Introduction to RDBMS

Database Management System (DBMS)

- DBMS contains information about a particular enterprise
  - Collection of interrelated data
  - Set of programs to access the data
  - An environment that is both *convenient* and *efficient* to use
- Database Applications:
  - Banking: transactions
  - Airlines: reservations, schedules
  - Universities: registration, grades
  - Sales: customers, products, purchases
  - Online retailers: order tracking, customized recommendations
  - Manufacturing: production, inventory, orders, supply chain
  - Human resources: employee records, salaries, tax deductions
- Databases can be very large.
- Databases touch all aspects of our lives

University Database Example

- Application program examples
  - Add new students, instructors, and courses
  - Register students for courses, and generate class rosters
  - Assign grades to students, compute grade point averages (GPA) and generate transcripts
- In the early days, database applications were built directly on top of file systems

Drawbacks of using file systems to store data

- Data redundancy and inconsistency
  - Multiple file formats, duplication of information in different files
- Difficulty in accessing data
  - Need to write a new program to carry out each new task
- Data isolation — multiple files and formats
- Integrity problems
  - Integrity constraints (e.g., account balance > 0) become “buried” in program code rather than being stated explicitly
  - Hard to add new constraints or change existing ones

Drawbacks of using file systems to store data (Cont.)

- Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
  - Concurrent access needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
  - Hard to provide user access to some, but not all, data

Database systems offer solutions to all the above problems

Levels of Abstraction

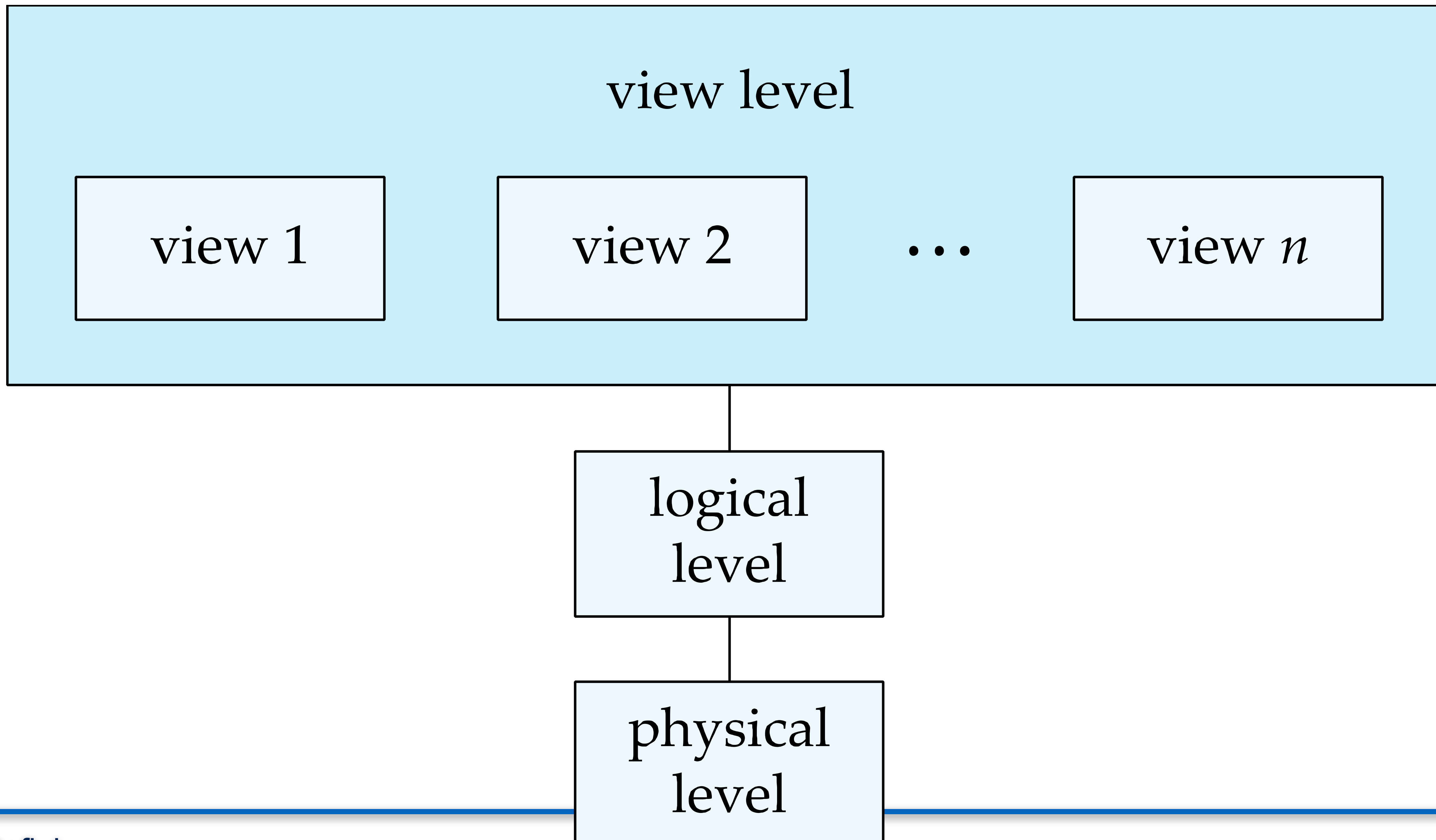
- **Physical level:** describes how a record (e.g., customer) is stored.
- **Logical level:** describes data stored in database, and the relationships among the data.

  type *instructor* = **record**
  ID : string;
  name : string;
  dept_name : string;
  salary : integer;
  end;

- **View level:** application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

View of Data

An architecture for a database system



Instances and Schemas

- Similar to types and variables in programming languages
- **Schema** – the logical structure of the database
  - Example: The database consists of information about a set of customers and accounts and the relationship between them
  - Analogous to type information of a variable in a program
  - **Physical schema**: database design at the physical level
  - **Logical schema**: database design at the logical level
- **Instance** – the actual content of the database at a particular point in time
  - Analogous to the value of a variable
- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema
  - Applications depend on the logical schema
  - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

Database

- A database consists of multiple relations
- Information about an enterprise is broken up into parts

  instructor
  student
  advisor

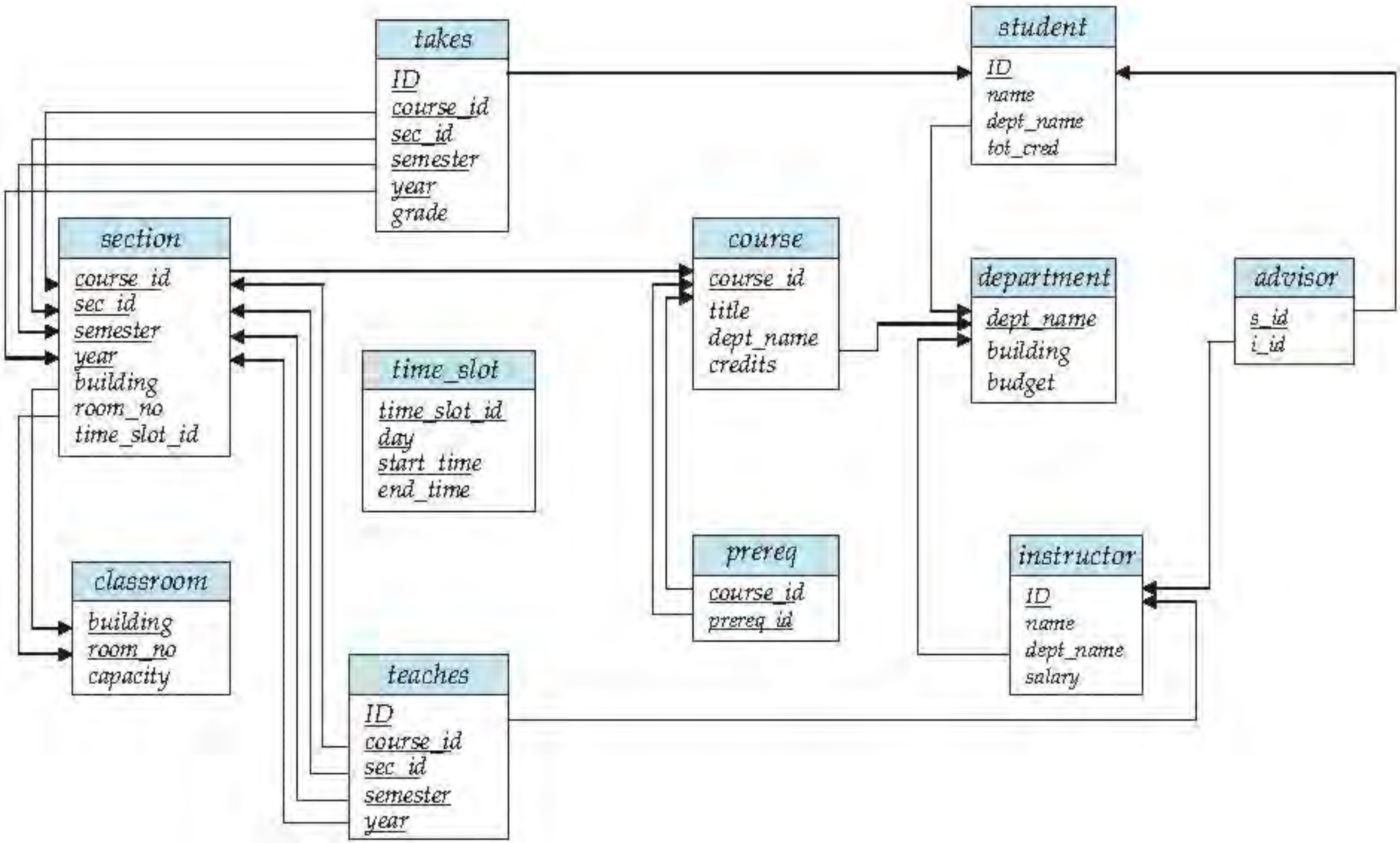
- Bad design:
  univ (*instructor -ID, name, dept_name, salary, student_Id*, ..)
  results in
  - repetition of information (e.g., two students have the same instructor)
  - the need for null values (e.g., represent an student with no advisor)
- Normalization theory (Chapter 7) deals with how to design “good” relational schemas

Keys

- Let K ⊆ R
- *K* is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
  - Example: {*ID*} and {ID,name} are both superkeys of *instructor.*
- Superkey *K* is a **candidate key** if *K* is minimal

  Example: {*ID*} is a candidate key for *Instructor*
- One of the candidate keys is selected to be the **primary key**.
  - which one?
- **Foreign key** constraint: Value in one relation must appear in another
  - **Referencing** relation
  - **Referenced** relation

Schema Diagram for University Database



Normalization

Normalization

- Transformation of improperly designed tables into tables with better structure
- Iterative process
- Extent of normalization depends upon the characteristics (application)
  - 1NF
  - 2NF
  - 3NF
  - ...

First Normal Form

- Domain is **atomic** if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **first normal form** if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - Example: Set of accounts stored with each customer, and set of owners stored with each account
  - We assume all relations are in first normal form (and revisit this in Chapter 22: Object Based Databases)

First Normal Form (Cont'd)

- Atomicity is actually a property of how the elements of the domain are used.
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
  - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
  - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

2nd Normal Form (Cont'd)

- If it is 1NF, and every non-key attribute is fully functionally dependent on the primary key.

- Students (IDSt, StudentName, IDProf, ProfName, Grade)
  - The Attributes IDSt, IDProf are keys
  - All attributes are single valued (1NF)
- The following functional dependencies exist:
  - The attribute ProfName is functionally dependent on IDProf (IDProf -> ProfiName)
  - The attribute StudentName is functionally dependent on IDSt (IDSt→StudentName)
  - The attribute Grade is fully functionally dependent on IDSt and IDProf (IDSt, IDProf → Grade)

2NF

Students

IDSt	LastName	IDProf	Prof	Grade
1	Mueller	3	Schmid	5
2	Meier	2	Borner	4
3	Tobler	1	Bernasconi	6

Result after normalisation

Students

ID	LastName
1	Mueller
2	Meier
3	Tobler

Professors

IDProf	Professor
1	Bernasconi
2	Borner
3	Schmid

Grades

IDStIDProf	Grade	
1	3	5
2	2	4
3	1	6

- 1NF since all attributes are single value
- Not 2NF.
- If student 1 leaves University and the tuple is delete we lose info about Schmidt
- Decompose
  - Add Professor
  - Grade – combine students & Prof

3NF

- If it is 2NF, and no non-key attribute is transitively dependent on the primary key. (All attributes are determined by the primary key)
- Bank uses the following relation
  - The ID is the key
  - All attributes are single valued (1NF)
  - Table is also in 2NF

- The following functional dependencies exist:
  - Name, Account_No, Bank_CodeNo are functionally dpenedent on ID
    - (ID → Name, Account_No, Bank_code_No)
    - Bank is functionality dependent on Bank_code_name
      - (Bank_Code_No → Bank)

Vendor

ID	Name	Account_No	Bank_Code_No	Bank

3NF

Result after normalisation

Vendor

ID	Name	Account_No	Bank_Code_No

Bank

Bank_Code_No	Bank

- There is a transitive dependency between Bank_Code_No and Bank.
- Bank_Code_No is not the primary key of this relation.