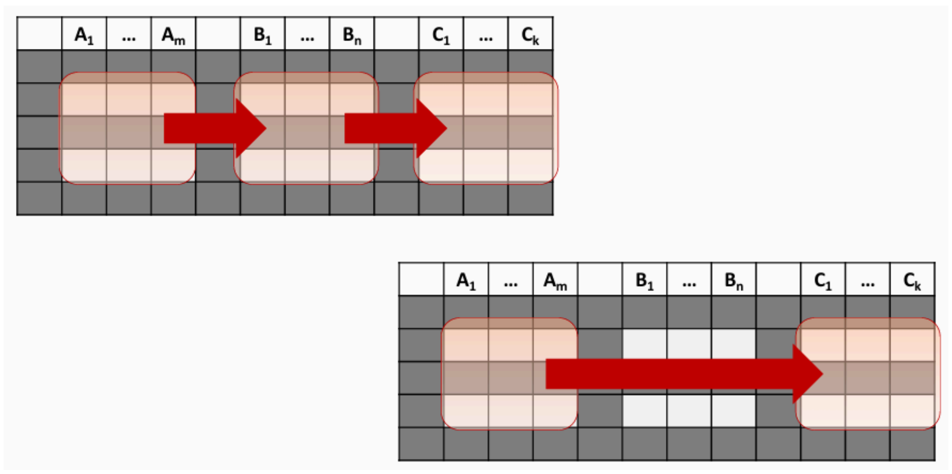
Transitive Rule:

It states that if an attribute is functionally dependent on another attribute **through an intermediate attribute**, then a direct dependency can be inferred.
If we have the following two functional dependencies:

- X → Y
- Y → Z

Then, we can conclude:

- X → Z



Example:

- StudentID → Email (A student's ID determines their email.)
- Email → UniversityDomain (An email determines the domain.)

By the transitive rule, we can conclude:

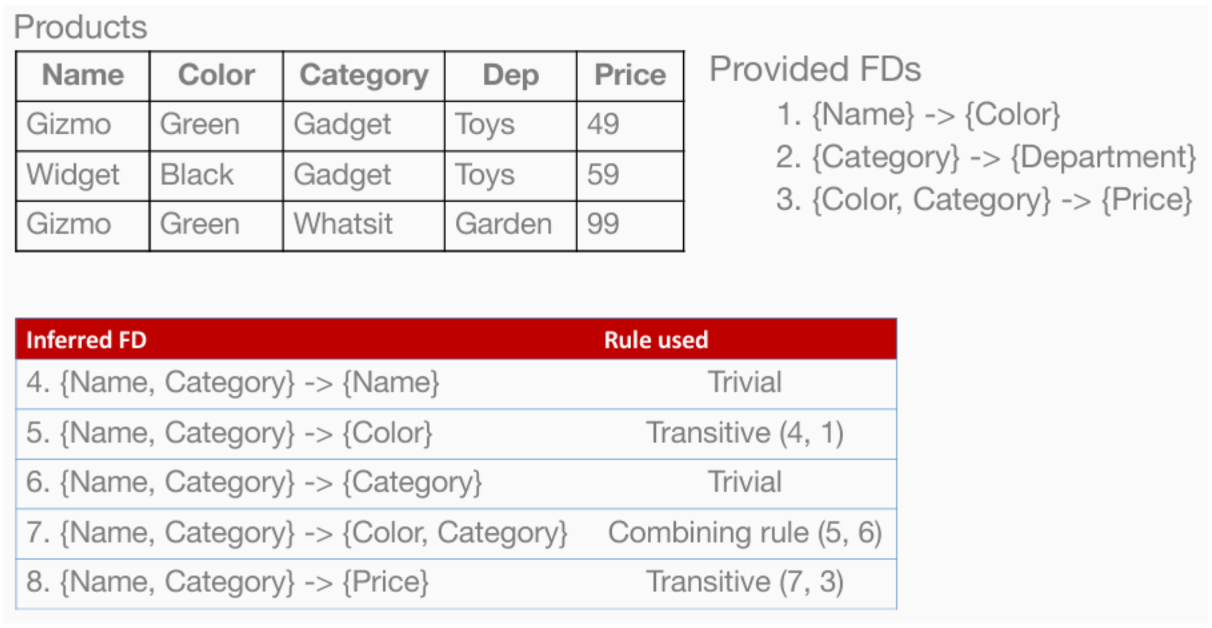
- StudentID → UniversityDomain

This means knowing StudentID alone is enough to determine the UniversityDomain, even though there was an intermediate step.

Why is the Transitive Rule important?

- Normalization: Helps identify transitive dependencies, which should be eliminated in 3rd Normal Form (3NF).
- Redundancy Reduction: Prevents unnecessary storage of derived attributes.
- Efficient Querying: Helps simplify database relationships.

Example on Finding FDs:



Products

Name	Color	Category	Dep	Price
Gizmo	Green	Gadget	Toys	49
Widget	Black	Gadget	Toys	59
Gizmo	Green	Whatsit	Garden	99

Provided FDs

1. {Name} -> {Color}
2. {Category} -> {Department}
3. {Color, Category} -> {Price}

Inferred FD	Rule used
4. {Name, Category} -> {Name}	Trivial
5. {Name, Category} -> {Color}	Transitive (4, 1)
6. {Name, Category} -> {Category}	Trivial
7. {Name, Category} -> {Color, Category}	Combining rule (5, 6)
8. {Name, Category} -> {Price}	Transitive (7, 3)

Attribute Closure (X^+):

The **closure** of an attribute set **X^+** (denoted as "X plus") is the set of all attributes that can be **functionally determined** by X using the given functional dependencies.

Why Is a Closure Important?

- Used to find candidate keys (minimal unique identifiers for a table).
    - A candidate key is a minimal superkey—a superkey that has no unnecessary attributes (more on this below)
- Helps in normalization by identifying redundant dependencies.
- Determines if an attribute set can uniquely determine all attributes in a relation.

How to Compute Attribute Closure (X^+)?

To compute X^+, follow these steps:

- Start with X: Put all attributes of X in the closure.
- Apply FDs: Add attributes to the closure if they can be determined by FDs.
- Repeat Until No More Attributes Can Be Added.

Example:

Consider a relation **R(A, B, C, D, E)** with FDs:

- $A \rightarrow B$
- $B \rightarrow C$
- $A, C \rightarrow D$
- $D \rightarrow E$

Find A^+ (closure of A):

1. Start: $A^+ = \{A\}$
2. Since $A \rightarrow B$, add B: $A^+ = \{A, B\}$
3. Since $B \rightarrow C$, add C: $A^+ = \{A, B, C\}$
4. Since $A, C \rightarrow D$ and we already have A and C, add D: $A^+ = \{A, B, C, D\}$
5. Since $D \rightarrow E$, add E: $A^+ = \{A, B, C, D, E\}$

Since A^+ = **all attributes of R**, this means A is a candidate key.

Superkeys:

A **superkey** is any **set of attributes that uniquely identifies a row** in a relation.

Example:
For a relation Student(StudentID, Name, Email, Phone), possible superkeys:

- {StudentID} ✅ (Unique on its own)
- {StudentID, Name} ✅ (Still unique, but Name is unnecessary.)
- {StudentID, Email, Phone} ✅ (Still unique, but redundant.)

Idea: Superkeys may contain unnecessary attributes.

Candidate Keys and Primary Keys:

Remembering from before that a candidate key is a minimal superkey—a superkey that has no unnecessary attributes.

How to find candidate keys?

1. Compute X^+ for potential superkeys.
2. If X^+ = all attributes in the relation, then X is a superkey.
3. Remove any unnecessary attributes → If removing an attribute still makes X unique, then it's not minimal.
4. The remaining minimal superkeys are candidate keys.

What Is a Primary Key?

- A primary key is one of the candidate keys chosen by the designer to uniquely identify rows.
- Every relation must have one primary key.

Example:
For a relation **Employee(EmpID, Name, SSN, Department)**:

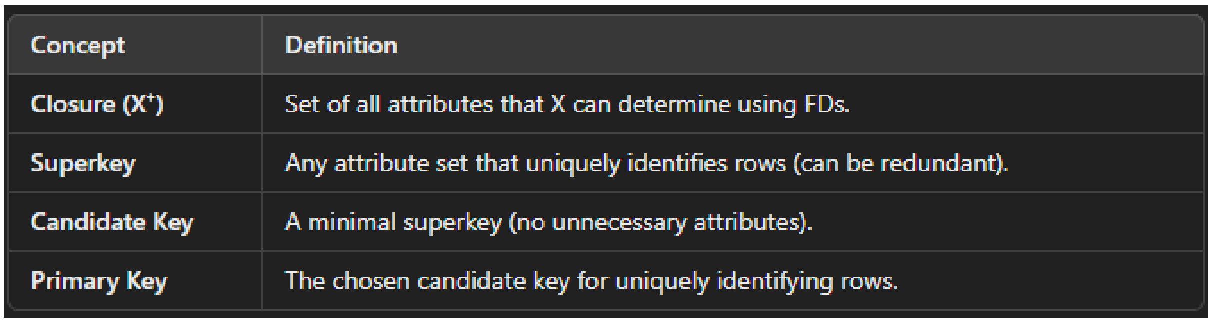
- FDs:
  - EmpID → Name, Department
  - SSN → Name, Department

Candidate keys:

- {EmpID} ✅
- {SSN} ✅

The primary key is selected from these candidate keys.

Main takeaways:



Concept	Definition
Closure (X^+)	Set of all attributes that X can determine using FDs.
Superkey	Any attribute set that uniquely identifies rows (can be redundant).
Candidate Key	A minimal superkey (no unnecessary attributes).
Primary Key	The chosen candidate key for uniquely identifying rows.

Inferring Functional Dependencies (FDs):

What Does "Inferring" Mean?

- It means deriving new functional dependencies from an initial set of given dependencies.
- This is done using Armstrong's Axioms, which provide rules to logically infer new FDs.

Armstrong's Axioms are **3 fundamental rules** that help us **infer** FDs.
Knowing the rules themselves instead of names is more relevant tbh

1. Reflexivity Rule:
If Y ⊆ X, then X → Y

- If the right-hand side (RHS) is already part of the left-hand side (LHS), the FD always holds (trivial FD).

Example:
{A, B} → A (Since A is in {A, B}, this is trivial.)

2. Augmentation Rule:
If X → Y, then XZ→YZ
- You can add the same attribute(s) to both sides of an FD, and it will still hold.

Example:
Given A → B, then we can infer AC → BC (Adding C to both sides).

3. Transitivity Rule:
If X → Y and Y → Z, then X → Z
- If X determines Y, and Y determines Z, then X determines Z (like a chain reaction).

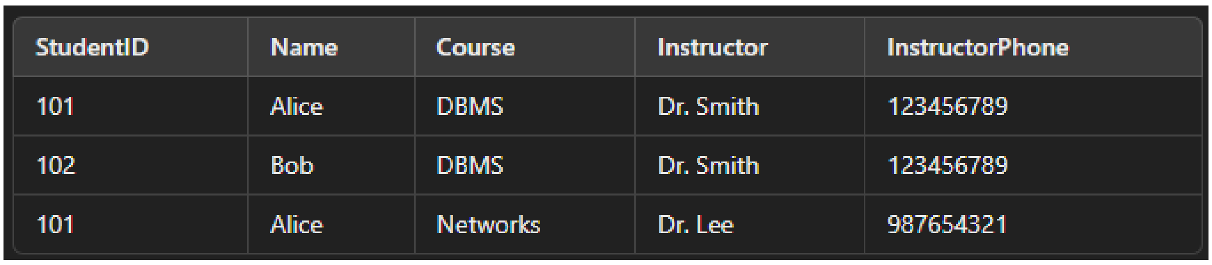
Example:
Given A → B and B → C, we can infer A → C.

Normalization & Normal Forms:

What Is Normalization?
Normalization is the process of organizing a database to:
- Minimize **redundancy** (no repeated data).
- Prevent **anomalies** (problems with insertion, update, deletion).
- Ensure data **integrity** (data remains consistent).

Types of Anomalies:
- **Insertion** Anomaly: Can't insert a row without adding unrelated data.
- **Update** Anomaly: Changing data in one row requires changing it everywhere.
- **Deletion** Anomaly: Deleting a row removes important data.

Example: Poorly designed table



StudentID	Name	Course	Instructor	InstructorPhone
101	Alice	DBMS	Dr. Smith	123456789
102	Bob	DBMS	Dr. Smith	123456789
101	Alice	Networks	Dr. Lee	987654321

- Redundancy: Dr. Smith’s phone number is repeated.
- Update Anomaly: If Dr. Smith changes his number, it must be updated everywhere.
- Deletion Anomaly: If we delete Alice’s Networks record, we lose Dr. Lee’s contact.

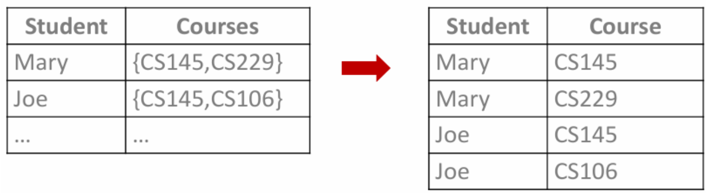
How do we fix this? We apply **normal forms**.

The Normal Forms (1NF to BCNF):

Each normal form (NF) applies certain rules to improve database design.

1st Normal Form (1NF) → Remove repeating values:

- Every column must contain atomic (indivisible) values.
- No repeating groups (multiple values in a single field).

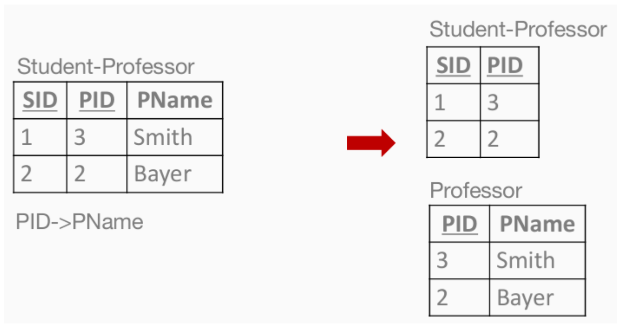


Student	Courses
Mary	{CS145,CS229}
Joe	{CS145,CS106}
...	...

Student	Course
Mary	CS145
Mary	CS229
Joe	CS145
Joe	CS106

2nd Normal Form (2NF) → Remove Partial Dependencies:

- Must be in 1NF
- No attribute that isn’t **prime** is functionally dependent on a proper subset of a candidate key
    - An attribute that is member of some key is **prime**



Student-Professor

SID	PID	PName
1	3	Smith
2	2	Bayer

PID->PName

Student-Professor

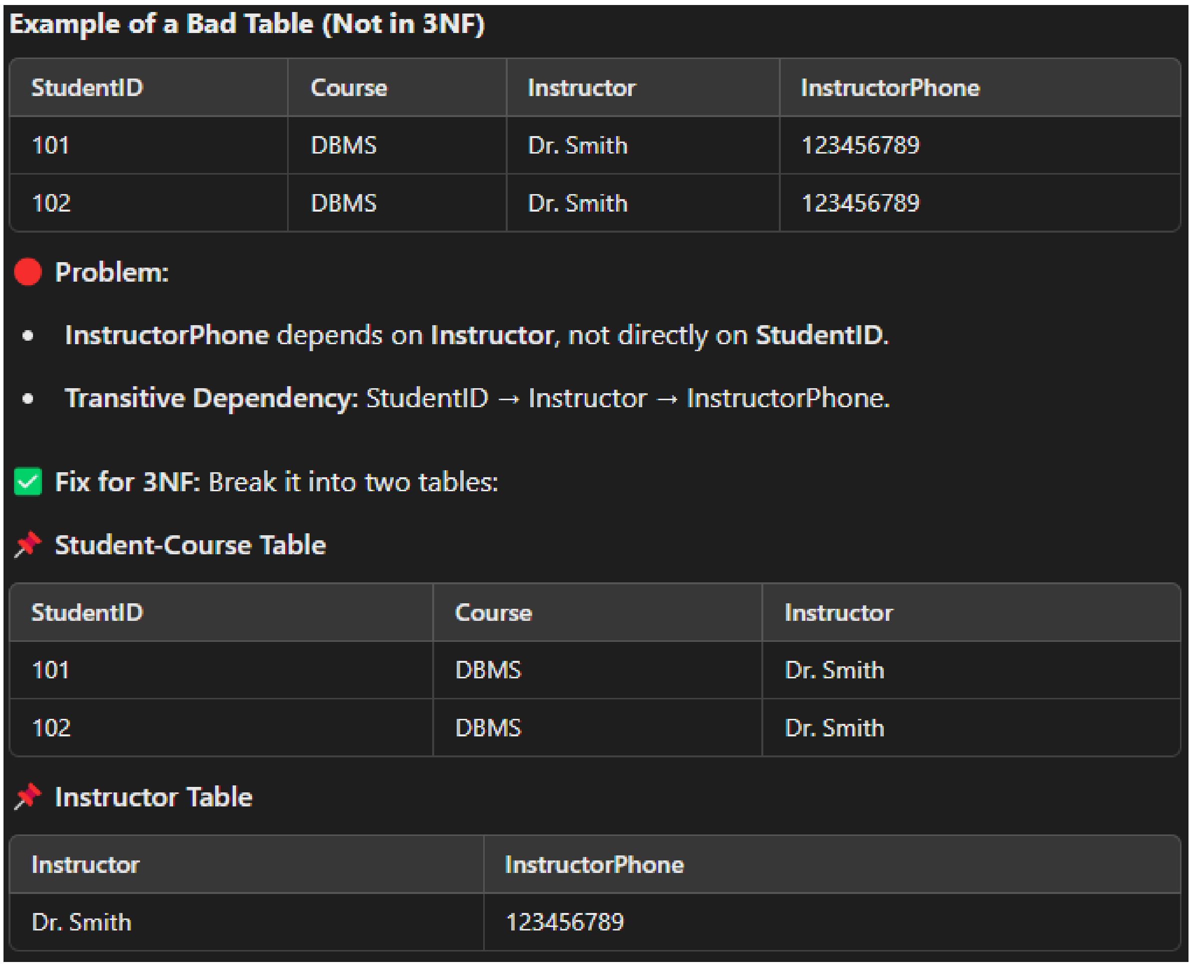
SID	PID
1	3
2	2

Professor

PID	PName
3	Smith
2	Bayer

3rd Normal Form (3NF) → Remove Transitive Dependencies:

- Must be in 2NF
- No transitive dependencies (non-key attributes must depend only on the primary key, not another non-key attribute).



Example of a Bad Table (Not in 3NF)

StudentID	Course	Instructor	InstructorPhone
101	DBMS	Dr. Smith	123456789
102	DBMS	Dr. Smith	123456789

🔴 **Problem:**

- **InstructorPhone** depends on **Instructor**, not directly on **StudentID.**
- **Transitive Dependency:** StudentID → Instructor → InstructorPhone.

✅ **Fix for 3NF:** Break it into two tables:

📌 **Student-Course Table**

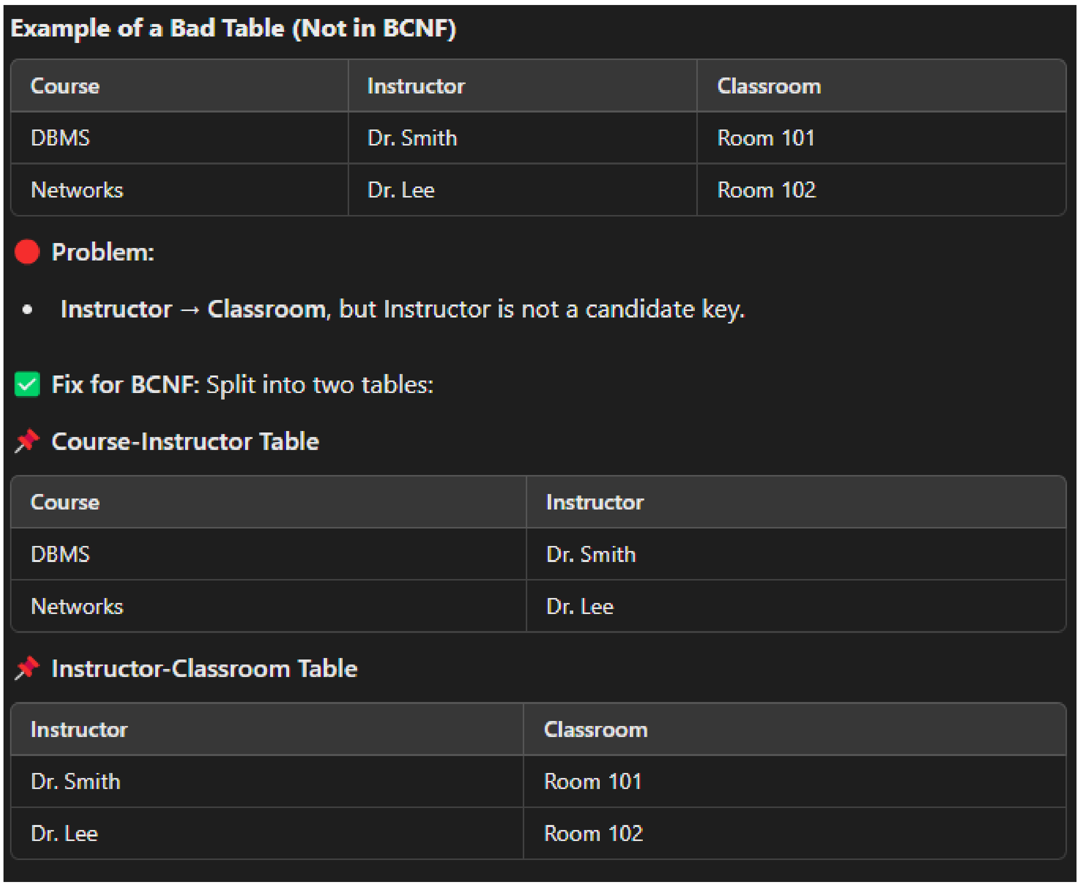
StudentID	Course	Instructor
101	DBMS	Dr. Smith
102	DBMS	Dr. Smith

📌 **Instructor Table**

Instructor	InstructorPhone
Dr. Smith	123456789

Boyce-Codd Normal Form (BCNF) → Stronger 3NF:

- Must be in 3NF
- Every determinant must be a candidate key (No non-trivial FD where a non-key determines a key).



Example of a Bad Table (Not in BCNF)

Course	Instructor	Classroom
DBMS	Dr. Smith	Room 101
Networks	Dr. Lee	Room 102

🔴 **Problem:**

- **Instructor → Classroom**, but Instructor is not a candidate key.

✅ **Fix for BCNF:** Split into two tables:

📌 **Course-Instructor Table**

Course	Instructor
DBMS	Dr. Smith
Networks	Dr. Lee

📌 **Instructor-Classroom Table**

Instructor	Classroom
Dr. Smith	Room 101
Dr. Lee	Room 102

Summary:

- 1NF (No repeating groups): Break multi-valued attributes into separate rows.
- 2NF (No partial dependencies): Ensure non-key attributes depend on the **whole** primary key.
- 3NF (No transitive dependencies): Break table so non-key attributes depend **only** on the primary key.
- BCNF (No non-key determinant): Ensure every determinant is a **candidate key**.

SQL - slide 251

Table Declaration:

```
CREATE TABLE MovieStar (
            id          INTEGER,
            name        CHAR(30),
            address     VARCHAR(255),
            gender      CHAR(1),
            birthdate   DATE
);
```

Table and relative tuples deletion:

```
DROP TABLE table_name;
```

Modify the schema of an existing relation:

```
ALTER TABLE table_name ADD column_name data_type;
ALTER TABLE table_name DROP column_name;
```

Default Values:

We can define a default value to a column by explicitly writing "DEFAULT" when creating a table. If nothing is defined, it is NULL.

```
CREATE TABLE <table_name> (
<column_name> <data_type> DEFAULT <default_value>,
...
<column_name> <data_type>
);
```

Example:

```
CREATE TABLE MovieStar (
            id          INTEGER,
            name        CHAR(30),
            address     VARCHAR(255),
            gender      CHAR(1) DEFAULT '?',
            birthdate   DATE
);

ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT
'unlisted';
```

Integrity Constraints:

Impose restrictions on allowable data, beyond those imposed by structure and types.
Examples:

- $0.0 < GPA \leq 4.0$
- enrolment < 50,0000
- Decision: ‘Y’, ‘N’ or NULL
- $Major = CS \Rightarrow decision = NULL$
- *sizeHS* < 200 ⇒ *not admitted in colleges with enroll* > 30,000

Classification of Constraints:

Non-null constraints:

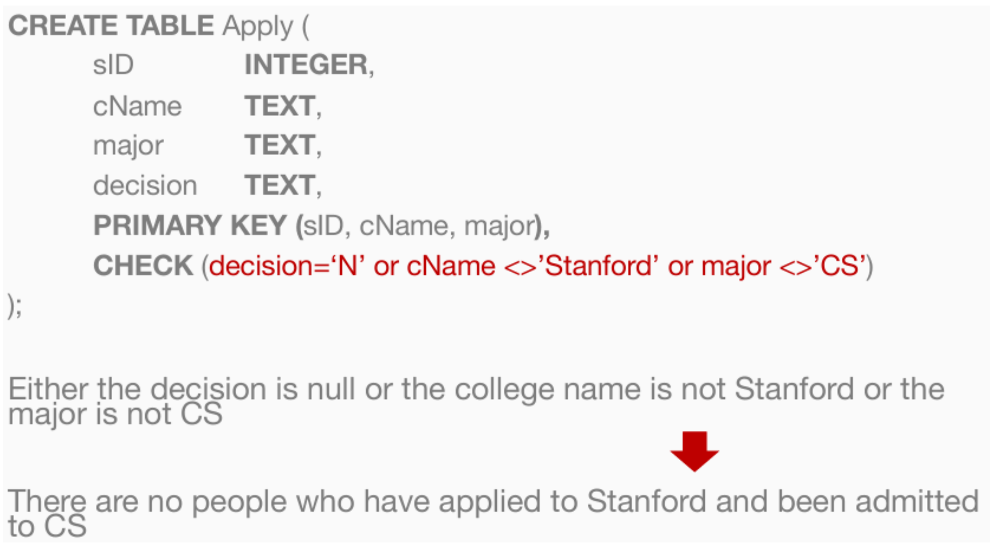
- Defines that a column does not have NULL values
- Ex.: name CHAR(30) **NOT NULL**

Key constraints:

- Primary:
    - We can define one, and only one, primary key for a table
    - A PK cannot be NULL and cannot have repeated values
    - Ex.: id INTEGER **PRIMARY KEY**
    - Multiple column primary key is also possible to implement
    - Ex.: **PRIMARY KEY** (column_1_name, column_2_name)
- Unique:
    - We can define multiple unique keys for a table. Unique keys allow NULL values.
    - A unique value cannot have repeated values.
    - Ex.: address VARCHAR(255) **UNIQUE**
    - Multiple column unique key is also possible to implement
    - Ex.: **UNIQUE** (column_1_name, column_2_name)

Attribute-based and tuple-based constraints:

- **Att-Bsd CHECK** Constraint:
  - Constrain the value of a particular attribute. This is checked whenever we insert or update a tuple.
  - Ex.: GPA REAL CHECK (GPA<=4.0 and GPA>0.0)
- **Tup-Bsd CHECK** Constraint:
  - Constrain relationships between different values in each tuple. This is checked whenever we insert or update a tuple.

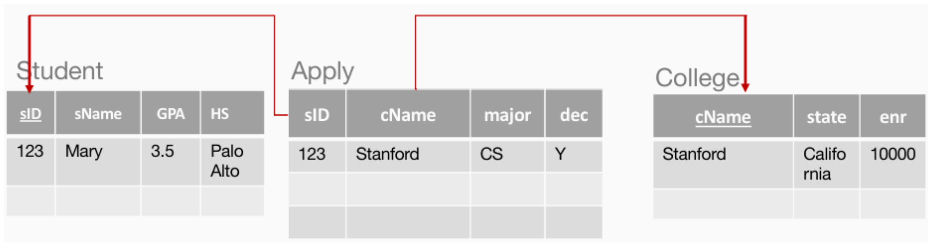


```
CREATE TABLE Apply (
        sID         INTEGER,
        cName       TEXT,
        major       TEXT,
        decision    TEXT,
        PRIMARY KEY (sID, cName, major),
        CHECK (decision='N' or cName <>'Stanford' or major <>'CS')
);
```

Either the decision is null or the college name is not Stanford or the major is not CS

There are no people who have applied to Stanford and been admitted to CS

Referential integrity (foreign key):



Student

sID	sName	GPA	HS
123	Mary	3.5	Palo Alto

Apply

sID	cName	major	dec
123	Stanford	CS	Y

College

cName	state	enr
Stanford	California	10000

Referential integrity from R.A to S.B

- A is called the "foreign key".
- B is usually required to be the primary key for table S or at least unique.
- Multi-attribute foreign keys are allowed.

Foreign Key Declaration:

- column_z data_type **REFERENCES** table_A (column_A)

Example:

```
CREATE TABLE College (cName text PRIMARY KEY, state text,
enrollment int);

CREATE TABLE Student (sID int PRIMARY KEY, sName text, GPA
real, sizeHS int);

CREATE TABLE Apply (
    sID REFERENCES Student(sID),
    cName text REFERENCES College (cName),
    major text,
    decision text,
    PRIMARY KEY(sID, cName)
);
```

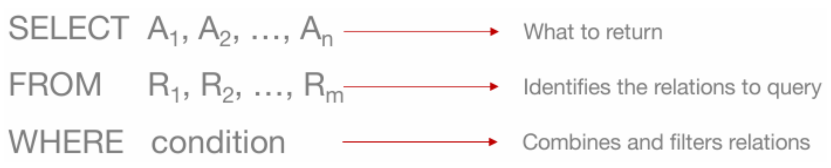
If the referenced column is the primary key of the other table, we can omit the name of the column

```
CREATE TABLE <table_A> (
    <column_A> <data_type> PRIMARY KEY,
    <column_B> <data_type>,
     ...
    <column_C> <data_type>
);

CREATE TABLE <table_B> (
    <column_X> <data_type> PRIMARY KEY,
    <column_Y> <data_type>,
    ...
    <column_Z> <data_type> REFERENCES <table_A>
);
```

SQL deeper dive - slide 400

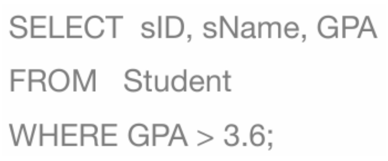
SELECT Statement:



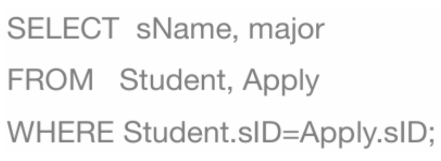
SELECT $A_1, A_2, \ldots, A_n$ $\longrightarrow$ What to return

FROM $R_1, R_2, \ldots, R_m$ $\longrightarrow$ Identifies the relations to query

WHERE condition $\longrightarrow$ Combines and filters relations



```
SELECT  sID, sName, GPA
FROM   Student
WHERE GPA > 3.6;
```



```
SELECT  sName, major
FROM   Student, Apply
WHERE Student.sID=Apply.sID;
```

LIKE operator:
Allows for string matching on attribute values.

- **%** is used to define any sequence of characters (ex.: *pkc*, *fghijkl*)
- **_** is used to define a single character.

Suppose you have:

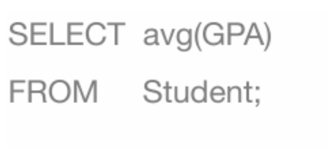
- **SELECT sID, major FROM Apply WHERE major like ‘%bio%’;**

This could provide you with results like:

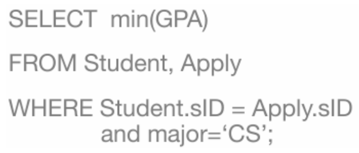
- *biology*, *bioengineering*, *marine biology*, …

While something like ‘_bio%’, could get you: *abiotic*, …

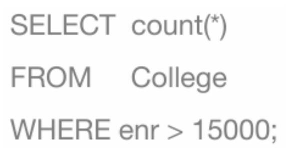
Aggregation: - relevant because of group by
Perform aggregation over sets of values in multiple rows.
Basic functions: min, max, sum, avg, count.
Examples:



```
SELECT  avg(GPA)
FROM    Student;
```



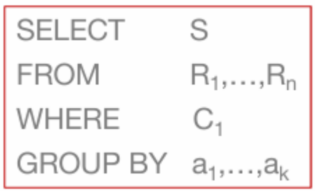
```
SELECT  min(GPA)
FROM Student, Apply
WHERE Student.sID = Apply.sID
      and major=‘CS’;
```



```
SELECT  count(*)
FROM    College
WHERE enr > 15000;
```

Group By clause:
Only used in conjunction with aggregation. Used to partition a relation by values of a given attribute or set of attributes.

1. Compute the FROM and WHERE clauses
2. Group by the attributes in the GROUP BY
3. Compute the SELECT clause

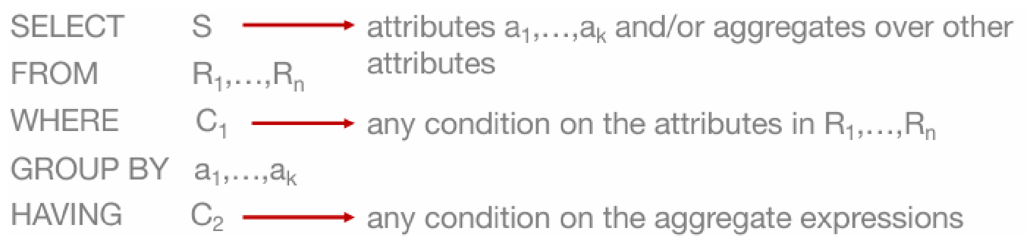


SELECT S
FROM $R_1,\ldots,R_n$
WHERE C_1
GROUP BY $a_1,\ldots,a_k$

Having clause:
Main take: The HAVING clause filters grouped results after aggregation.
Applies conditions to the results of the aggregate functions.

- Check conditions that involve the entire group
- WHERE clause applies to one tuple at a time

Applied **after the GROUP BY clause**:



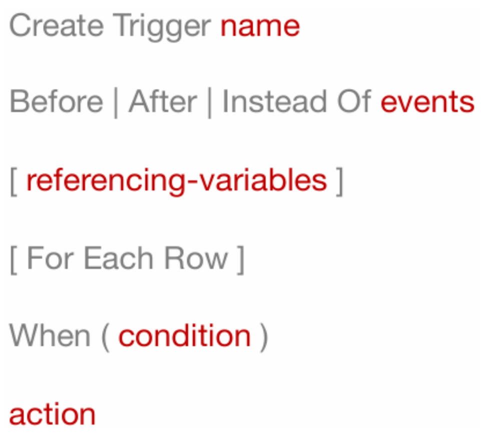
SELECT S $\longrightarrow$ attributes $a_1,\dots,a_k$ and/or aggregates over other attributes
FROM $R_1,\dots,R_n$
WHERE C_1 $\longrightarrow$ any condition on the attributes in $R_1,\dots,R_n$
GROUP BY $a_1,\dots,a_k$
HAVING C_2 $\longrightarrow$ any condition on the aggregate expressions

Example: List the colleges with fewer than 5 applications

- SELECT cName FROM Apply GROUP BY cName HAVING count(*) < 5

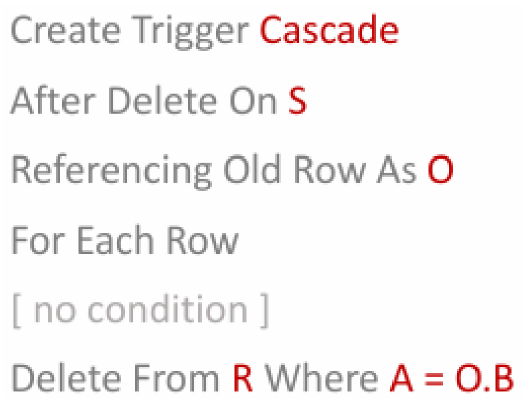
Triggers:
Event-condition-action rules.

Structure:



Create Trigger name
Before | After | Instead Of events
[referencing-variables]
[For Each Row]
When (condition)
action

Example:



Create Trigger Cascade
After Delete On S
Referencing Old Row As O
For Each Row
[no condition]
Delete From R Where A = O.B

1. Events:
    - insert on T
    - delete on T
    - update [C1,...,Cn] on T
        - columns are optional

2. Referencing Values
    - To reference the data that was modified and caused the trigger to be activated.
        - old row as var
        - new row as var
        - old table as var
        - new table as var
    - *New* variables for inserts and updates
    - *Old* variables for deletes and updates

3. [For Each Row]
    - Optional Clause
    - States that the trigger is activated once for each modified tuple.
    - If, for example, a delete command deletes 10 tuples
        - With “for each row”, trigger will run 10 times, once for each deleted tuple
        - Without it, trigger will run once for the entire statement
    - The trigger is always activated at the end of the statement
        - Either once, or x times if “for each row” is present
4. Condition
    - Tests the condition, if it is true, action will be performed
5. Action
    - SQL Statement

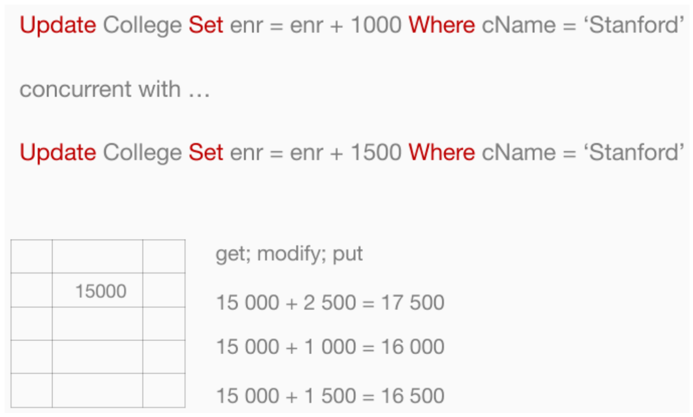
Note:

- If the trigger is **row-level**, [For Each Row] tends to be on
- If the trigger is **statement-level**, [For Each Row] tends to be off

Transactions:
They are necessary to deal with concurrent database access and assure resilience to system failures.

To contextualize and help you understand this, here are some
Examples of Inconsistency:

1. on Attribute-level



Update College Set enr = enr + 1000 Where cName = ‘Stanford’

concurrent with …

Update College Set enr = enr + 1500 Where cName = ‘Stanford’

15000

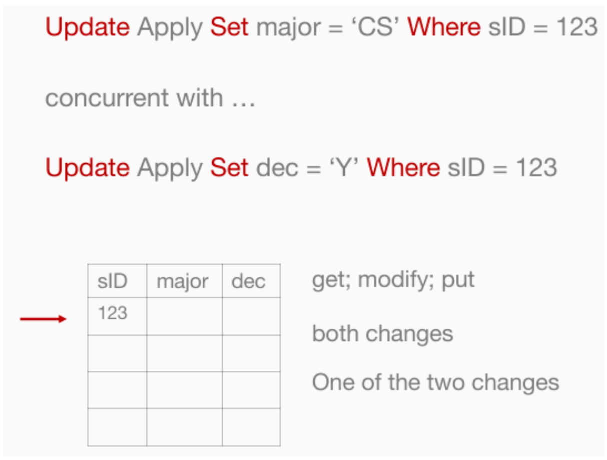
get; modify; put

15 000 + 2 500 = 17 500

15 000 + 1 000 = 16 000

15 000 + 1 500 = 16 500

2. on Tuple-level



Update Apply Set major = ‘CS’ Where sID = 123

concurrent with …

Update Apply Set dec = ‘Y’ Where sID = 123

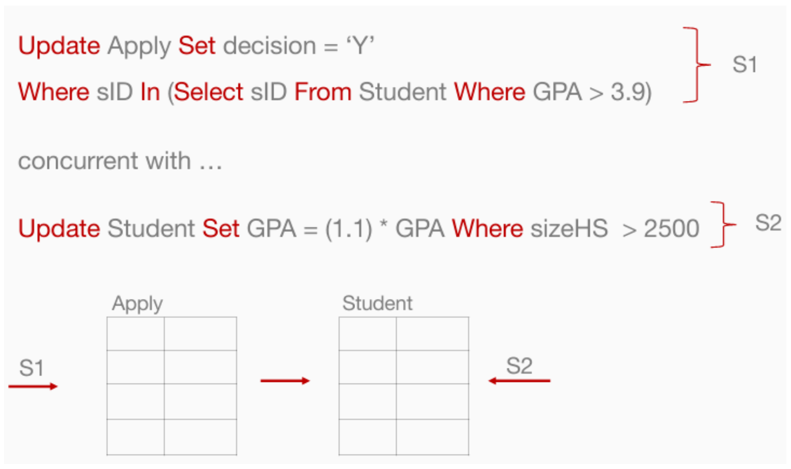
sID	major	dec
123		

get; modify; put

both changes

One of the two changes

3. on Table-level



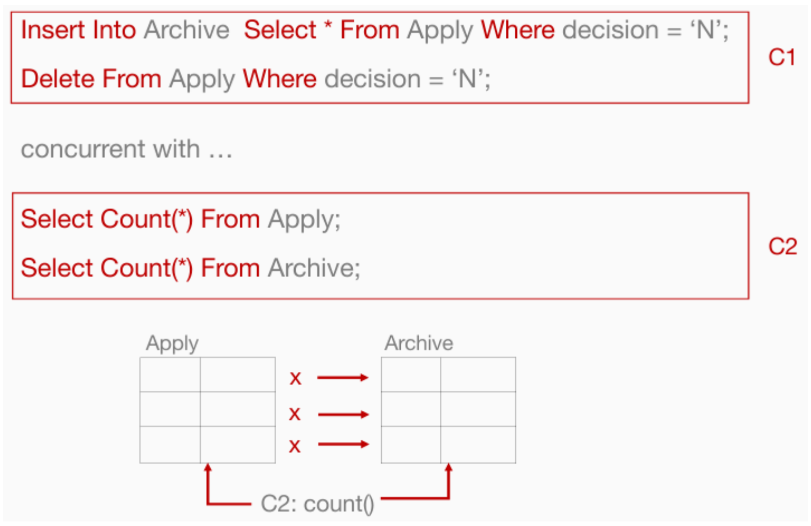
Update Apply Set decision = ‘Y’
Where sID In (Select sID From Student Where GPA > 3.9) } S1

concurrent with …

Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500 } S2

Apply
Student
S1
S2

4. Multi-Statement

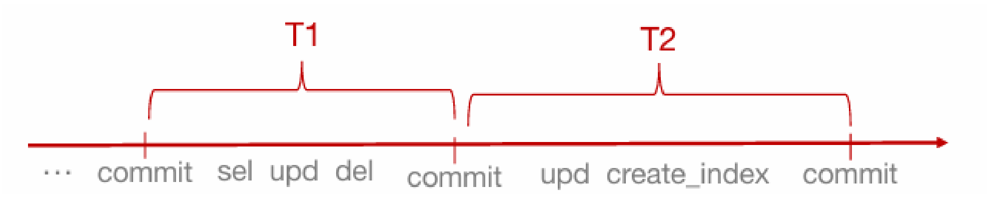


```
Insert Into Archive  Select * From Apply Where decision = 'N';
Delete From Apply Where decision = 'N';
```
C1

concurrent with …

```
Select Count(*) From Apply;
Select Count(*) From Archive;
```
C2

So,
Transactions are the solution for both concurrency and failures.



A transaction is a sequence of one or more SQL operations treated as a unit.

- Transactions appear to run in isolation.
- If the system fails, each transaction's changes are reflected either entirely or not at all.

And:

- Transaction begins automatically on first SQL statement
- On "commit" transaction ends and new one begins
- Current transaction ends on session termination
- "Autocommit" turns each statement into transaction

ACID Properties in the topic of Transactions:

- Atomicity
- Consistency
- Isolation
- Durability

Atomicity:

- A transaction is treated as a single unit. **It either completes fully or does not execute at all** (typically called “all-or-nothing”).
- Example: If a bank transfer debits an account but fails to credit another, the entire transaction is rolled back.

Consistency:

- A transaction brings the database from one **valid state to another**, maintaining all defined rules, such as constraints or triggers.
- Example: If a transfer violates a balance constraint, it is aborted.

Isolation:

- Transactions **execute independently** of one another, as if they were running sequentially, even if they occur simultaneously.
- Example: Two users booking the last available seat at the same time won’t overwrite each other's actions.

Durability:

- Once a transaction is **committed**, its changes are **permanent**, even in case of a system failure.
- Example: After saving a file in a database, a power outage won't result in data loss.

QUESTIONS

Answers can be checked below

Q1: Which of the following statements is **true** regarding row-level and statement-level triggers?
a) Row-level triggers execute once for each row affected by the triggering statement.
b) Statement-level triggers execute once for each row affected.
c) Row-level triggers cannot be defined for AFTER triggers.
d) Statement-level triggers cause recursion when affecting multiple rows.

Q2: What could happen if a trigger performs an update on the same table that triggered it?
a) An infinite loop may occur.
b) The database automatically prevents recursion.
c) It will always execute the trigger only once.
d) This is not allowed by most database systems.

Q3: What is typically **not included** in the study of system load during physical schema design?
a) Query execution patterns and frequency.
b) Estimation of tuple growth rates for each table.
c) Storage and indexing requirements.
d) The identification of integrity constraints.

Q4: During physical schema design, the **growth rate of tuples** in a table affects which of the following?
a) The primary key selection.
b) The schema normalization process.
c) The choice of indexing strategies.
d) Query execution order.

Q5: What is the **main objective** of normalization in relational database design?
a) Reducing the number of tables in the schema.
b) Eliminating redundancy and ensuring data integrity.
c) Optimizing query performance by reducing joins.
d) Increasing the complexity of relationships for scalability.

Q6: Which of the following is **true** about integrity constraints?
a) They can include primary keys, foreign keys, and check constraints.
b) They are always enforced at the application layer.
c) They do not affect physical schema design.
d) They are optional in relational database systems.

Q7: A row-level trigger is created to execute after an INSERT statement on a table. What happens if the INSERT statement affects 10 rows?
a) The trigger executes once.
b) The result depends on the database system.
c) The trigger executes 10 times.
d) The trigger cannot execute on multiple rows.

Q8: A statement-level trigger is fired by an UPDATE query. Which of the following is **true**?
a) It will execute for each row affected by the update.
b) It will execute once for the entire update statement.
c) It will not execute if no rows are updated.
d) It executes before row-level triggers for the same event.

Q9: Why is it important to estimate tuple growth rates during physical schema design?
a) To determine the relational schema normalization level.
b) To identify redundant attributes in tables.
c) To decide the primary key and foreign key relationships.
d) To optimize storage and ensure scalability.

Q10: During system load analysis, what is the primary focus when studying query execution patterns?
a) Identifying the most complex queries for redesign.
b) Determining the table relationships to normalize.
c) Estimating the frequency and resource usage of queries.
d) Reducing the need for indexing on frequently queried columns.

ANSWERS

Q1: Which of the following statements is **true** regarding row-level and statement-level triggers?
a) Row-level triggers execute once for each row affected by the triggering statement.
b) Statement-level triggers execute once for each row affected.
c) Row-level triggers cannot be defined for AFTER triggers.
d) Statement-level triggers cause recursion when affecting multiple rows.

Explanation: Row-level triggers run for each affected row, while statement-level triggers run once per statement.

Q2: What could happen if a trigger performs an update on the same table that triggered it?
a) An infinite loop may occur.
b) The database automatically prevents recursion.
c) It will always execute the trigger only once.
d) This is not allowed by most database systems.

Explanation: Self-triggering updates can lead to infinite loops unless explicitly managed.

Q3: What is typically **not included** in the study of system load during physical schema design?
a) Query execution patterns and frequency.
b) Estimation of tuple growth rates for each table.
c) Storage and indexing requirements.
d) The identification of integrity constraints.

Explanation: While integrity constraints are considered, system load focuses more on storage, indexing, and query patterns.

Q4: During physical schema design, the **growth rate of tuples** in a table affects which of the following?
a) The primary key selection.
b) The schema normalization process.
c) The choice of indexing strategies.
d) Query execution order.

Explanation: Growth rate influences index design to optimize query performance and scalability.

Q5: What is the **main objective** of normalization in relational database design?
a) Reducing the number of tables in the schema.
b) Eliminating redundancy and ensuring data integrity.
c) Optimizing query performance by reducing joins.
d) Increasing the complexity of relationships for scalability.

Explanation: Normalization minimizes data duplication and prevents anomalies.

Q6: Which of the following is **true** about integrity constraints?
a) They can include primary keys, foreign keys, and check constraints.
b) They are always enforced at the application layer.
c) They do not affect physical schema design.
d) They are optional in relational database systems.

Explanation: Integrity constraints ensure consistency and valid data in the database.

Q7: A row-level trigger is created to execute after an INSERT statement on a table. What happens if the INSERT statement affects 10 rows?
a) The trigger executes once.
b) The result depends on the database system.
c) The trigger executes 10 times.
d) The trigger cannot execute on multiple rows.

Explanation: Row-level triggers fire for each row affected.

Q8: A statement-level trigger is fired by an UPDATE query. Which of the following is **true**?
a) It will execute for each row affected by the update.
b) It will execute once for the entire update statement.
c) It will not execute if no rows are updated.
d) It executes before row-level triggers for the same event.

Explanation: Statement-level triggers fire once per statement.

Q9: Why is it important to estimate tuple growth rates during physical schema design?
a) To determine the relational schema normalization level.
b) To identify redundant attributes in tables.
c) To decide the primary key and foreign key relationships.
d) To optimize storage and ensure scalability.

Explanation: Tuple growth rate impacts indexing, storage allocation, and scalability.

Q10: During system load analysis, what is the primary focus when studying query execution patterns?
a) Identifying the most complex queries for redesign.
b) Determining the table relationships to normalize.
c) Estimating the frequency and resource usage of queries.
d) Reducing the need for indexing on frequently queried columns.

Explanation: This ensures performance optimization based on usage.