

Unit 3: Relational Model and Schema Refinement

Dr. Ahmed E. Khaled

Department of Computer Science



MPCS 53001- Databases

- Different Database Models
- Intro to Relational Model
- Properties of Relational Model
- Transforming Entity sets
- Transforming Weak entity sets
- Transforming Relationship sets
- Database Normalization and Denormalization
 - First Normal Form (1NF)
 - Second Normal Form (2NF)
 - Third Normal Form (3NF)

Step 1- *Requirements Analysis*

Step 2- *Conceptual Database Design*

Step 3- *Logical Database Design* (The focus of the first part)

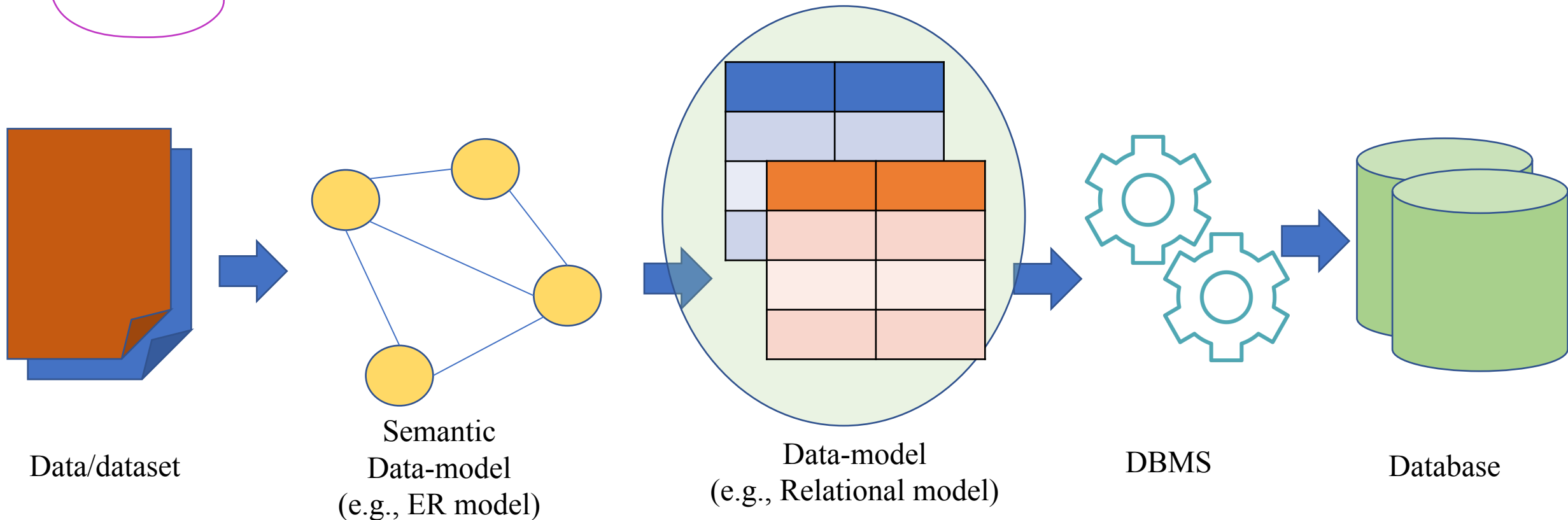


Step 4- *Schema Refinement* (The focus of the second part)

Step 5- *Physical Database Design*

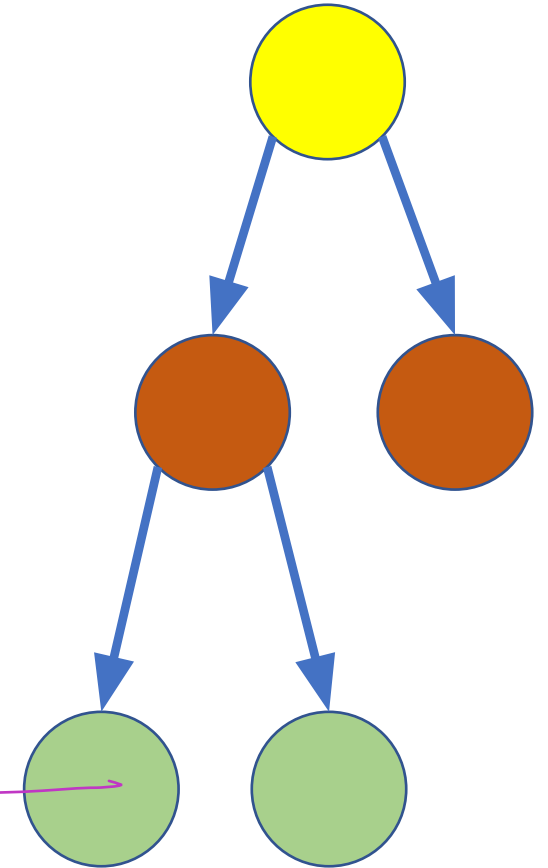
Step 6- *Application Design*

- A data model is a collection of high-level data description constructs (e.g., tables and links) that hide many low-level storage details.
- A DBMS –through data model- allows the user to focus on organizing and structuring the data to be stored.

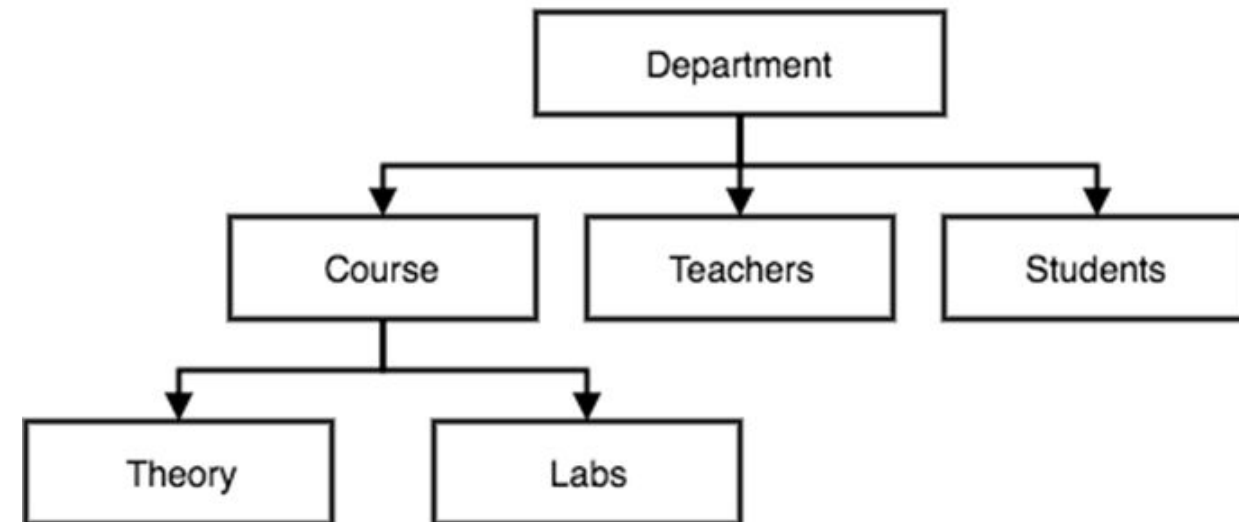


Different Data models used in designing databases

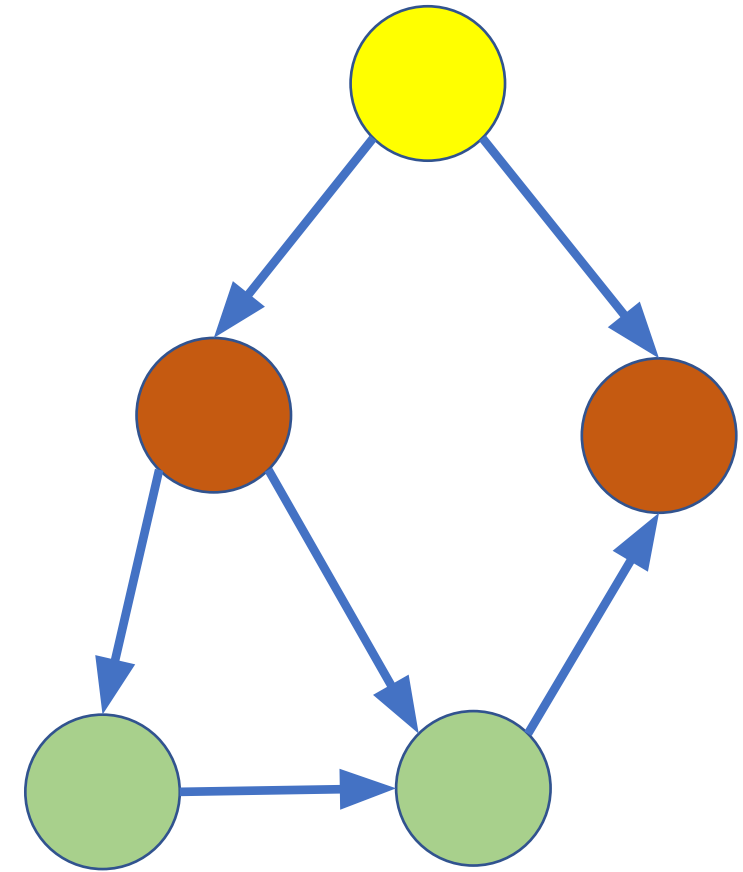
- The ***Hierarchical Data Model*** was the first model for database systems (1966) as an improvement on ***the general file-systems*** (as we discussed in unit 1).
- This model allows the creation of ***logical relationships*** between data items in a database.
- This model arranges data in a ***tree-like structure*** with a main root and children nodes, and children nodes to the previous children nodes.
- One essential property in a tree-like structure is that each child node ***has a single parent***.



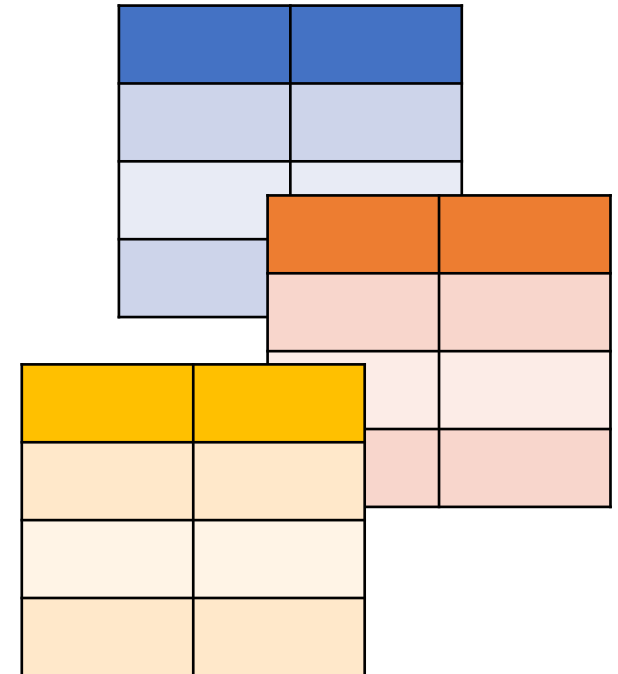
- The Hierarchical Data Model mainly represents the *one-to-many* (1:M) relationship type formed from the *parent-child* relationship (which is like folders and files on a computer).
- This model efficiently describes simple *real-world scenarios* like one department can have many courses, many professors and many students.
- This model *does not fit more complicated scenarios* that require other forms of relationships (e.g., M:1 or M:M).



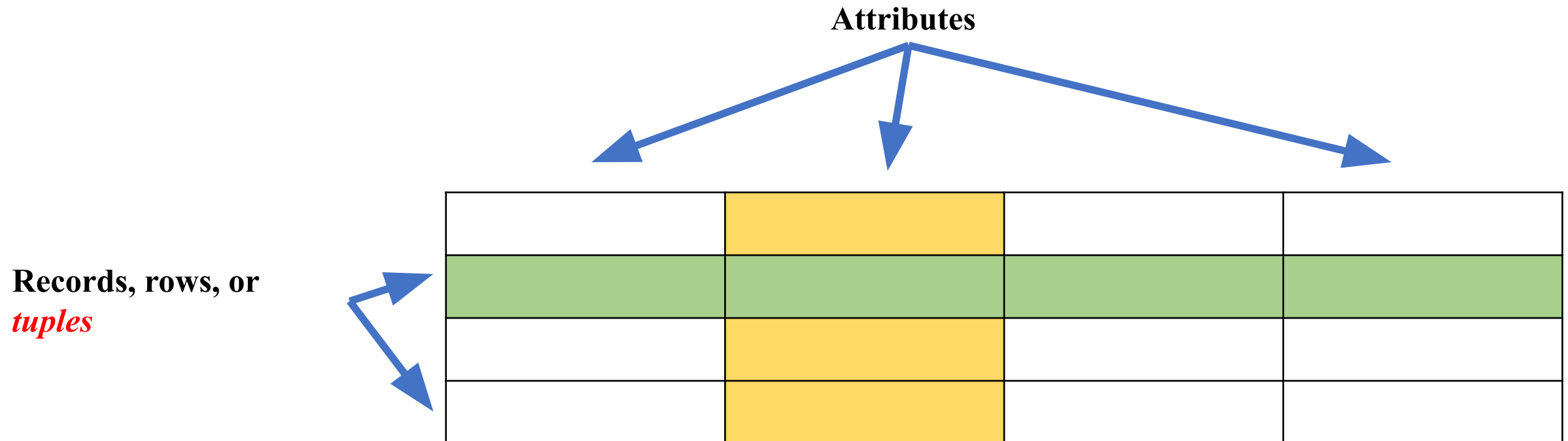
- The Network Data Model extends the hierarchical model through organizing data in a *graph-like structure*.
- In a graph-like structure, a child node can have *multiple parents*.
- This model allows the representation of both the *one-to-Many* (1:M) and *Many-to-Many* (M:M) relationships.



- In the relational model, the data is *logically structured* as tables (also known as relations - *not relationships*).
- A database in this model is a collection of one or more tables or relations. This representation enables the use of simple, high-level languages *to query the data* (e.g., SQL).
- The main objectives of this model:
 - High degree of *data independence*: applications must not be affected by modifications to the internal data representation (e.g., record ordering, change to file organization).
 - Enable expressing *complex queries*.

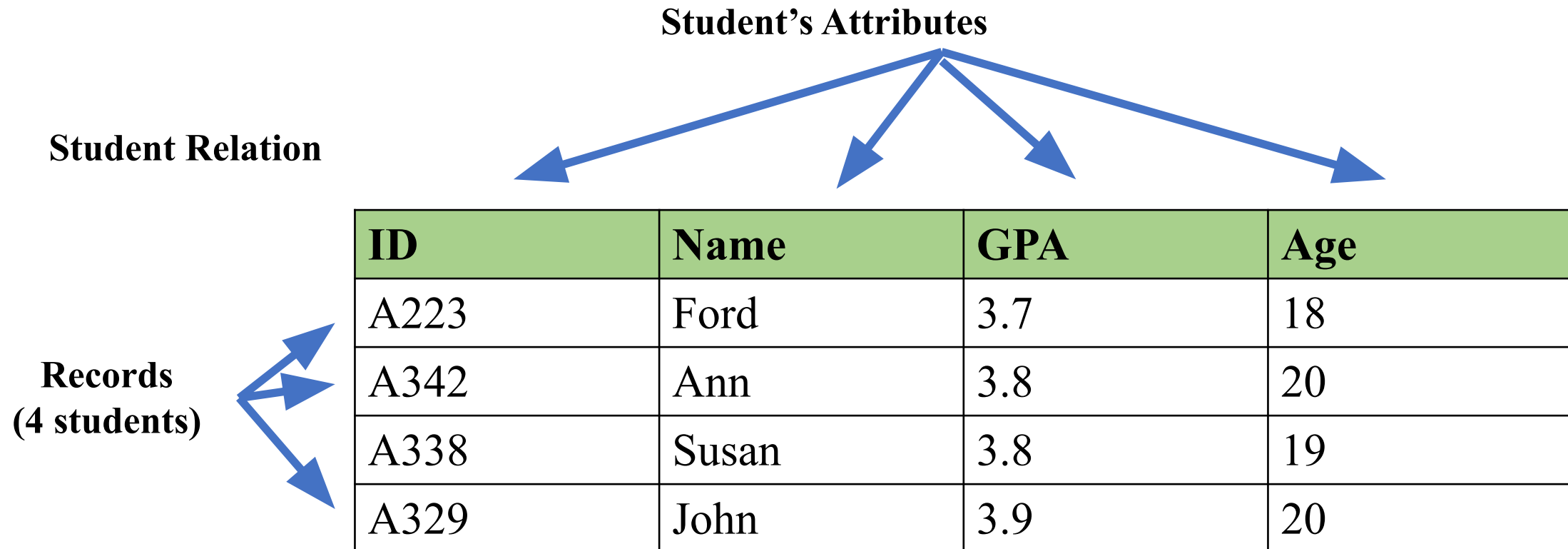


A **relation** is represented as two-dimensional table in which the rows of the table correspond to **individual records** and the columns of the table correspond to **attributes (or fields)**.



This perception applies to the **logical structure of the database** and doesn't apply to the **physical structure**, which can be implemented using a variety of storage structures (e.g., bin files).


- For example, the information of Student is represented by the *Student relation* (table), with *attributes* (columns): Student's ID, name, GPA, age.
- Records (tuples) represent the *instances* of the Student relation – where a column contains values of a single attribute. In this example, the relation stores info for 4 students.



Any relation (table) in a relational database has the following properties:

- The name is *distinct*, we cannot have two or more relations with the same name
- Each attribute has a *distinct name* (in the same relation, we cannot have two or more attributes with the same name, but two attributes in different relations can have same name)
- Each cell of the relation contains *exactly one atomic (single) value*
- The values of an attribute are all from the *same domain*

	GPA	
	3.8	
	3.5	



Name		
John		

A relation has the following properties:

- Each tuple is distinct, there are *no duplicate tuples – how ?!*

ID	Age	GPA
11	20	3.8
12	20	3.8



Two different Students with similar info

- The order of attributes has *no importance*

ID	GPA	Age
11	3.8	20
12	3.5	21

=

ID	Age	GPA
11	20	3.8
12	21	3.5

- Tuples can appear *in any order* and the relation will still convey the same meaning (practically the order may affect *the efficiency of accessing tuples*).

ID	GPA	Age
11	3.8	20
12	3.5	21

=

ID	GPA	Age
12	3.5	21
11	3.8	20

- The Relational Database Management System (RDBMS), the dominant type of DBMS, is based on the *relational data model* and was proposed in 1970.
- IBM Research Laboratory in California developed *System R* late 70's – that provided an *implementation of this model's data structures and operations*.
- The project also handled *transaction management*, concurrency control, *recovery techniques*, query optimization, data security and user interfaces.

System R from IBM led to two major developments:

- The development of a *Structured Query Language* called SQL.
- The production of various commercial relational DBMS products (e.g., DB2 from IBM and Oracle from Oracle Corporation).

Logical Database Design – Relational Schema

- The ER diagram (as a high-level model) we designed in Unit2 should be translated using the relational model into a collection of relations (tables) with associated constraints.
- In this unit, we would like to do such translation and build something called *relational database schemas*
- Think of a schema as the header of a relation (a table) specifying the following:
 - The relation's name
 - The name of each field (or column, or attribute)
 - The domain of each field.
- The student information in a university database may be stored in a relation with the *following schema*:

Students (*sid* : string , *name* : string , *age* : integer , *gpa* : real)

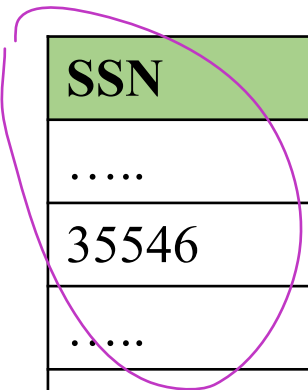
↑
Relation Name

↑
Attribute : Data Type “domain”

Employee (***SSN*** : integer , ***Name*** : string , ***Age*** : integer , ***Salary*** : real)

- Such schema says that each record (each employee) in the Employee relation (table) has four attributes (value for each attribute), with field names and types as indicated.

**Employee
Relation**



SSN	Name	Age	Salary
....
35546	Susan	26	50000
....
....

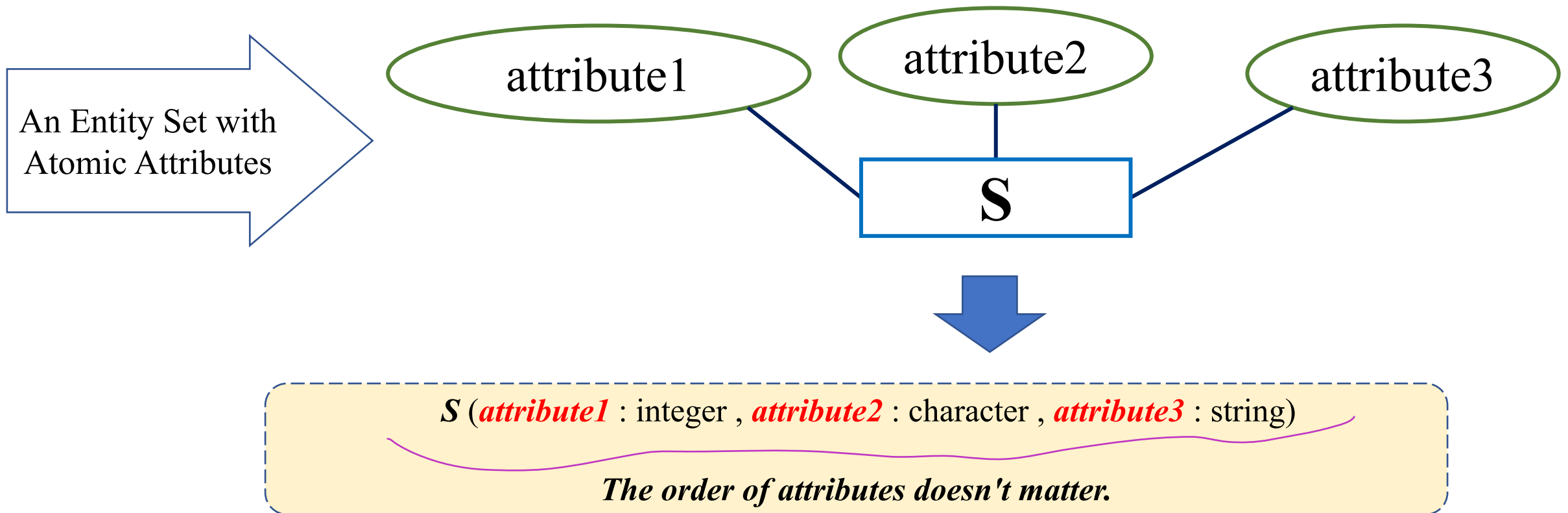
- Think of the schema as a ***template*** for describing a student, and each row in the Students relation is a ***record*** that describes a student in the university.

From ER diagram to relations:-

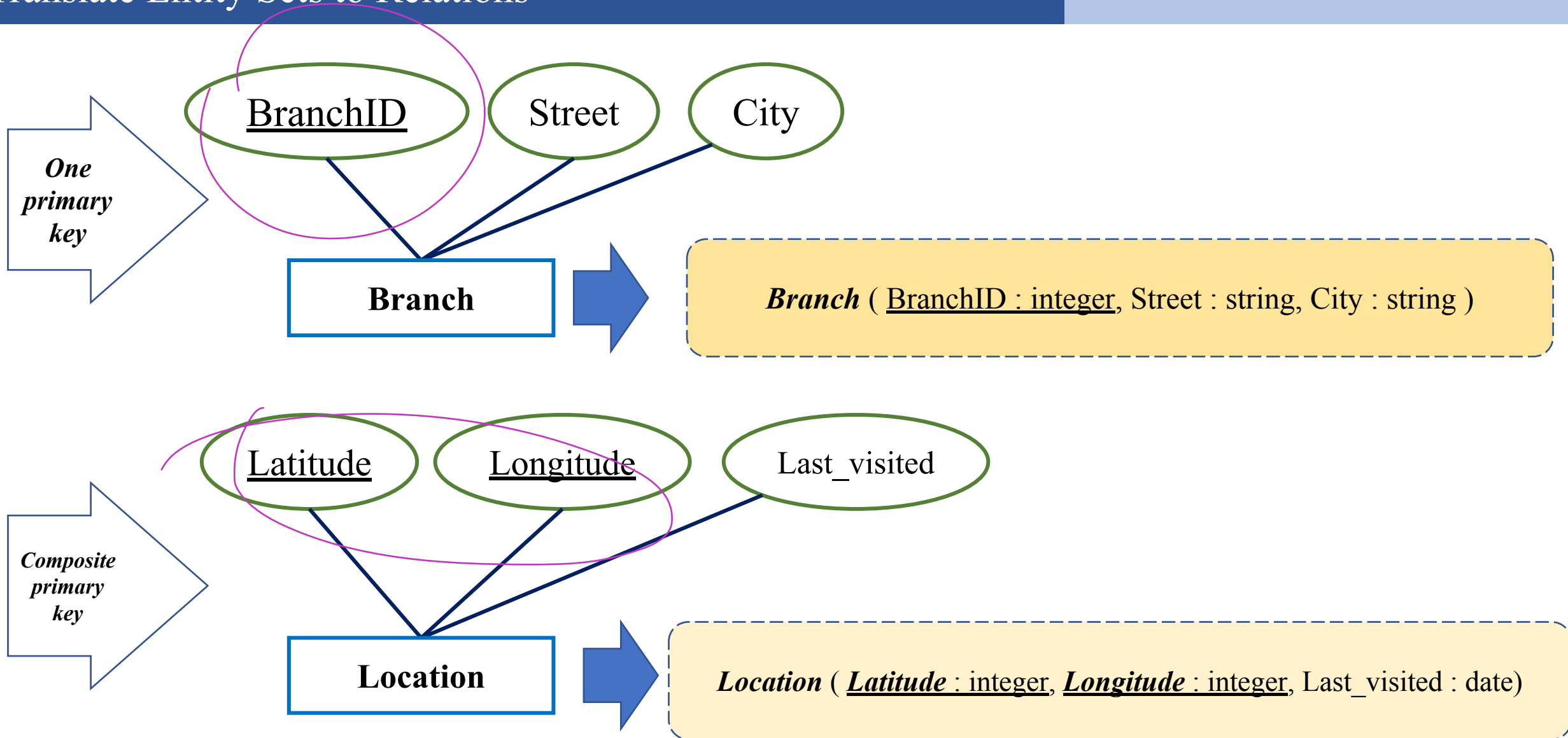
1st step - translate Entity Sets to Relations

Translate Entity Sets to Relations

Each atomic attribute of an entity set becomes an *attribute (field) in the relation schema*.

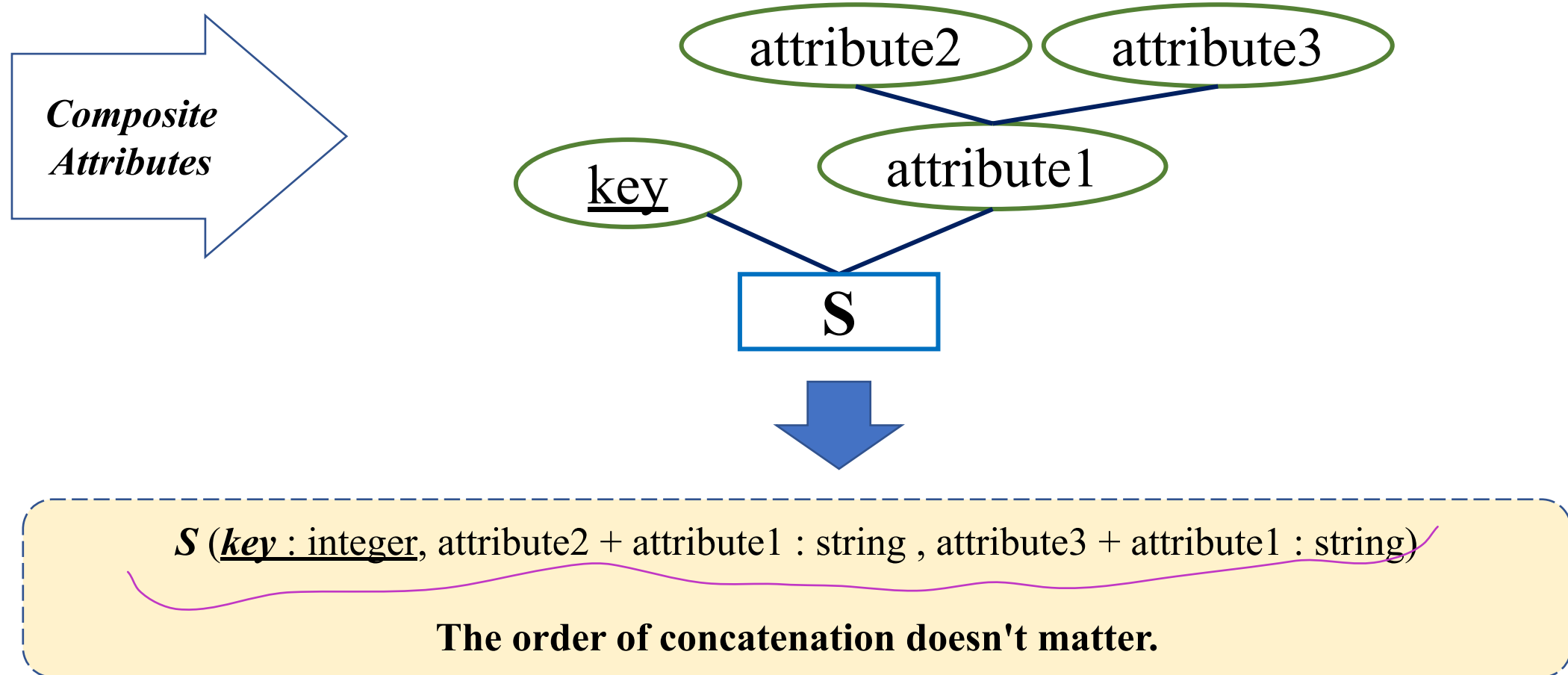


Translate Entity Sets to Relations

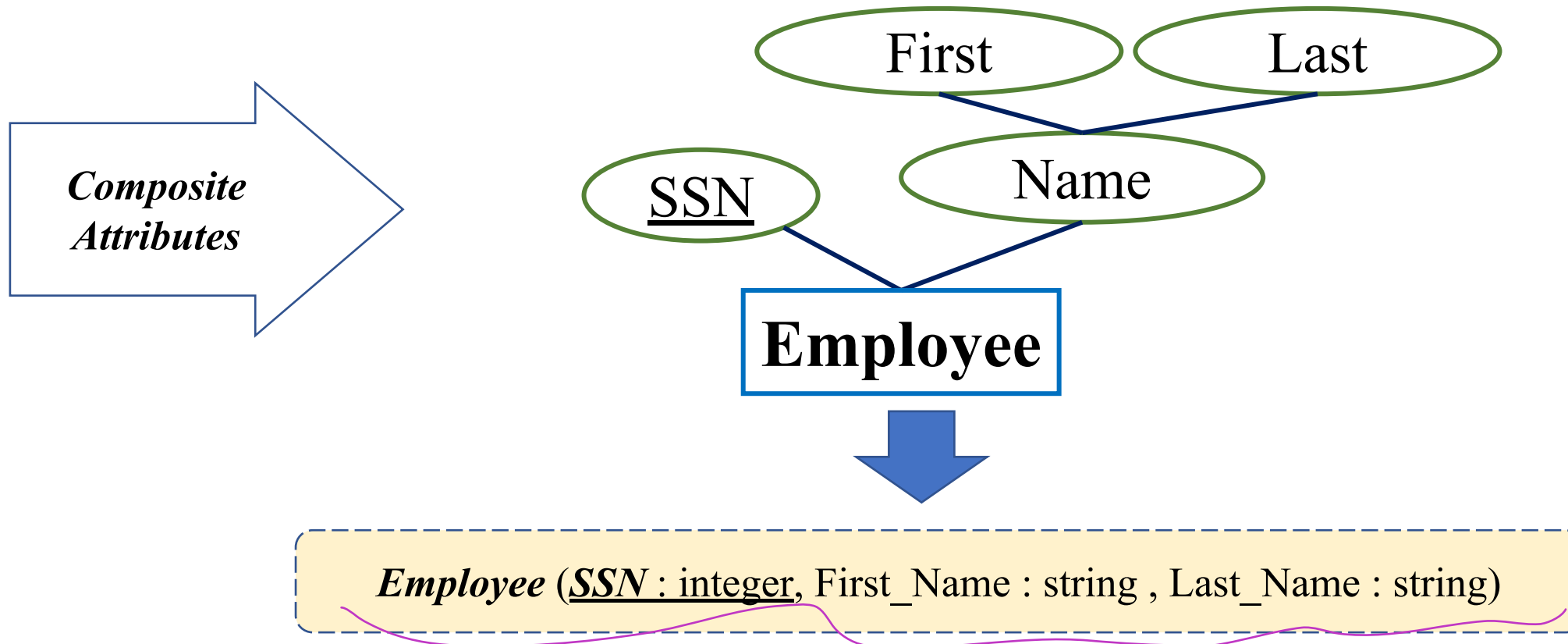


Translate Entity Sets to Relations

Each attribute of a composite attribute along with the root attribute of an entity set becomes an *attribute (field) in the relation schema*.

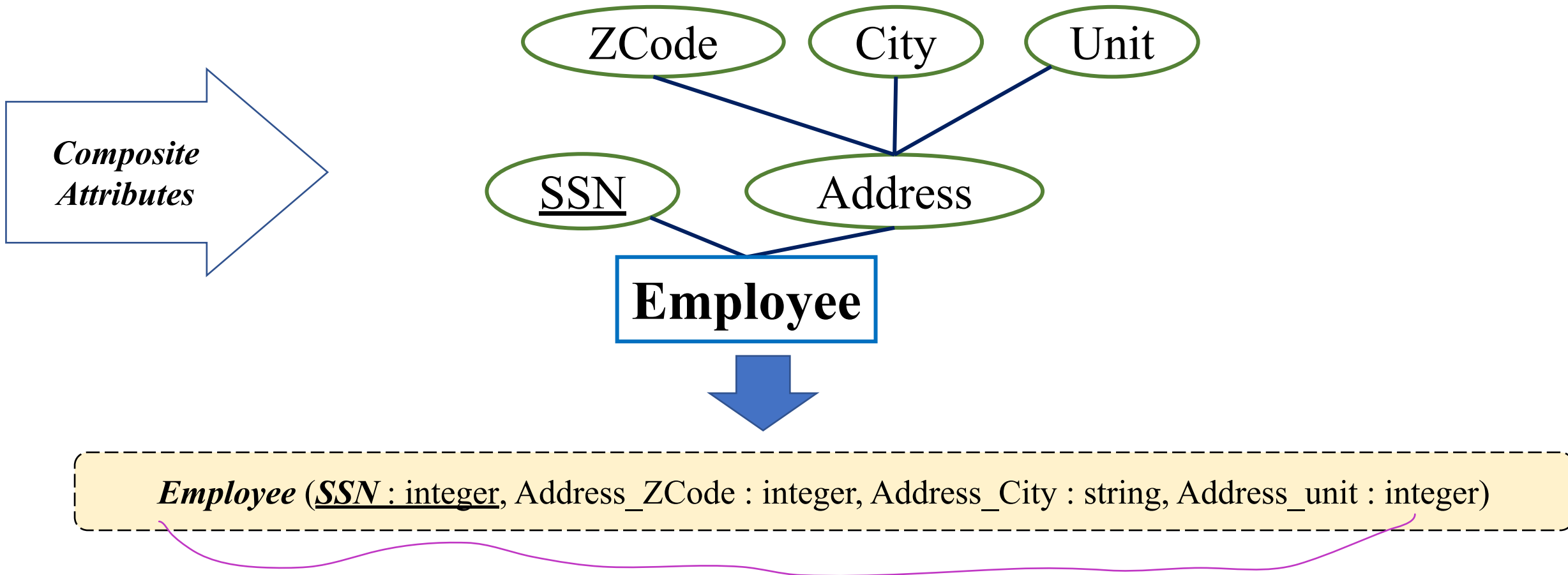


Each attribute of a composite attribute along with the root attribute of an entity set becomes an *attribute (field) in the relation schema*.

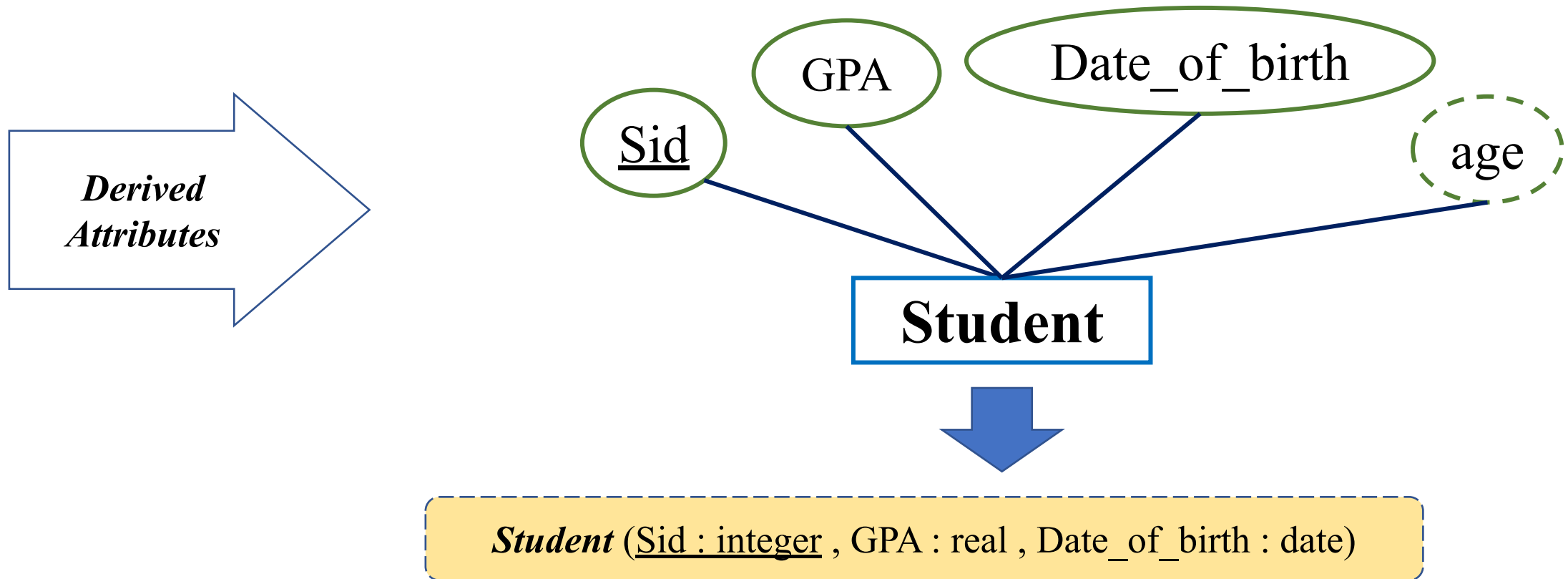


Translate Entity Sets to Relations

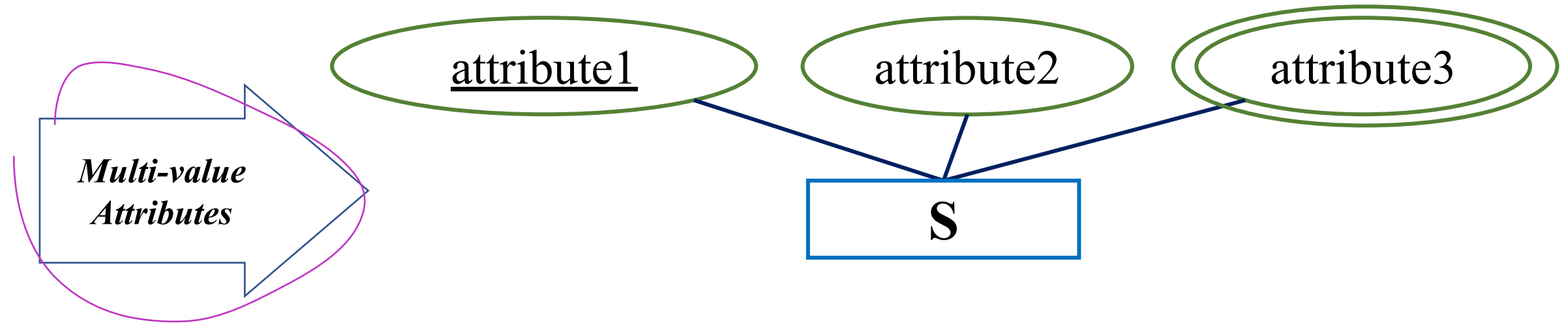
Each attribute of a composite attribute along with the root attribute of an entity set becomes an *attribute (field) in the relation schema*.



The derived attributes are not included in the relational schema - we can include and find their values *by a query language (as will be discussed later)*.



- As we discussed before, each cell in any relation should contain *exactly one atomic (single) value*
- Multi-value attribute may have *more than one value*, and the schema should be of fixed size (pre-defined number of attributes).

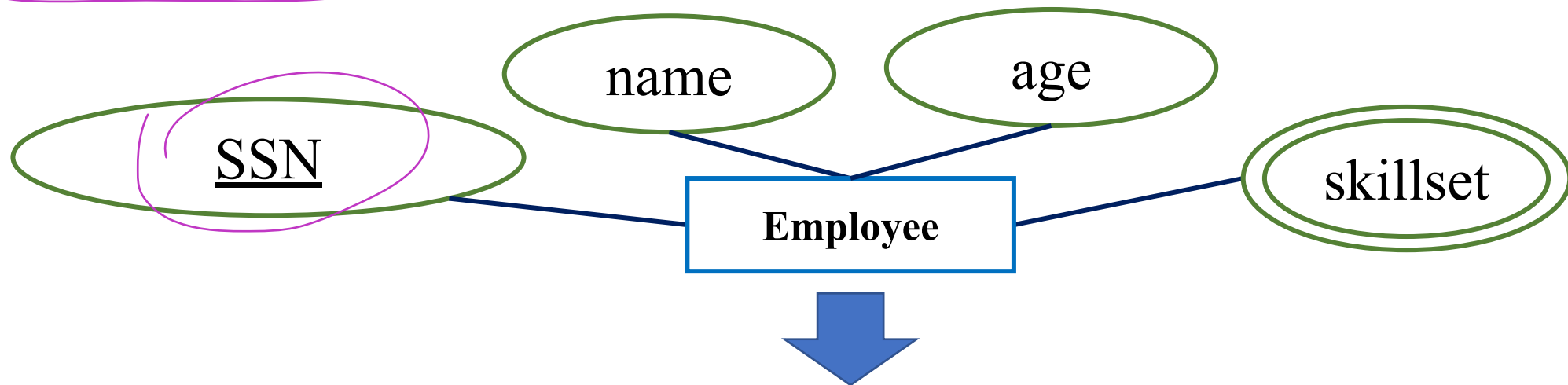


- How can we handle multivalued attributes?

Translate Entity Sets to Relations

To answer this question, let's consider a company with the following Employee entity set:

- Developer1 with skill set of Java, C++, Python
- Developer2 with skill set of Java, C
- Developer3 with skill set of Python, Java



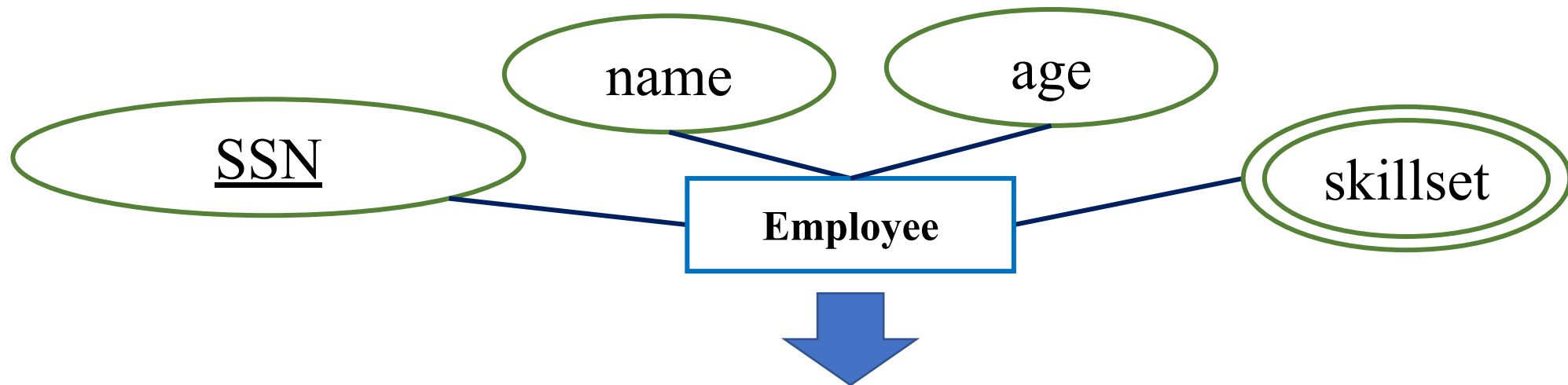
Employee

<u>SSN</u>	name	age	skillset
111	John	25	Java, C++, Python
222	Olivia	25	Java, C
333	Adam	25	Python, Java

Translate Entity Sets to Relations

To answer this question, let's consider a company with the following Employee entity set:

- Developer1 with skill set of Java, C++, Python
- Developer2 with skill set of Java, C
- Developer3 with skill set of Python, Java



Employee

<u>SSN</u>	name	age	skillset
111	John	25	Java, C++, Python
222	Olivia	25	Java, C
333	Adam	25	Python, Java

But each cell should host at most atomic (single) value

So, *we need to divide this table into two*

Translate Entity Sets to Relations

Employee

<u>SSN</u>	name	age	skillset
111	John	25	Java, C++, Python
222	Olivia	25	Java, C
333	Adam	25	Python, Java

Employee

<u>SSN</u>	name	age
111	John	25
222	Olivia	25
333	Adam	25

Skills

skillset
Java
C++
Python
Java
C
Python
Java

For each table (relation), *no duplicate tuples/rows* are allowed (we need a key)

Translate Entity Sets to Relations

Employee

<u>SSN</u>	name	age	skillset
111	John	25	Java, C++, Python
222	Olivia	25	Java, C
333	Adam	25	Python, Java

Employee

<u>SSN</u>	name	age
111	John	25
222	Olivia	25
333	Adam	25

Skills

skillset	SSN
Java	111
C++	111
Python	111
Java	222
C	222
Python	333
Java	333

In this case, we add an identifier from the first table to the second table.

Which of these attributes should be the primary key?

Translate Entity Sets to Relations

Employee

<u>SSN</u>	name	age	skillset
111	John	25	Java, C++, Python
222	Olivia	25	Java, C
333	Adam	25	Python, Java

Employee

<u>SSN</u>	name	age
111	John	25
222	Olivia	25
333	Adam	25

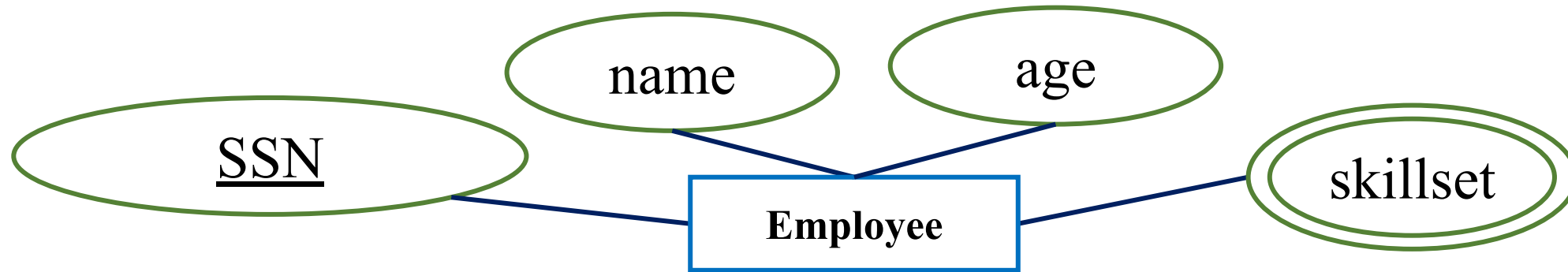
The primary key of Employee table is:
SSN (foreign key)

Skills

<u>skillset</u>	<u>SSN</u>
Java	111
C++	111
Python	111
Java	222
C	222
Python	333
Java	333

SSN in Skills table now works as an identifier - we refer to as *foreign key*.

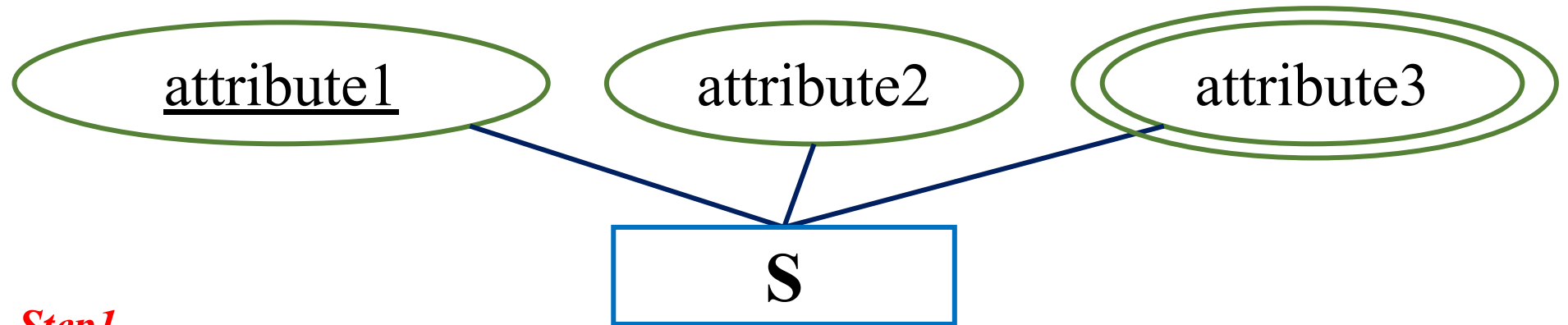
The primary key of Skills table is:
Skillset + SSN (foreign key)



Employee (SSN : integer, name : string, age : integer)

Skills (skillset : string, SSN : integer)

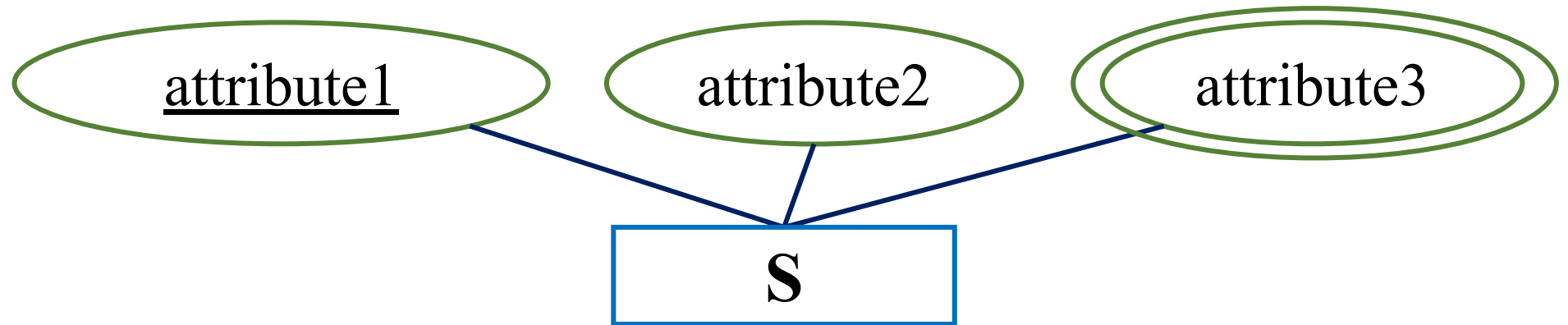
The last step is to add *an arrow from the foreign key to the primary key* it is taken from.



Step1

Place all single-value attributes (either atomic or composite) in the main relation

S (attribute1 , attribute2)



$S (\underline{\text{attribute1}} , \text{attribute2})$

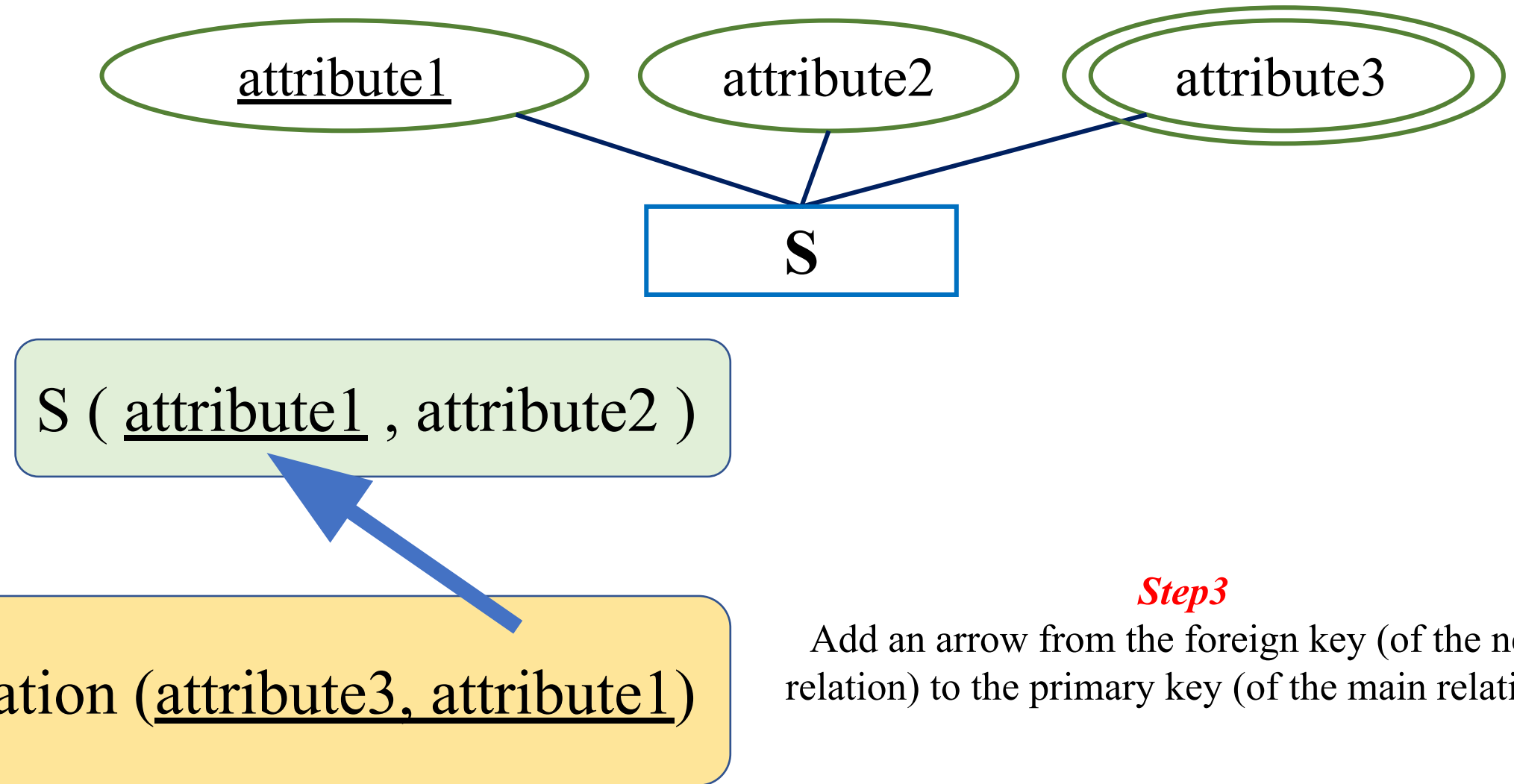
NewRelation (attribute3, attribute1)

Step2

For each multi-value attribute, create a new relation

New relation (you can name it) has a ***composite primary key***:

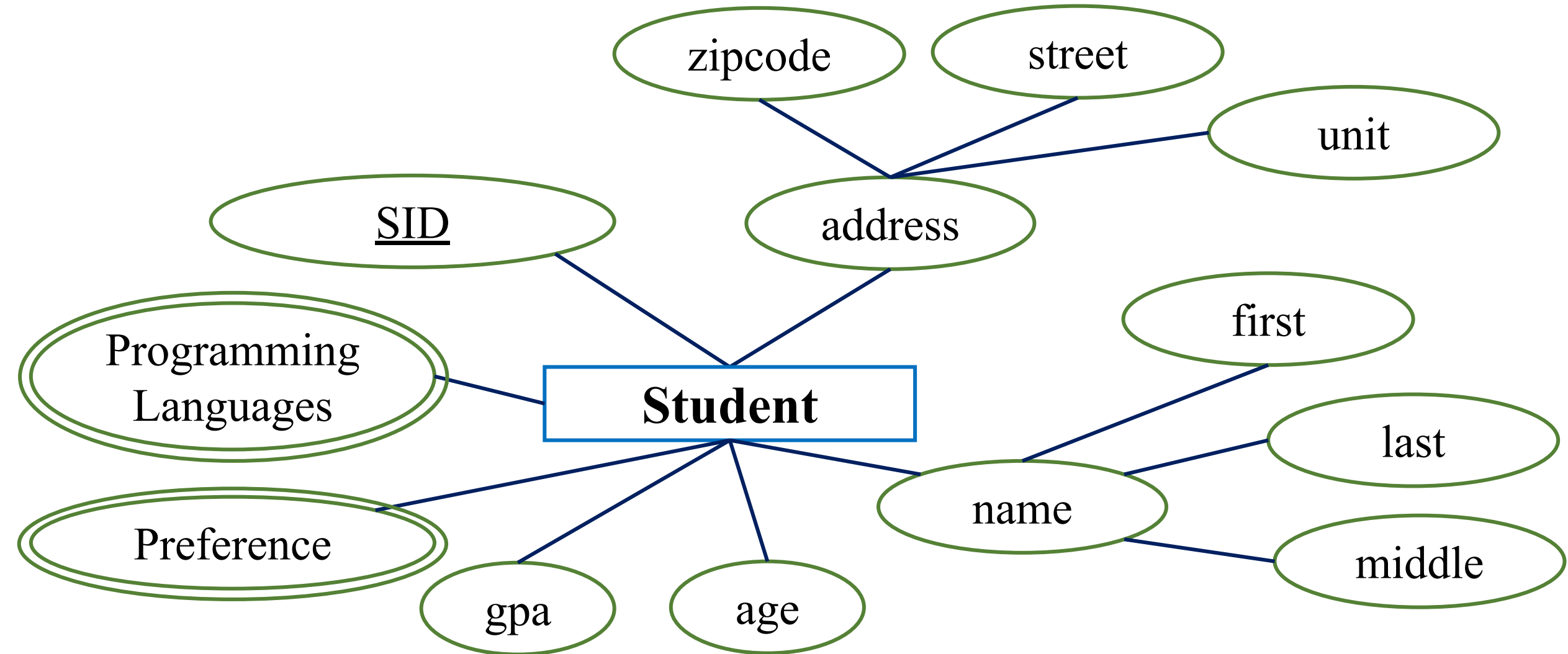
the multi-valued attribute ***as partial key***
+
the primary key of the main table ***as foreign key***

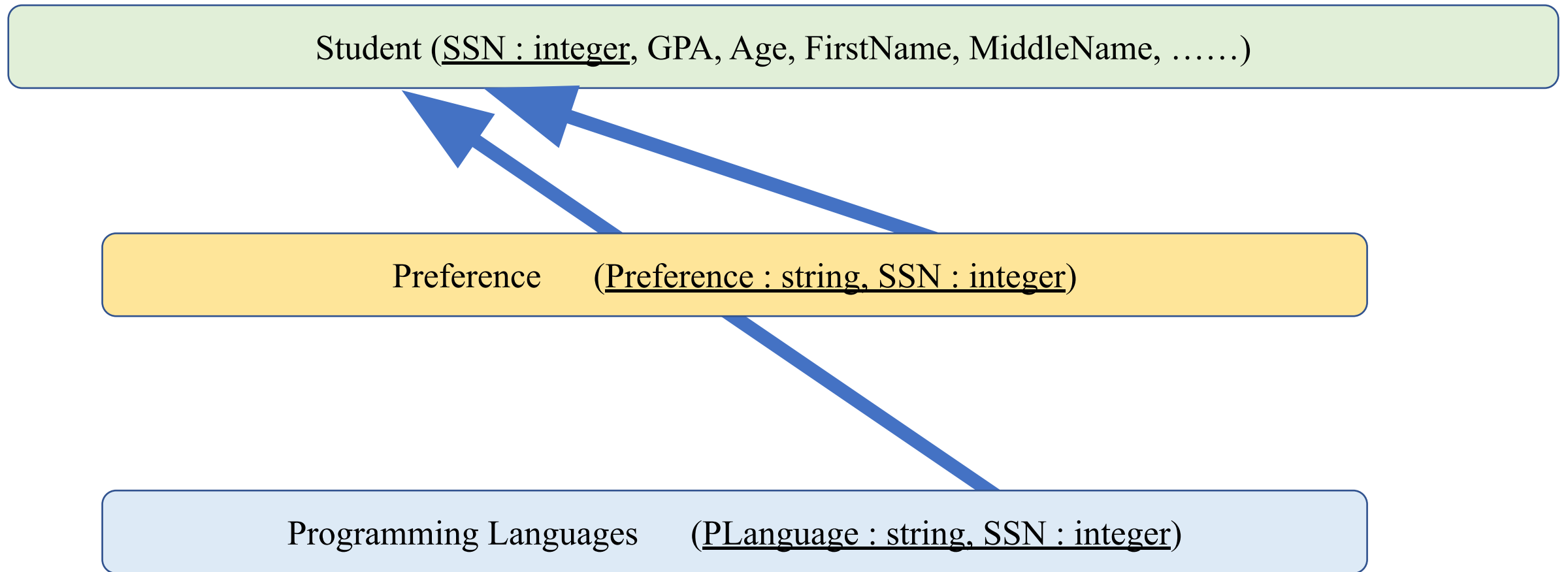


Step3

Add an arrow from the foreign key (of the new relation) to the primary key (of the main relation).

- Build *a relational schema* from the following ER diagram:



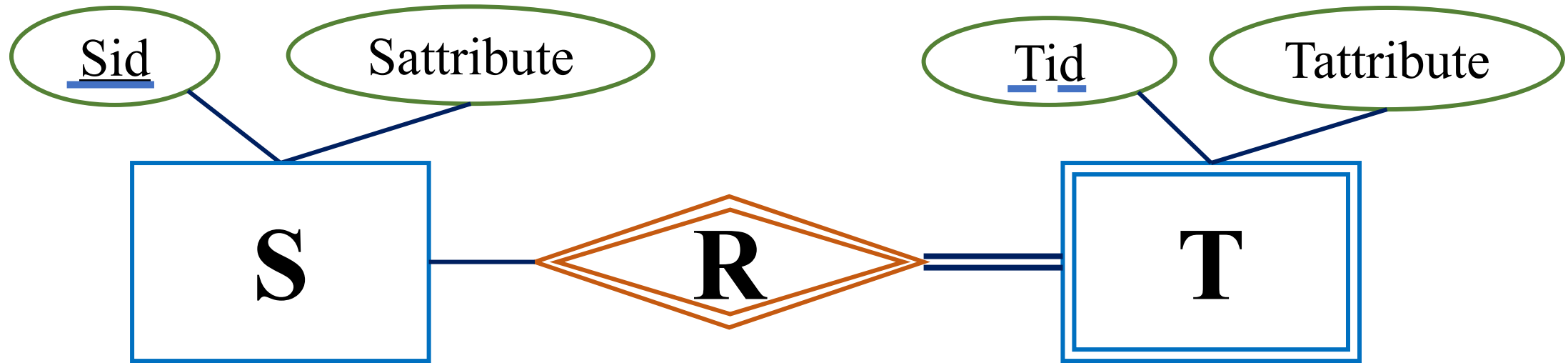


From ER diagram to relations:-

2nd step - translate Weak Entity Sets to Relations

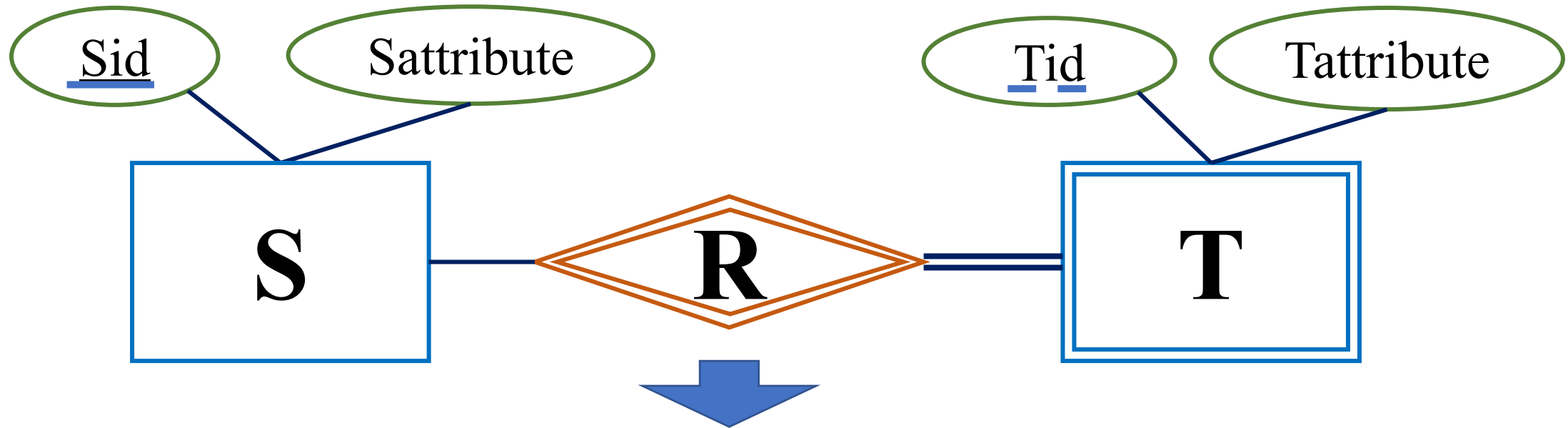
Translate Weak Entity Sets to Relations

Consider the following ER diagram with S (strong entity set), T (weak entity set), and R (identifying relationship)



Translate Weak Entity Sets to Relations

Consider the following ER diagram with S (strong entity set), T (weak entity set), and R (identifying relationship)

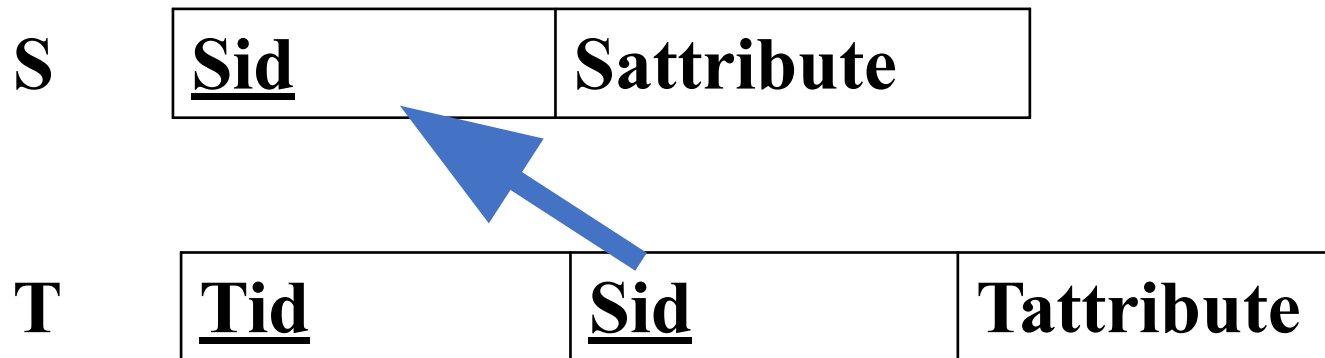
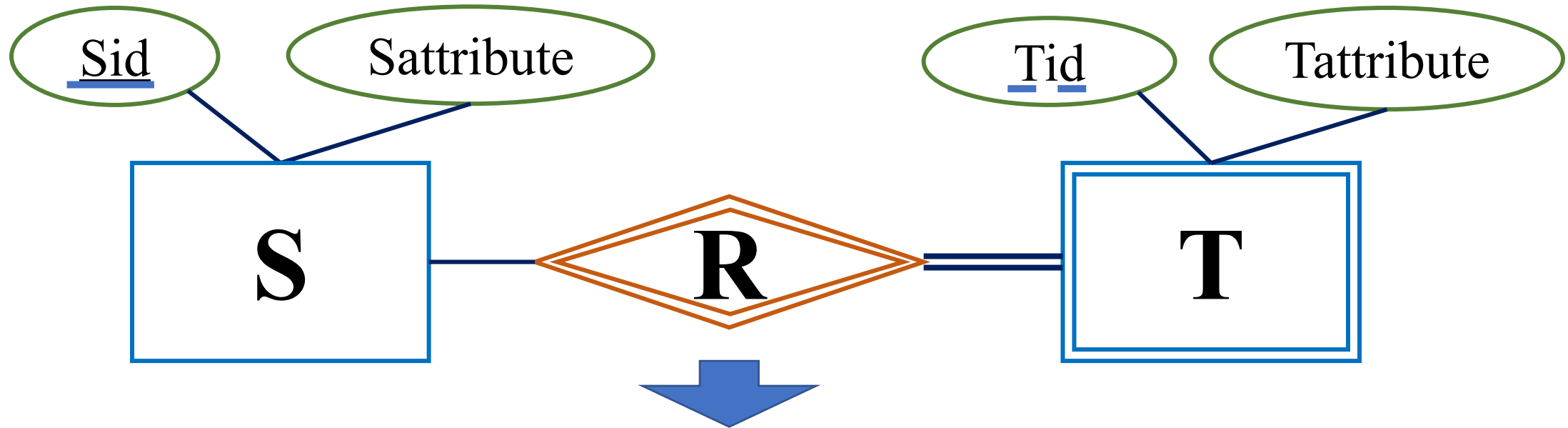


Tid + Sid = the *Composite Primary key* of T

- Tid is the *Partial* key of T
- Sid is a *Foreign* key from S

Translate Weak Entity Sets to Relations

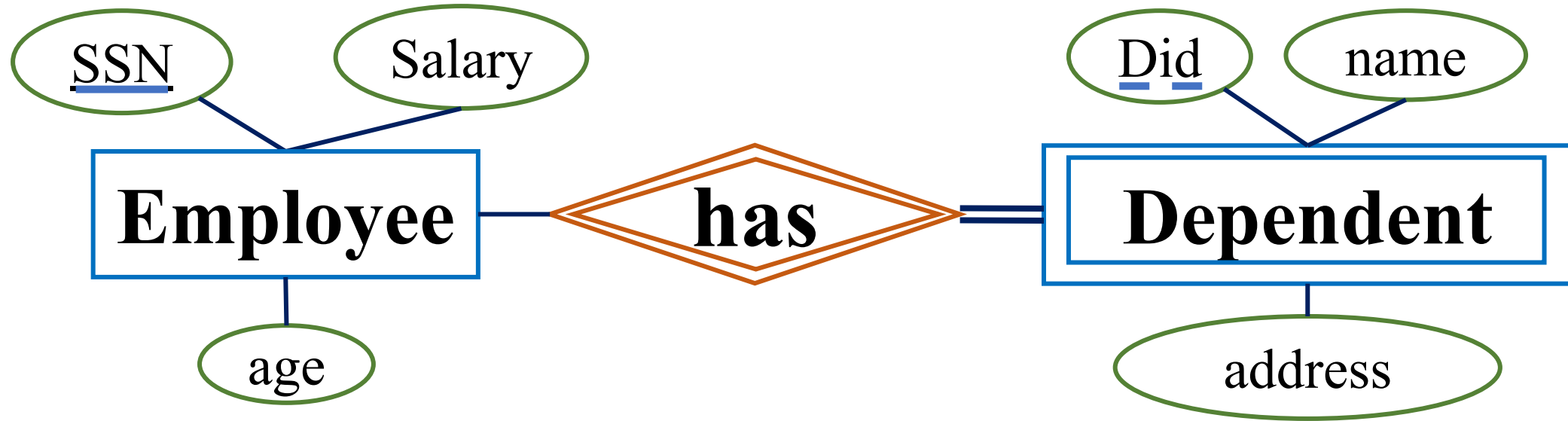
Consider the following ER diagram with S (strong entity set), T (weak entity set), and R (identifying relationship)



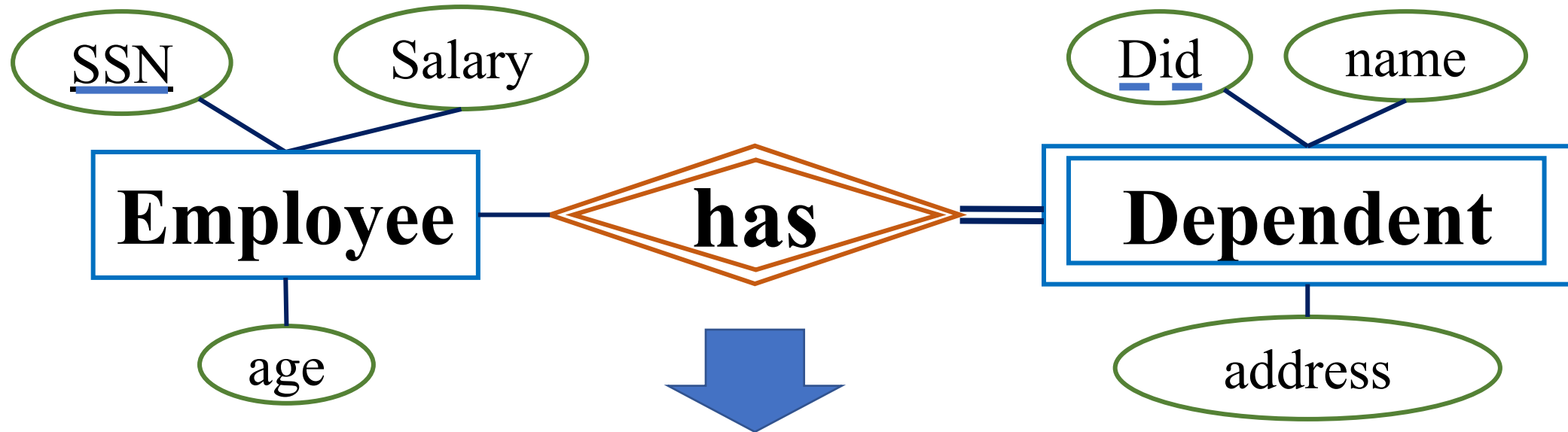
Tid + Sid = the *Composite Primary key* of T

- Tid is the *Partial* key of T
- Sid is a *Foreign* key from S

Imagine a company database that stores information about the employees and their dependents, build relational schema to capture that requirement from the following ER diagram:



Imagine a company database that stores information about the employees and their dependents, build relational schema to capture that requirement from the following ER diagram:

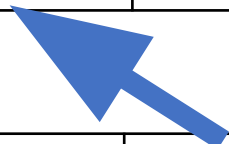


Employee

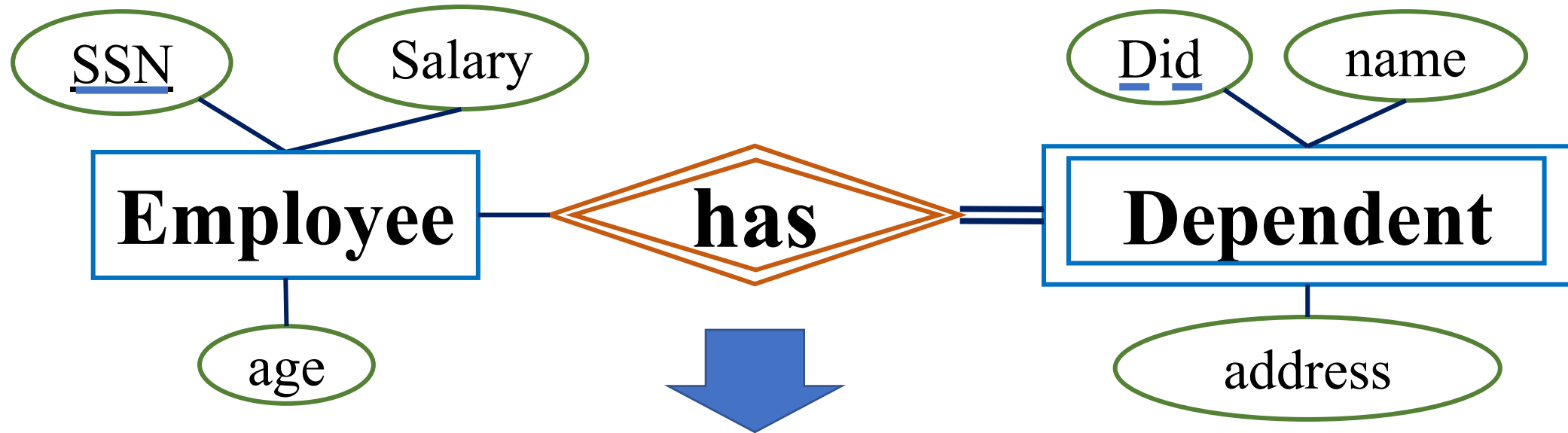
<u>SSN</u>	Salary	age
------------	--------	-----

Dependent

<u>Did</u>	<u>EmployeeSSN</u>	name	address
------------	--------------------	------	---------



Imagine a company database that stores information about the employees and their dependents, build relational schema to capture that requirement from the following ER diagram:

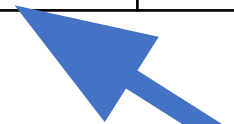


Employee

<u>SSN</u>	Salary	age
------------	--------	-----

Dependent

<u>Did</u>	<u>EmployeeSSN</u>	name	address
------------	--------------------	------	---------



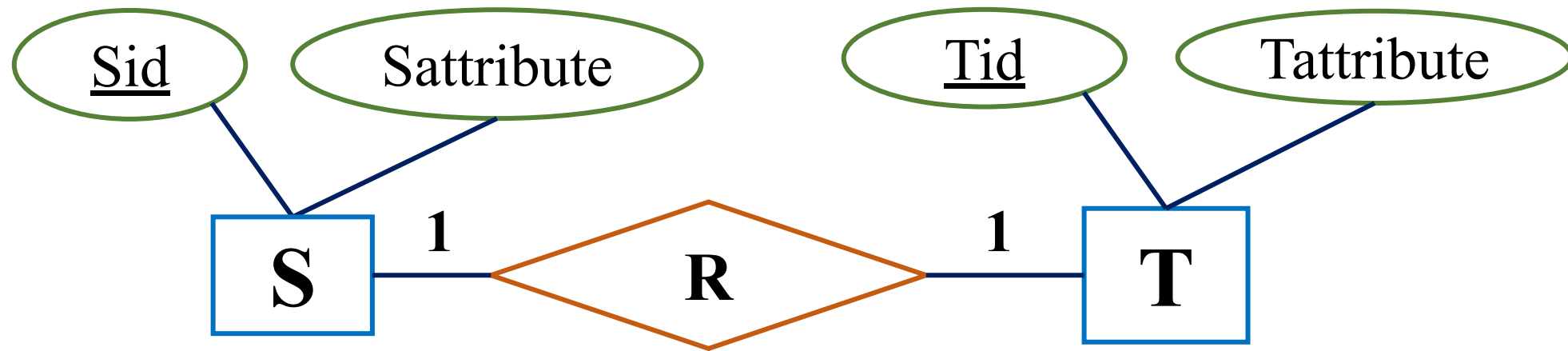
- It is up to the you to chose names when you build relational schemas and ER diagrams.
- The names in the schemas *may not match* that of the ER diagram - that is why it is *mandatory to draw the arrows*.

From ER diagram to relations:-

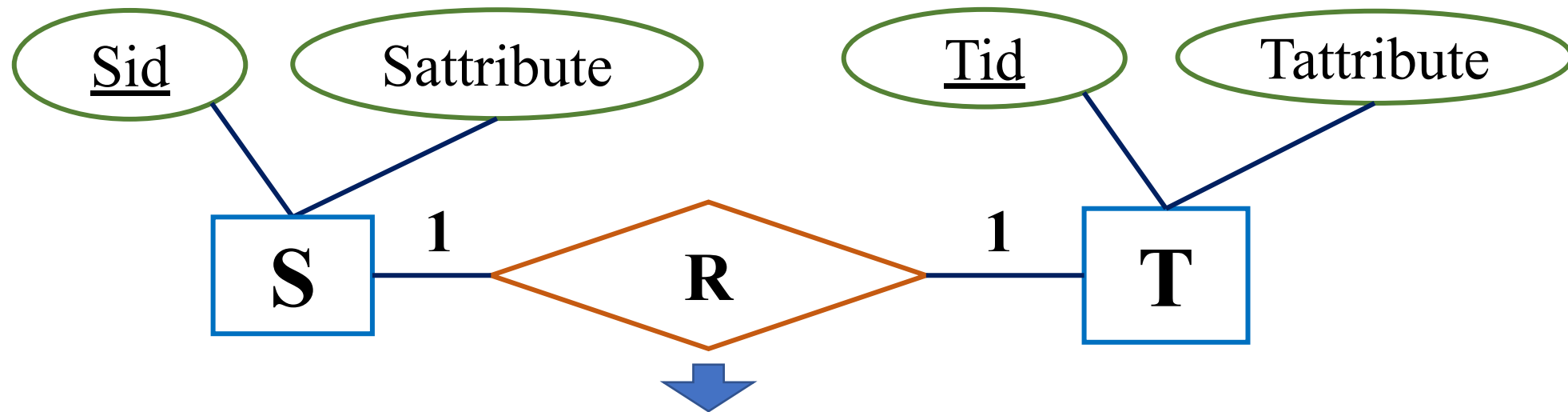
3rd step - translate Relationship Sets to Relations

Translating 1:1 relationship sets

Consider the following binary 1:1 relationship set R connecting two entity sets S and T



Consider the following binary 1:1 relationship set R connecting two strong entity sets S and T

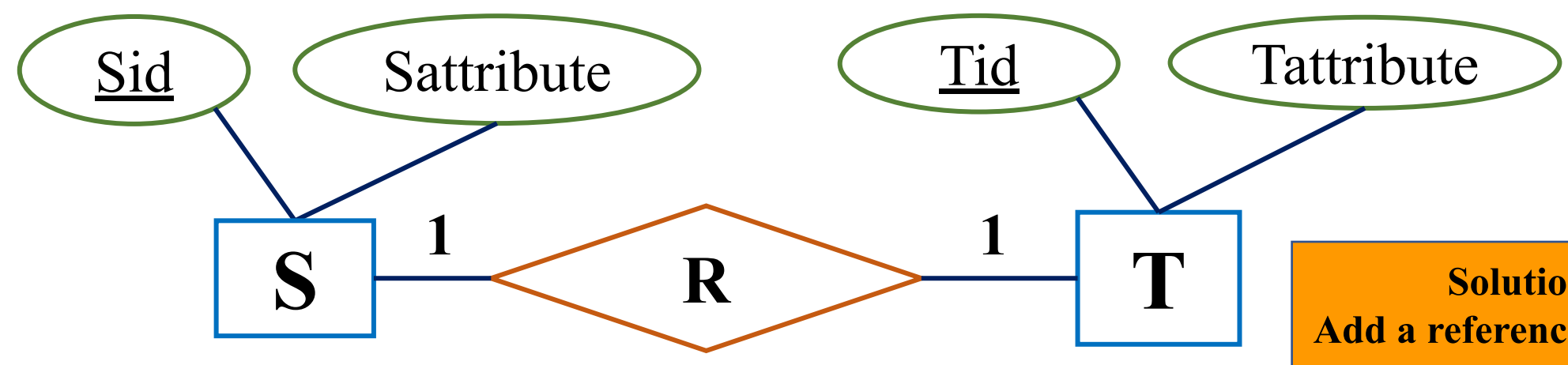


S	<u>Sid</u>	Sattribute

T	<u>Tid</u>	Tattribute

- Sid is the primary key of S
- Tid is the primary key of T
- How can we represent the relation R?

Consider the following binary 1:1 relationship set R connecting two strong entity sets S and T



Solution1:
Add a reference for S at T

S

<u>Sid</u>	Sattribute
------------	------------

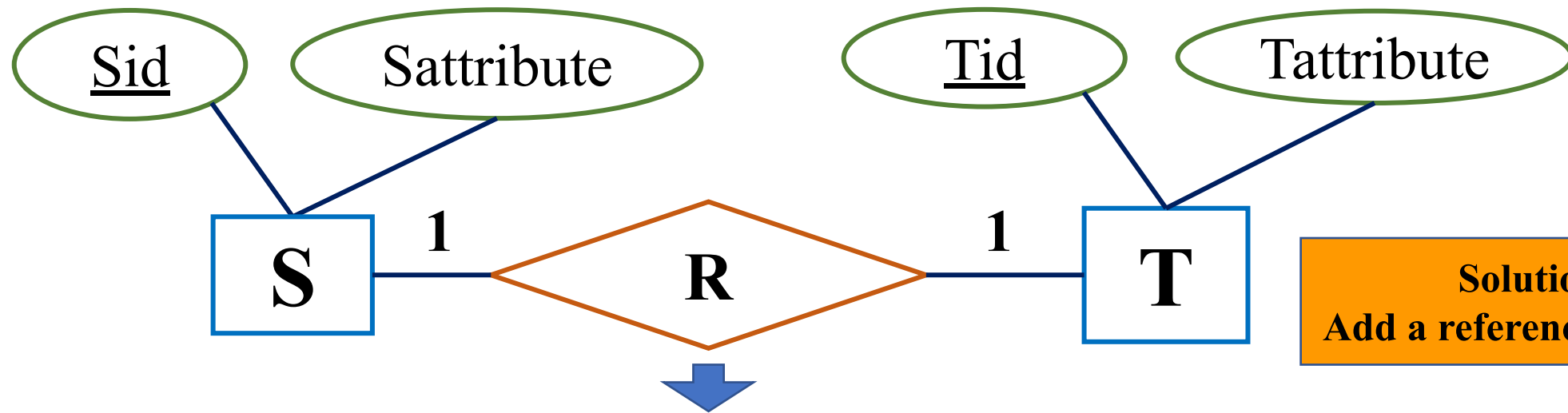
T

<u>Tid</u>	Tattribute	Sid
------------	------------	-----

Since T is a strong entity set - we have:

- Tid is the primary key of T
- Sid is a foreign key of S *but not part of the primary key*

Consider the following binary 1:1 relationship set R connecting two strong entity sets S and T



Solution2:
Add a reference for T at S

OR

S	<u>Sid</u>	Sattribute	Tid
---	------------	------------	-----

T	<u>Tid</u>	Tattribute
---	------------	------------

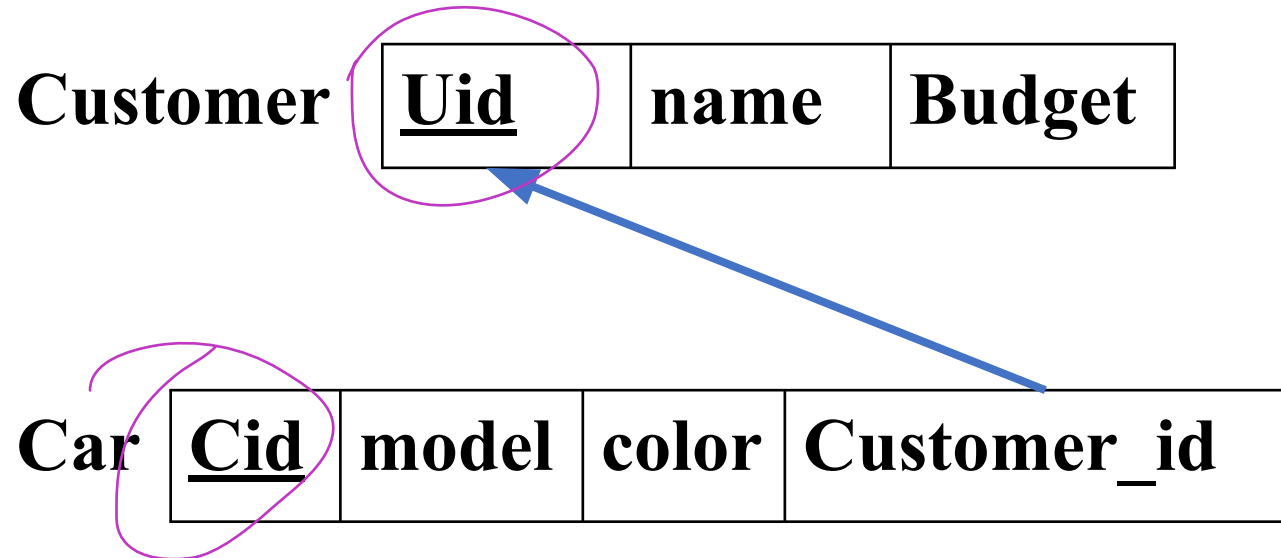
Since S is a strong entity set - we have:

- Sid is the primary key of S
- Tid is a foreign key of T *but not part of the primary key*

Translate the following ER diagram into relational schema(s):



Translate the following ER diagram into relational schema(s):



- As mentioned before, It is up to the you to chose names when you build relational schemas and ER diagrams.
- The names in the schemas *may not match* that of the ER diagram - that is why it is *mandatory to draw the arrows*.

Translate the following ER diagram into relational schema(s):



Customer

<u>Uid</u>	name	Budget
U1	John	5000
U2	Olivia	5000
U3	Adam	5000

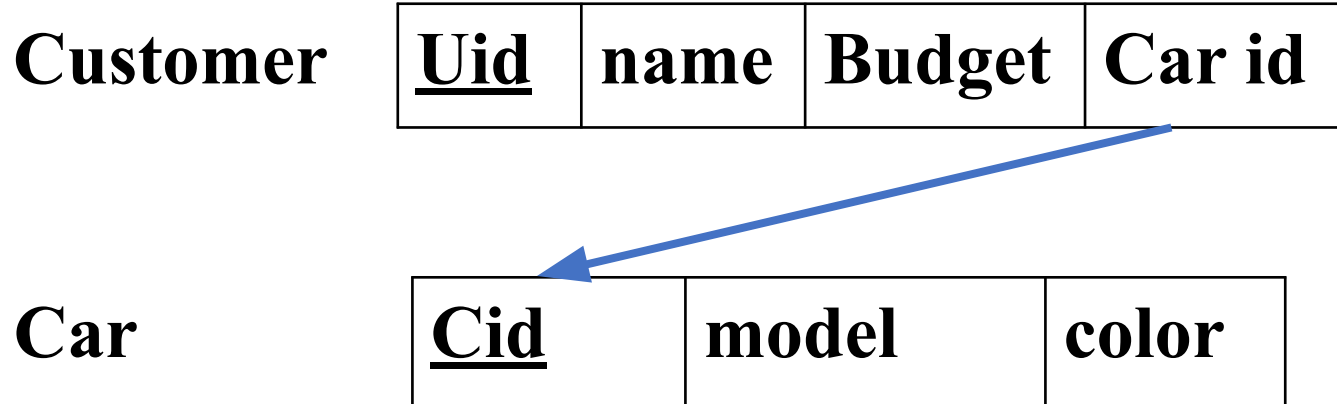
Car

<u>Cid</u>	model	color	Customer id
C1	Toyota	Blue	U2
C2	Honda	Gray	U1
C3	Merc	Black	NULL
C4	BMW	Red	U3
C5	Toyota	Blue	NULL

Translate the following ER diagram into relational schema(s):



OR



Translate the following ER diagram into relational schema(s):



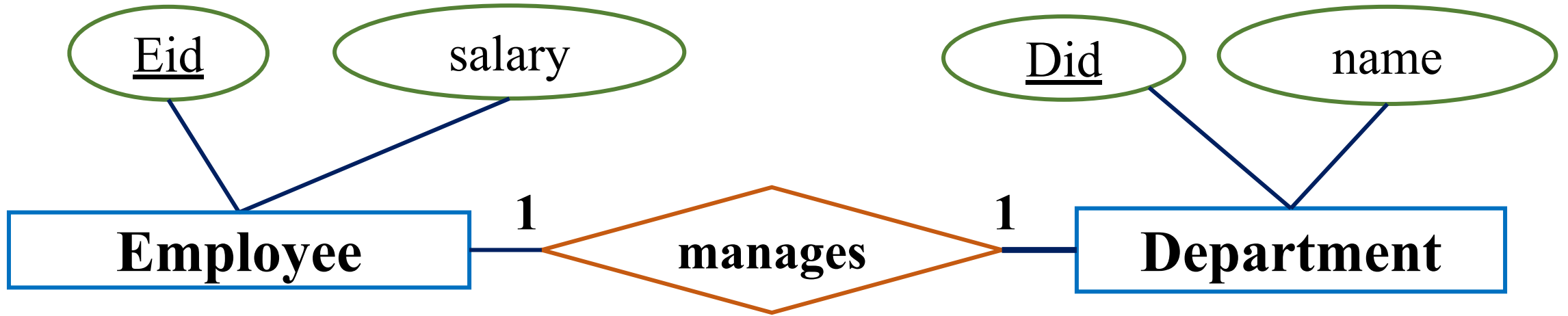
Customer

<u>Uid</u>	name	Budget	Car ID
U1	John	5000	C1
U2	Olivia	5000	NULL
U3	Adam	5000	C5

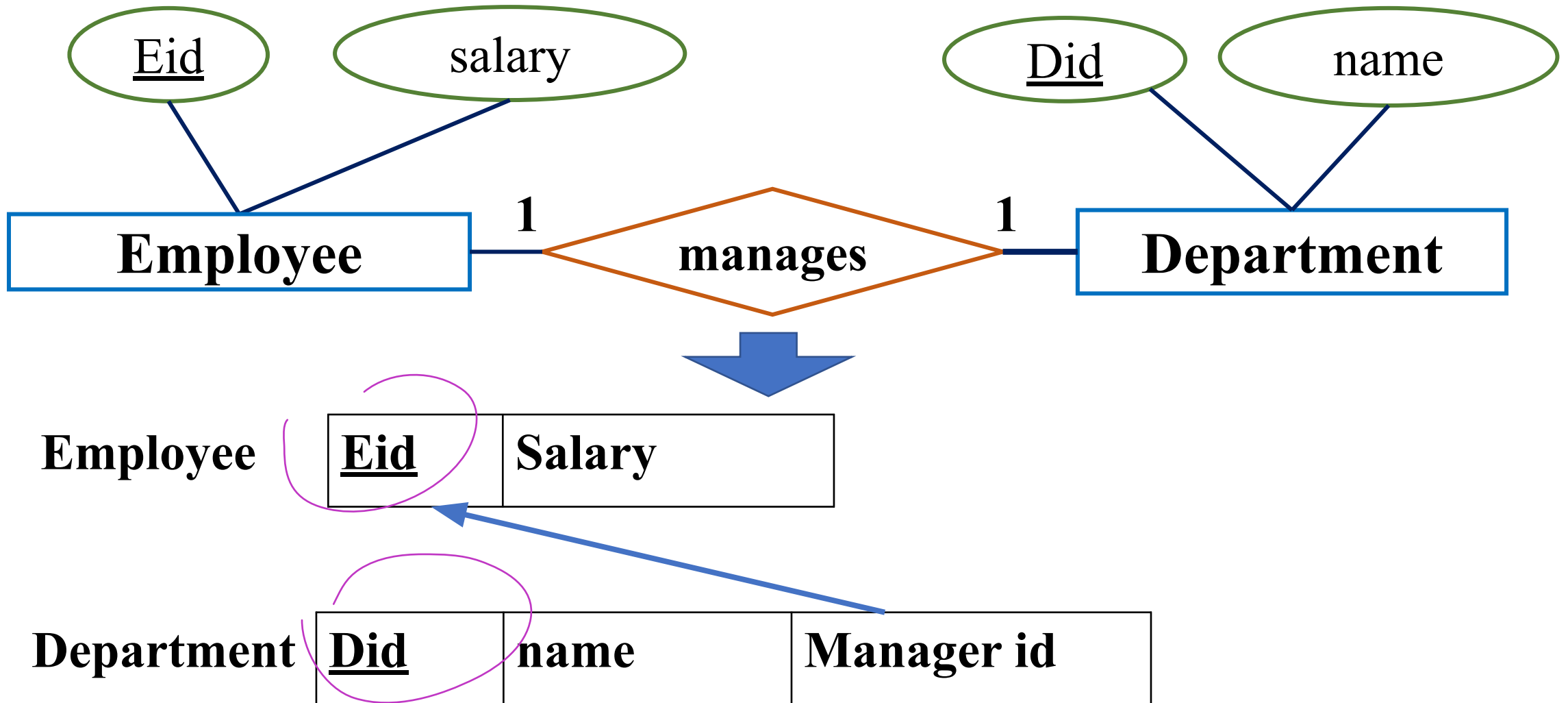
Car

<u>Cid</u>	model	color
C1	Toyota	Blue
C2	Honda	Gray
C3	Merc	Black
C4	BMW	Red
C5	Toyota	Blue

Translate the following ER diagram into relational schema(s):



Translate the following ER diagram into relational schema(s):



Translate the following ER diagram into relational schema(s):



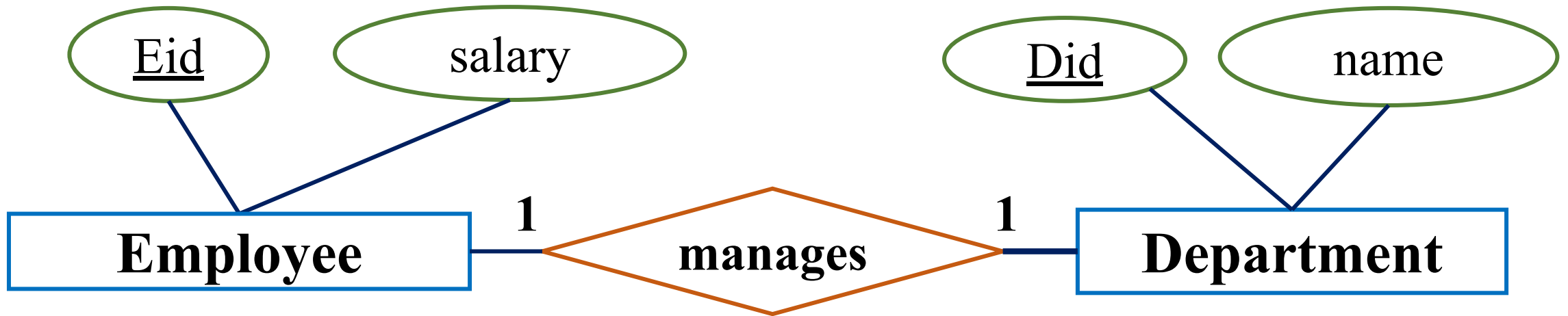
Employee

<u>Eid</u>	salary
E1	5000
E2	6000
E3	7000
E4	5000
E5	6000
E6	5500
E7	6700

Dep

<u>Did</u>	Name	Manager ID
D1	Sales	E1
D2	Ads	E4
D3	Tech	E7

Translate the following ER diagram into relational schema(s):



As *not every* employee is a manager, and for each department there *should be a manager* – so it is logical to add the reference of employee in the department, otherwise this will end up with a lot of *NULL entries*.



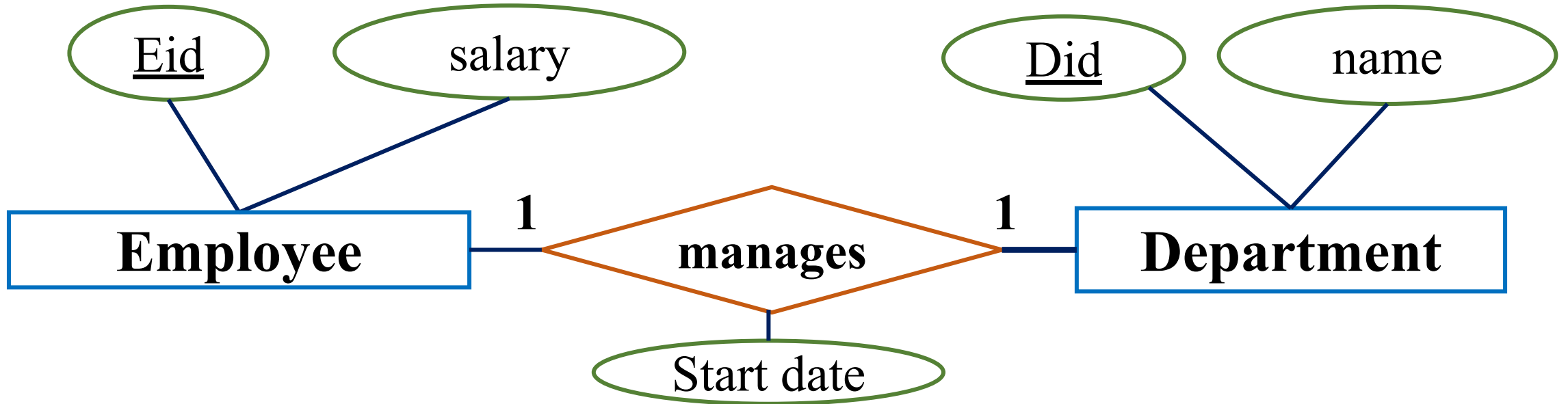
Employee

<u>Eid</u>	salary	Dep ID
E1	5000	D1
E2	6000	NULL
E3	7000	NULL
E4	5000	D2
E5	6000	NULL
E6	5500	NULL
E7	6700	D3

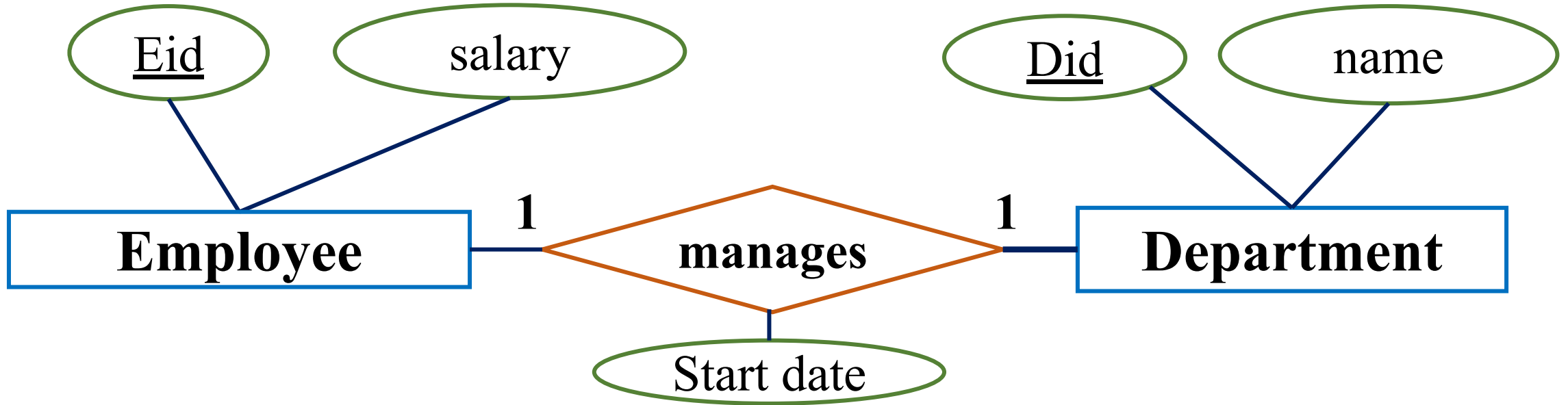
Dep

<u>Did</u>	Name
D1	Sales
D2	Ads
D3	Tech

Translate the following ER diagram into relational schema(s):



Translate the following ER diagram into relational schema(s):

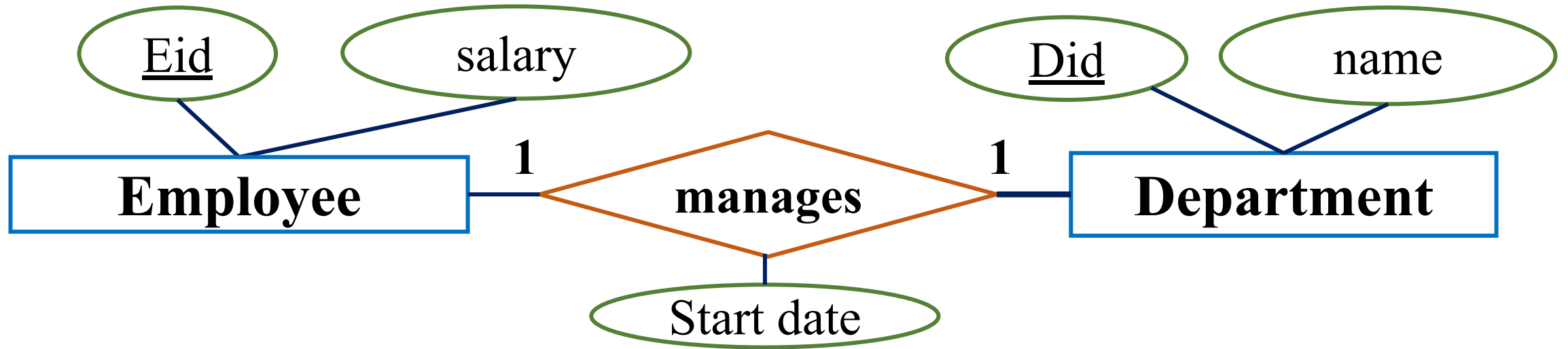


Employee

<u>Eid</u>	Salary
-------------------	---------------

Department

<u>Did</u>	name	Manager id	Start date
-------------------	-------------	-------------------	-------------------



Employee

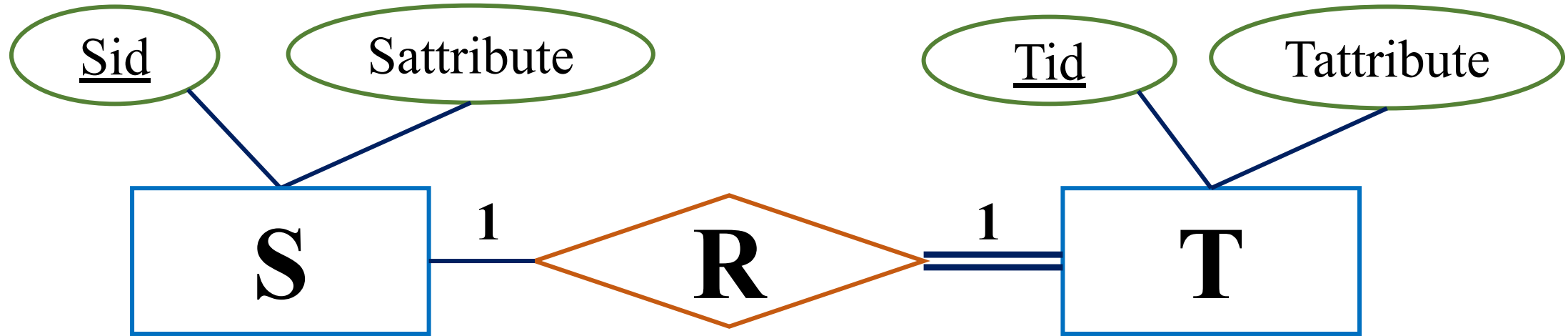
<u>Eid</u>	salary
E1	5000
E2	6000
E3	7000
E4	5000
E5	6000
E6	5500
E7	6700

Dep

<u>Did</u>	Name	Manager ID	Start Date
D1	Sales	E1	Jan20
D2	Ads	E4	Feb20
D3	Tech	E7	March20

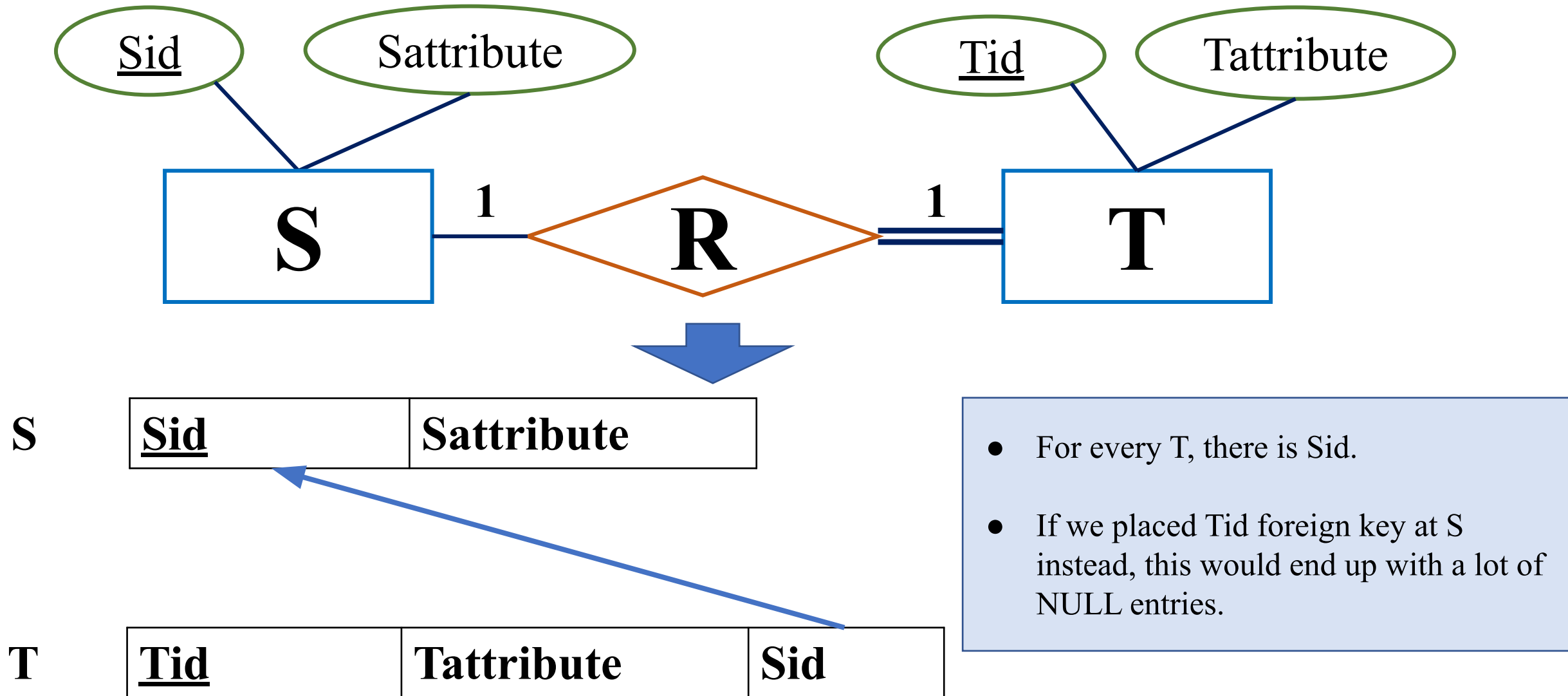
Translating 1:1 relationship sets

Consider the following binary 1:1 relationship set R connecting two strong entity sets S and T, with total participation:

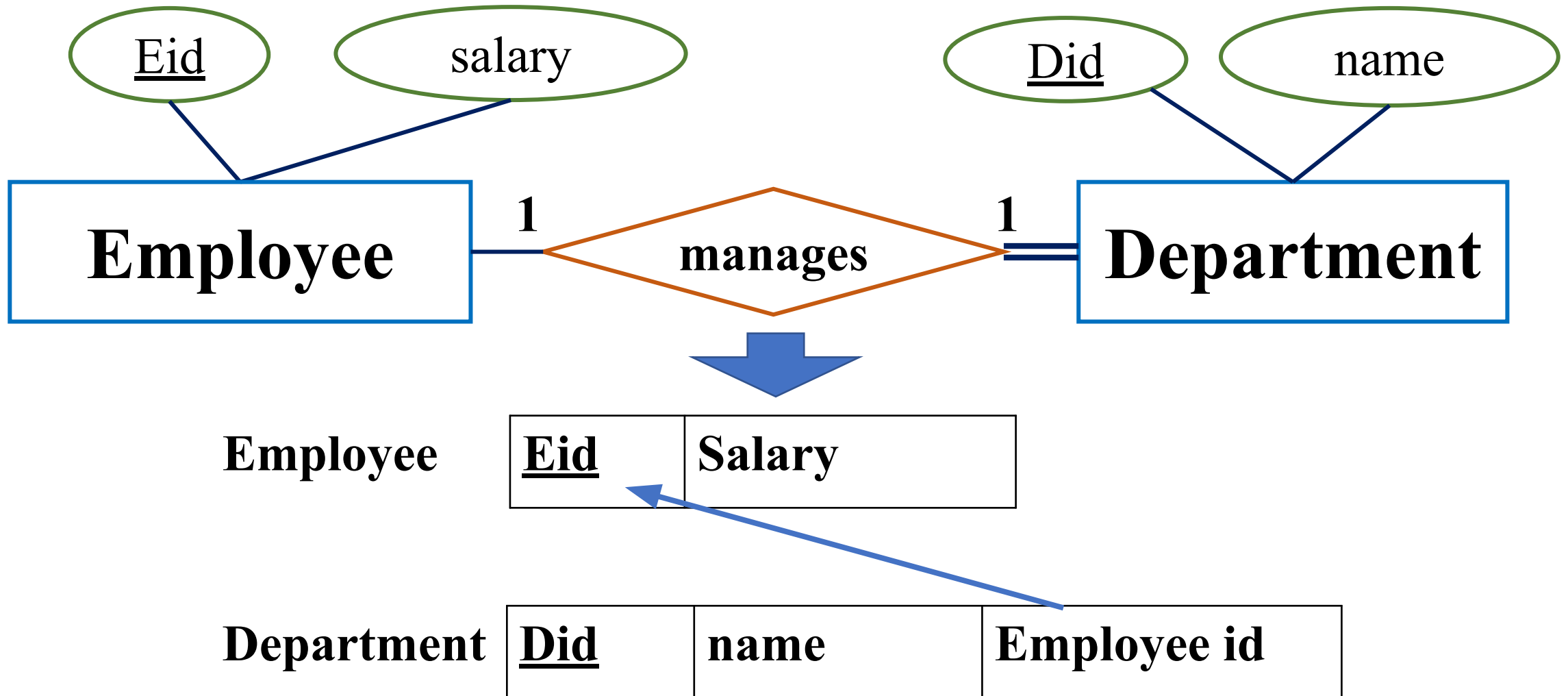


Translating 1:1 relationship sets

Consider the following binary 1:1 relationship set R connecting two strong entity sets S and T, with total participation:

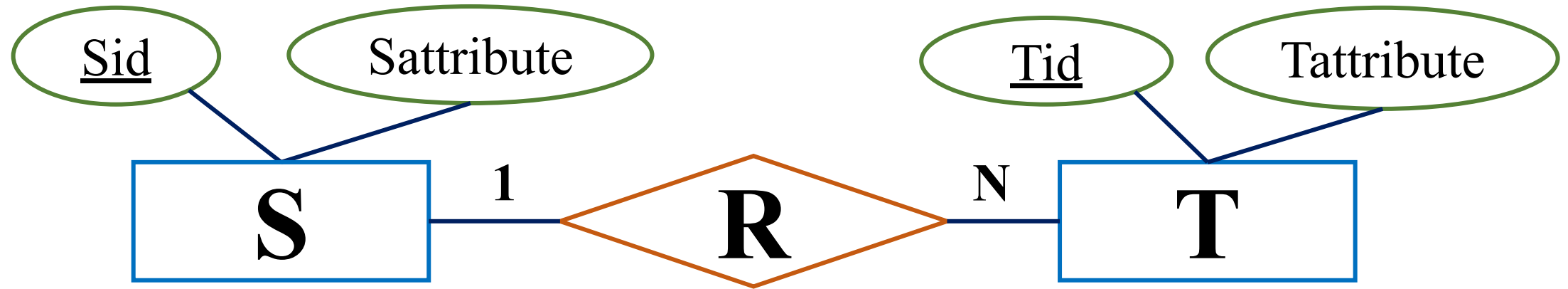


Translate the following ER diagram into relational schema(s):

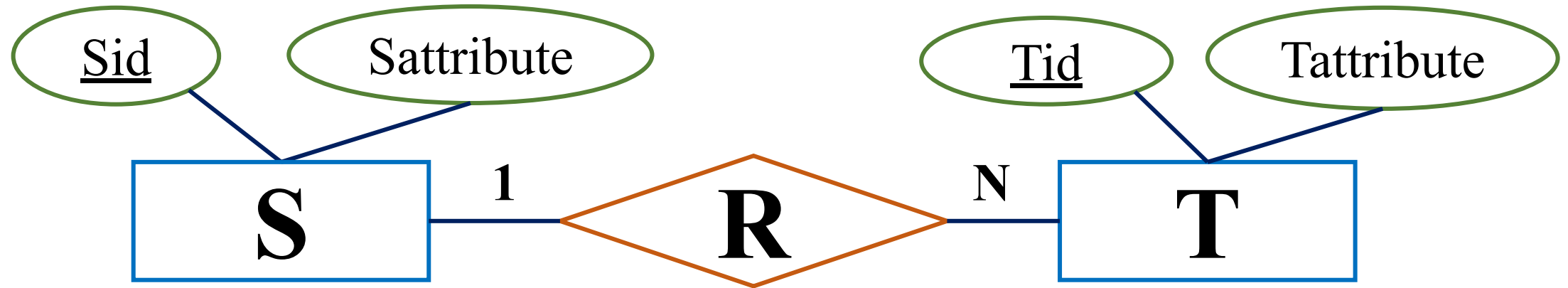


Translating 1:N relationship sets

Consider the following binary 1:N relationship set R connecting two strong entity sets S and T



Consider the following binary 1:N relationship set R connecting two strong entity sets S and T

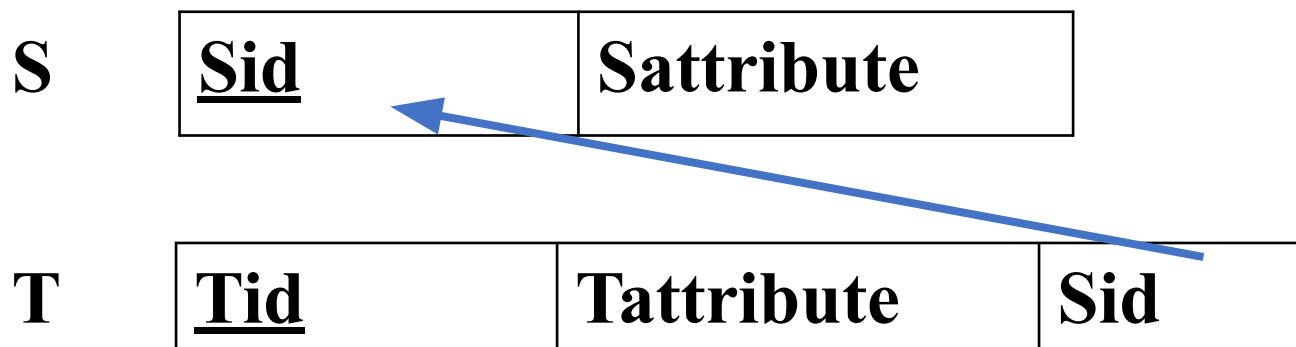
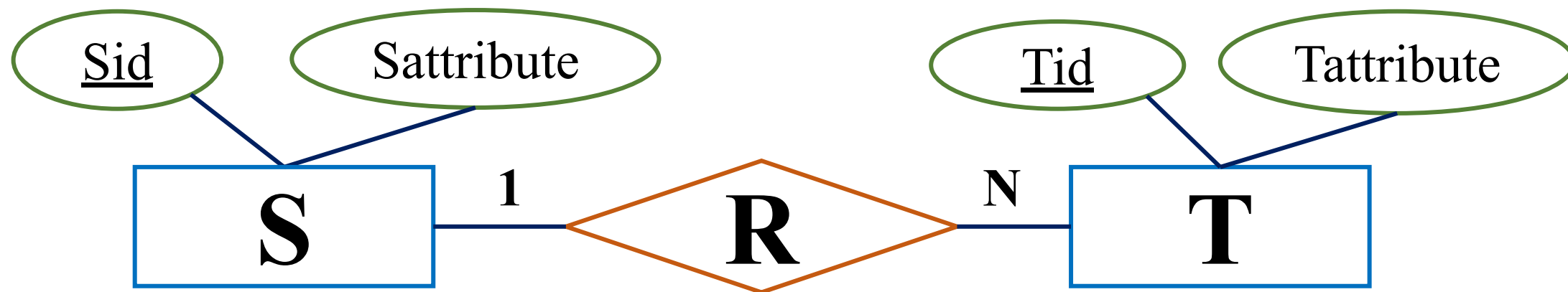


S	<u>Sid</u>	Sattribute
T	<u>Tid</u>	Tattribute

If More than one T may be linked to the same S,
how can we represent such relationship R?

**Very close to the way we handled multivalued
attributes**

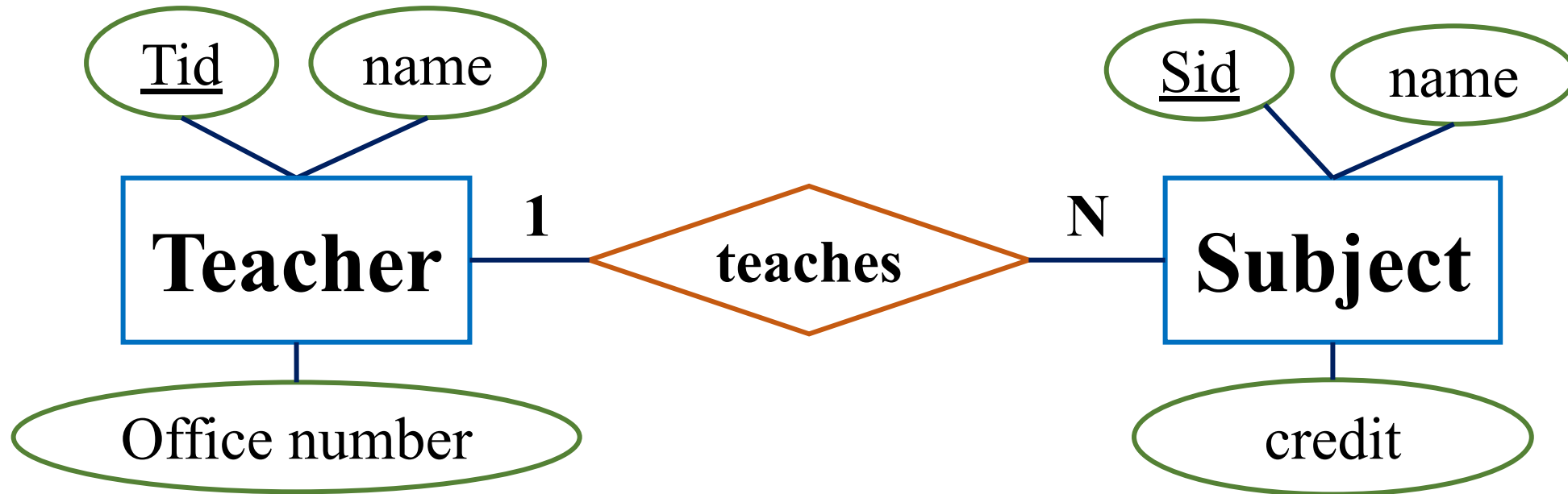
Consider the following binary 1:N relationship set R connecting two strong entity sets S and T



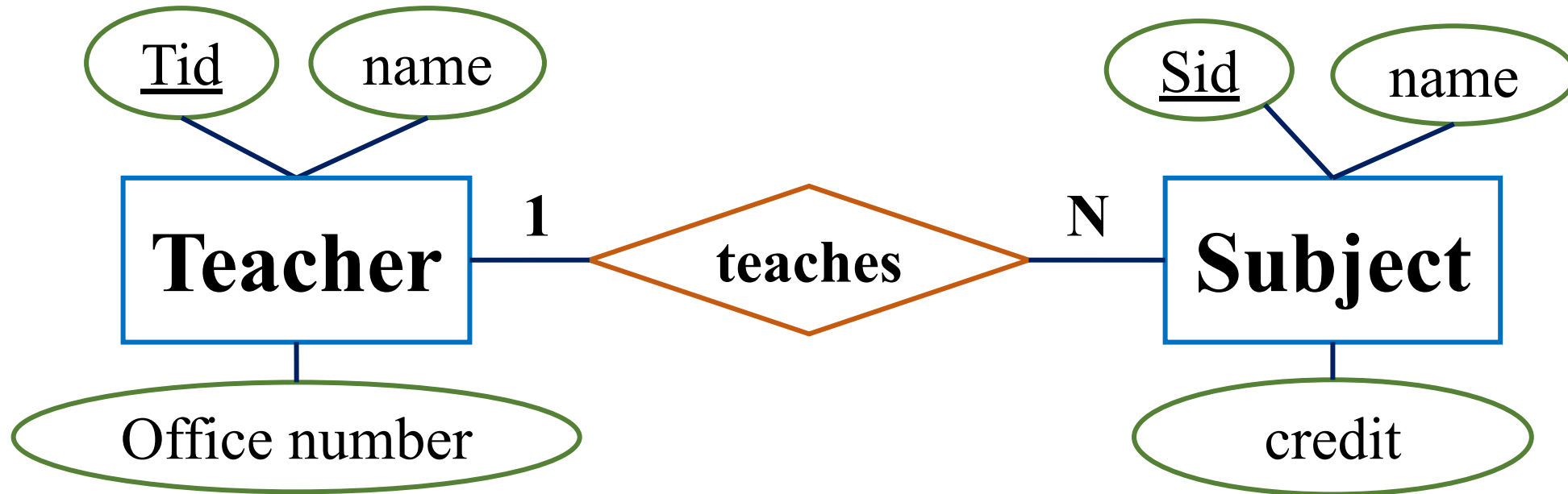
Since T is a strong entity set - we have:

- Tid is the primary key of T
- Sid is a foreign key of S *but not part of the primary key*

Translate the following ER diagram into relational schema(s):

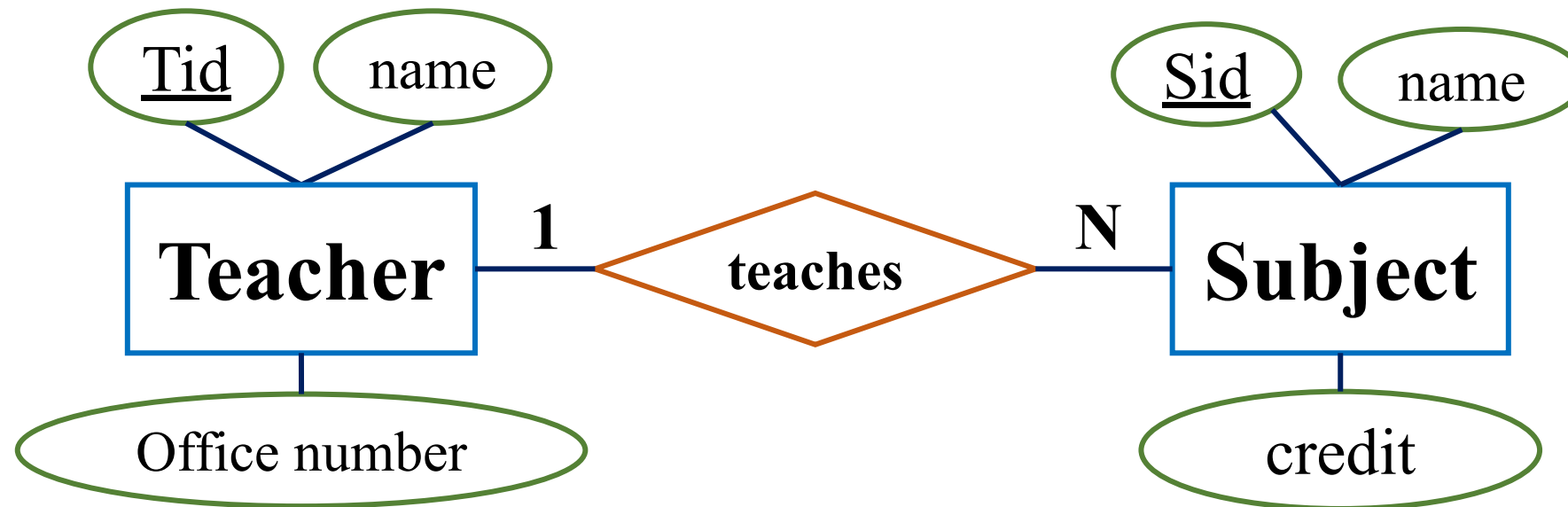


Translate the following ER diagram into relational schema(s):



Teacher	<u>Tid</u>	name	Office Number

Subject	<u>Sid</u>	name	credit	Teacher_id



Teacher

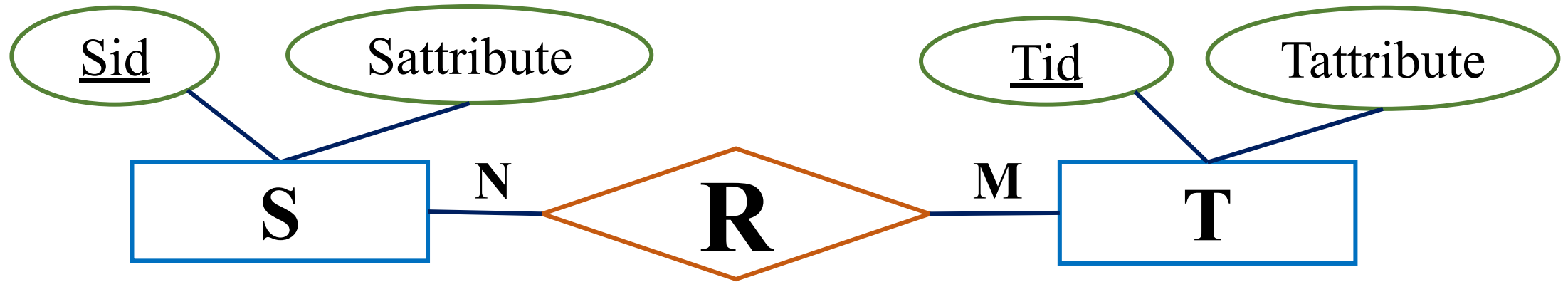
<u>Tid</u>	Name	salary
T1	John	5000
T2	Olivia	6000
T3	Adam	7000

Subject

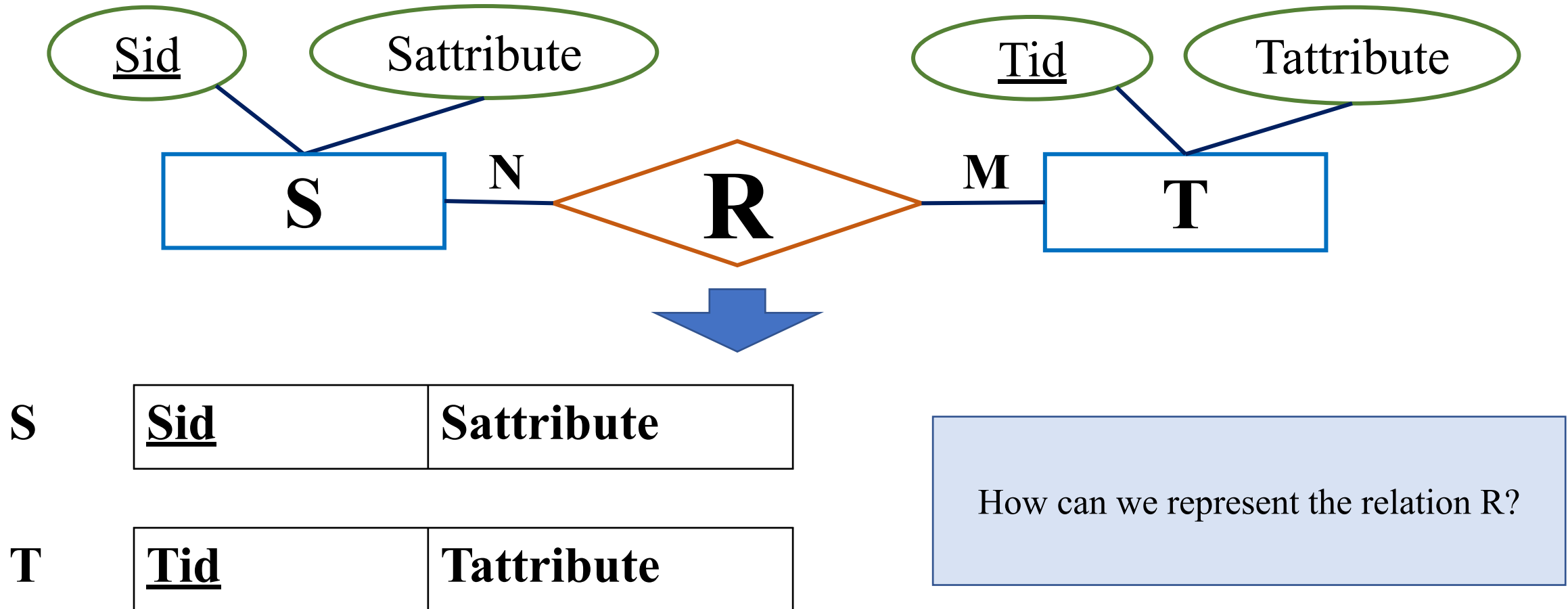
<u>Sid</u>	Name	Credit	Teacher ID
S1	OS	3	T1
S2	DB	3	T3
S3	CN	3	T1
S4	Java	3	T2
S5	Python	3	T2
S6	Math	3	T1

Translating $N:M$ relationship sets

Consider the following binary M:N relationship set R connecting two strong entity sets S and T

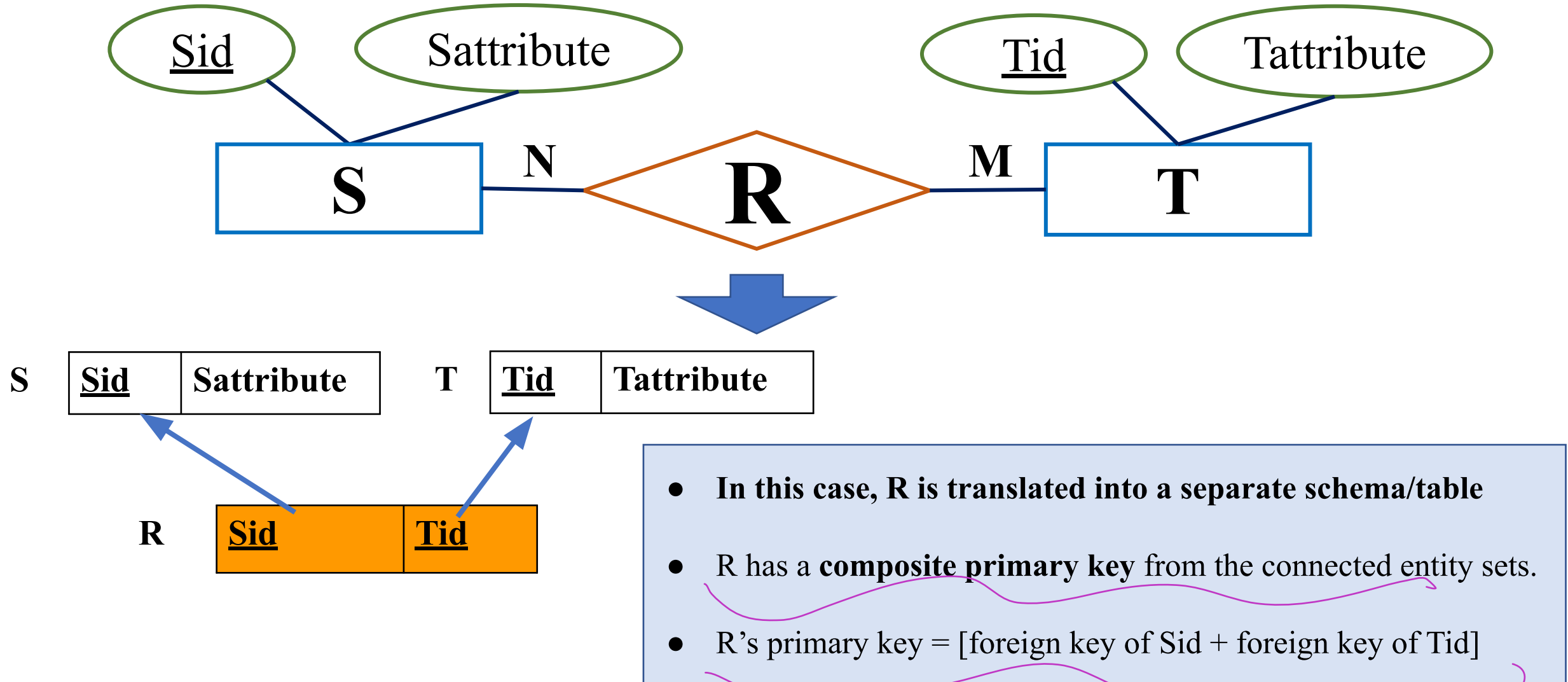


Consider the following binary M:N relationship set R connecting two strong entity sets S and T

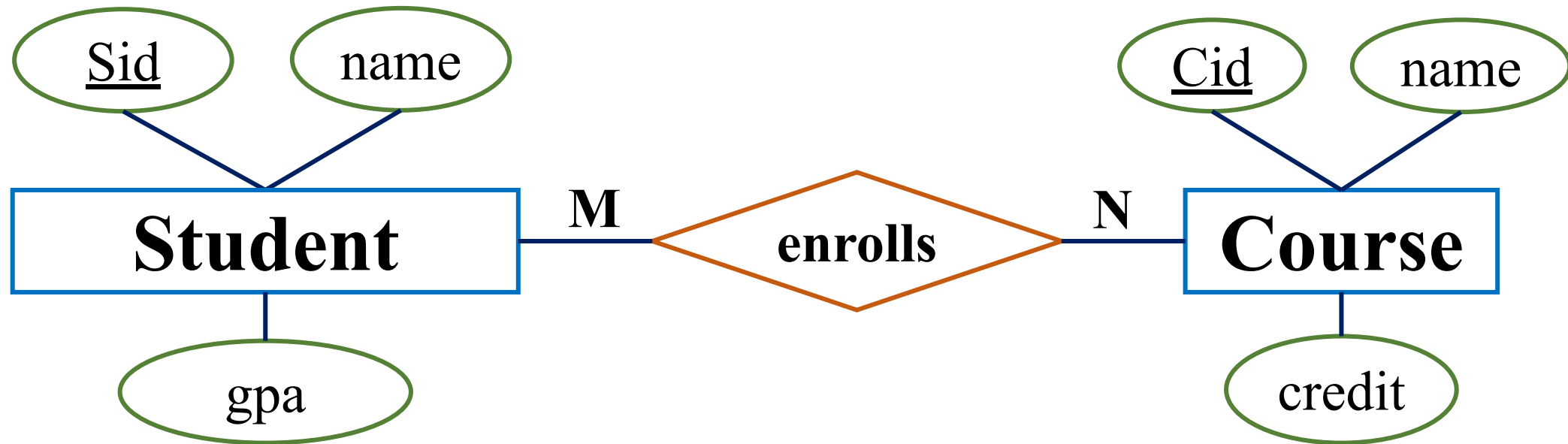


Translating N:M relationship sets

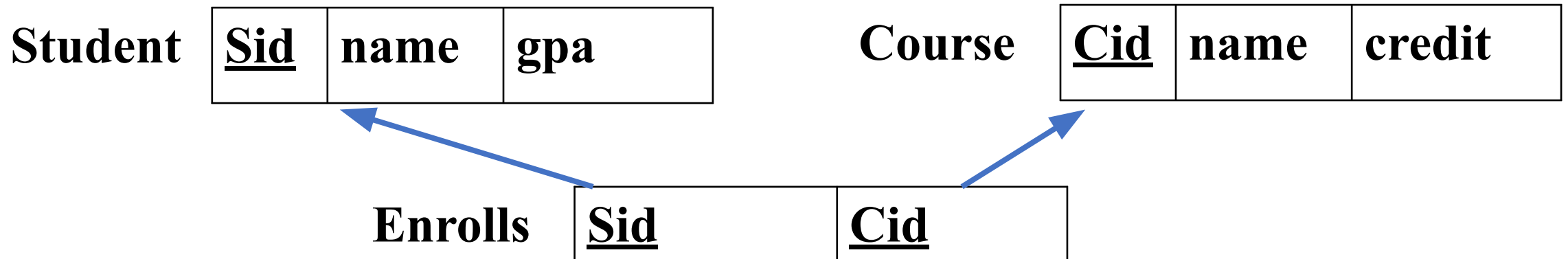
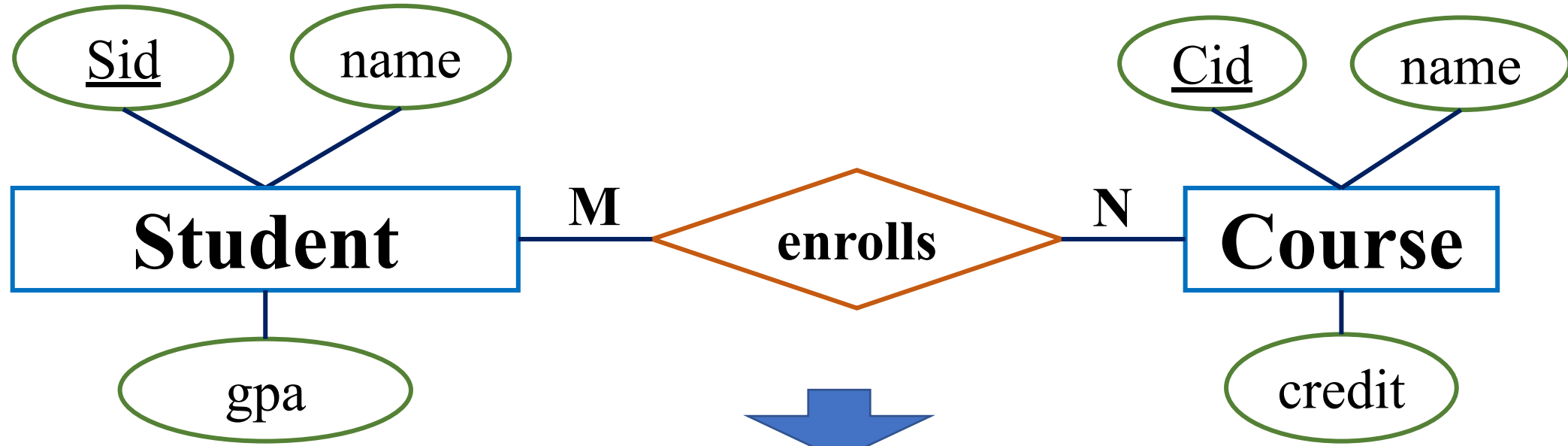
Consider the following binary M:N relationship set R connecting two strong entity sets S and T

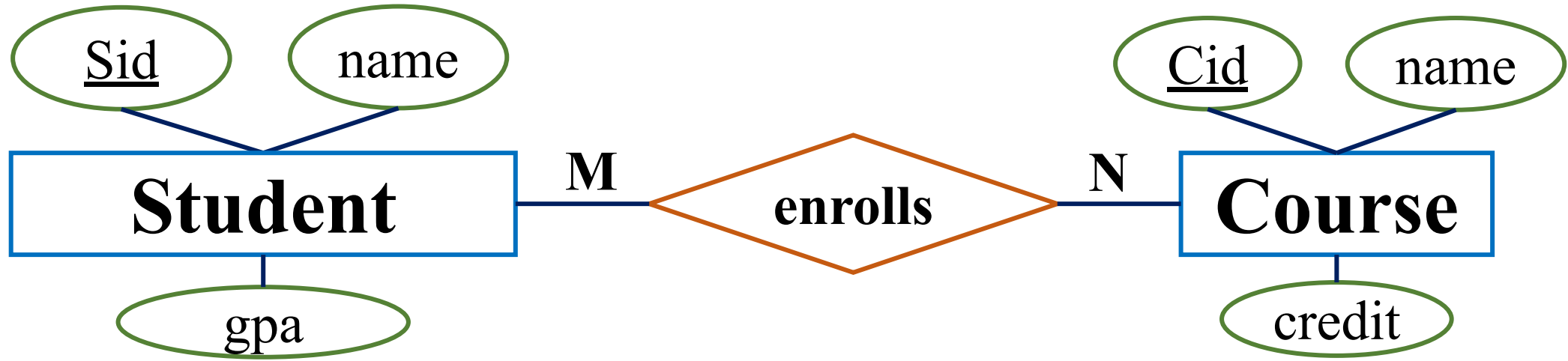


Translate the following ER diagram into relational schema(s):



Translate the following ER diagram into relational schema(s):





Student

<u>Sid</u>
S1
S2
S3

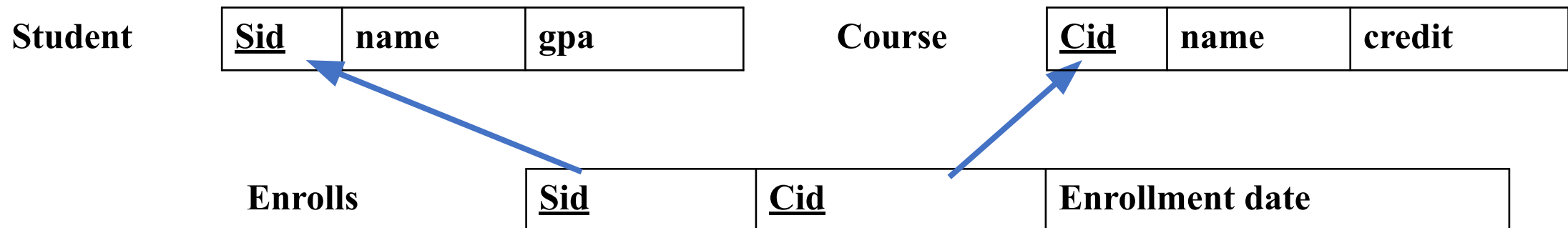
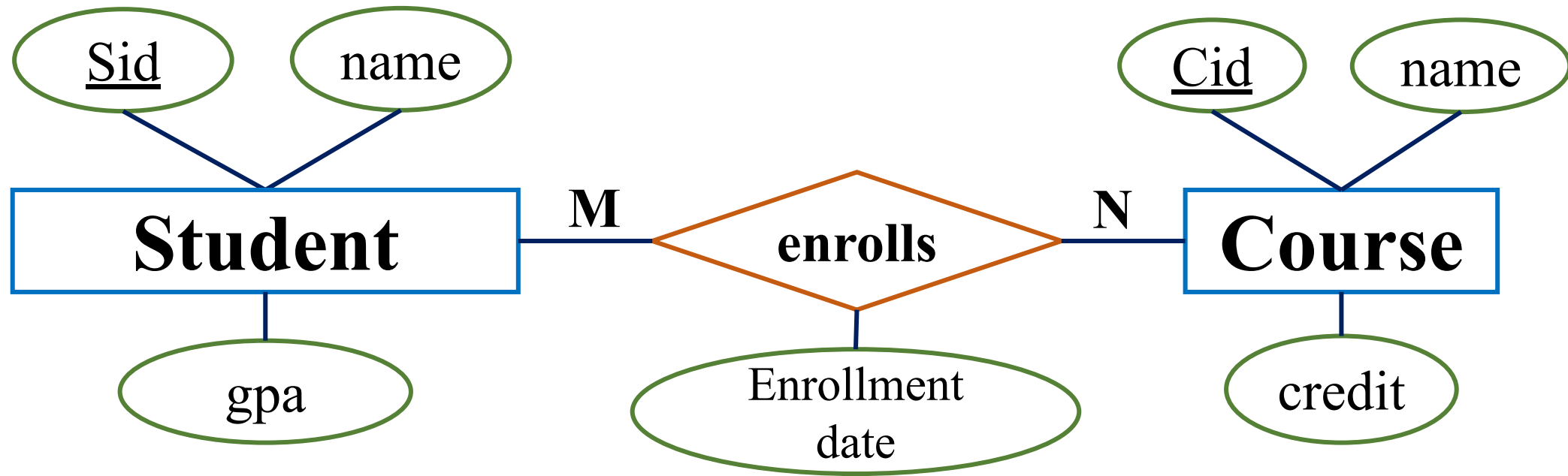
Enroll

<u>Sid</u>	<u>Cid</u>
S1	C1
S1	C2
S3	C1
S3	C2
S3	C3
S2	C1

Course

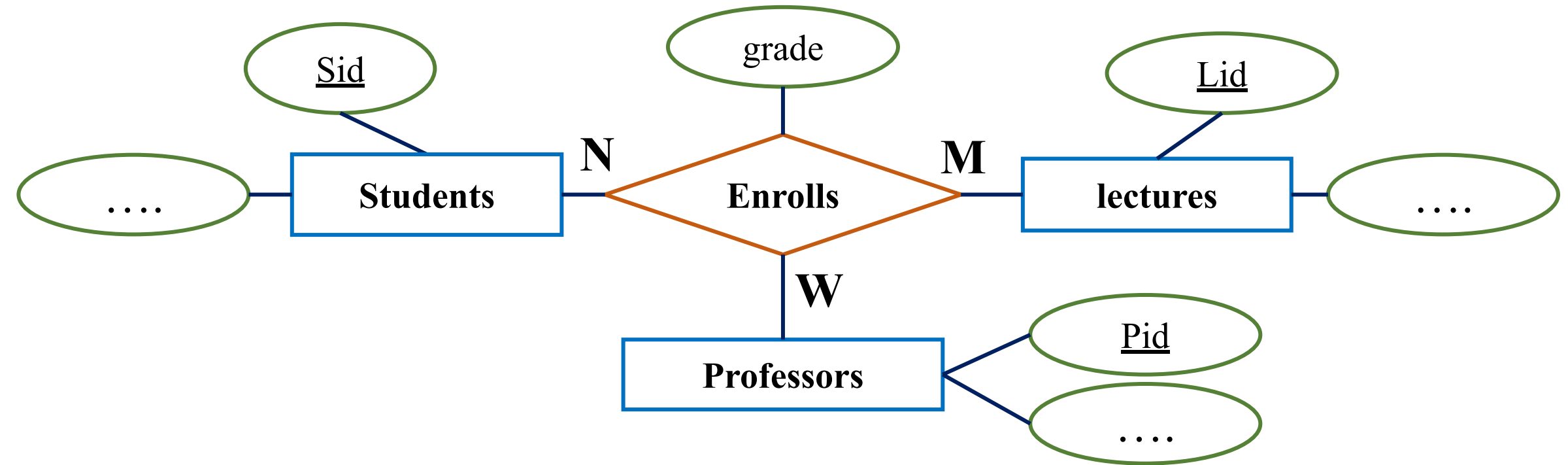
<u>Cid</u>
C1
C2
C3

Translate the following ER diagram into relational schema(s):



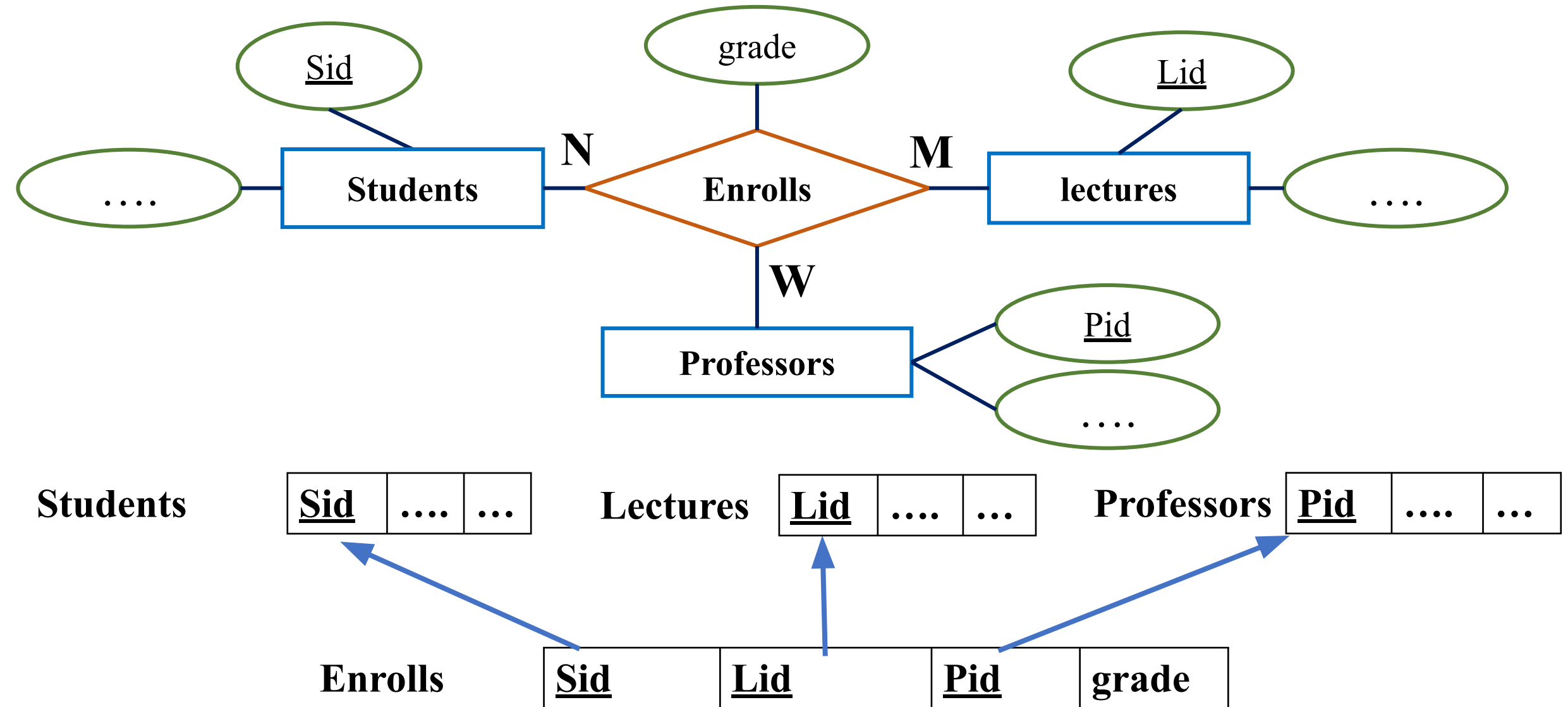
Translating N:M relationship sets

Translate the following ER diagram into relational schema(s):

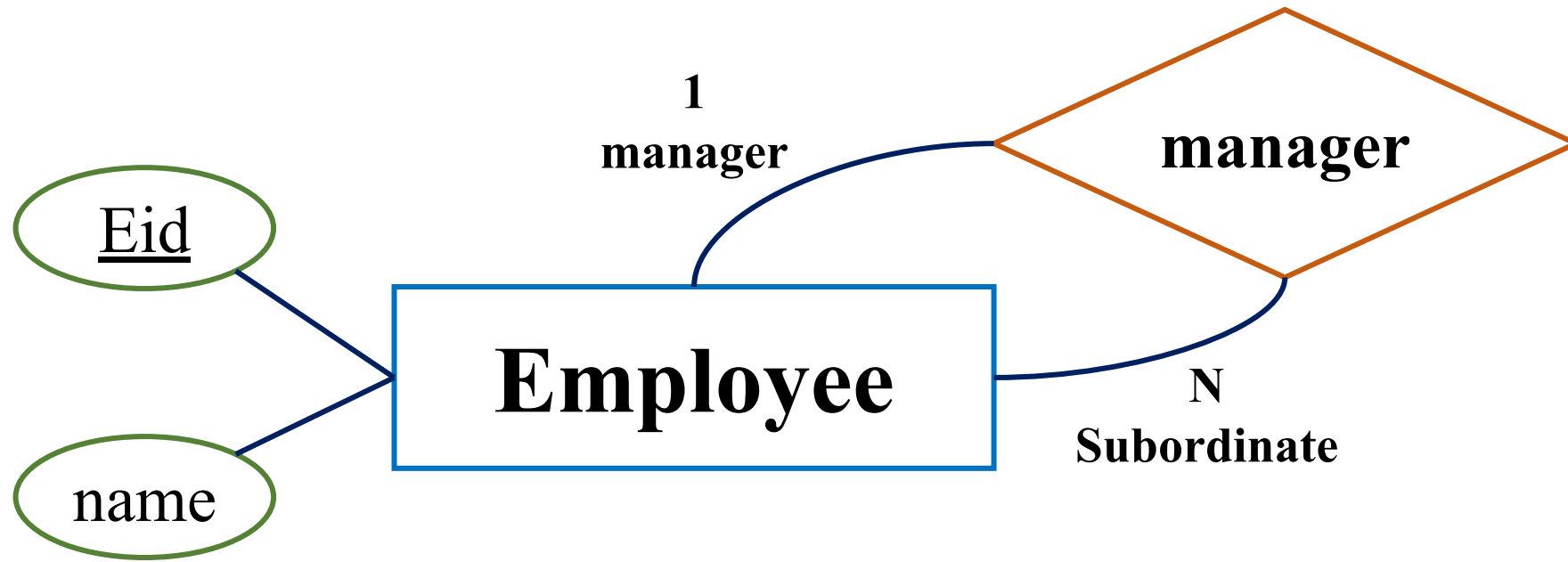


Translating N:M relationship sets

Translate the following ER diagram into relational schema(s):



Translate the following ER diagram into relational schema(s):

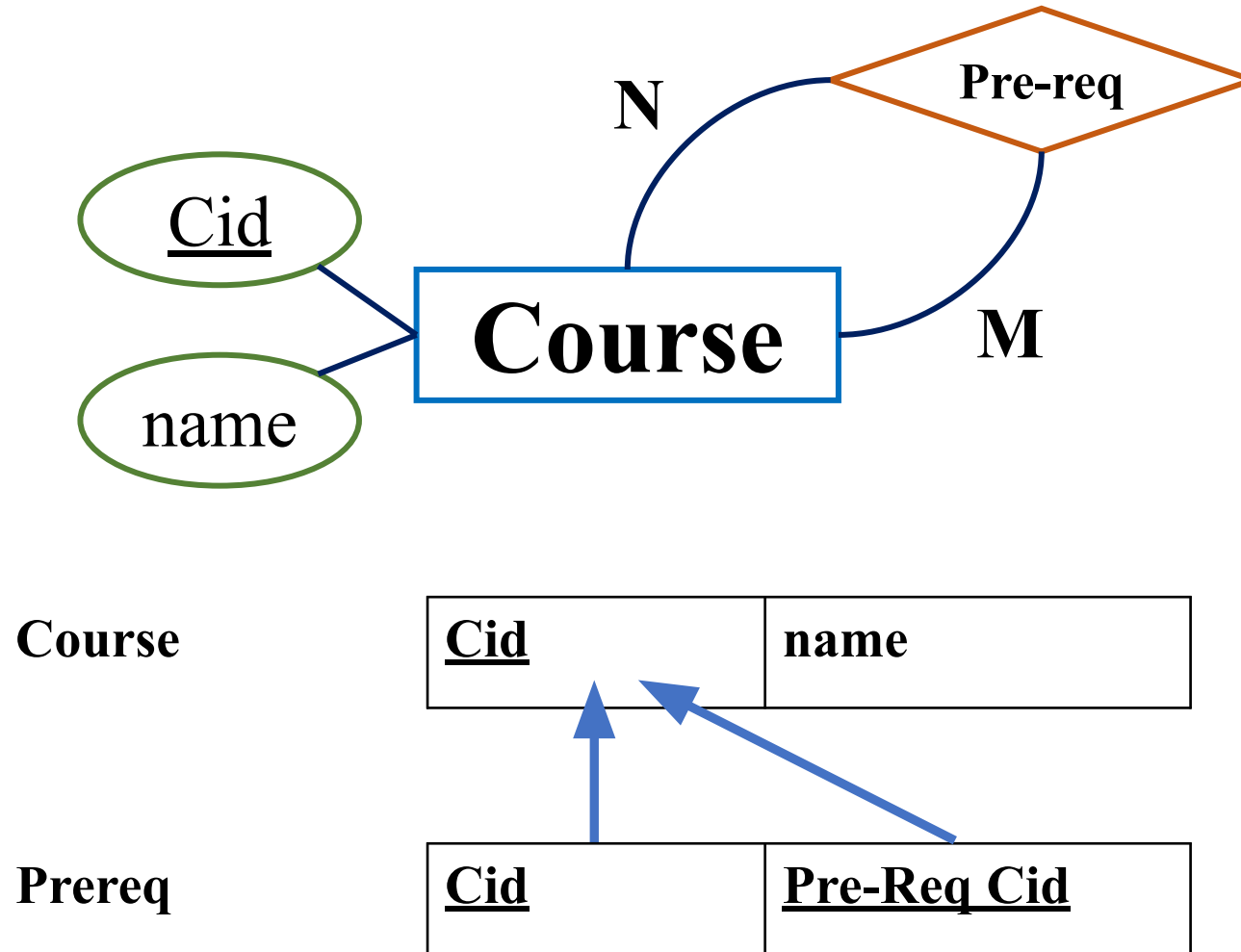


Employee

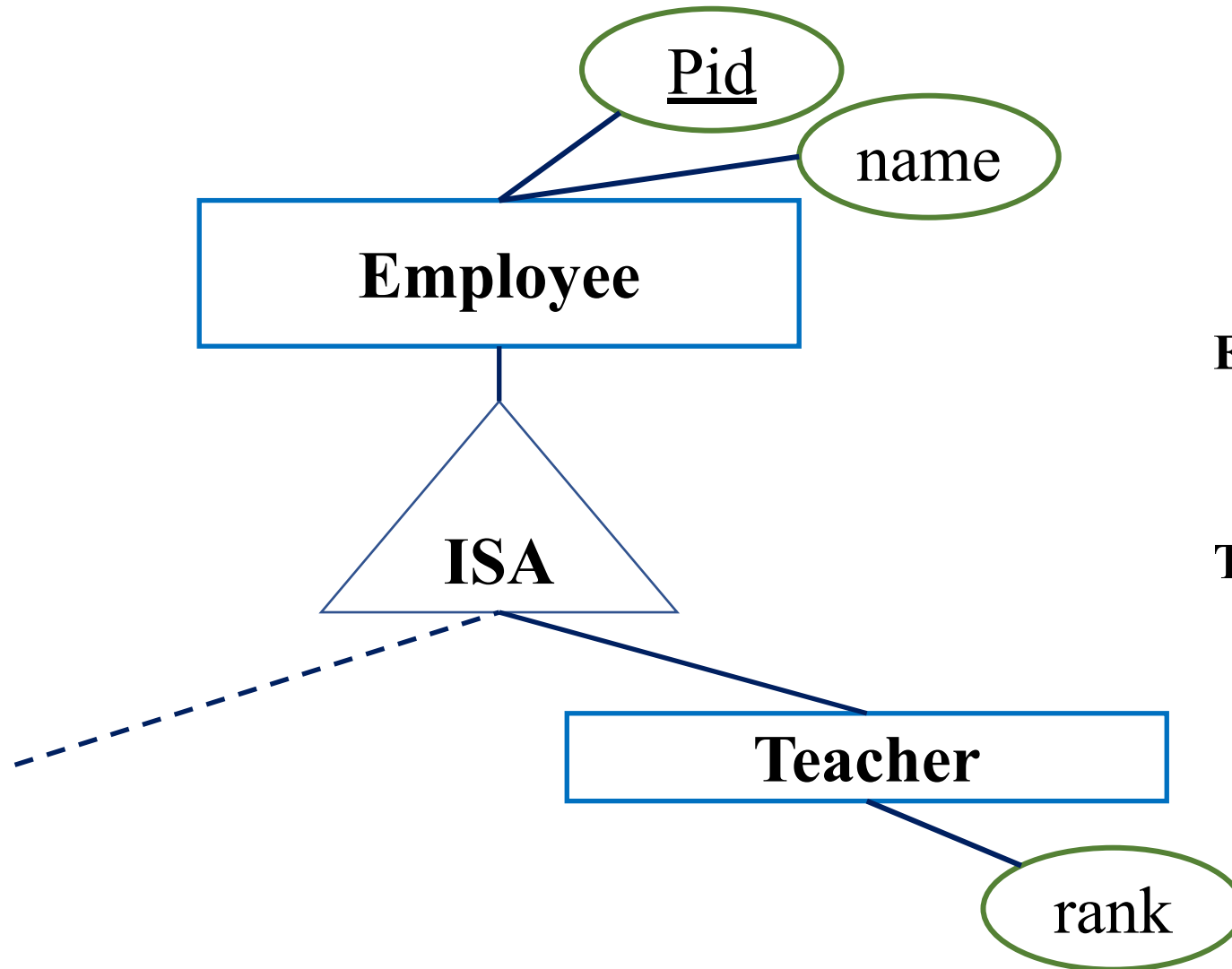
<u>Eid</u>	name	Manager id
-------------------	-------------	-------------------

A blue arrow originates from the **Manager id** attribute and points back to the **Eid** attribute, indicating a foreign key relationship.

Translate the following ER diagram into relational schema(s):



Translate the following ER diagram into relational schema(s):



Employee

<u>Pid</u>	name
------------	------

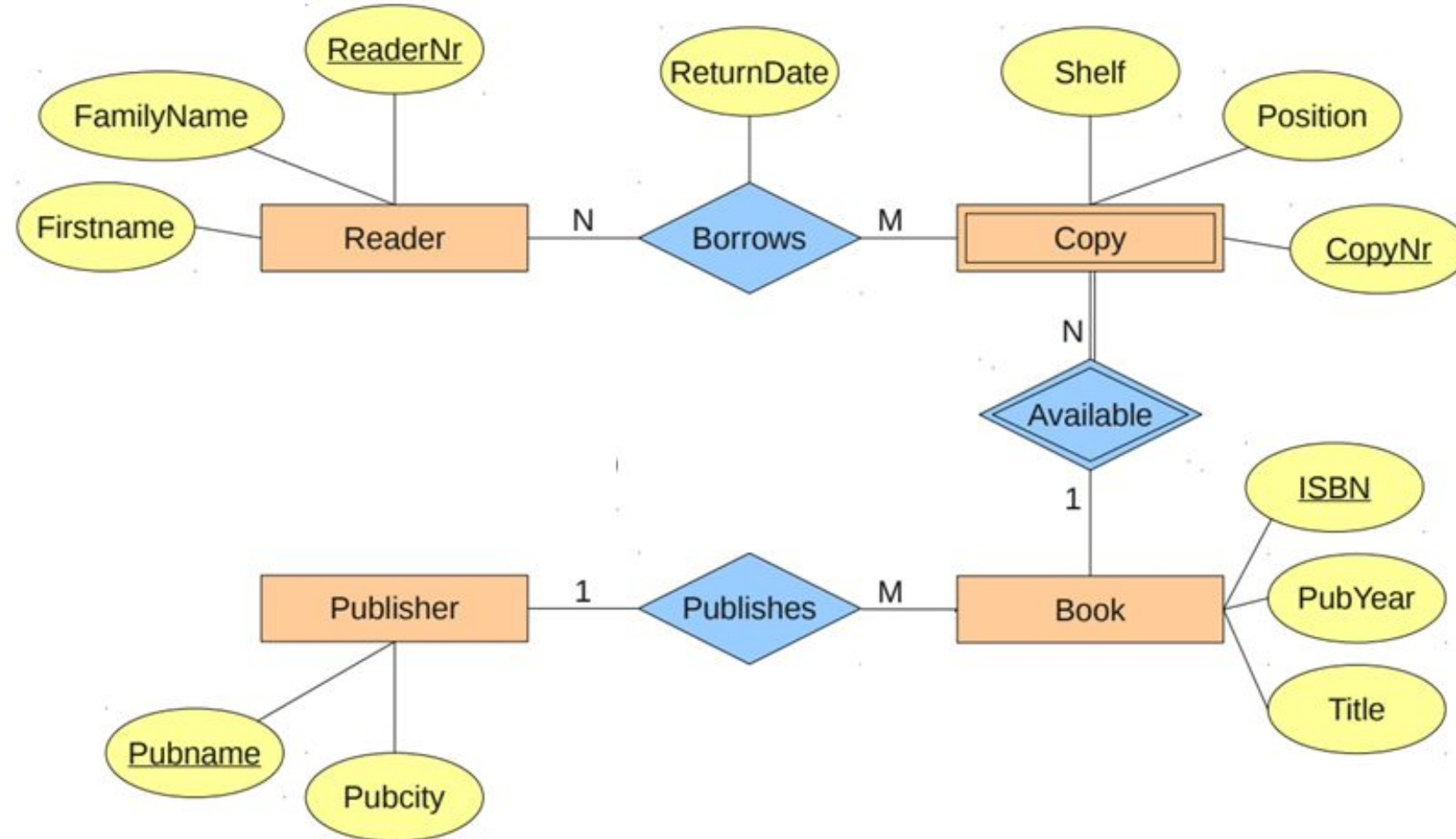
Teacher

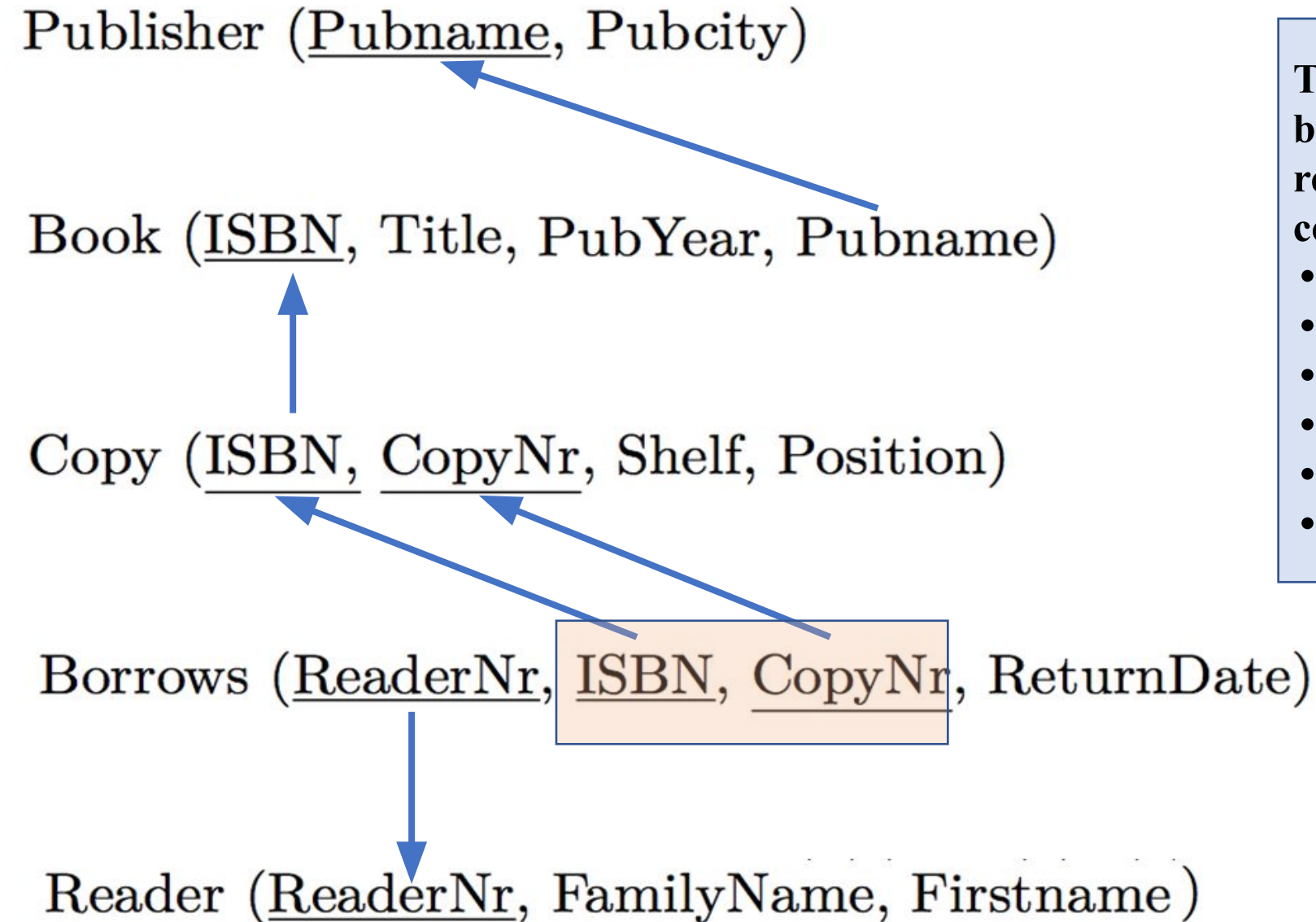
<u>Teacher id</u>	rank
-------------------	------



Review and Revision

Translate the following ER diagram into relational schema(s):



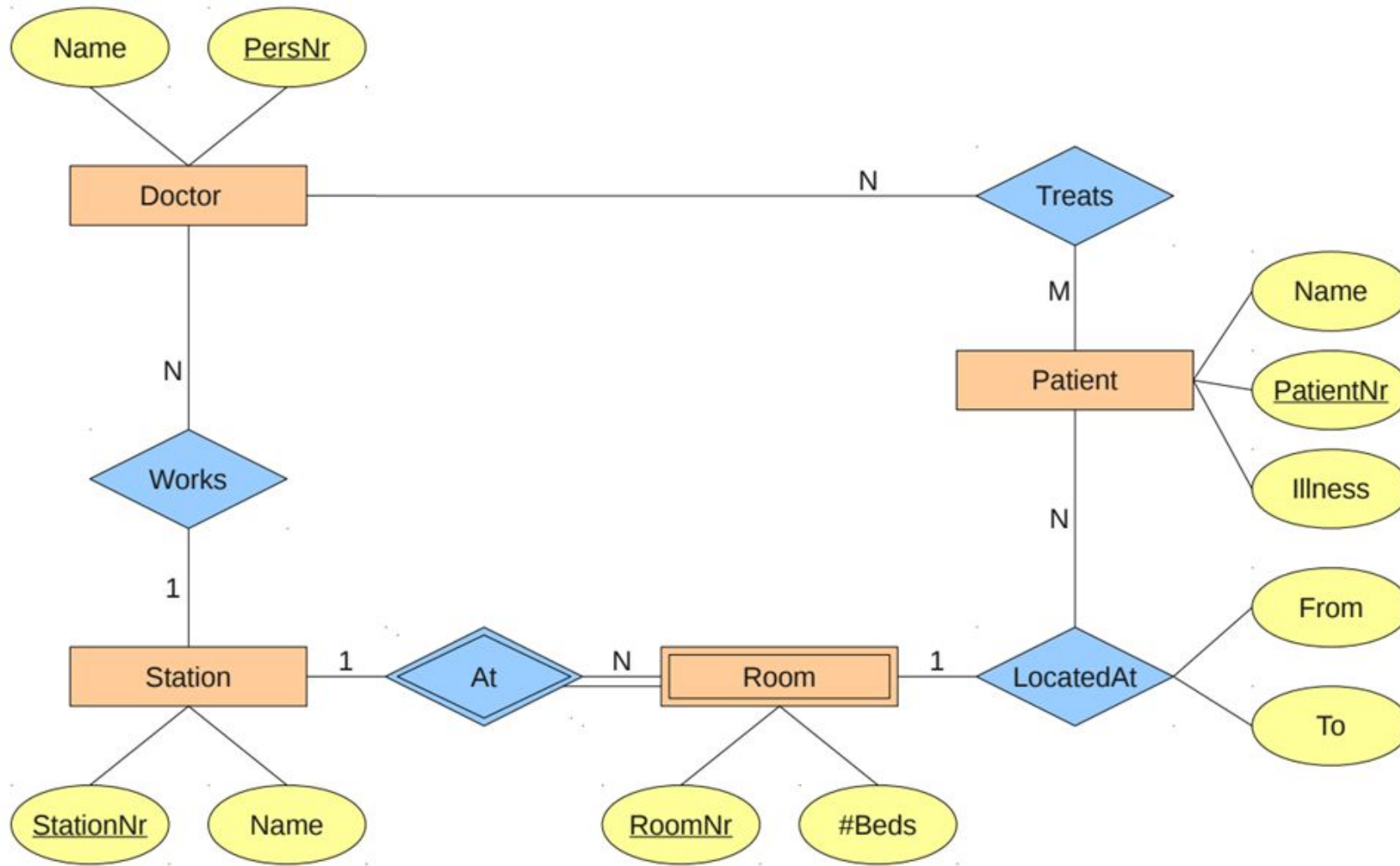


The datatypes/domains are removed because of the limited space and for readability – the available domains to consider are:

- Integer
- String
- Character
- Date
- Boolean
- Real

The Primary key of the “Copy” is ISBN+CopyNr

Translate the following ER diagram into relational schema(s):



The datatypes/domains are removed for space problems – the available domains to consider are:

- Integer,
- String,
- Character,
- Date,
- Boolean,
- Real

Worker (PersNr, Name, StationNr)

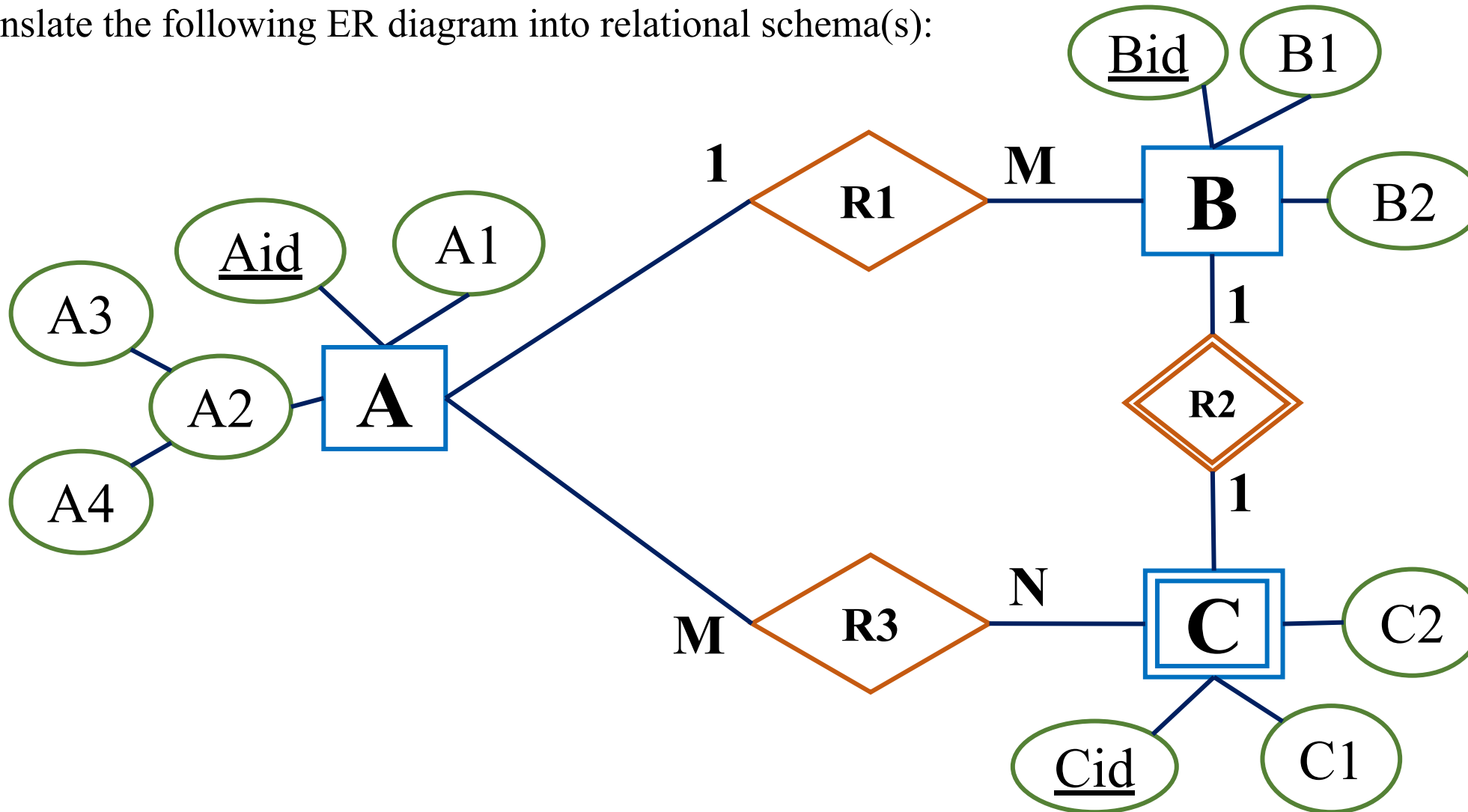
Station (StationNr, Name)

Room (StationNr, RoomNr, NumBeds)

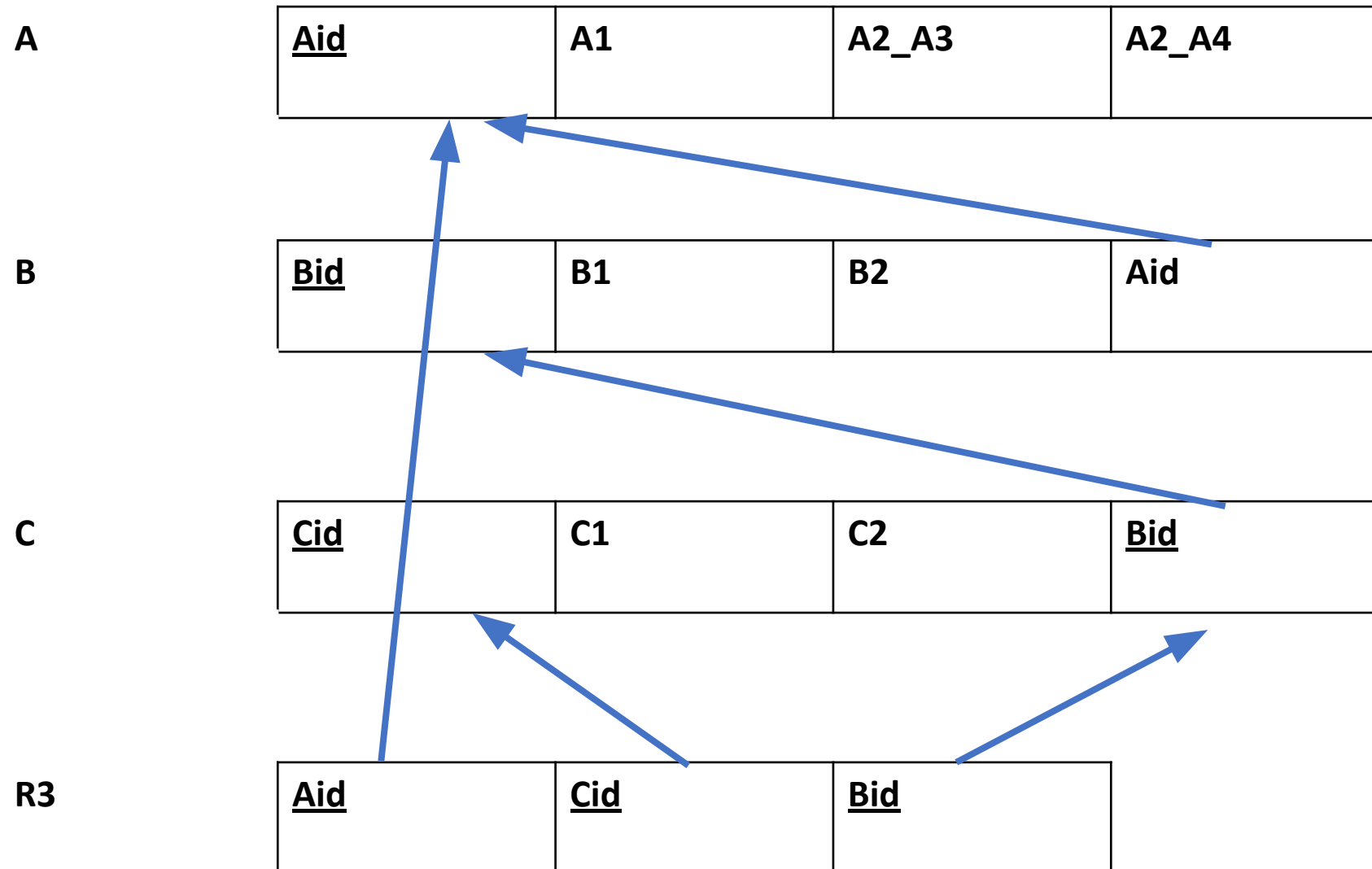
Patient (PatientNr, Name, Illness, From, To, StationNr, RoomNr)

Treats (PatientNr, PersNr)

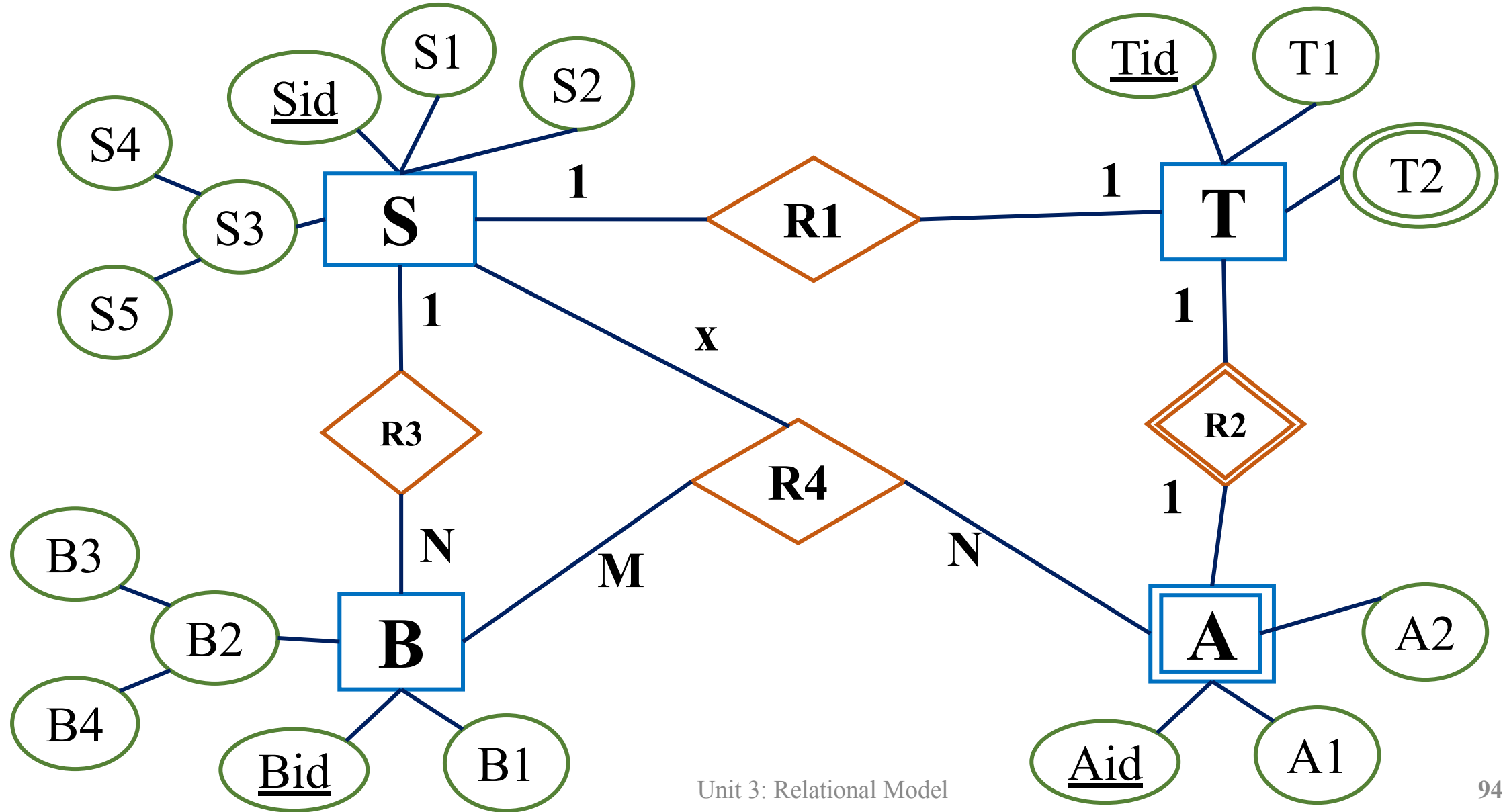
Translate the following ER diagram into relational schema(s):



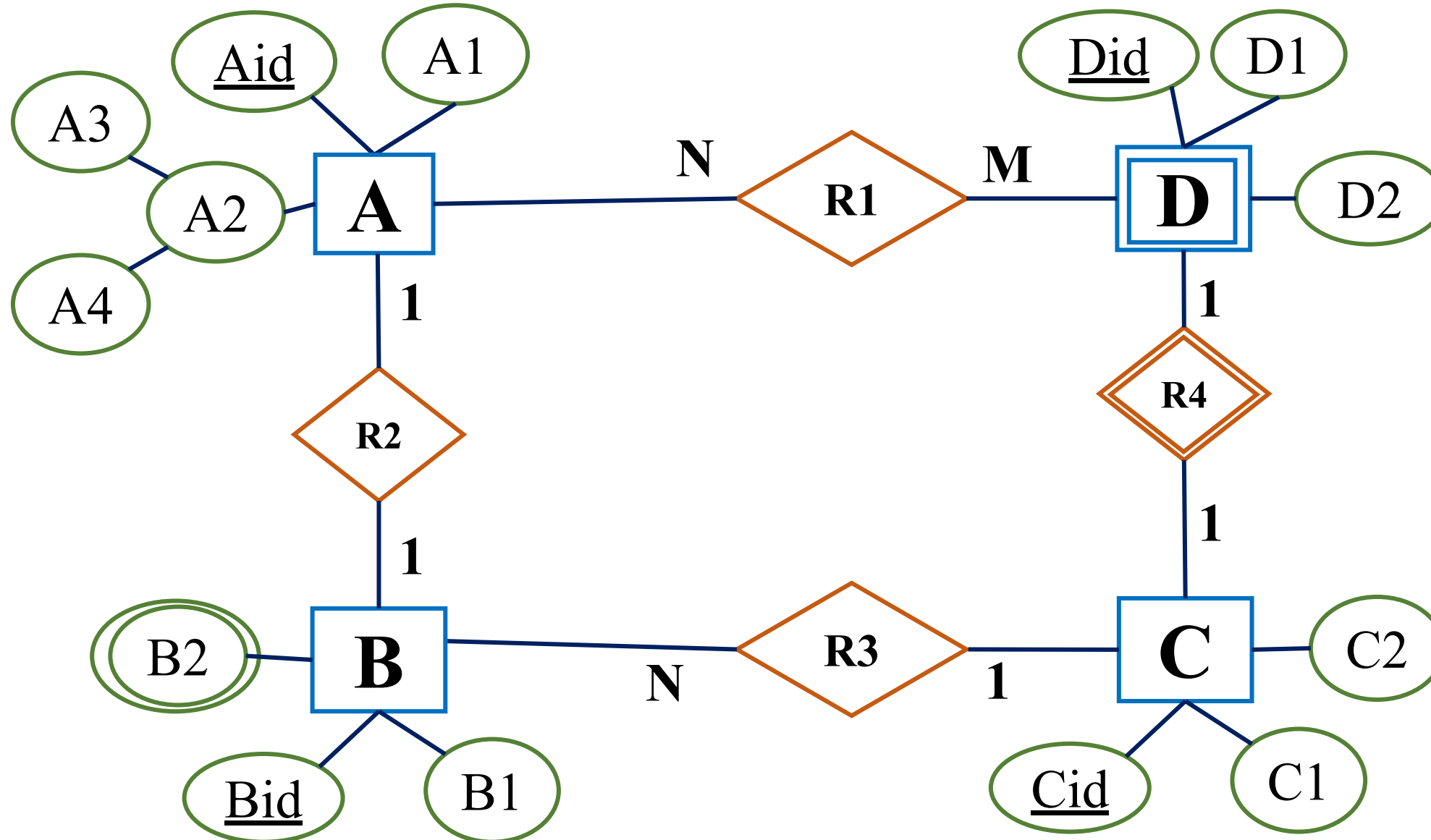
Answer:



Translate the following ER diagram into relational schema(s):



Translate the following ER diagram into relational schema(s):



Problems with Data Redundancy

- Consider the following Student relation (table) as an example, for every student, we store the student's name, the name of his/her department, the name of the department's chair, and the office telephone of the department's chair.

<u>Student_id</u>	Name	Department	Head of department	Office telephone
<u>401</u>	John	Computer Science	Dr. William	7404
<u>402</u>	Olivia	Computer Science	Dr. William	7404
<u>403</u>	Adam	Computer Science	Dr. William	7404
...
...
<u>4100</u>	Amelia	Computer Science	Dr. William	7404

- What is the problem?

- Consider the following Student relation (table) as an example, for every student, we store the student's name, the name of his/her department, the name of the department's chair, and the office telephone of the department's chair.

<u>Student_id</u>	Name	Department	Head of department	Office telephone
<u>401</u>	John	Computer Science	Dr. William	7404
<u>402</u>	Olivia	Computer Science	Dr. William	7404
<u>403</u>	Adam	Computer Science	Dr. William	7404
...
...
<u>4100</u>	Amelia	Computer Science	Dr. William	7404

- What is the problem? - The record [data for the department's name, head of the department, and office telephone] is repeated for the students who are in the same department, this is ***Data Redundancy***.

- Consider the following Student relation (table) as an example, for every student, we store the student's name, the name of his/her department, the name of the department's chair, and the office telephone of the department's chair.

<u>Student_id</u>	Name	Department	Head of department	Office telephone
<u>401</u>	John	Computer Science	Dr. William	7404
<u>402</u>	Data Redundancy leads to three main problems: <ul style="list-style-type: none">• Insertion anomalies• Update anomalies• Deletion anomalies			04
<u>403</u>				04
...				
...				
<u>4100</u>				04

- What is the problem? - The record [data for the department's name, head of the department, and office telephone] is repeated for the students who are in the same department, this is **Data Redundancy**.

- The 1st problem is “Insertion Anomalies”: What if we added 1000 students to the same department?

<u>Student_id</u>	Name	Department	Head of department	Office telephone
<u>401</u>	John	Computer Science	Dr. William	7404
<u>402</u>	Olivia	Computer Science	Dr. William	7404
<u>403</u>	Adam	Computer Science	Dr. William	7404
...
...
<u>4100</u>	Amelia	Computer Science	Dr. William	7404

The department information must be *repeated “correctly”* for all those 1000 students

- The 2nd problem “Update anomalies”: What if Dr. William is no longer the head of the department?

<u>Student_id</u>	Name	Department	Head of department	Office telephone
<u>401</u>	John	Computer Science	Dr. William	7404
<u>402</u>	Olivia	Computer Science	Dr. William	7404
<u>403</u>	Adam	Computer Science	Dr. William	7404
...
...
<u>4100</u>	Amelia	Computer Science	Dr. William	7404

- In that case we need to update all the student records
- If by mistake we miss any student, this will lead to *data inconsistency* (students from same department have different info about the Head of department).

- The 3rd problem “Deletion anomalies”: As two *different* information (info of student and info of dept.) are kept together in one table, what will happen after the graduation of the last student in such department?

The Student Info		The Department Info		
<u>Student_id</u>	Name	Department	Head of department	Office telephone
<u>401</u>	John	Computer Science	Dr. William	7404
<u>402</u>	Olivia	Computer Science	Dr. William	7404
<u>403</u>	Adam	Computer Science	Dr. William	7404
...
...
<u>4100</u>	Amelia	Computer Science	Dr. William	7404

If the last student in the department graduated, we will *lose all stored info about such a department!*

- We need some techniques to guide our database design and to avoid these problems. This is the *schema refinement* step of designing databases.
- As we discussed in the introduction, the steps of designing a database can be summarized as follows:

Step 1- *Requirements Analysis*

Step 2- *Conceptual Database Design*

Step 3- *Logical Database Design*

Step 4- *Schema Refinement* (The focus of this part)



Step 5- *Physical Database Design*

Step 6- *Application Design*

- A table with data redundancy is known as a *non-normalized* table.
- Normalization is a *systematic design technique* of organizing data in a database by decomposing non-normalized tables with insertion/deletion/update anomalies to produce *well-structured relations*.
- Normalization techniques are tools to *validate and improve the logical design* to avoid unnecessary redundancy and anomalies, and to allow users to insert/delete/update rows without causing *data inconsistencies*.

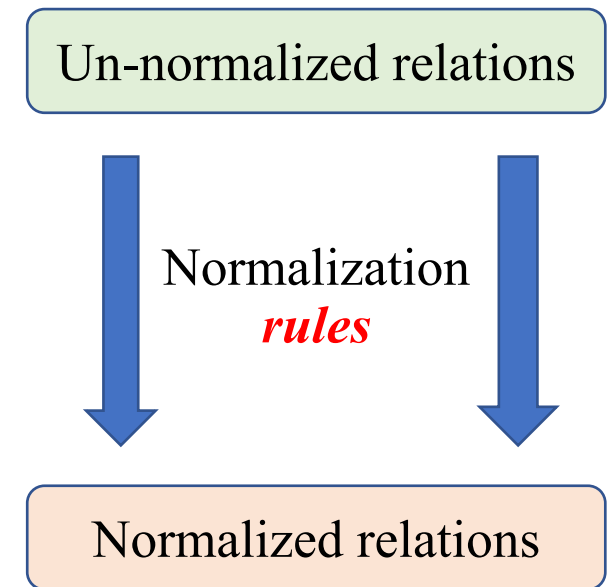
□ **Normalization** is a set of techniques/rules where more relations (tables) are created to store non-redundant data by dividing a relation into sub-relations.

□ **Denormalization** is the reverse process where **redundant data is included** in order to improve the performance and for faster execution of certain applications and scenarios.

- In this process, relations (tables) are combined to speed up the query process by reducing the use of the Join operator (sub-relations are combined in one relation)

Normalization *rules* are divided into the following *normal forms*:

1. First Normal Form (1NF)
2. Second Normal Form (2NF)
3. Third Normal Form (3NF)
4. Boyce-Codd Normal Form (BCNF)
5. Fourth Normal Form (4NF)



Normalization *rules* are divided into the following *normal forms*:

1. First Normal Form (1NF)
2. Second Normal Form (2NF)
3. Third Normal Form (3NF)
4. Boyce-Codd Normal Form (BCNF)
5. Fourth Normal Form (4NF)

We will focus on
the first three
Normal Forms

Un-normalized relations

Normalization
rules

Normalized relations

First Normal Form (1NF)

- The 1NF is about re-designing a table/relation to be easily extended and to enable an *easier/faster retrieval* of data whenever required.
- Processing an array (list of values) -for some relational databases- may negatively impact the performance.
- For a table to be in the 1NF, it should follow the following rules:
 1. Each cell should only hold *single (atomic) valued attributes*
 2. Values stored in the same column should be of the *same domain (integer, string)*
 3. All columns in a table should have *unique names*
 4. The order in which data is stored does not matter (the order of tuples/rows)

Consider the following table as an example – is it in 1NF?

<u>Student_id</u>	Name	Subject	GPA
401	John	OS, CN	3.8
402	Olivia	Java	3.9
403	Adam	PYT, C++	3.7

1. It should only have single(atomic) valued attributes. ✗
2. Values stored in same column should be of the same domain ✓
3. All the columns in a table should have unique names. ✓
4. And the order in which data is stored, does not matter. ✓

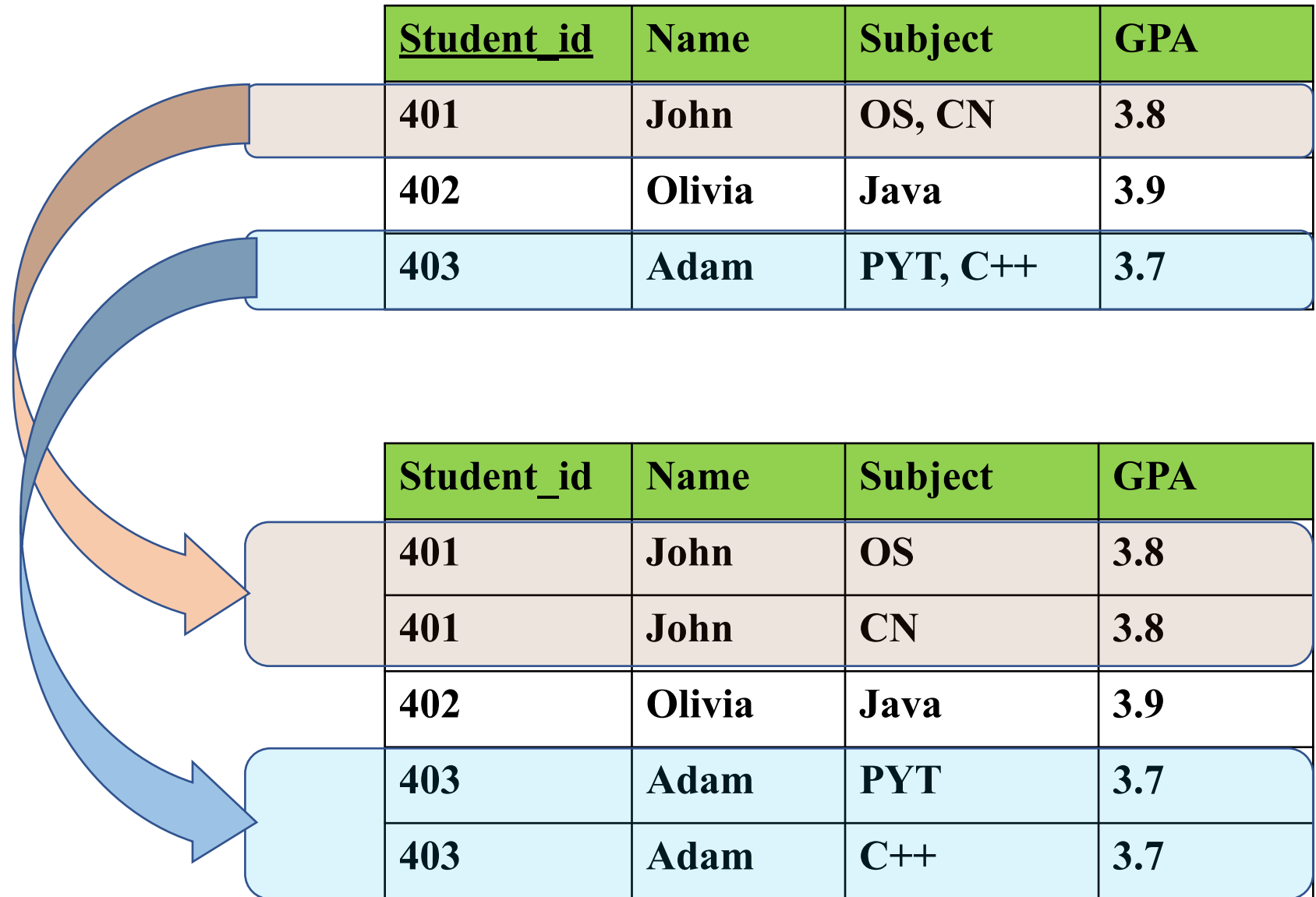
Consider the following table as an example – is it in 1NF?

<u>Student_id</u>	Name	Subject	GPA
401	John	OS, CN	3.8
402	Olivia	Java	3.9
403	Adam	PYT, C++	3.7

1. It should only have single(atomic) valued attributes. ✗
2. Values stored in same column should be of the same domain ✓
3. All the columns in a table should have unique names. ✓
4. And the order in which data is stored, does not matter. ✓

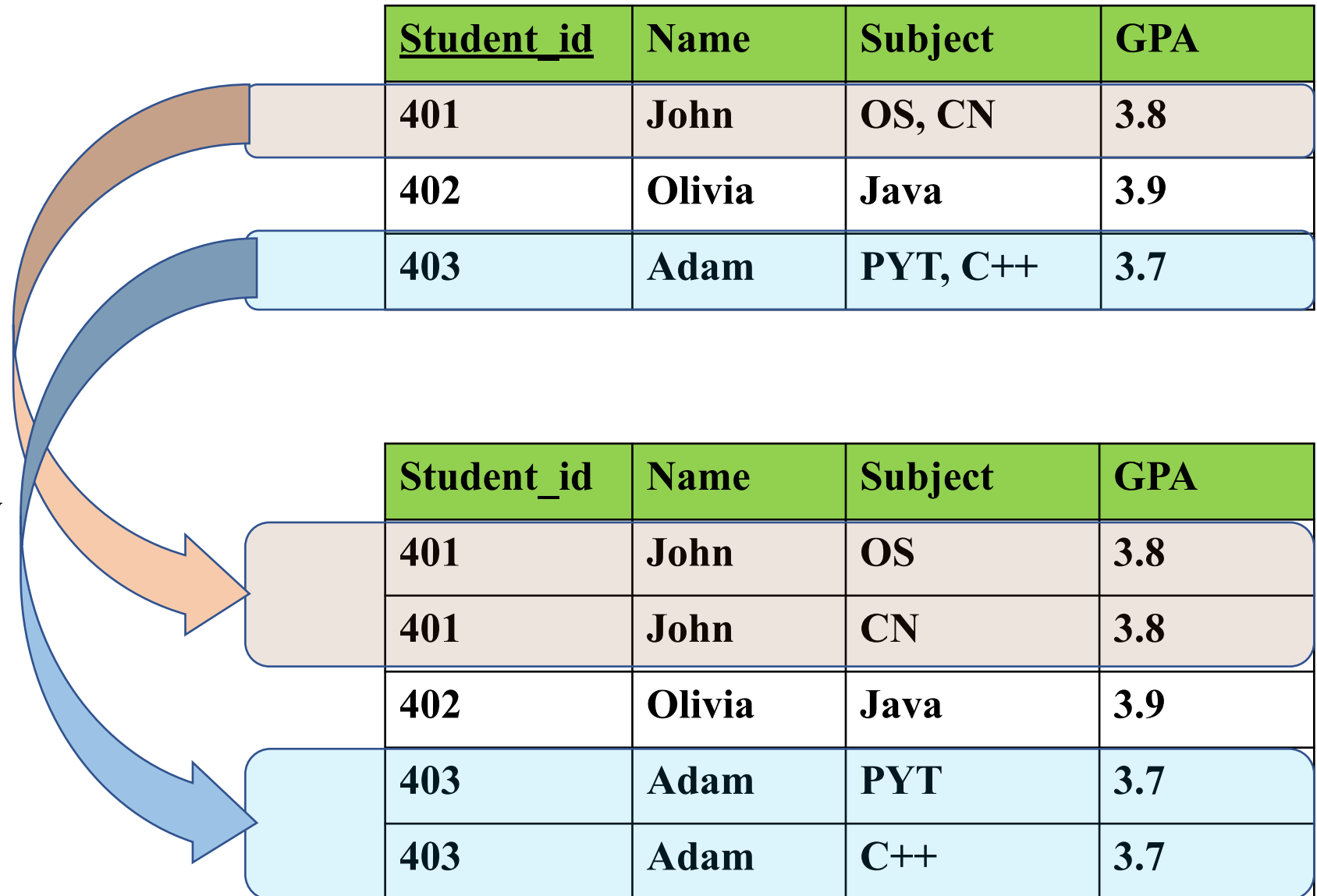
We need to re-consider such multivalued attribute column to carry single values only

The first step is to extend the non 1NF table to have a record per value from the multi-valued attribute



The first step is to extend the non 1NF table to have a record per value from the multi-valued attribute

Let us Ignore the new table's primary key constraint for now - What is the *problem* with such change?



The first step is to extend the non 1NF table to have a record per value from the multi-valued attribute

	<u>Student_id</u>	Name	Subject	GPA
	401	John	OS, CN	3.8
	402	Olivia	Java	3.9
	403	Adam	PYT, C++	3.7

Let us Ignore the new table's primary key constraint for now - What is the *problem* with such change?

Data redundancy increases - The name and GPA are repeated unnecessarily

<u>Student_id</u>	Name	Subject	GPA
401	John	OS	3.8
401	John	CN	3.8
402	Olivia	Java	3.9
403	Adam	PYT	3.7
403	Adam	C++	3.7

This can be solved by *dividing the new table* into two - linked through the foreign and primary key relationship.

Student_id	Name	Subject	gpa
401	John	OS	3.8
401	John	CN	3.8
402	Olivia	Java	3.9
403	Adam	PYT	3.7
403	Adam	C++	3.7

*The original
Student Table*

This can be solved by *dividing the new table* into two - linked through the foreign and primary key relationship.

Student_id	Name	Subject	gpa
401	John	OS	3.8
401	John	CN	3.8
402	Olivia	Java	3.9
403	Adam	PYT	3.7
403	Adam	C++	3.7

The original Student Table

Student Table

Student_id	Name	gpa
401	John	3.8
402	Olivia	3.9
403	Adam	3.7

Where is the primary key here?

Student Subject Table

Student_id	Subject
401	OS
401	CN
402	Java
403	PYT
403	C++

Where is the primary key here?

First Normal Form

This can be solved by *dividing the new table* into two - linked through the foreign and primary key relationship.

Student_id	Name	Subject	gpa
401	John	OS	3.8
401	John	CN	3.8
402	Olivia	Java	3.9
403	Adam	PYT	3.7
403	Adam	C++	3.7

The original Student Table

Student Table

<u>Student_id</u>	Name	gpa
401	John	3.8
402	Olivia	3.9
403	Adam	3.7

Student Subject Table

<u>Student_id</u>	<u>Subject</u>
401	OS
401	CN
402	Java
403	PYT
403	C++

- This table saves the ID, Name, and GPA
- *the Primary key is ID*

- This table saves the ID and the updated column -
the Primary key is: ID + Subject
- This ID is a foreign key for the ID attribute of the first table

If the following table is not normalized, normalize it to follow the 1NF

Employee Table

<u>Employee ID</u>	Name	Department	Projects
401	John	Marketing	334,354
402	Olivia	Advertising	234,443

*The original
Employee Table*

<u>Employee ID</u>	Name	Department	Projects
401	John	Marketing	334,354
402	Olivia	Advertising	234,443

Projects column is multi-valued, the Employee table is not in 1NF.

We should divide the main table into two tables:



Employee Table

<u>Employee_id</u>	Name	Department
401	John	Marketing
402	Olivia	Advertising



Employee Project Table

<u>Employee_id</u>	<u>Projects</u>
401	334
401	354
402	234
402	443

Second Normal Form (2NF)

Redundant data across multiple rows of a table must be moved to a separate table, the resulting tables must be related to *each other by use of foreign key(s)*



For a table to be in the 2NF, it should follow the following rules:


1. It should be in the First Normal form (1NF).
 2. It should not have *Partial Dependency*.
- 

Redundant data across multiple rows of a table must be moved to a separate table, the resulting tables must be related to *each other by use of foreign key(s)*

For a table to be in the 2NF, it should follow the following rules:

1. It should be in the First Normal form (1NF).
2. It should not have *Partial Dependency*.

What is dependency?

- FD is a relationship between one attribute (or attribute) to another, where *one defines the other*.
- If i know the employee's ID, i can know the employee's First Name. In this case, we can say that employee's First Name functionally depends on the employee's ID.A decorative wavy purple line that spans across the width of the slide, positioned below the second bullet point.
- If i know the employee's years of experience, i can't for sure know the employee's Name or ID, however i can know the employee's seniority.

- FD is a kind of *integrity constraints* that generalizes the concept of a key and defines links between attributes.
- Let R be a relational schema with X and Y to be sets of attributes in R:

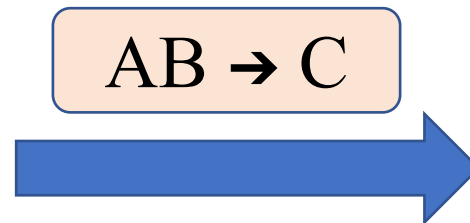
A FD “ $X \rightarrow Y$ ” is read as:

X functionally determines Y, or Y is determined by X

- For every occurrence of X, you will get the same value of Y
 - Thus, if tuples agree on the values in attributes X, they must also agree on the values in attributes Y.
 - This doesn't usually mean that X is a primary key

- Consider the following relation as an example, the combined value of A and B determines the value of C. In another word, the value of C depends on the value of A and B:

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1



A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

- Consider the following relation as an example:

ID	Name	Project	Department
100	Adam	Proj1	CS
101	Olivia	Proj2	Accounting
102	John	Proj3	Marketing
103	Melissa	Proj1	CS
104	Bob	Proj3	Marketing

- Consider the following relation as an example:

ID	Name	Project	Department
100	Adam	Proj1	CS
101	Olivia	Proj2	Accounting
102	John	Proj3	Marketing
103	Melissa	Proj1	CS
104	Bob	Proj3	Marketing

From the provided data,
we can infer few functional dependencies:
FD1: ID \rightarrow Name
FD2: ID \rightarrow Project

whenever you use ID, you will get the corresponding
value for Name and Project

- Consider the following relation as an example:

ID	Name	Project	Department
100	Adam	Proj1	CS
101	Olivia	Proj2	Accounting
102	John	Proj3	Marketing
103	Melissa	Proj1	CS
104	Bob	Proj3	Marketing

You may also notice from the data that for every project, there is only one department.

From the provided data,
we can infer few functional dependencies:
FD3: Project → Department

- Consider the following Hourly Emps relation:

HourlyEmployees (SSN, name, rating, hourly_wages, hours_worked)

- As the value of a primary key uniquely identifies specific record in a table, a *primary key constraint is a special case of an FD*, thus an FD on key can be expressed as:

$\{\text{SSN}\} \rightarrow \{\text{SSN}, \text{name}, \text{rating}, \text{hourly wages}, \text{hours worked}\}$

- We can also have an FD describing the constraint that the hourly wages attribute is determined by the rating attribute:

$\{\text{rating}\} \rightarrow \{\text{hourly wages}\}$

- In the domain of functional dependencies, there is a wide set of rules that can be used to set initial set of FDs and to infer or claim more FDs - for example:
- The *Transitivity rule*:

If $X \rightarrow Y$ and $Y \rightarrow Z$ Then $X \rightarrow Z$

- Using the *transitivity* rule on the following example:

ID	Name	Project	Department
100	Adam	Proj1	CS
101	Olivia	Proj2	Accounting
102	John	Proj3	Marketing
103	Melissa	Proj1	CS
104	Bob	Proj3	Marketing

FD1: ID \rightarrow Name

FD2: ID \rightarrow Project

FD3: Project \rightarrow Department

we can infer/claim that ID \rightarrow Department

- In the domain of functional dependencies, there is a wide set of rules that can be used to set initial set of FDs and to infer or claim more FDs.
- Assume a relation schema R with a set of attributes X, Y, and Z – There are some rules, known as the Armstrong's Axioms, that can be applied to *infer* FDs from a set of FDs:

- The *Transitivity rule*:

If $X \rightarrow Y$ and $Y \rightarrow Z$ Then $X \rightarrow Z$

- The *Reflexive rule*:

If $X \supseteq Y$ (X is a superset of Y, Y is a subset of X, X includes Y), then $X \rightarrow Y$

- In the domain of functional dependencies, there is a wide set of rules that can be used to set initial set of FDs and to infer or claim more FDs.
- Assume a relation schema R with a set of attributes X, Y, and Z – There are some rules, known as the Armstrong's Axioms, that can be applied to *infer* FDs from a set of FDs:

- The *Augmentation rule*:

If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z

- The *Union rule*:

If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

- The *Decomposition rule*:

If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

- Functional dependencies are logical constraints derived from the *meaning and interrelationships* between the attributes.
- Functional dependencies can be used as formal methods to *measure how good your design* for a relational database is.
- In the 2nd and 3rd levels of normalization forms (2NF and 3NF), we will focus on two essential types of functional dependencies:
 1. Partial dependency
 2. Transitive dependency
- Other levels of normalization forms (4NF and BCNF) focus on wider set of functional dependencies for more efficient designs.

Redundant data across multiple rows of a table must be moved to a separate table, the resulting tables must be related to *each other by use of foreign key(s)*

For a table to be in the 2NF, it should follow the following rules:

1. It should be in the First Normal form (1NF).
2. It should not have *Partial Dependency*.

Redundant data across multiple rows of a table must be moved to a separate table, the resulting tables must be related to *each other by use of foreign key(s)*

For a table to be in the 2NF, it should follow the following rules:

1. It should be in the First Normal form (1NF).
2. It should not have *Partial Dependency*.

- The relation is automatically in 2NF if, and only if, the primary key is a *single attribute*.
- If the relation has a *composite primary key*, then each non-key attribute must be fully dependent on the entire primary key and not on a subset (i.e., there must be no partial dependency or augmentation).

Consider the following Student relation, where the student ID (primary key) is unique for every row (for every student), hence can be used to fetch a student record *even if we have students with exact same records*.

Student

<u>id</u>	Name	GPA	Department	Address
-----------	------	-----	------------	---------

Consider the following Student relation, where the student ID (primary key) is unique for every row (for every student), hence can be used to fetch a student record *even if we have students with exact same records*.

Student

<u>id</u>	Name	GPA	Department	Address
-----------	------	-----	------------	---------

Two different students,
with exact same info

Student

<u>id</u>	Name	GPA	Department	Address
<u>22</u>	John	3.8	CS	Chicago
<u>23</u>	John	3.8	CS	Chicago

Consider the following Student relation, where the student ID (primary key) is unique for every row (for every student), hence can be used to fetch a student record *even if we have students with exact same records*.

Student	<u>id</u>	Name	GPA	Department	Address
---------	-----------	------	-----	------------	---------

- We need the student's ID in order to access/read the student's record and the different attributes - this known as *functional dependency*
- The access of the different attributes *“functionally depends”* on the access of the ID.

Student

<u>id</u>	Name	GPA	Department	Address
-----------	------	-----	------------	---------

Along with the student table, let's create another table for the *Subjects*, with subject id (primary key) and name as attributes.

Subjects

<u>Subject id</u>	Name
<u>1</u>	Java
<u>2</u>	C++
<u>3</u>	PHP

- Let us also create another table to store the *marks* received by students in the different subjects and to store the *teacher's name* who teaches that subject.

Score

<u>Student id</u>	<u>Subject id</u>	marks	teacher
10	1	70	John
10	2	75	Olivia
11	1	80	John

- Both student's id and subject's id form *a composite primary key* – both can uniquely identify the different rows.

- So far, we have the following:

Student

<u>id</u>	Name	GPA	Department	Address
-----------	------	-----	------------	---------

Subjects

<u>Subject id</u>	Name
-------------------	------

Score

<u>Student id</u>	<u>Subject id</u>	marks	teacher
-------------------	-------------------	-------	---------

What is the problem in this design?

Score

<u>Student id</u>	<u>Subject id</u>	marks	teacher
10	1	70	John
10	2	75	Olivia
11	1	80	John

- If at anytime there are no students linked to Subject id 1 - *there will be no record for the teacher's name* of that subject (John) !
- What if we have 100 students taking certain exam – should we store the teacher's name 100 times ?!
- If teacher's name is changed (e.g., new assignment), every row referring to him/her *must be updated* - otherwise we will have data *inconsistency problem*.

Score

<u>Student id</u>	<u>Subject id</u>	marks	teacher
10	1	70	John
10	2	75	Olivia
11	1	80	John

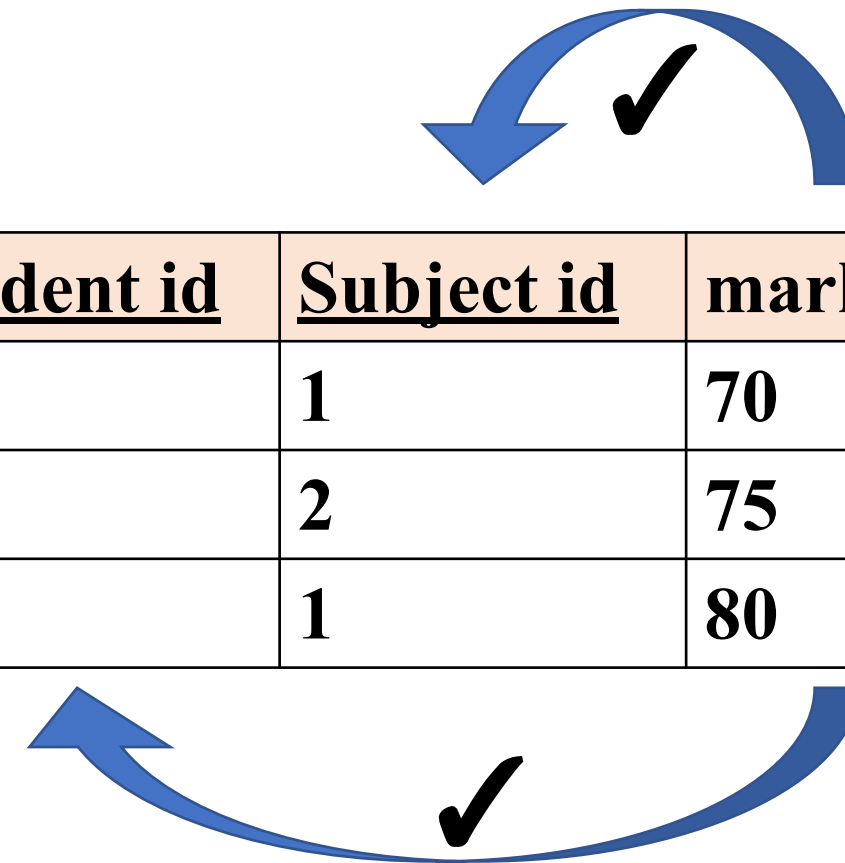
- These problems occurred because the teacher's name is *logically dependent* only on the subject (Subject id) - for Java it's John and for C++ it's Olivia.
- The teacher's name has no *logical relation* with the student id (*part of the composite primary key*).
- Second Normal Form (2NF) addresses such *Partial Dependency*

Score

<u>Student id</u>	<u>Subject id</u>	marks	teacher
10	1	70	John
10	2	75	Olivia
11	1	80	John

This is *Partial Dependency*, where an attribute in a table *logically* depends on a part of the composite primary key and not on the whole key.

Score



<u>Student id</u>	<u>Subject id</u>	marks	teacher
10	1	70	John
10	2	75	Olivia
11	1	80	John

The marks attribute *does not face such problem*, marks attribute logically depends on both the student id and the subject id.

Score

<u>Student id</u>	<u>Subject id</u>	marks	teacher
10	1	70	John
10	2	75	Olivia
11	1	80	John

One way to solve this partial dependency is to remove teacher's name from Score table *and add it to the Subject table.*

Score

<u>Student id</u>	<u>Subject id</u>	marks
10	1	70
10	2	75
11	1	80

Subject

<u>Subject id</u>	Name	Teacher
1	Java	John
2	C++	Olivia
3	PHP	Adam

Old
Design

Student

<u>id</u>	Name	GPA	Department	Address
-----------	------	-----	------------	---------

Subjects

<u>Subject id</u>	Name
-------------------	------

Score

<u>Student id</u>	<u>Subject id</u>	marks	teacher
-------------------	-------------------	-------	---------

New
Design

Student

<u>id</u>	Name	GPA	Department	Address
-----------	------	-----	------------	---------

Subjects

<u>Subject id</u>	Name	Teacher
-------------------	------	---------

Score

<u>Student id</u>	<u>Subject id</u>	marks
-------------------	-------------------	-------

- This **Catalog** table links the different mechanical parts with the warehouses that sell such parts. The catalog also holds the quantity each warehouse stores for the different parts and the address of such Warehouse.
- If the following table is not formalized, normalize it to follow the 2NF

Catalog

<u>Part ID</u>	<u>Warehouse</u>	Quantity	Warehouse Address
42	Chicago	200	M street
33	Chicago	100	M street
39	NYC	300	B street

*The primary key is a composite key: **Part ID + the warehouse name.**

Catalog

<u>Part ID</u>	<u>Warehouse</u>	Quantity	Warehouse Address
42	Chicago	200	M street
33	Chicago	100	M street
39	NYC	300	B street

- The “Quantity” attribute is a fact about the both the part and the warehouse.
- The “Warehouse Address” attribute is a fact about the Warehouse, not logically related to the Part ID – *Partial dependency* problem, thus the Catalog table is not in 2NF.

Catalog

<u>Part ID</u>	<u>Warehouse</u>	Quantity	Warehouse Address
42	Chicago	200	M street
33	Chicago	100	M street
39	NYC	300	B street

Split the table into *two smaller tables* to resolve *the partial dependency*, each sub-table is now in 2NF

Catalog

<u>Part ID</u>	<u>Warehouse</u>	Quantity
42	Chicago	200
33	Chicago	100
39	NYC	300

Warehouse

<u>Warehouse</u>	Warehouse Address
Chicago	M street
NYC	B street

Third Normal Form (3NF)

Any field that logically depends not only on the primary key but *also on another non-key field* must be moved to another table

For a table to be in the 3NF, it should follow the following rules:

1. It should be first in the 2NF.
2. It doesn't have *Transitive Dependency*.

All transitive dependencies must be removed; a non-key attribute may not be functionally dependent on another non-key attribute.

Imagine the following table for customers:

<u>Customer ID</u>	Name	city
--------------------	------	------

- The different attributes (name and city) are *logically related* to the customer (in terms of the customer ID), as we described in 2NF

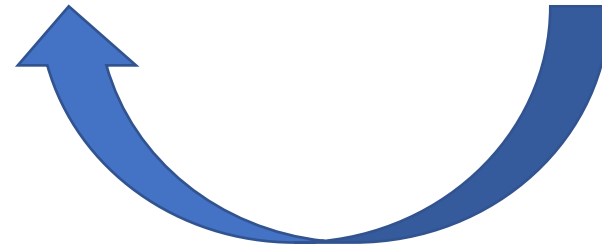
But, Imagine the following updated table for customers:

<u>Customer ID</u>	Name	City	PostalCode
--------------------	------	------	------------

What is the problem in this design?

But, Imagine the following updated table for customers:

<u>Customer ID</u>	Name	City	PostalCode
--------------------	------	------	------------



- The postal code *logically depends only on the city*, not one the name or the id of the customer.

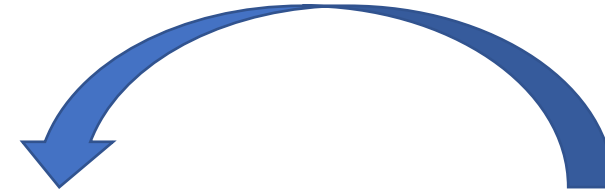
Imagine the following table for customers.

<u>Customer ID</u>	Name	City	PostalCode
--------------------	------	------	------------

- But the city is a regular attribute – *not a primary key or even a part of the primary key.*
- This is Transitive Dependency: When a *non-primary key attribute* depends on other *non-primary key attribute* rather than depending upon the full primary key.

Customer

<u>Customer ID</u>	Name	City	PostalCode
--------------------	------	------	------------



To solve the transitive dependency, take out the *City* and *PostalCode* in a separate table.

Customer

<u>Customer ID</u>	Name	City
--------------------	------	------



City

<u>City</u>	PostalCode
-------------	------------



Customer

<u>Customer ID</u>	Name	City	PostalCode
--------------------	------	------	------------



To solve the transitive dependency, take out the *City* and *PostalCode* in a separate table.

Customer

<u>Customer ID</u>	Name	City
--------------------	------	------



City

<u>City</u>	PostalCode
-------------	------------




City attribute is *primary key* and *foreign key* referencing City of Customer's table

- If the following table is not formalized, normalize it to follow the 3NF

Employee

<u>EmployeeID</u>	Dept.Name	Dept._Locaiton
234	Finance	Boston
223	Finance	Boston
399	Operations	Washington


Employee



<u>EmployeeID</u>	Dept.Name	Dept._Locaiton
234	Finance	Boston
223	Finance	Boston
399	Operations	Washington


- The primary key of this table is EmployeeID, but the Department location is a function (logically dependent) of the Department Name, *which is not a key*.
- This is *Transitive Dependency* problem (not in 3NF)

Employee



<u>EmployeeID</u>	Dept.Name	Dept._Locaiton
234	Finance	Boston
223	Finance	Boston
399	Operations	Washington

Problems:

- Department location is repeated for each employee
 - If department location is changed, we need to update every record, and this may leads to data inconsistent if any record/row is missed
- 

*The original
Employee table*

<u>EmployeeID</u>	Dept.Name	Dept._Locaiton
234	Finance	Boston
223	Finance	Boston
399	Operations	Washington



Employee

<u>EmployeeID</u>	Dept.Name
234	Finance
223	Finance
399	Operations

Department

<u>Dept.Name</u>	Dept_Location
Finance	Boston
Operations	Washington

Another Employee table with transitive dependency:



- As discussed, the main goal of normalization is to reduce data redundancy (duplication), the higher the normalization the lower the redundancy of data in the tables.
- However, in some scenarios, normalized designs are not fast enough for real-world needs. Imagine the following: A user with several email addresses. In fully normalized model, we will design a table for the users and another table for the emails.
- If a core (frequently asked) query requires to display the user's info along with their primary and secondary emails, this will require join operators (as will be discussed in the following units) to be used - affecting the overall performance (e.g., speed) of the query.
- In such scenarios, we can denormalize tables (combine smaller tables) in order to gain query performance on the cost of duplicating data (substantially in some cases). In this case, we can expand the user's table with two columns (one for the primary email address and the other for the secondary email address).

Let's summarize what we discussed so far:

