Database Systems

Introduction

Dr P Sreenivasa Kumar

Professor

CS&E Department

I I T Madras

Introduction

What is a Database?

A collection of related pieces of data:

- Representing/capturing the information about a real-world enterprise or part of an enterprise.
- Collected and maintained to serve specific data management needs of the enterprise.
- Activities of the enterprise are supported by the database and continually update the database.

An Example

University Database:

Data about students, faculty, courses, research-laboratories, course registration/enrollment etc.

Reflects the state of affairs of the academic aspects of the university.

Purpose: To keep an accurate track of the academic activities of the university.

Database Management System (DBMS)

A *general purpose* software system enabling:

- Creation of large disk-resident databases.
- Posing of data retrieval queries in a standard manner.
- Retrieval of query results efficiently.
- Concurrent use of the system by a large number of users in a consistent manner.
- Guaranteed availability of data irrespective of system failures.

OS File System Storage Based Approach

- Files of records – used for data storage
  - data redundancy – wastage of space
  - maintaining consistency becomes difficult
- Record structures – hard coded into the programs
  - structure modifications – hard to perform
- Each different data access request (a query)
  - performed by a separate program
  - difficult to anticipate all such requests
- Creating the system
  - requires a lot of effort
- Managing concurrent access and failure recovery are difficult

DBMS Approach

DBMS

- separation of data and metadata
- flexibility of changing metadata
- program-data independence

Data access language

- standardized – SQL
- ad-hoc query formulation – easy

System development

- less effort required
- concentration on logical level design is enough
- components to organize data storage

  process queries, manage concurrent access,
  recovery from failures, manage access control

  are all available

Data Model

Collection of conceptual tools to describe the database at a certain level of abstraction.

- *Conceptual Data Model*
  - a high level description
  - useful for requirements understanding.
- *Representational Data Model*
  - describing the logical representation of data without giving details of physical representation.
- *Physical Data Model*
  - description giving details about record formats, file structures etc.

E/R (Entity/Relationship) Model

- A conceptual level data model.
- Provides the concepts of *entities*, *relationships* and *attributes*.

The University Database Context

Entities: *student*, *faculty member*, *course*, *departments* etc.

Relationships: *enrollment* relationship between student & course,
employment relationship between faculty
member, department etc.

Attributes: *name*, *rollNumber*, *address* etc., of *student* entity,
name, *empNumber*, *phoneNumber* etc., of faculty
entity etc.

More details will be given in the E/R Model Module.

Representational Level Data Model

Relational Model : Provides the concept of a relation.

In the context of university database:

Relation name

Attributes

student

SName	**RollNumber**	**JoiningYear**	**BirthDate**	**Program**	**Dept**
Sriram	CS04B123	2004	15Aug1982	BTech	CS
⋮	⋮	⋮	⋮	⋮	⋮

Data tuple

Relation scheme: Attribute names of the relation.

Relation data/instance: set of data tuples.

More details will be given in Relational Data Model Module.

Data versus Schema or Meta-Data

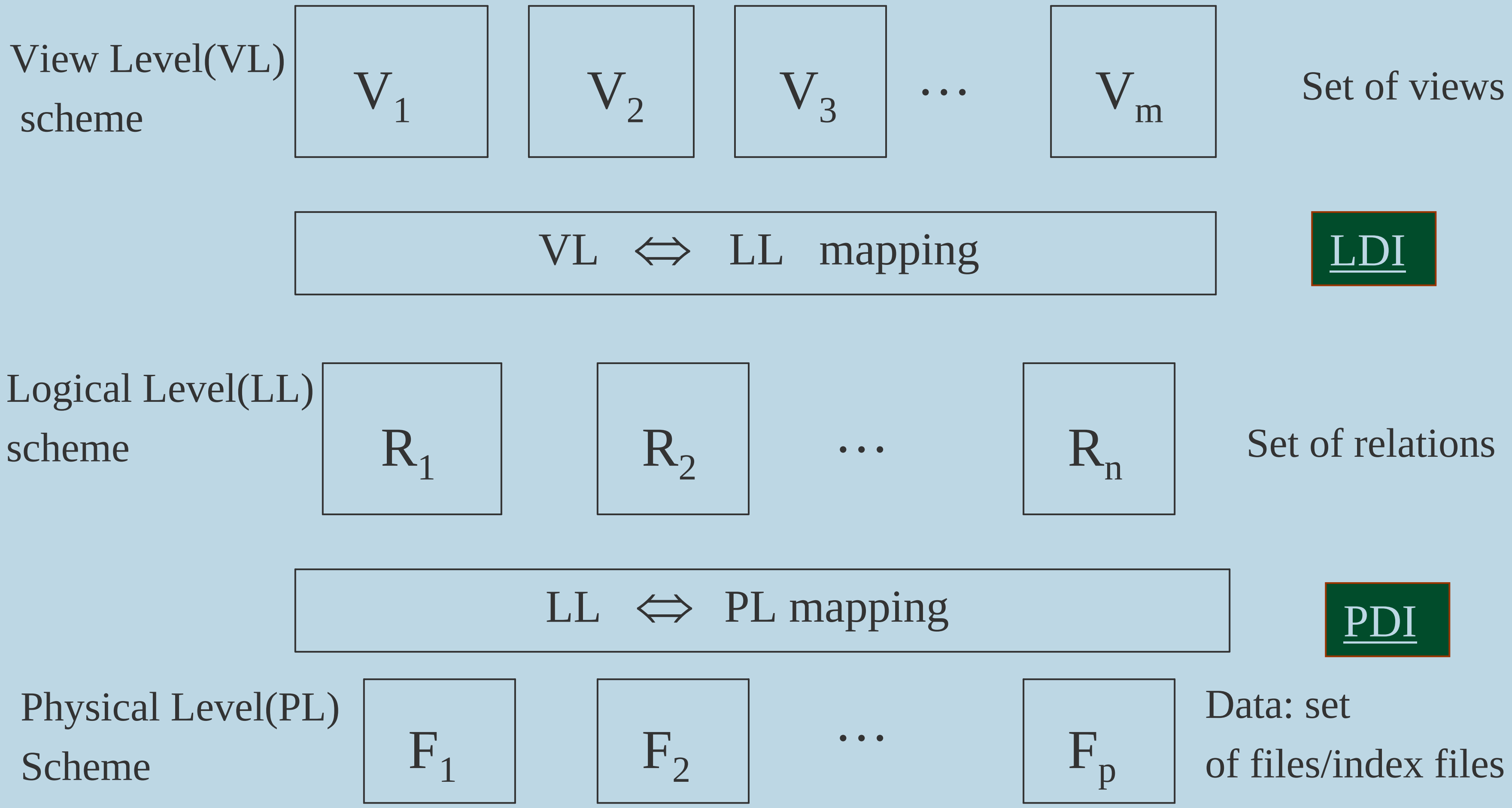
- DBMS is generic in nature
    - not tied to a single database
    - capable of managing several databases at a time
- Data and schema are stored separately.

- In RDBMS context:

  Schema – table names, attribute names with their data types for each table and constraints etc.

- Database definition – setting up the skeleton structure
- Database Loading/populating – storing data

Abstraction Levels in a DBMS: Three-Schema Architecture



Three-schema Architecture(1/2)

View Level Schema

Each view describes an aspect of the database relevant to a particular group of users.

For instance, in the context of a library database:

- Books Purchase Section
- Issue/Returns Management Section
- Users Management Section

Each section views/uses a portion of the entire data.

Views can be set up for each section of users.

Three-schema Architecture(2/2)

Logical Level Schema

- Describes the logical structure of the entire database.
- No physical level details are given.

Physical Level Schema

- Describes the physical structure of data in terms of record formats, file structures, indexes etc.

Remarks

- Views are optional
  - Can be set up if the DB system is very large and if easily identifiable user-groups exist
- The logical scheme is essential
- Modern RDBMS's hide details of the physical layer

Physical Data Independence

The ability to modify physical level schema without affecting the logical or view level schema.

Performance tuning – modification at physical level
creating a new index etc.

Physical Data Independence – modification is localized

- achieved by suitably modifying PL-LL mapping.
- a very important feature of modern DBMS.

Three Schema Arch

Logical Data Independence

The ability to change the logical level scheme without affecting the view level schemes or application programs

Adding a new attribute to some relation
- no need to change the programs or views that don’t require to use the new attribute

Deleting an attribute
- no need to change the programs or views that use the remaining data
- view definitions in VL-LL mapping only need to be changed for views that use the deleted attribute

Three-schema Architecture

Development Process of a Database System (1/2)

Step 1. Requirements collection

- *Data model requirements*
  - various pieces of data to be stored and the interrelationships.
  - presented using a conceptual data model such as E/R model.
- *Functional requirements*
  - various operations that need to be performed as part of running the enterprise.
    - acquiring a new book, enrolling a new user, issuing a book to the user, recording the return of a book etc.

Development process of a database system (2/2)

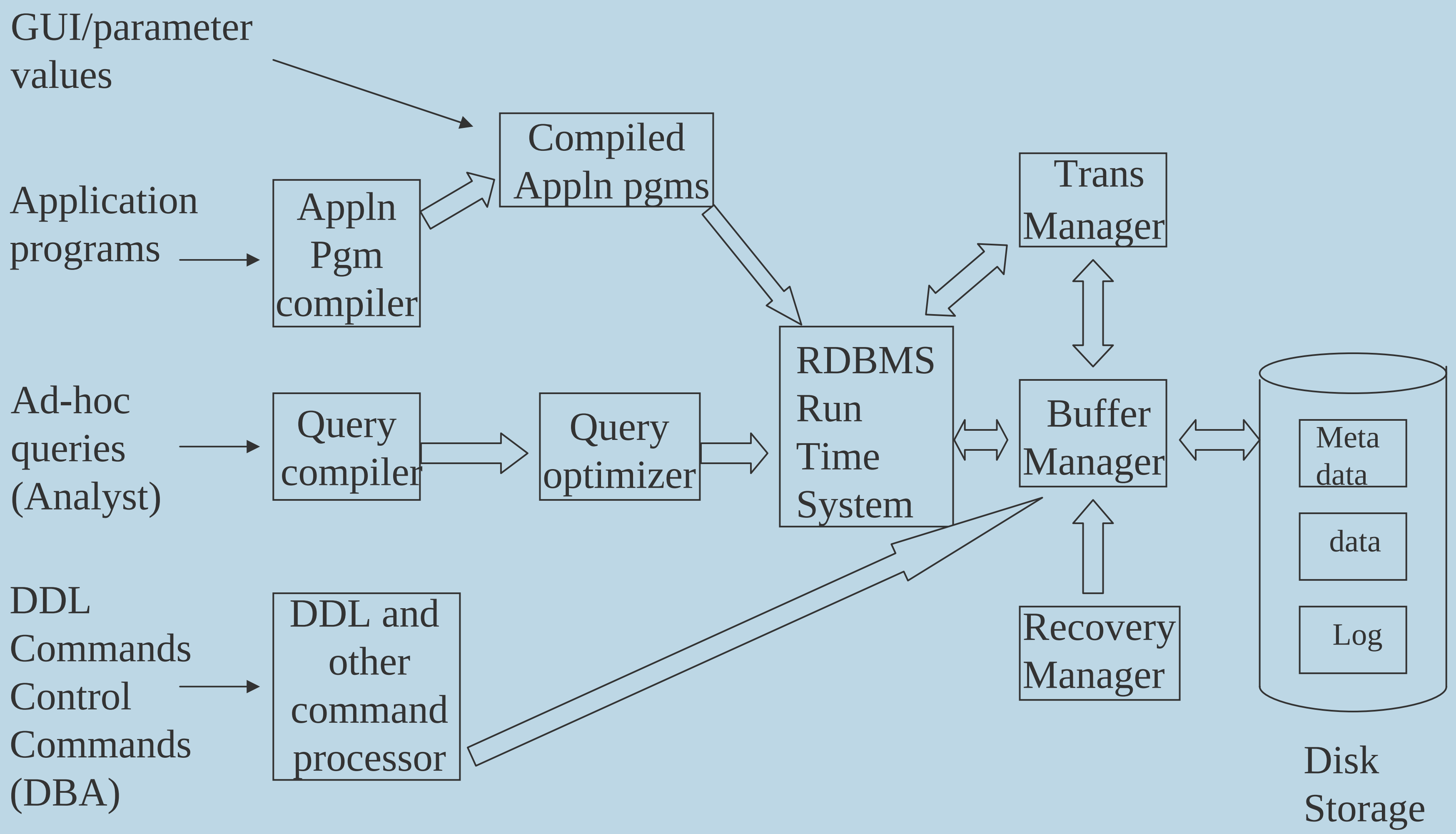
Step 2. Convert the data model into a representational level model

- typically relational data model.
- choose an RDBMS system and create the database.

Step 3. Convert the functional requirements into application programs

- programs in a high-level language that use embedded SQL to interact with the database and carry out the required tasks.

Architecture of an RDBMS system



Architecture Details (1/3)

Disk Storage:

Meta-data – schema

- table definition, view definitions, mappings

Data – relation instances, index structures

statistics about data

Log – record of database update operations

essential for failure recovery

DDL and other command processor:

Commands for relation scheme creation

Constraints setting

Commands for handling authorization and data access control

Architecture Details (2/3)

Query compiler

$$\begin{tikzcd}[column sep=small, row sep=tiny] & \text{SQL adhoc queries} \\ \text{Compiles} \arrow[ur, dash] \arrow[dr, dash] & \\ & \text{update / delete commands} \end{tikzcd}$$

Query optimizers

Selects a near optimal plan for executing a query
- relation properties and index structures are utilized

Application Program Compiler

Preprocess to separate embedded SQL commands
Use host language compiler to compile rest of the program
Integrate the compiled program with the libraries for SQL commands supplied by RDBMS

Architecture Details (3/3)

RDBMS Run Time System:
- Executes Compiled queries, Compiled application programs
- Interacts with Transaction Manager, Buffer Manager

Transaction Manager:
- Keeps track of start, end of each transaction
- Enforces concurrency control protocols

Buffer Manager:
- Manages disk space
- Implements paging mechanism

Recovery Manager:
- Takes control as restart after a failure
- Brings the system to a consistent state before it can be resumed

Roles for people in an Info System Management (1/2)

Naive users / Data entry operators

- Use the GUI provided by an application program
- Feed-in the data and invoke an operation
  - e.g., person at the train reservation counter, person at library issue / return counter
- No deep knowledge of the IS required

Application Programmers

- Embed SQL in a high-level language and develop programs to handle functional requirements of an IS
- Should thoroughly understand the logical schema or relevant views
- Meticulous testing of programs - necessary

Roles for people in an Info System management (2/2)

Sophisticated user / data analyst:
- Uses SQL to generate answers for complex queries

DBA (Database Administrator)
- Designing the logical scheme
- Creating the structure of the entire database
- Monitor usage and create necessary index structures to speed up query execution
- Grant / Revoke data access permissions to other users etc.

Text Books

- Ramez Elmasri and Shamkant B Navathe, *Fundamentals of Database Systems*, 3rd Edition, Addison Wesley, 2000.
- Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*, 3rd Edition, McGraw Hill, 2003.
- A Silberschatz, H F Korth and S Sudarshan, *Database System Concepts*, 5th Edition, 2006.
- H Garcia-Molina, J D Ullman, and Jennifer Widom, Database Systems-The Complete Book, Pearson Education, 2004.

Entity-Relationship (E/R) Model

Dr P Sreenivasa Kumar

Professor

CS&E Dept I I T Madras

Entity-Relationship (E/R) Model

- Widely used conceptual level data model
  - proposed by Peter P Chen in 1970s
- Data model to describe the database system at the requirements collection stage
  - high level description.
  - easy to understand for the enterprise managers.
  - rigorous enough to be used for system building.
- Concepts available in the model
  - entities and attributes of entities.
  - relationships between entities.
  - diagrammatic notation.

Entities

- *Entity* - a thing (animate or inanimate) of independent physical or conceptual existence and *distinguishable*.
  In the University database context, an individual *student*, *faculty member*, a *class room*, a *course* are entities.

- *Entity Set* or *Entity Type*-
  Collection of entities all having the same properties.
  Student entity set – collection of all *student* entities.
  Course entity set – collection of all *course* entities.

Attributes

Each entity is described by a set of attributes/properties.

student entity

- *StudName* – name of the student.
- *RollNumber* – the roll number of the student.
- *Sex* – the gender of the student etc.

All entities in an Entity set/type have the same set of attributes.

Chosen set of attributes – amount of detail in modeling.

Types of Attributes (1/2)

- Simple Attributes
  - having atomic or indivisible values.

    `example`: *Dept* – a string

    PhoneNumber – an eight digit number
- Composite Attributes
  - having several components in the value.

    `example`: *Qualification* with components

    (*DegreeName*, *Year*, *UniversityName*)
- Derived Attributes
  - Attribute value is dependent on some other attribute.

    `example`: *Age* depends on *DateOf Birth*.

    So age is a derived attribute.

Types of Attributes (2/2)

- Single-valued
  - having only one value rather than a set of values.
  - for instance, *PlaceOfBirth* – single string value.
- Multi-valued
  - having a set of values rather than a single value.
  - for instance, *CoursesEnrolled* attribute for student
    EmailAddress attribute for student
    PreviousDegree attribute for student.
- Attributes can be:
  - simple single-valued, simple multi-valued,
  - composite single-valued or composite multi-valued.

Diagrammatic Notation for Entities

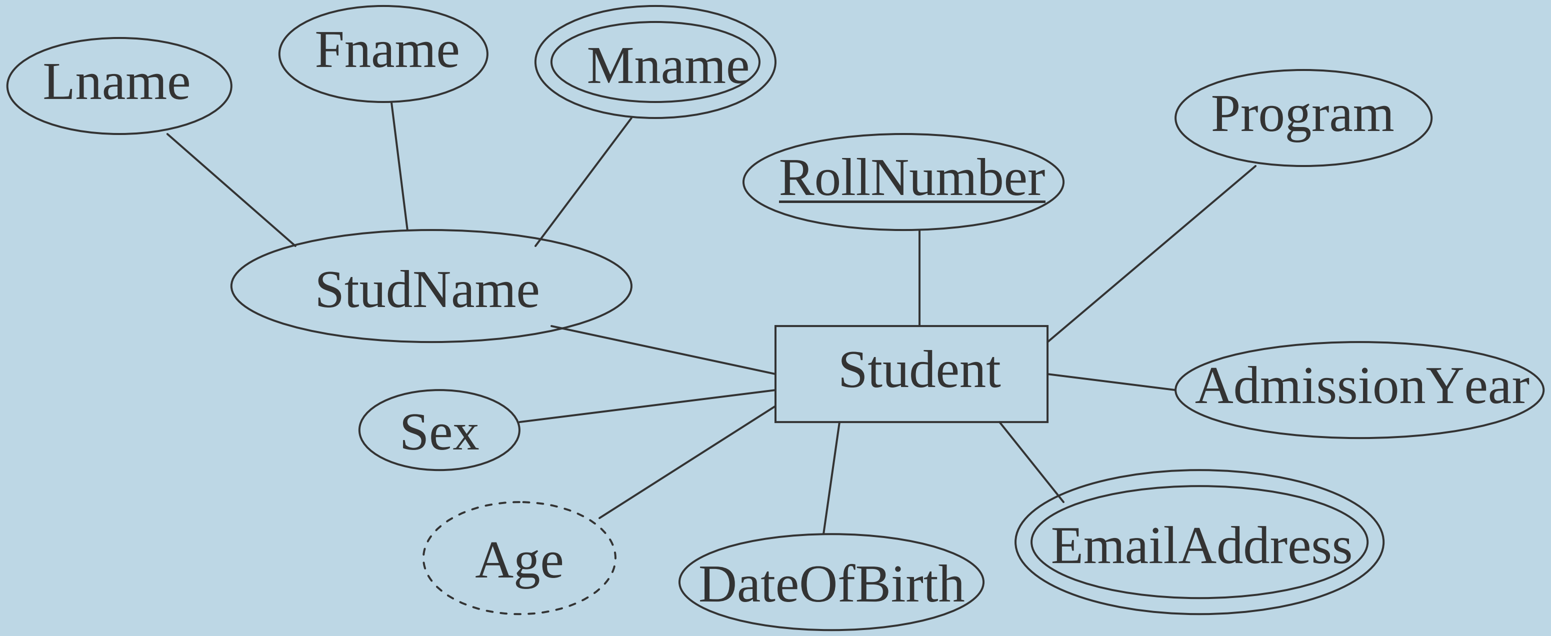
entity - rectangle

attribute - ellipse connected to rectangle

multi-valued attribute - double ellipse

composite attribute - ellipse connected to ellipse

derived attribute - dashed ellipse



Domains of Attributes

Each attribute takes values from a set called its *domain*

For instance, *studentAge* – {17,18, …, 55}

HomeAddress – character strings of length 35

Domain of composite attributes –

cross product of domains of component attributes

Domain of multi-valued attributes –

set of subsets of values from the basic domain

Entity Sets and Key Attributes

- *Key* – an attribute or a collection of attributes whose value(s) uniquely identify an entity in the entity set.
- For instance,
  - *RollNumber* - Key for *Student* entity set
  - *EmpID* - Key for *Faculty* entity set
  - *HostelName*, *RoomNo* - Key for *Student* entity set (assuming that each student gets to stay in a single room)
- A key for an entity set may have more than one attribute.
- An entity set may have more than one key.
- Keys can be determined only from the meaning of the attributes in the entity type.
  - Determined by the designers

Relationships

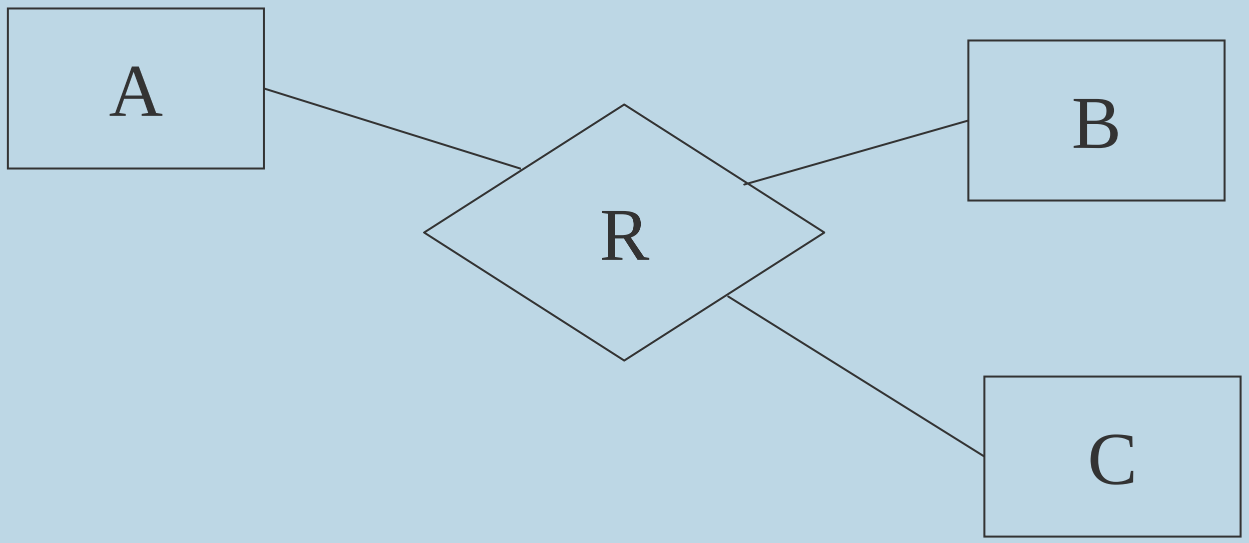
- When two or more entities are associated with each other, we have an instance of a *Relationship*.
- E.g.: *student* Ramesh *enrolls* in Discrete Mathematics *course*
- Relationship *enrolls* has *Student* and *Course* as the *participating* entity sets.
- Formally, $enrolls \subseteq Student \times Course$
    - $(s,c) \in enrolls \Leftrightarrow$ Student ‘s’ has enrolled in Course ‘c’
    - Tuples in *enrolls* – relationship instances
    - *enrolls* is called a relationship Type/Set.

Degree of a relationship

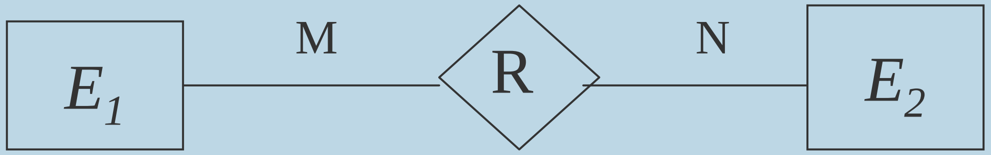
- *Degree* : the number of participating entities.
    - Degree 2: *binary*
    - Degree 3: *ternary*
    - Degree n: *n-ary*
- Binary relationships are very common and widely used.

Diagrammatic Notation for Relationships

- Relationship – diamond shaped box
  - Rectangle of each participating entity is connected by a line to this diamond. Name of the relationship is written in the box.



Binary Relationships and Cardinality Ratio



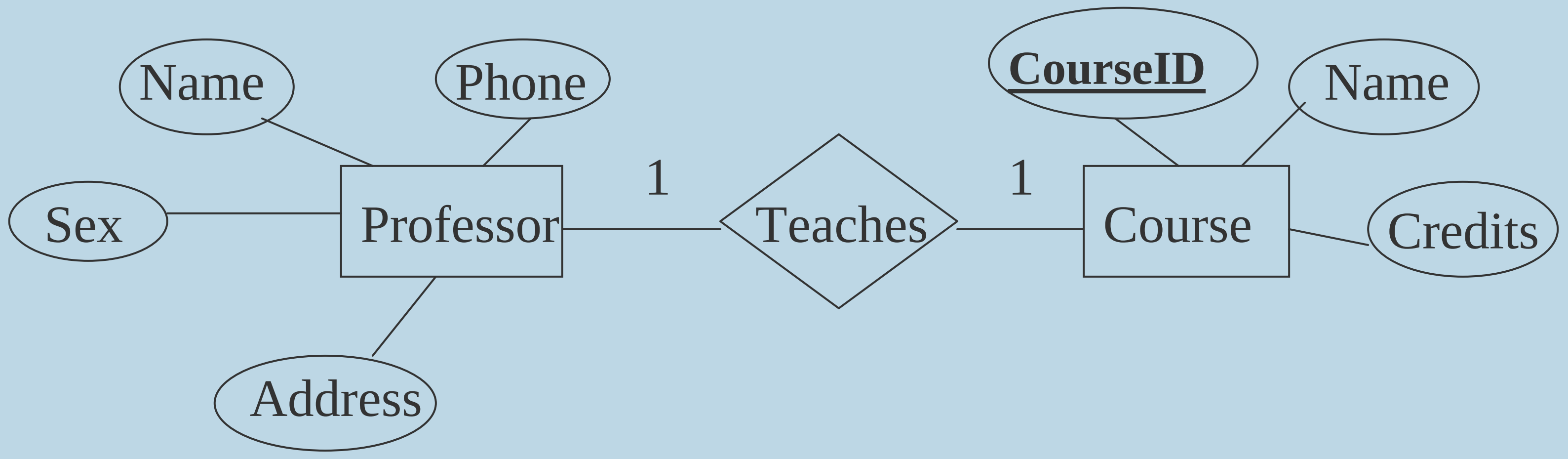
- The number of entities from E_2 that an entity from E_1 can possibly be associated thru R (and vice-versa) determines the *cardinality ratio* of R.

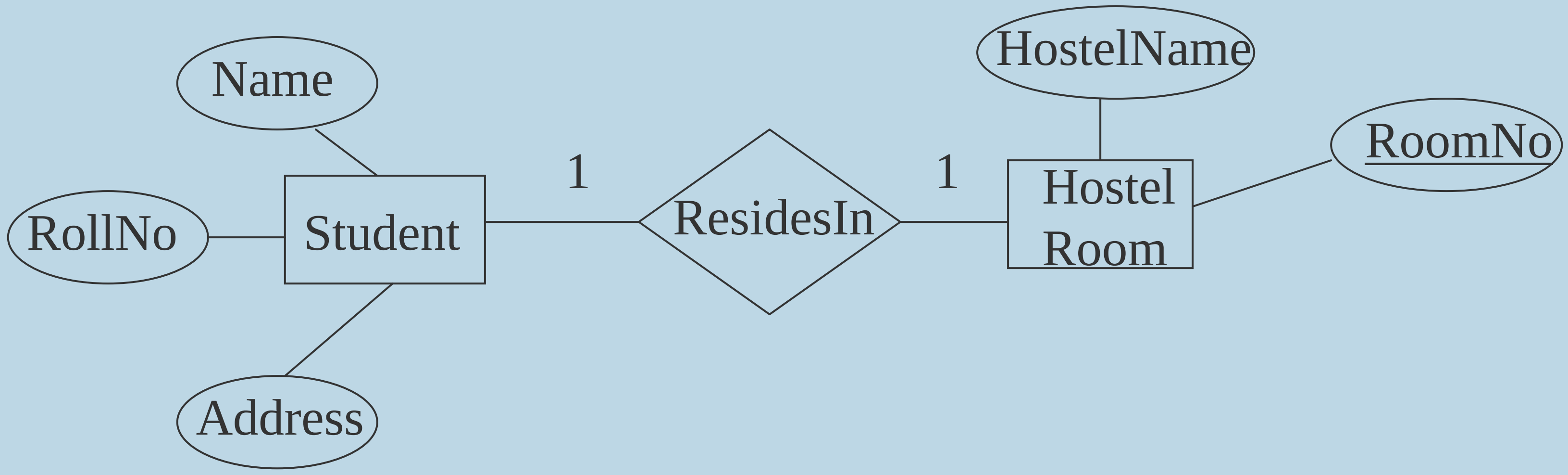
- Four possibilities are usually specified:
  - *one-to-one (1:1)*
  - *one-to-many (1:N)*
  - *many-to-one (N:1)*
  - *many-to-many (M:N)*

Cardinality Ratios

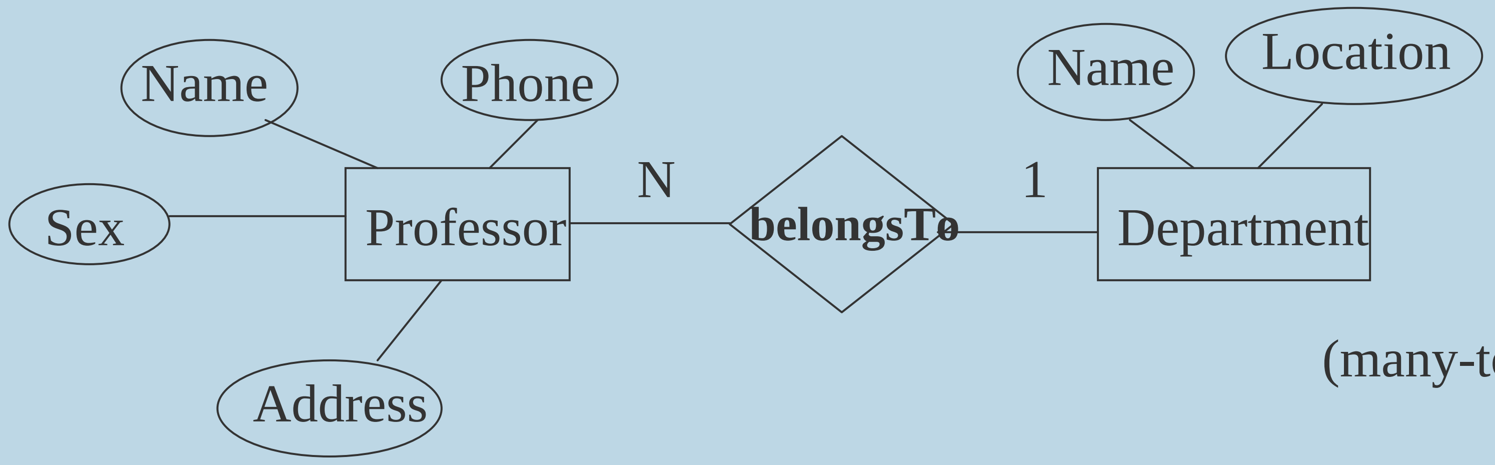
• *One-to-one:*	An E_1 entity may be associated with at most one E_2 entity and similarly an E_2 entity may be associated with at most one E_1 entity.
• *One-to-many:*	An E_1 entity may be associated with many E_2 entities whereas an E_2 entity may be associated with at most one E_1 entity.
• *Many-to-one:*	… (similar to above)
• *Many-to-many:*	Many E_1 entities may be associated with a single E_2 entity and a single E_1 entity may be associated with many E_2 entities.

Cardinality Ratio – example (*one-to-one*)

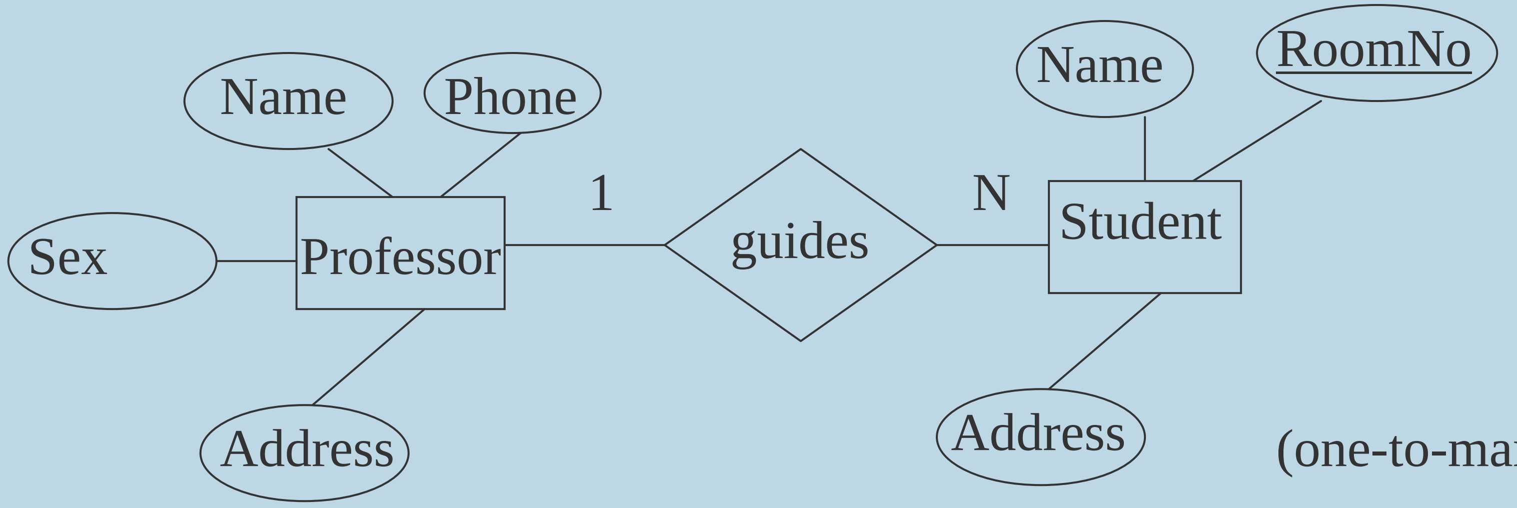




Cardinality Ratio – example (*many-to-one/one-to-many*)

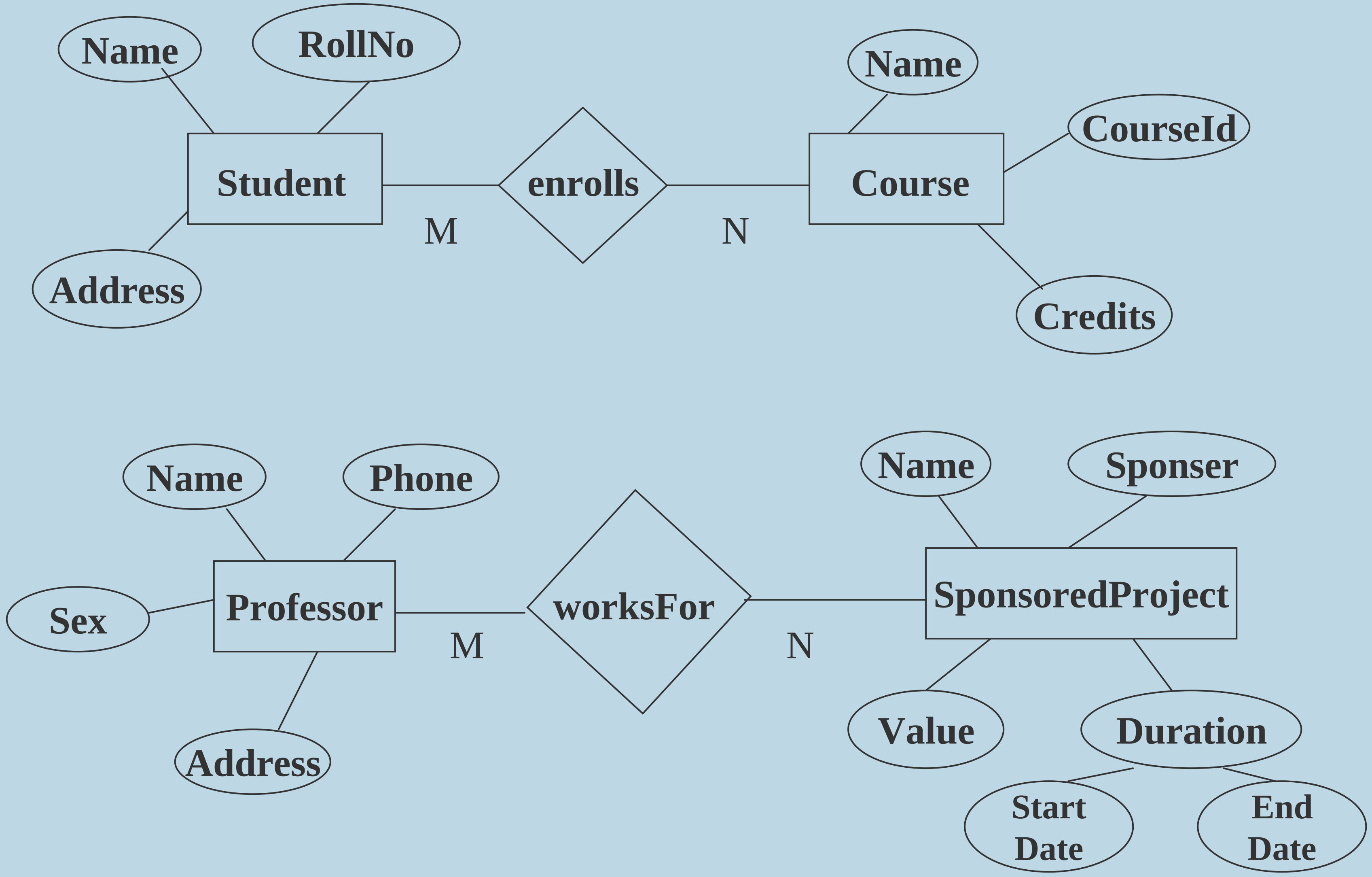


(many-to-one)



(one-to-many)

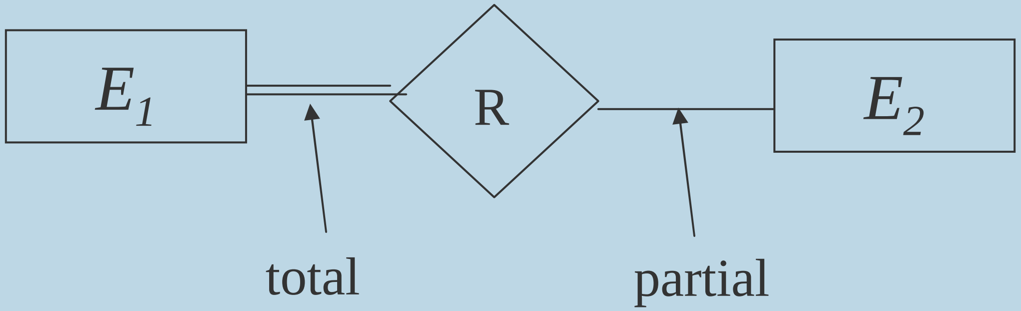
Cardinality Ratio – example (*many-to-many*)



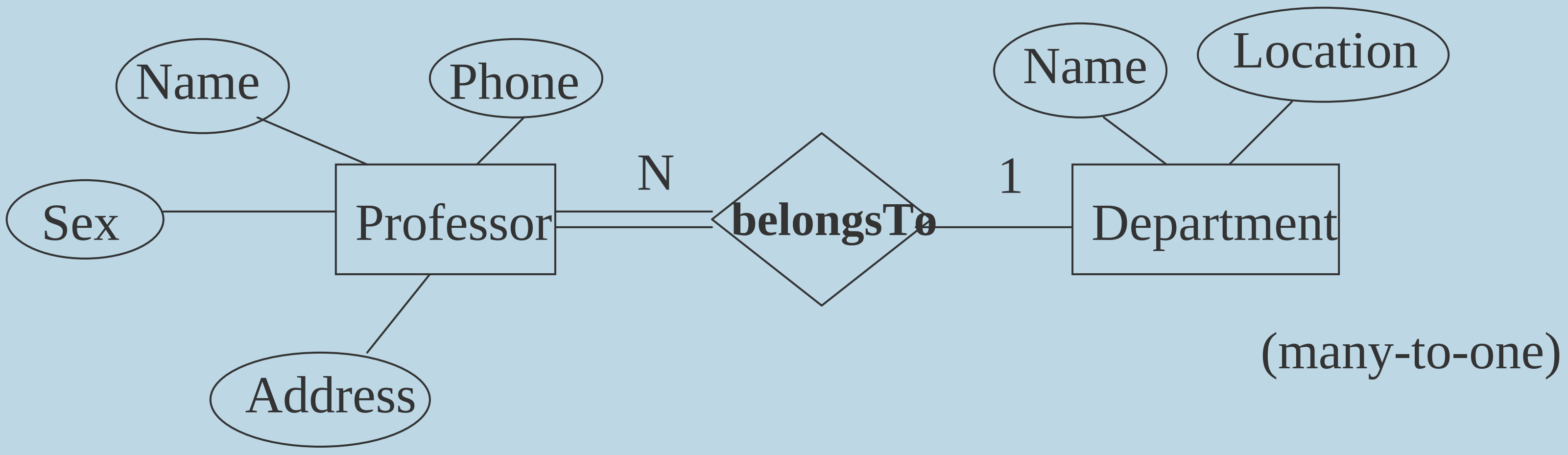
Participation Constraints

- An entity set may participate in a relation either *totally* or *partially*.
  - *Total participation*: Every entity in the set is involved in some association (or tuple) of the relationship.
  - *Partial participation*: Not all entities in the set are involved in association (or tuples) of the relationship.

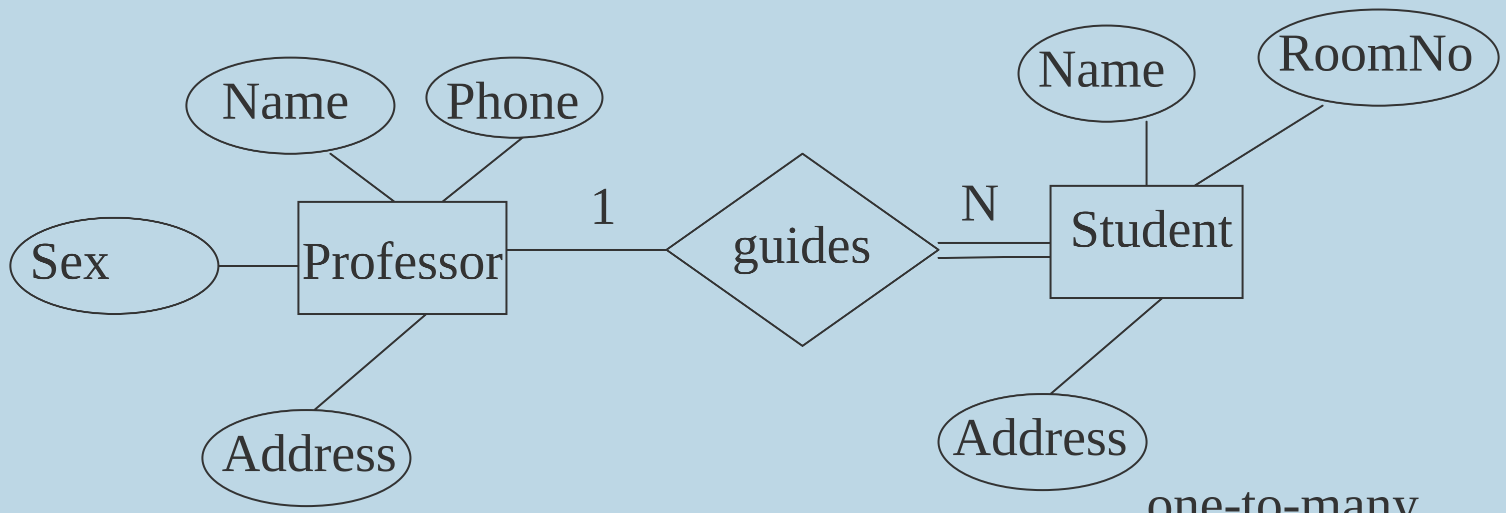
Notation:



Example of total/partial Participation



(many-to-one)

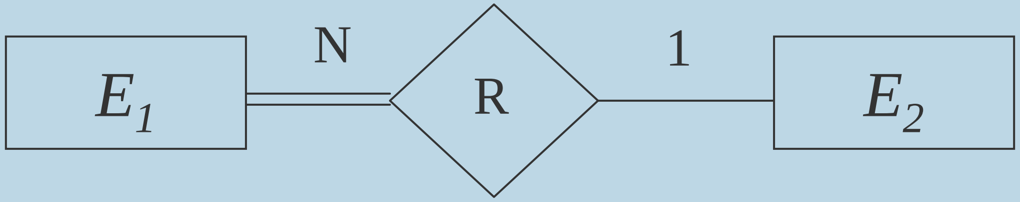


one-to-many

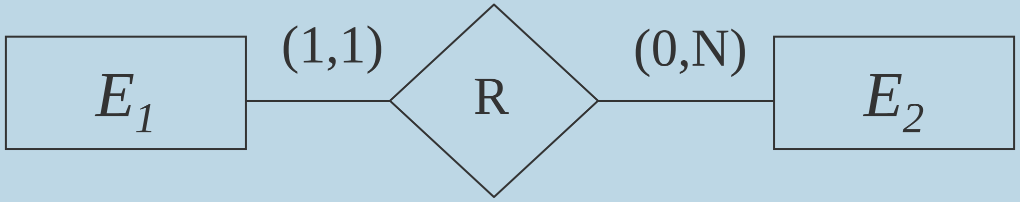
Structural Constraints

- Cardinality Ratio and Participation Constraints are together called *Structural Constraints*.
- They are called *constraints* as the *data* must satisfy them to be consistent with the requirements.
- *Min-Max notation*: pair of numbers (*m*,*n*) placed on the line connecting an entity to the relationship.
- *m*: the minimum number of times a particular entity *must appear* in the relationship tuples at any point of time
  - 0 – partial participation
  - ≥ 1 – total participation
- *n*: similarly, the maximum number of times a particular entity *can appear* in the relationship tuples at any point of time

Comparing the Notations



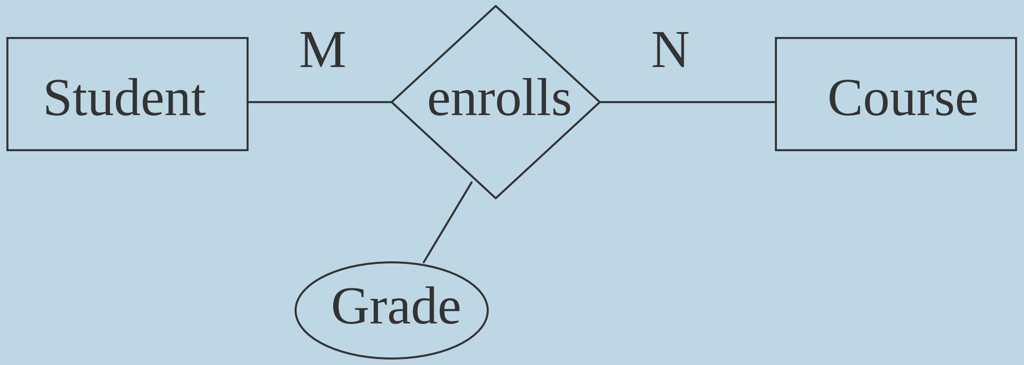
is equivalent to



Attributes for Relationship Types

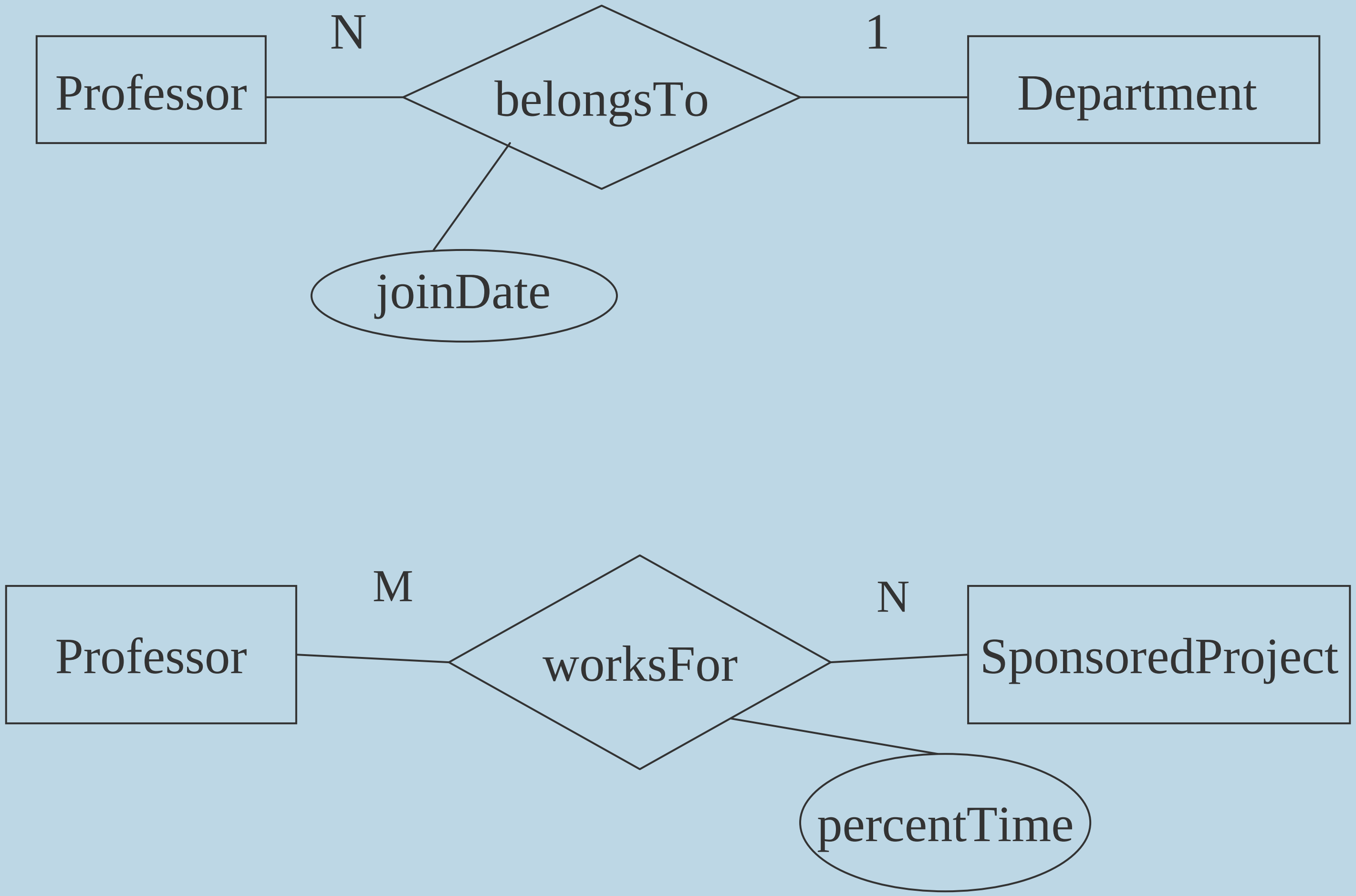
Relationship types can also have attributes.

- properties of the association of entities.



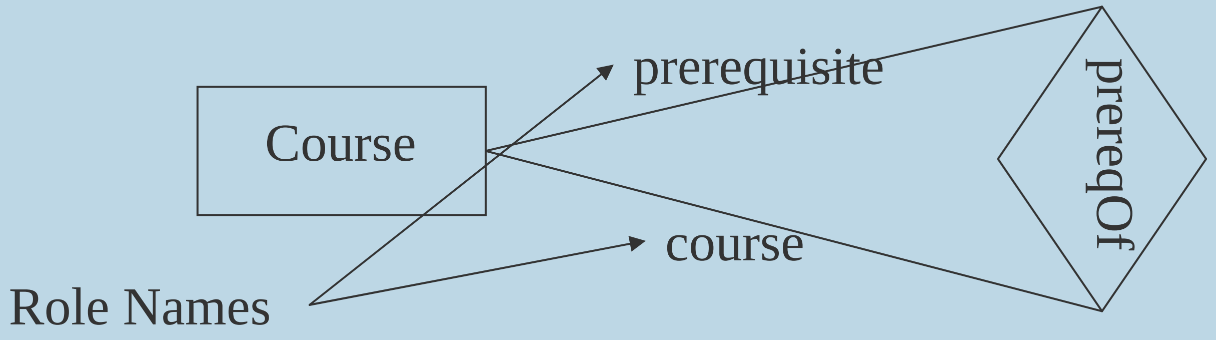
- *grade* gives the letter grade (S,A,B, etc.) earned by the student for a course.
  - neither an attribute of *student* nor that of *course*.

Attributes for Relationship Types – More Examples



Recursive Relationships and Role Names

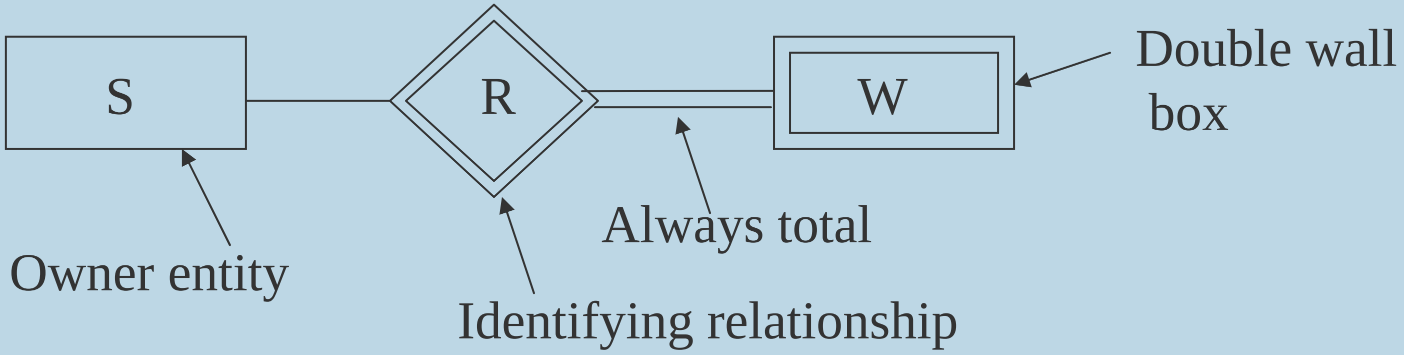
- Recursive relationship: An entity set relating to itself gives rise to a *recursive* relationship
- E.g., the relationship *prereqOf* is an example of a recursive relationship on the entity *Course*
- Role Names – used to specify the exact role in which the entity participates in the relationships
  - Essential in case of recursive relationships
  - Can be optionally specified in non-recursive cases



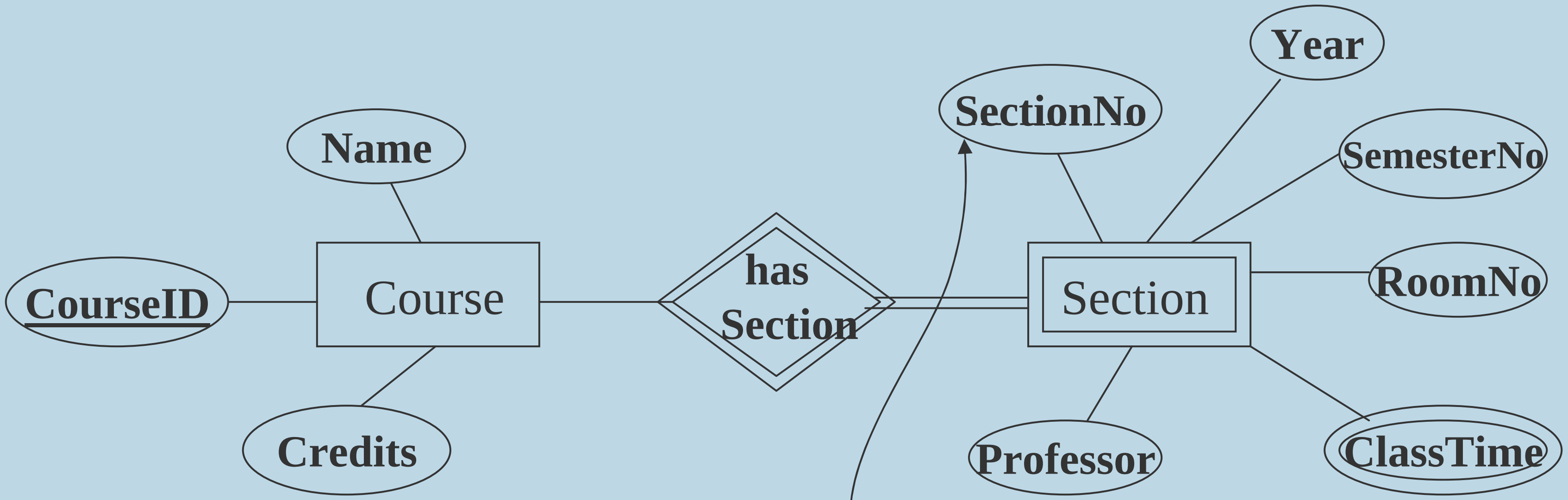
Weak Entity Sets

Weak Entity Set: An entity set whose members owe their existence to some entity in a *strong entity set.*

- entities are not of independent existence.
- each weak entity is associated with some entity of the *owner* entity set through a special relationship.
- weak entity set may not have a key attribute.



Weak Entity Sets - Example



A popular course may have several sections each taught by a different professor and having its own class room and meeting times

Partial key:
Uniquely identifies a section among the set of sections of a particular course

Complete Example for E/R schema: Specifications (1/2)

In an educational institute, there are several departments and students belong to one of them. Each department has a unique department number, a name, a location, phone number and is headed by a professor. Professors have a unique employee Id, name, phone number.

We like to keep track of the following details regarding students: name, unique roll number, sex, phone number, date of birth, age and one or more email addresses. Students have a local address consisting of the hostel name and the room number. They also have home address consisting of house number, street, city and PIN. It is assumed that all students reside in the hostels.

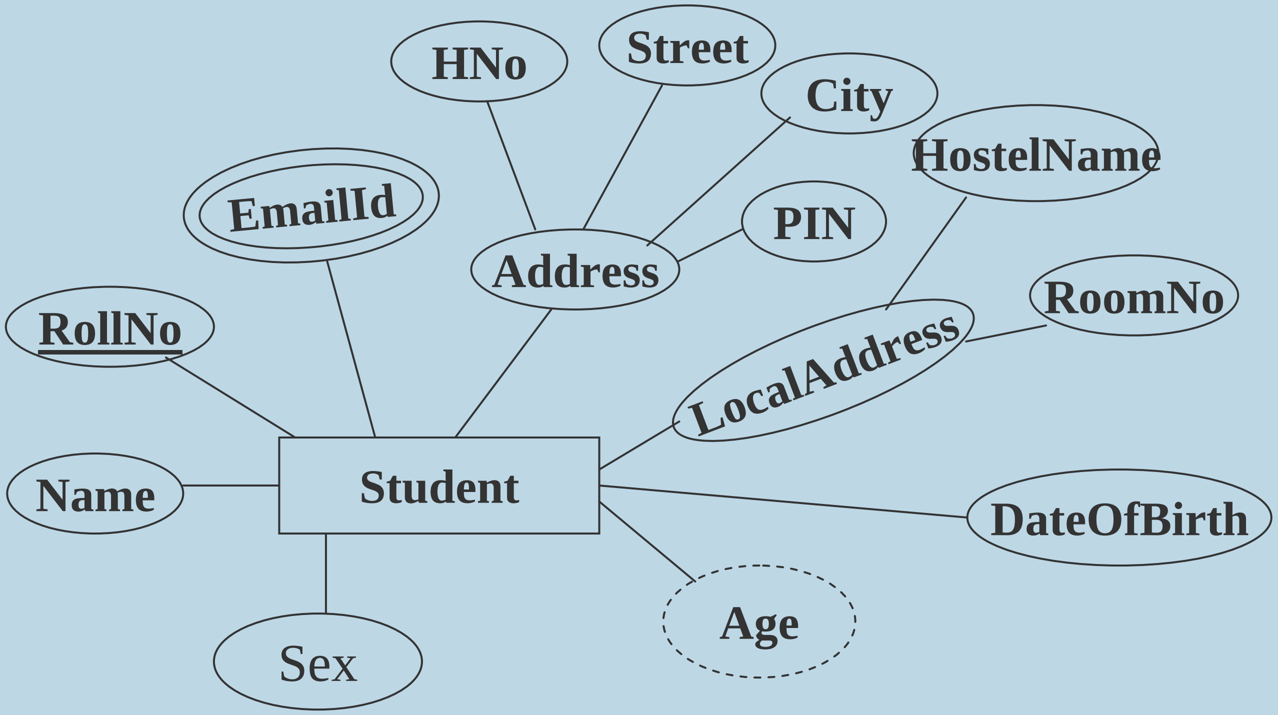
Complete Example for E/R schema: Specifications (2/2)

A course taught in a semester of the year is called a *section*. There can be several sections of the same course in a semester; these are identified by the *section number*. Each section is taught by a different professor and has its own timings and a room to meet. Students enroll for several sections in a semester.

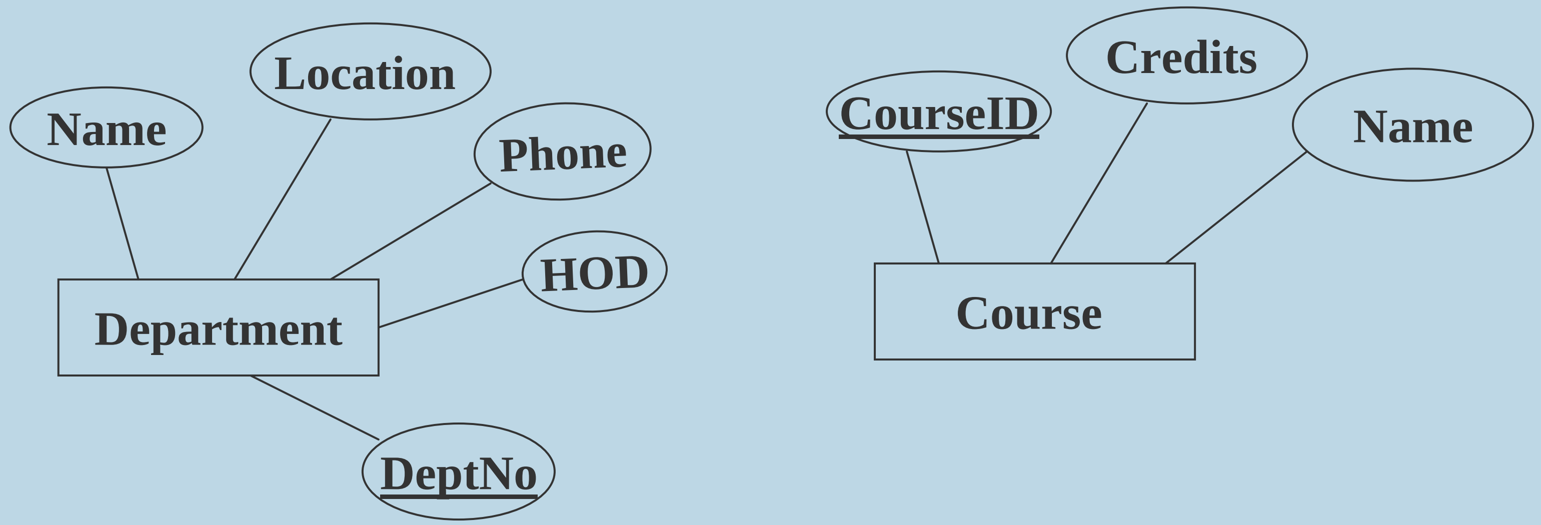
Each course has a name, number of credits and the department that offers it. A course may have other courses as pre-requisites i.e, courses to be completed before it can be enrolled in.

Professors also undertake research projects. These are sponsored by funding agencies and have a specific start date, end date and amount of money given. More than one professor can be involved in a project. Also a professor may be simultaneously working on several projects. A project has a unique *projectId*.

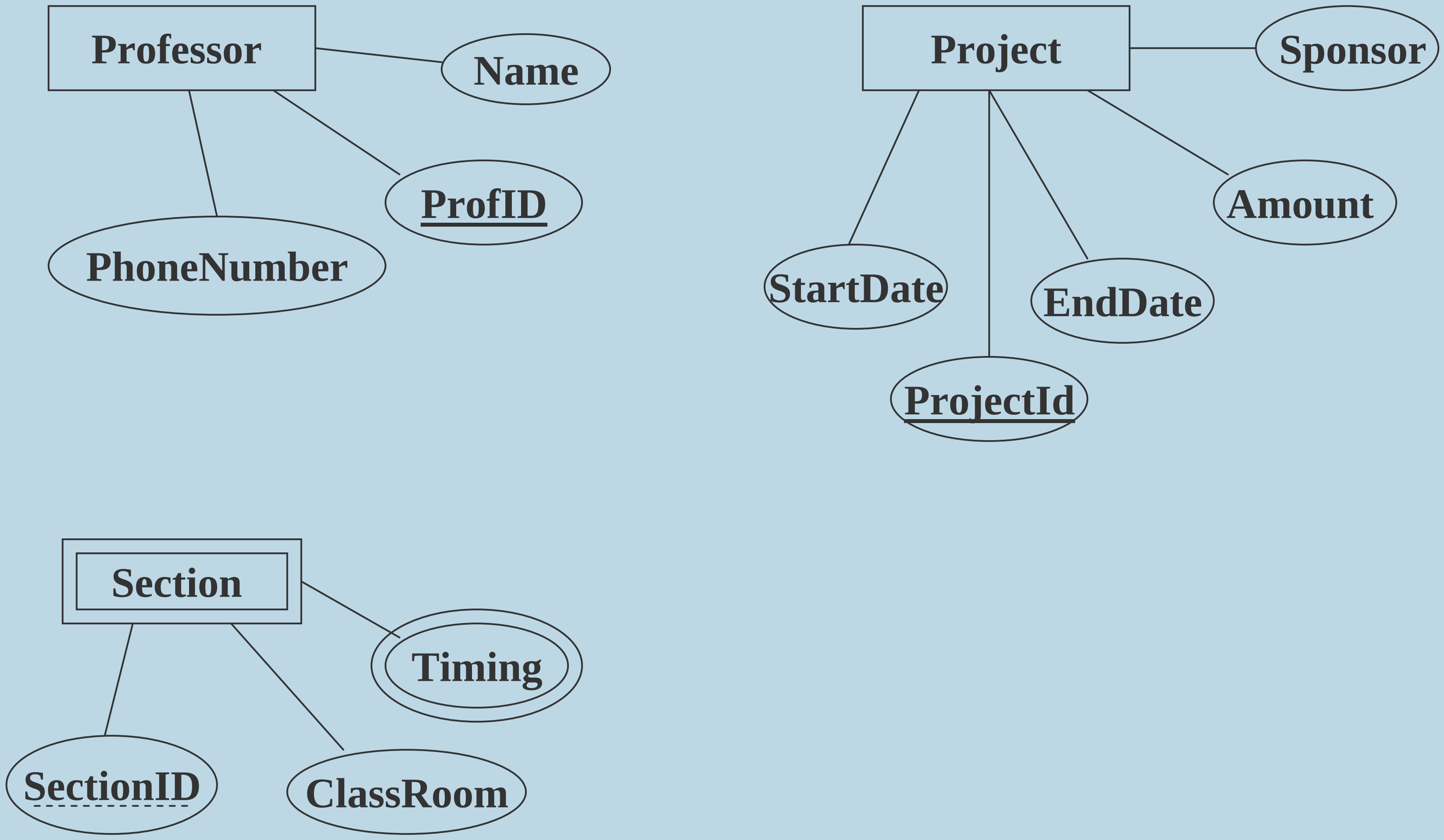
Entities - Student



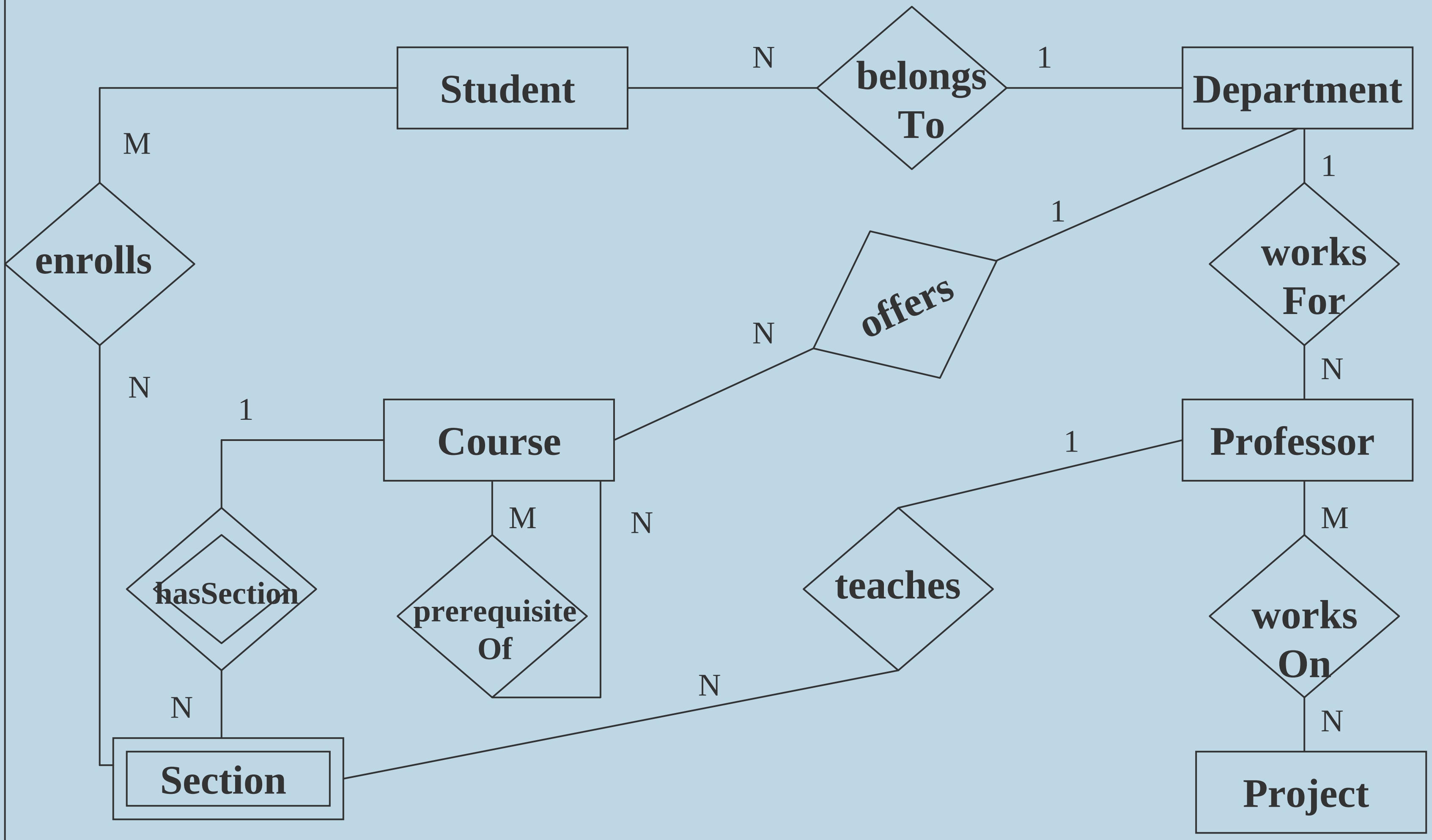
Entities – Department and Course



Entities – Professor, Project and Sections



E/R Diagram showing relationships



Design Choices: Attribute versus Relationship

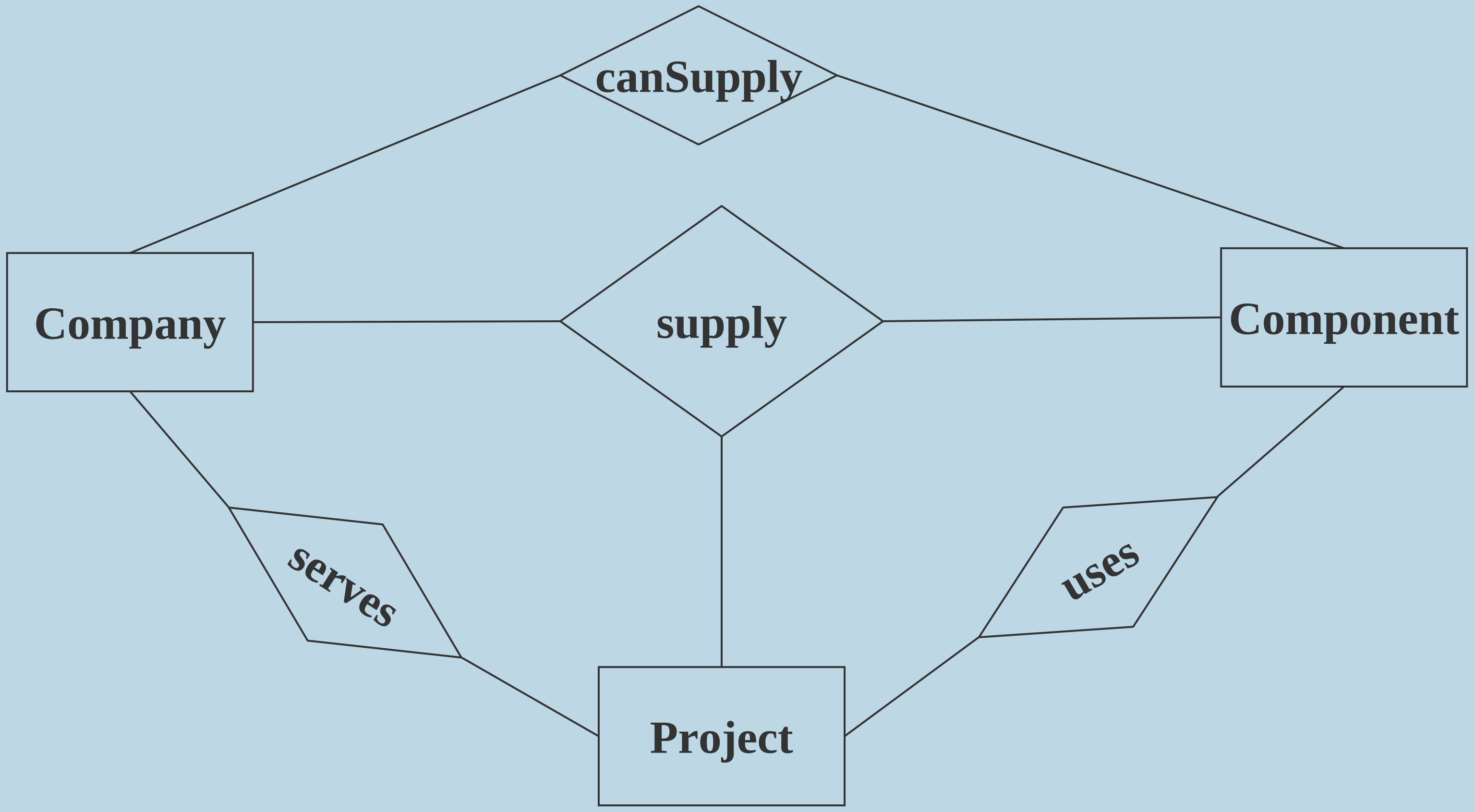
- Should *offering department* be an attribute of a course or should we create a relationship between Course and Dept entities called, say, *offers* ?
    - Later approach is preferable when the necessary entity, in this case the Department, already exists.
- Should *class room* be an attribute of Section or should we create an entity called ClassRoom and have a relationship, say, meetsIn, connecting Section and ClassRoom?
    - In this case, the option of making classRoom as an attribute of Section is better as we do not want to give a lot of importance to class room and make it a an *entity*.

Design Choices:
Weak entity versus composite multi-valued attributes

- Note that *section* could be a composite multi-valued *attribute* of Course entity.
  - However, if so, *section* can not participate in relationships, such as, *enrolls* with Student entity.
- In general, if a thing, even though not of independent existence, participates in other relationships on its own, it is best captured as a *weak entity*.
  - If the above is not the case, composite multi-valued attribute may be enough.

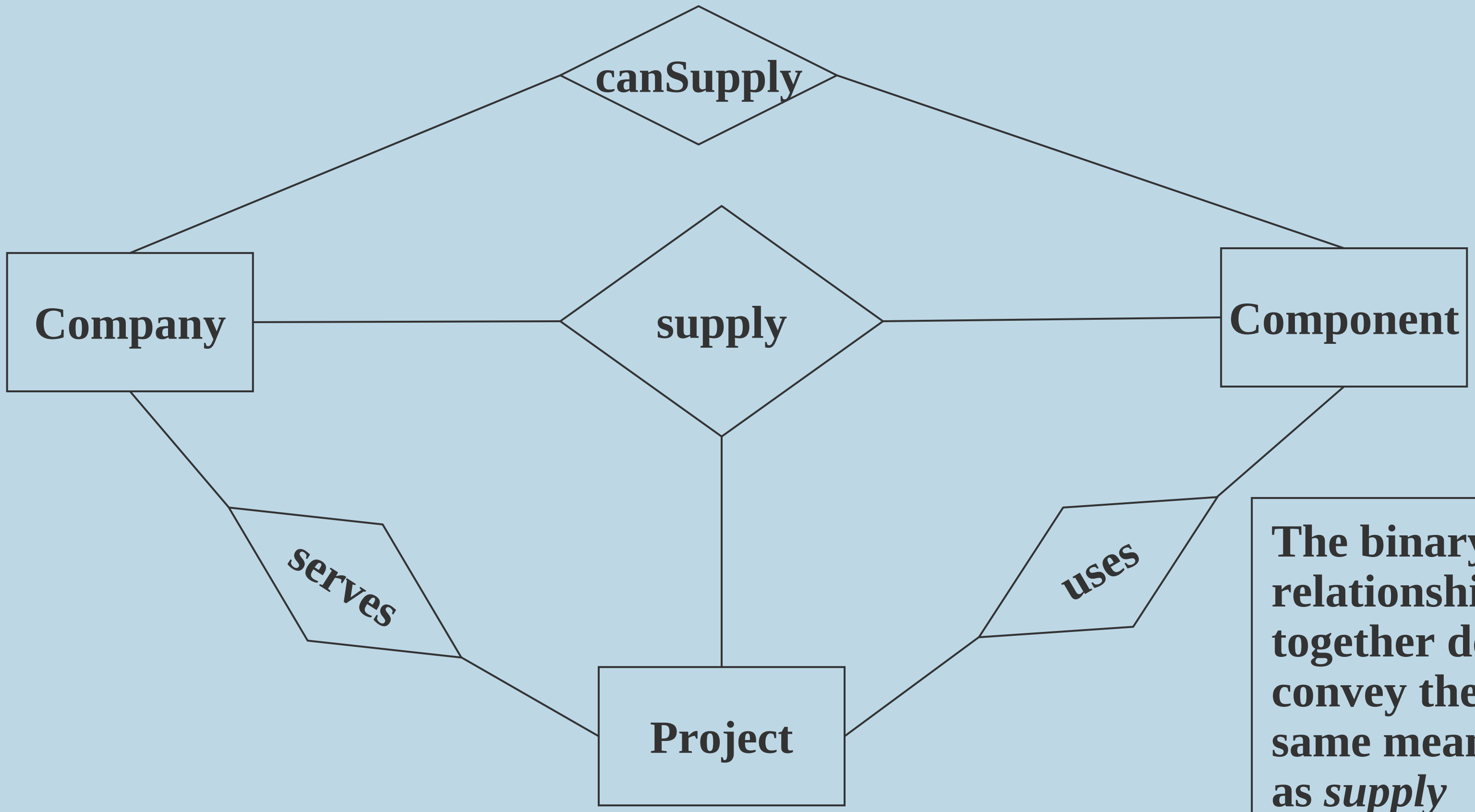
Ternary Relationships

Relationship instance (c, p, j) indicates that
company c supplies a component p that is made use of by the project j



Ternary Relationships

(c,p) in *canSupply*, (j,p) in *uses*, (c,j) in *serves* may not together imply (c,p,j) is in *supply*. Whereas the other way round is of course true.



The binary relationships together do not convey the same meaning as *supply*

Tuple Relational Calculus (TRC)

Introduction

Procedural Query language

- query specification involves giving a step by step process of obtaining the query result
  e.g., relational algebra
- usage calls for detailed knowledge of the operators involved
- difficult for the use of non-experts

Declarative Query language

- query specification involves giving the logical conditions the results are required to satisfy
- easy for the use of non-experts

TRC – a declarative query language

Tuple variable – associated with a relation
(called the *range relation*)

- takes tuples from the range relation as its values
- *t*: tuple variable over relation *r* with scheme R(A,B,C)
  t.A stands for value of column *A* etc

TRC Query – basic form:

$\{ t_1.A_{i_1}, t_2.A_{i_2}, \ldots t_m.A_{i_m} \mid \theta \}$

predicate calculus expression involving tuple variables $t_1, t_2, \ldots, t_m, t_{m+1}, \ldots, t_s$
- specifies the condition to be satisfied

An example TRC query

student (*<u>rollNo</u>*, *name*, *degree*, *year*, *sex*, *deptNo*, *advisor*)
department (*<u>deptId</u>*, *name*, *hod*, *phone*)

```
Obtain the rollNo, name of all girl students
in the Maths Dept (deptId = 2)
{s.rollNo,s.name| student(s)^ s.sex='F'^ s.deptNo=2}
```

attributes required in the result

This predicate is true whenever value of *s* is a tuple from the student relation, false otherwise

In general, if *t* is a tuple variable with range relation *r*, *r*(*t*) is taken as a predicate which is true if and only if the value of *t* is a tuple in *r*

General form of the condition in TRC queries

Atomic expressions are the following:

1. *r (t)* -- true if *t* is a tuple in the relation instance *r*
2. $t_1.A_i$ *<compOp>* $t_2.A_j$ *compOp* is one of $\{<, \leq, >, \geq, =, \neq\}$
3. $t.A_i$ *<compOp> c* *c* is a constant of appropriate type

Composite expressions:

1. Any atomic expression
2. $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$ where F_1 and F_2 are expressions
3. $(\forall t)$ (F), $(\exists t)$ (F) where F is an expression and *t* is a tuple variable

Free Variables

Bound Variables – quantified variables

Interpretation of the query in TRC

All possible tuple assignments to the free variables in the query are considered.

For any specific assignment,
  if the expression to the right of the vertical bar evaluates to true,
    that combination of tuple values
    would be used to produce a tuple in the result relation.

While producing the result tuple, the values of the attributes for the corresponding tuple variables as specified on the left side of the vertical bar would be used.

Note: The only free variables are the ones that appear to the left
    of the vertical bar

Example TRC queries

```
Obtain the rollNo, name of all girl students in the
      Maths Dept

{s.rollNo,s.name | student(s) ^ s.sex=‘F’ ^
                   (∃d)(department(d) ^ d.name=‘Maths’
                               ^ d.deptId = s.deptNo)}
```

s: free tuple variable *d*: existentially bound tuple variable

Existentially or universally quantified tuple variables can be used on the RHS of the vertical bar to specify query conditions

Attributes of free (or unbound) tuple variables can be used on LHS of vertical bar to specify attributes required in the results

Example Relational Scheme

student (rollNo, name, degree, year, sex, deptNo, advisor)

department (deptId, name, hod, phone)

professor (empId, name, sex, startYear, deptNo, phone)

course (courseId, cname, credits, deptNo)

enrollment (rollNo, courseId, sem, year, grade)

teaching (empId, courseId, sem, year, classRoom)

preRequisite (preReqCourse, courseID)

Q2

Q3

Q4

Q5

Example queries in TRC (1/5)

```
1)Determine the departments that do not have
  any girl students
```

student (<u>rollNo</u>, name, degree, year, sex, deptNo, advisor)
department (<u>deptId</u>, name, hod, phone)

```
{d.name|department(d) ^
            ¬(∃ s)(student(s) ^
                   s.sex =‘F’ ^ s.deptNo = d.deptId)
```

Examples queries in TRC (2/5)

Schema

2)Obtain the names of courses enrolled by student named Mahesh

```
{c.name | course(c) ^
          (∃s)(∃e)( student(s) ^ enrollment(e)
                  ^ s.name = "Mahesh"
                  ^ s.rollNo = e.rollNo
                  ^ c.courseId = e.courseId }
```

Examples queries in TRC (3/5)

Schema

3)Get the names of students who have scored ‘S’ in all subjects they have enrolled. Assume that every student is enrolled in at least one course.

```
{s.name | student(s) ^
        (∀e)(( enrollment(e) ^
              e.rollNo = s.rollNo) → e.grade =‘S’)}
```

person P with all S grades:

for enrollment tuples not having her roll number, LHS is false

for enrollment tuples having her roll number, LHS is true, RHS also true

so the implication is true for all e tuples

person Q with some non-S grades:

for enrollment tuples not having her roll number, LHS is false

for enrollment tuples having her roll number, LHS is true, but RHS is false for at least one tuple.

So the implication is not true for at least one tuple.

Examples queries in TRC (4/5)

Schema

4) Get the names of students who have taken at least one course taught by their advisor

```
{s.name | student(s) ^
          (∃e)(∃t)(enrollment(e) ^ teaching(t) ^
                   e.courseId = t.courseId ^
                   e.rollNo = s.rollNo ^
                   t.empId = s.advisor}
```

5) Display the departments whose HODs are teaching at least one course in the current semester

```
{d.name | department(d) ^(∃t)(teaching(t) ^
                   t.empid = d.hod
          ^ t.sem = ‘odd’ ^ t.year = ‘2008’)}
```

Examples queries in TRC (5/5)

Schema

6)Determine the students who are enrolled for *every* course taught by Prof Ramanujam. Assume that Prof Ramanujam teaches at least one course.

```
1.  {s.rollNo | student (s) ^
2.    (∀c)(course (c) ^
3.         ((∃t),(∃p)( teaching(t) ^ professor(p) ^
4.              t.courseId = c.courseId ^
5.              p.name = "Ramanujam" ^
6.              p.empId = t.empId )) →
7.                (∃e) (enrollment(e) ^
8.                      e.courseId = c.courseId ^
9.                      e.rollNo = s.rollNo)
10.   )
11.  }
```

Problem with unrestricted use of Negation

What is the result of the query:

```
{s.rollNo | ¬ student(s)} ?
```

Infinite answers !!

Unsafe TRC expression :

Any expression whose result uses "constants / values" that do not appear in the instances of any of the database relations.

Unsafe expressions are to be avoided while specifying TRC queries.

Expressive power of TRC and Relational Algebra

It can be shown that
both Tuple Relational Calculus and Relational Algebra
have the same expressive power

A query can be formulated in (safe) TRC
if and only if it can be formulated in RA

Both *can not* be used to formulate queries involving
transitive closure
-- find all direct or indirect pre-requisites of a course
-- find all subordinates of a specific employee etc.

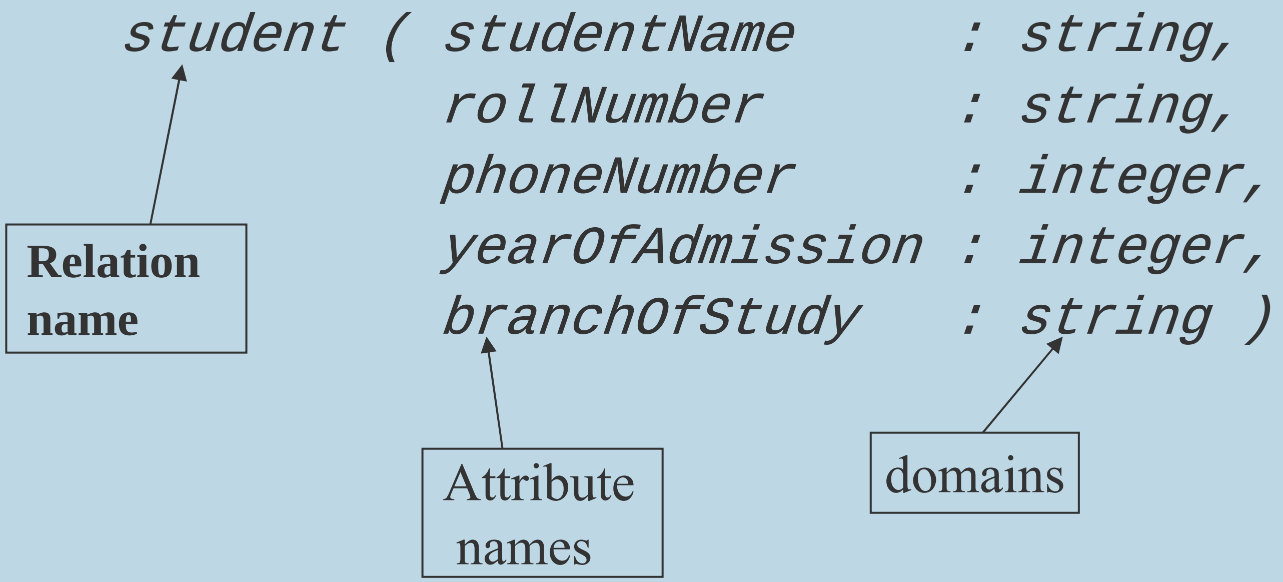
Relational Model

Introduction

- Proposed by Edgar. F. Codd (1923-2003) in the early seventies. [Turing Award – 1981]
- Most of the modern DBMS are relational.
- Simple and elegant model with a mathematical basis.
- Led to the development of a theory of data dependencies and database design.
- Relational algebra operations –
  crucial role in query optimization and execution.
- Laid the foundation for the development of
  - Tuple relational calculus and then
  - Database standard SQL

Relation Scheme

- Consists of relation name, and a set of attributes or field names or column names. Each attribute has an associated domain.
- *Example:*



```
student ( studentName      : string,
          rollNumber       : string,
          phoneNumber      : integer,
          yearOfAdmission  : integer,
          branchOfStudy    : string )
```

- *Domain* – set of *atomic* (or *indivisible*) values – data type

Relation Instance

- A finite *set* of tuples constitute a relation instance.
- A tuple of relation with scheme $R = (A_1, A_2, \ldots, A_m)$ is an ordered sequence of values $(v_1, v_2, \ldots, v_m)$ such that $v_i \in$ domain (A_i), $1 \leq i \leq m$

student

studentName	**rollNumber**	**yearOf Admission**	**phoneNumber**	**branch Of Study**
Ravi Teja	CS05B015	2005	9840110489	CS
Rajesh	CS04B125	2004	9840110490	EC
		⋮		

No duplicate tuples (or rows) in a relation instance.

We shall later see that in SQL, duplicate rows would be allowed in tables.

Another Relation Example

enrollment (studentName, rollNo, courseNo, sectionNo)

enrollment

studentName	rollNumber	courseNo	sectionNo
Rajesh	CS04B125	CS320	2
Rajesh	CS04B125	CS370	1
Suresh	CS04B130	CS320	2
		⋮	

Keys for a Relation (1/2)

- **Key**: A set of attributes K, whose values uniquely identify a tuple in any instance. And none of the proper subsets of K has this property

  Example: {*rollNumber*} is a key for *student* relation.

  {*rollNumber*, *name*} – values can uniquely identify a tuple
    - but the set is not *minimal*
    - not a Key
- A key can not be determined from any particular instance data
  - it is an intrinsic property of a scheme
  - it can only be determined from the meaning of attributes

Keys for a Relation (2/2)

- A relation can have more than one key.
- Each of the keys is called a *candidate* key

  Example: *book* (*isbnNo*, *authorName*, *title*, *publisher*, *year*)
  (Assumption : books have only one author)
  Keys: {*isbnNo*}, {*authorName, title*}
- A relation has at least one key

  - the set of all attributes, in case no proper subset is a key.
- **Superkey**: A set of attributes that contains any key as a subset.
  - A key can also be defined as a *minimal superkey*
- **Primary Key**: One of the candidate keys chosen for indexing purposes (More details later…)

Relational Database Scheme and Instance

Relational database scheme: *D* consist of a finite no. of relation schemes and a set *I* of integrity constraints.

Integrity constraints: Necessary conditions to be satisfied by the data values in the relational instances so that the set of data values constitute a meaningful database

- domain constraints
- key constraints
- referential integrity constraints

Database instance: Collection of relational instances satisfying the integrity constraints.

Domain and Key Constraints

- **Domain Constraints**: Attributes have associated domains

  Domain – set of atomic data values of a specific type.

  Constraint – stipulates that the actual values of an attribute in any tuple must belong to the declared domain.

- **Key Constraint**: Relation scheme – associated keys

  Constraint – if *K* is supposed to be a key for scheme *R*, any relation instance *r* on *R* should not have two tuples that have identical values for attributes in *K*.

  Also, none of the key attributes can have null value.

Foreign Keys

- Tuples in one relation, say $r_1(R_1)$, often need to refer to tuples in another relation, say $r_2(R_2)$
    - to capture relationships between entities
- Primary Key of R_2 : $K = \{B_1, B_2, \ldots, B_j\}$
- A set of attributes $F = \{A_1, A_2, \ldots, A_j\}$ of R_1 such that

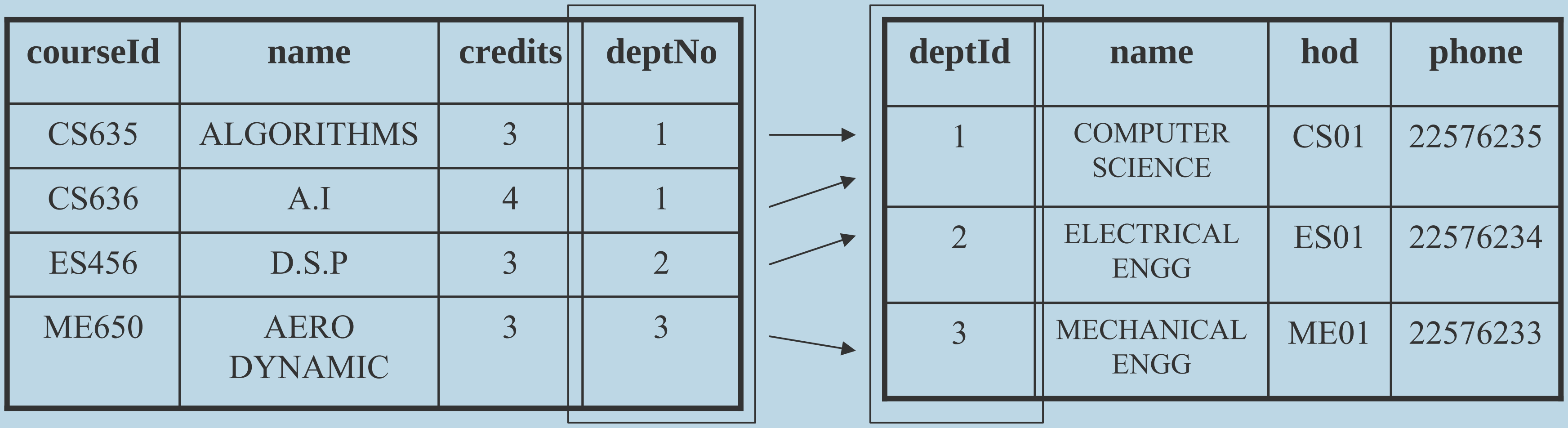
  $$\mathrm{dom}(A_i) = \mathrm{dom}(B_i), \quad 1 \leq i \leq j \text{ and}$$

  whose values are used to refer to tuples in r_2
  is called a *foreign key* in R_1 *referring* to R_2.

- R_1, R_2 can be the same scheme also.
- There can be more than one foreign key in a relation scheme

Foreign Key – Examples(1/2)

Foreign key attribute *deptNo* of *course* relation refers to Primary key attribute *deptID* of *department* relation



Course

courseId	name	credits	deptNo
CS635	ALGORITHMS	3	1
CS636	A.I	4	1
ES456	D.S.P	3	2
ME650	AERO DYNAMIC	3	3

Department

deptId	name	hod	phone
1	COMPUTER SCIENCE	CS01	22576235
2	ELECTRICAL ENGG	ES01	22576234
3	MECHANICAL ENGG	ME01	22576233

Foreign Key – Examples(2/2)

It is possible for a foreign key in a relation
to refer to the primary key of the relation itself

An Example:

univEmployee (<u>empNo</u>, name, sex, salary, dept, reportsTo)

reportsTo is a foreign key referring to *empNo* of the same relation

Every employee in the university reports to some other employee for administrative purposes
- except the *vice-chancellor*, of course!

Referential Integrity Constraint (RIC)

- Let *F* be a foreign key in scheme R_1 referring to scheme R_2 and let *K* be the primary key of R_2.
- **RIC**: any relational instance $r_1$on R_1, r_2 on R_2 must be s.t for any tuple *t* in r_1, either its *F*-attribute values are *null* or they are identical to the *K*-attribute values of *some* tuple in r_2.
- RIC ensures that references to tuples in r_2 are for *currently existing* tuples.
  - That is, there are no *dangling* references.

Referential Integrity Constraint (RIC) - Example

COURSE

courseId	name	credits	deptNo
CS635	ALGORITHMS	3	1
CS636	A.I	4	1
ES456	D.S.P	3	2
ME650	AERO DYNAMIC	3	3
CE751	MASS TRANSFER	3	4

DEPARTMENT

deptId	name	hod	phone
1	COMPUTER SCIENCE	CS01	22576235
2	ELECTRICAL ENGG.	ES01	22576234
3	MECHANICAL ENGG.	ME01	22576233

The new course refers to a non-existent department and thus violates the RIC

Example Relational Scheme

student (rollNo, name, degree, year, sex, deptNo, advisor)

Here, *degree* is the program (B Tech, M Tech, M S, Ph D etc) for which the student has joined. *Year* is the year of admission and *advisor* is the EmpId of a faculty member identified as the student's advisor.

department (deptId, name, hod, phone)

Here, *phone* is that of the department's office.

professor (empId, name, sex, startYear, deptNo, phone)

Here, *startYear* is the year when the faculty member has joined the department *deptNo*.

Example Relational Scheme

course (<u>courseId</u>, cname, credits, deptNo)

Here, *deptNo* indicates the department that offers the course.

enrollment (<u>rollNo, courseId, sem, year</u>, grade)

Here, *sem* can be either "odd" or "even" indicating the two semesters of an academic year. The value of *grade* will be null for the current semester and non-null for past semesters.

teaching (<u>empId, courseId, sem, year</u>, classRoom)

preRequisite (<u>preReqCourse, courseID</u>)

Here, if (c1, c2) is a tuple, it indicates that c1 should be successfully completed before enrolling for c2.

Example Relational Scheme

student (rollNo, name, degree, year, sex, deptNo, advisor)

department (deptId, name, hod, phone)

professor (empId, name, sex, startYear, deptNo, phone)

course (courseId, cname, credits, deptNo)

enrollment (rollNo, courseId, sem, year, grade)

teaching (empId, courseId, sem, year, classRoom)

preRequisite (preReqCourse, courseID)

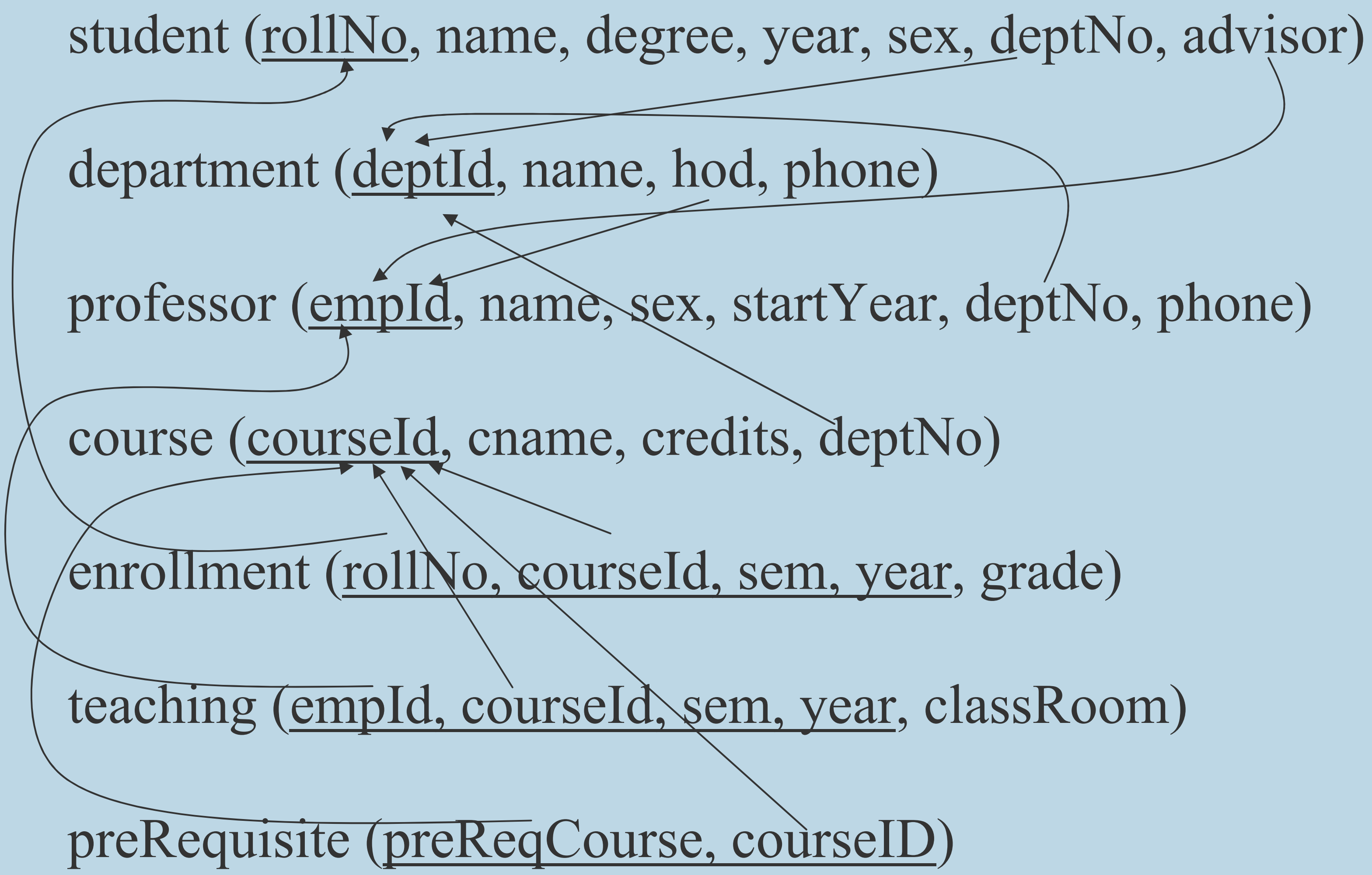
queries-1

queries-2

queries-3

TCQuery

Example Relational Scheme with RIC's shown



student (rollNo, name, degree, year, sex, deptNo, advisor)

department (deptId, name, hod, phone)

professor (empId, name, sex, startYear, deptNo, phone)

course (courseId, cname, credits, deptNo)

enrollment (rollNo, courseId, sem, year, grade)

teaching (empId, courseId, sem, year, classRoom)

preRequisite (preReqCourse, courseID)

Relational Algebra

- A set of operators (unary and binary) that take relation instances as arguments and return new relations.
- Gives a procedural method of specifying a retrieval query.
- Forms the core component of a relational query engine.
- SQL queries are internally translated into RA expressions.
- Provides a framework for query optimization.

RA operations: *select* (σ), *project* (π), *cross product* ($\times$), *union* ($\cup$), *intersection* ($\cap$), *difference* ($-$), *join* ($\bowtie$)

The *select* Operator

- Unary operator.
- can be used to *select* those tuples of a relation that satisfy a given condition.
- *Notation*: $\sigma_\theta (r)$

  σ : select operator (read as *sigma*)

  θ : selection condition

  r : relation name
- Result: a relation with the same schema as r consisting of the tuples in r that satisfy condition θ
- Select operation is commutative:

  $\sigma_{c1} (\sigma_{c2} (r)) = \sigma_{c2} (\sigma_{c1} (r))$

Selection Condition

- *Select condition*:
  Basic condition or Composite condition
- *Basic condition*:
  Either A_i <compOp> A_j or A_i <compOp> c
- *Composite condition*:
  Basic conditions combined with logical operators AND, OR and NOT appropriately.
- Notation:

  <compOp> : one of $<, \leq, >, \geq, =, \neq$
  A_i, A_j : attributes in the scheme R of r
  c : constant of appropriate data type

Examples of *select* expressions

1. Obtain information about a professor with name "giridhar"

$$\sigma_{\text{name} = \text{"giridhar"}} (\text{professor})$$

2. Obtain information about professors who joined the university between 1980 and 1985

$$\sigma_{\text{startYear} \geq 1980 \wedge \text{startYear} < 1985} (\text{professor})$$

The *project* Operator

- Unary operator.
- Can be used to keep only the required attributes of a relation instance and throw away others.
- *Notation*: $\pi_{A_1, A_2, \ldots, A_k}(r)$ where $A_1, A_2, \ldots, A_k$ is a list L of desired attributes in the scheme of r.
- Result = $\{ (v_1, v_2, \ldots, v_k) \mid v_i \in dom(A_i),\ 1 \leq i \leq k$ and there is some tuple t in r *s.t* $t.A_1 = v_1, t.A_2 = v_2, \ldots, t.A_k = v_k \}$
- If $r_1 = \pi_L(r_2)$ then scheme of r_1 is L

Examples of *project* expressions

student

rollNo	name	degree	year	sex	deptNo	advisor
CS04S001	Mahesh	M.S	2004	M	1	CS01
CS03S001	Rajesh	M.S	2003	M	1	CS02
CS04M002	Piyush	M.E	2004	M	1	CS01
ES04M001	Deepak	M.E	2004	M	2	ES01
ME04M001	Lalitha	M.E	2004	F	3	ME01
ME03M002	Mahesh	M.S	2003	M	3	ME01

$\pi_{\text{rollNo, name}}$ (student)

rollNo	name
CS04S001	Mahesh
CS03S001	Rajesh
CS04M002	Piyush
ES04M001	Deepak
ME04M001	Lalitha
ME03M002	Mahesh

π_{name} ($\sigma_{\text{degree = “M.S”}}$ (student))

name
Mahesh
Rajesh

Note: Mahesh is displayed only once because project operation results in a set.

Size of *project* expression result

- If $r_1 = \pi_L(r_2)$ then scheme of r_1 is L
- What about the number of tuples in r_1?
- Two cases arise:
  - Projection List L contains some key of r_2
    - Then $|r_1| = |r_2|$
  - Projection List L does not contain any key of r_2
    - Then $|r_1| \leq |r_2|$

Set Operators on Relations

- As relations are sets of tuples, set operations are applicable to them; but not in all cases.
- **Union Compatibility** : Consider two schemes R_1, R_2 where $R_1 = (A_1, A_2, \ldots, A_k)$; $R_2 = (B_1, B_2, \ldots, B_m)$
- R_1 and R_2 are called *union-compatible* if
    - $k = m$ and
    - $dom(A_i) = dom(B_i)$ for $1 \leq i \leq k$
- **Set operations** – *union, intersection, difference*
    - Applicable to two relations if their schemes are union-compatible
- If $r_3 = r_1 \cup r_2$, scheme of r_3 is R_1 (as a convention)

Set Operations

r_1 - relation with scheme R_1

r_2 - relation with scheme R_2 - union compatible with R_1

$$r_1 \cup r_2 = \{t \mid t \in r_1 \text{ or } t \in r_2\};$$
$$r_1 \cap r_2 = \{t \mid t \in r_1 \text{ and } t \in r_2\}$$
$$r_1 - r_2 = \{t \mid t \in r_1 \text{ and } t \notin r_2\};$$

By convention, in all the cases, the scheme of the result is that of the first operand i.e r_1.

Cross product Operation

r_1	A_1	A_2	...	A_m
	a_{11}	a_{12}	...	a_{1m}
	a_{21}	a_{22}	...	a_{2m}
	a_{s1}	a_{s2}	...	a_{sm}

r_1 : *s* tuples

r_2	B_1	B_2	...	B_n
	b_{11}	b_{12}	...	b_{1n}
	b_{21}	b_{22}	...	b_{2n}
	b_{t1}	b_{t2}	...	b_{tn}

r_2 : *t* tuples

$r_1 \times r_2$

A_1	A_2	...	A_m	B_1	B_2	...	B_n
a_{11}	a_{12}	...	a_{1m}	b_{11}	b_{12}	...	b_{1n}
a_{11}	a_{12}	...	a_{1m}	b_{21}	b_{22}	...	b_{2n}
	.				.		
a_{11}	a_{12}	...	a_{1m}	b_{t1}	b_{t2}	...	b_{tn}
a_{21}	a_{22}	...	a_{2m}	b_{11}	b_{12}	...	b_{1n}
a_{21}	a_{22}	...	a_{2m}	b_{21}	b_{22}	...	b_{2n}
	.				.		
a_{21}	a_{22}	...	a_{2m}	b_{t1}	b_{t2}	...	b_{tn}

.
.
.

$r_1 \times r_2$: $s \times t$ tuples

Example Query using *cross product*

Obtain the list of professors along with the name of their departments

- profDetail (eId, pname,deptno) $\leftarrow \pi_{\text{empId, name, deptNo}}$ (professor)
- deptDetail (dId,dname) $\leftarrow \pi_{\text{deptId, name}}$ (department)
- profDept $\leftarrow$ profDetail $\times$ deptDetail
- desiredProfDept $\leftarrow \sigma_{\text{deptno = dId}}$ (profDept)
- result $\leftarrow \pi_{\text{eId, pname, dname}}$ (desiredProfDept)

Join Operation

- ***Cross product*** : produces all combinations of tuples
    - often only certain combinations are meaningful
    - cross product is usually followed by selection

- ***Join*** : combines tuples from two relations provided they satisfy a specified condition (join condition)
    - equivalent to performing *cross product* followed by *selection*
    - a very useful operation

- Depending on the type of condition we have
    - *theta join*
    - *equi join*

Theta join

- Let r_1 - relation with scheme $R_1 = (A_1, A_2, \ldots, A_m)$

  r_2 - relation with scheme $R_2 = (B_1, B_2, \ldots, B_n)$

  and $R_1 \cap R_2 = \phi$
- Notation for join expression : $r_1 \bowtie_\theta r_2$, θ - join condition

  θ is of the form : $C_1 \wedge C_2 \wedge \ldots \wedge C_s$

  C_i is of the form : A_j <CompOp> B_k

  <CompOp> : $=$, $\neq$, $<$, $\leq$, $>$, $\geq$
- Scheme of the result relation

  $Q = (A_1, A_2, \ldots, A_m, B_1, B_2, \ldots, B_n)$
- $r = \{(a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n) \mid (a_1, a_2, \ldots, a_m) \in r_1,$ $(b_1, b_2, \ldots, b_n) \in r_2$ and $(a_1, a_2, \ldots, a_m , b_1, b_2, \ldots, b_n)$ satisfies $\theta\}$

Professor

empId	name	sex	startYear	deptNo	phone
CS01	GIRIDHAR	M	1984	1	22576345
CS02	KESHAV MURTHY	M	1989	1	22576346
ES01	RAJIV GUPTHA	M	1980	2	22576244
ME01	TAHIR NAYYAR	M	1999	3	22576243

Courses

courseId	cname	credits	deptNo
CS635	Algorithms	3	1
CS636	A.I	4	1
ES456	D.S.P	3	2
ME650	Aero Dynamics	3	3

Department

deptId	name	hod	phone
1	Computer Science	CS01	22576235
2	Electrical Engg.	ES01	22576234
3	Mechanical Engg.	ME01	22576233

Examples

For each department, find its name and the name, sex and phone number of the head of the department.

Prof (empId, p-name, sex, deptNo, prof-phone)

$\leftarrow \pi_{\text{empId, name, sex, deptNo, phone}}$ (professor)

Result $\leftarrow$

$\pi_{\text{DeptId, name, hod, p-name, sex, prof-phone}}$ (Department $\bowtie_{\text{(empId = hod) ^ (deptNo = deptId)}}$ Prof)

deptId	name	hod	p-name	sex	prof-phone
1	Computer Science	CS01	Giridher	M	22576235
2	Electrical Engg.	EE01	Rajiv Guptha	M	22576234
3	Mechanical Engg.	ME01	Tahir Nayyar	M	22576233

Equi-join and *Natural join*

- *Equi-join* : Equality is the only comparison operator used in the join condition
- *Natural join* : R_1, R_2 - have common attributes, say X_1,X_2,X_3
  - Join condition:

    $(R_1.X_1 = R_2.X_1) \wedge (R_1.X_2 = R_2.X_2) \wedge (R_1.X_3 = R_2.X_3)$
    - values of common attributes should be equal
  - Schema for the result $Q = R_1 \cup (R_2 - \{X_1, X_2, X_3\})$
    - Only one copy of the common attributes is kept

- Notation for natural join : $r = r_1 * r_2$

Examples – Equi-join

Find courses offered by each department

$$\pi_{\text{deptId, name, courseId, cname, credits}} (\text{Department} \bowtie_{(\text{deptId} = \text{deptNo})} \text{Courses})$$

deptId	**name**	**courseId**	**cname**	**credits**
1	Computer Science	CS635	Algorithms	3
1	Computer Science	CS636	A.I	4
2	Electrical Engg.	ES456	D.S.P	3
3	Mechanical Engg.	ME650	Aero Dynamics	3

Teaching

empId	courseId	sem	year	classRoom
CS01	CS635	1	2005	BSB361
CS02	CS636	1	2005	BSB632
ES01	ES456	2	2004	ESB650
ME650	ME01	1	2004	MSB331

To find the courses handled by each professor

Professor * Teaching

result

empId	name	sex	startYear	deptNo	phone	courseId	sem	year	classRoom
CS01	Giridhar	M	1984	1	22576345	CS635	1	2005	BSB361
CS02	Keshav Murthy	M	1989	1	22576346	CS636	1	2005	BSB632
ES01	Rajiv Guptha	M	1989	2	22576244	ES456	2	2004	ESB650
ME01	Tahir Nayyar	M	1999	3	22576243	ME650	1	2004	MSB331

Division operator

- The necessary condition to determine $r \div s$ on instances $r(R)$ and $s(S)$ is $S \subseteq R$
- The relation $r \div s$ is a relation on schema $R – S$.

  A tuple t is in $r \div s$ if and only if

  1) t is in $\pi_{R\text{-}S}(r)$
  2) For every tuple t_s in s, there is t_r in r satisfying both

     a) $t_r[S] = t_s$

     b) $t_r[R – S] = t$

- <u>Another Definition</u> $r = r_1 \div r_2$

  Division operator produces a relation R(X) that includes all tuples t[X] in $R_1(Z)$ that appear in R_1 in combination with every tuple from $R_2(Y)$ where $Z = X \cup Y$

R = (A, B, C, D), S = (A, B), X = (C, D)

x = r $\div$ s

s

A	B
a_1	b_1
a_2	b_2

r

A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_2	c_1	d_1
a_1	b_1	c_2	d_2
a_1	b_1	c_3	d_3
a_2	b_2	c_3	d_3

x

C	D
c_1	d_1
c_3	d_3

(c_2, d_2) is not present in the result of division as it does not appear in combination with all the tuples of s in r

Query using division operation

Find those students who have registered for *all* courses offered in dept of Computer Science.

Step1: Get the course enrollment information for all students

studEnroll $\leftarrow \pi_{\text{name, courseId}}$ (student * enrollment)

Step2: Get the course Ids of all courses offered by CS dept

csCourse $\leftarrow \pi_{\text{courseId}}(\sigma_{\text{dname = “computer science”}}(\text{courses} \bowtie_{\text{deptId = deptNo}} \text{dept}))$

Result : studEnroll $\div$ csCourse

Schema

Suppose result of step 1 is

studEnroll

name	courseId
Mahesh	CS635
Mahesh	CS636
Rajesh	CS635
Piyush	CS636
Piyush	CS635
Deepak	ES456
Lalitha	ME650
Mahesh	ME650

result of step 2

csCourse

courseId
CS635
CS636

studEnroll $\div$ csCourse

result

name
Mahesh
Piyush

Let's assume for a moment that student names are unique!

Complete Set of Operators

- Are all Relational Algebra operators essential ?

  Some operators can be realized through other operators

- What is the minimal set of operators ?
  - The operators {σ , π, $\times$, $\cup$, - }constitute a *complete* set of operators
  - Necessary and sufficient set of operators.
  - Intersection – union and difference
  - Join – cross product followed by selection
  - Division – project, cross product and difference

Example Queries

Schema

Retrieve the list of female PhD students

$\sigma_{\text{degree = 'phD'} \wedge \text{sex = 'F'}} \text{(student)}$

Obtain the name and rollNo of all female Btech students

$\pi_{\text{rollNo, name}} (\sigma_{\text{degree = 'BTech'} \wedge \text{sex = 'F'}} \text{(student)})$

Obtain the rollNo of students who never obtained an 'E' grade

$\pi_{\text{rollNo}} (\sigma_{\text{grade} \neq \text{'E'}} \text{(enrollment)})$ is incorrect!!
(what if some student gets E in one course and A in another?)

$\pi_{\text{rollNo}} \text{(student)} - \pi_{\text{rollNo}} (\sigma_{\text{grade = 'E'}} \text{(enrollment)})$

More Example Queries

Obtain the department Ids for departments with no lady professor

$$\pi_{\text{deptId}}(\text{dept}) - \pi_{\text{deptId}}(\sigma_{\text{sex = 'F'}}(\text{professor}))$$

Obtain the rollNo of girl students who have obtained at least one S grade

$$\pi_{\text{rollNo}}(\sigma_{\text{sex = 'F'}}(\text{student})) \cap \pi_{\text{rollNo}}(\sigma_{\text{grade = 'S'}}(\text{enrollment}))$$

Another Example Query

Schema

Obtain the names, roll numbers of students who have got S grade in the CS370 course offered in 2006 odd semester along with his/her advisor name.

reqStudsRollNo $\leftarrow$

$\pi_{\text{rollNo}}(\sigma_{\text{courseId ='CS370'\& year ='2006'\& semester = 'odd'\&grade='S'}}(\text{enrollment}))$

reqStuds-Name-AdvId (rollNo, sName, advId) $\leftarrow$

$\pi_{\text{rollNo, name, advisor}}(\text{reqStudsRollNo} * \text{student})$

result(rollNo, studentName, advisorName) $\leftarrow$

$\pi_{\text{rollNo, sName, name}}(\text{reqStuds-Name-AdvId} \bowtie_{\text{advId=empId}} \text{professor})$

Transitive Closure Queries

Schema

Obtain the courses that are either direct or indirect prerequisites of the course CS767.

- Indirect prerequisite – (prerequisite of $)^+$ a prerequisite course
- Prerequisites at all levels are to be reported

$$\text{levelOnePrereq(cId1)} \leftarrow \pi_{\text{preReqCourse}}(\sigma_{\text{courseId} = \text{'CS767'}}(\text{preRequisite}))$$

$$\text{levelTwoPrereq(cId2)} \leftarrow \pi_{\text{preReqCourse}}(\text{preRequisite} \bowtie_{\text{courseId} = \text{cId1}} \text{levelOnePrereq}))$$

Similarly, level *k* prerequisites can be obtained.
But, prerequisites at all levels can not be obtained as there is
no looping mechanism.

Outer Join Operation (1/2)

- Theta join, equi-join, natural join are all called *inner joins* . The result of these operations contain only the matching tuples

- The set of operations called *outer joins* are used when all tuples in relation *r* or relation *s* or both in *r* and *s* have to be in result.

There are 3 kinds of outer joins:

Left outer join ⟕

Right outer join ⟖

Full outer join ⟗

Outer Join Operation (2/2)

Left outer join: $r ⟕ s$

It keeps all tuples in the first, or left relation *r* in the result. For some tuple *t* in *r*, if no matching tuple is found in *s* then S-attributes of *t* are made null in the result.

Right outer join: $r ⟖ s$

Same as above but tuples in the second relation are all kept in the result. If necessary, R-attributes are made null.

Full outer join: $r ⟗ s$

All the tuples in both the relations *r* and *s* are in the result.

Instance Data for Examples

Student

rollNo	name	degree	year	sex	deptNo	advisor
CS04S001	Mahesh	M.S	2004	M	1	CS01
CS05S001	Amrish	M.S	2003	M	1	null
CS04M002	Piyush	M.E	2004	M	1	CS01
ES04M001	Deepak	M.E	2004	M	2	null
ME04M001	Lalitha	M.E	2004	F	3	ME01
ME03M002	Mahesh	M.S	2003	M	3	ME01

Professor

empId	name	sex	startYear	deptNo	phone
CS01	GIRIDHAR	M	1984	1	22576345
CS02	KESHAV MURTHY	M	1989	1	22576346
ES01	RAJIV GUPTHA	M	1980	2	22576244
ME01	TAHIR NAYYAR	M	1999	3	22576243

Left outer join

$$\text{temp} \leftarrow (\text{student} ⟕_{\text{advisor = empId}} \text{professor})$$

$$\rho_{\text{rollNo, name, advisor}} (\pi_{\text{rollNo, student.name, professor.name}} (\text{temp}))$$

Result

rollNo	name	advisor
CS04S001	Mahesh	Giridhar
CS05S001	Amrish	Null
CS04M002	Piyush	Giridhar
ES04M001	Deepak	Null
ME04M001	Lalitha	Tahir Nayyer
ME03M002	Mahesh	Tahir Nayyer

Right outer join

$$\text{temp} \leftarrow (\text{student} \bowtie\!\!\!\!\lhd\!\!\sqsubset_{\text{advisor = empId}} \text{professor})$$

$$\rho_{\text{rollNo, name, advisor}} (\pi_{\text{rollNo, student.name, professor.name}} (\text{temp}))$$

Result

rollNo	**name**	**advisor**
CS04S001	Mahesh	Giridhar
CS04M002	Piyush	Giridhar
null	null	Keshav Murthy
null	null	Rajiv Guptha
ME04M001	Lalitha	Tahir Nayyer
ME03M002	Mahesh	Tahir Nayyer

Full outer join

$\text{temp} \leftarrow (\text{student} ⟗_{\text{advisor = empId}} \text{professor})$

$\rho_{\text{roll no, name, advisor}} (\pi_{\text{roll No, student.name, professor.name}} (\text{temp}))$

Result

rollNo	name	advisor
CS04S001	Mahesh	Giridhar
CS04M002	Piyush	Giridhar
CS05S001	Amrish	Null
null	null	Keshav Murthy
ES04M001	Deepak	Null
null	null	Rajiv Guptha
ME04M001	Lalitha	Tahir Nayyer
ME03M002	Mahesh	Tahir Nayyer

E/R diagrams to Relational Schema

- E/R model and the relational model are logical representations of real world enterprises
- An E/R diagram can be converted to a collection of tables
- For each entity set and relationship set in E/R diagram we can have a corresponding relational table with the same name as entity set / relationship set
- Each table will have multiple columns whose names are obtained from the attributes of entity types/relationship types

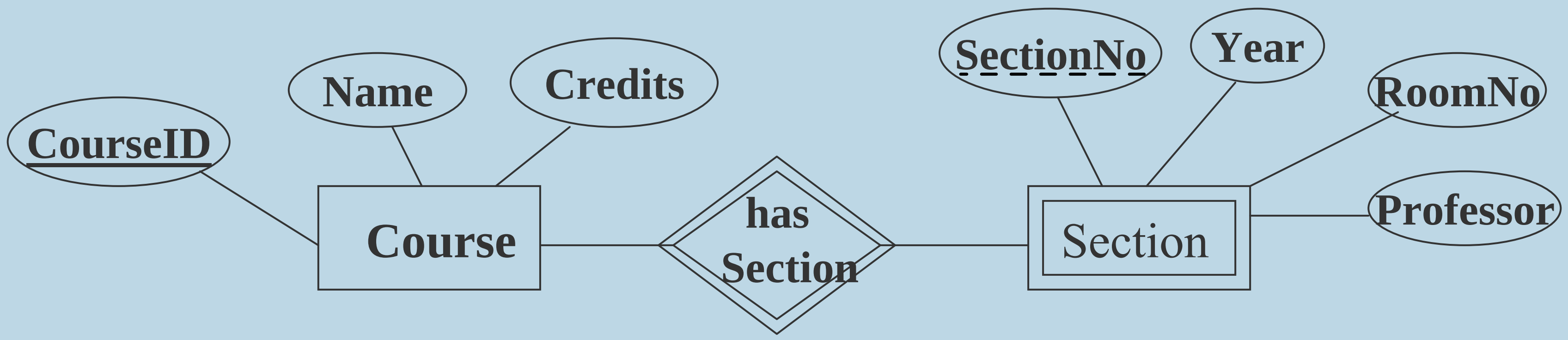
Relational representation of strong entity sets

- Create a table T_i for each strong entity set $E_{i.}$
- Include simple attributes and simple components of composite attributes of entity set E_i as attributes of T_i.
    - Multi-valued attributes of entities are dealt with separately.
- The primary key of E_i will also be the primary key of T_i.
- The primary key can be referred to by other tables via foreign keys in them to capture relationships as we see later

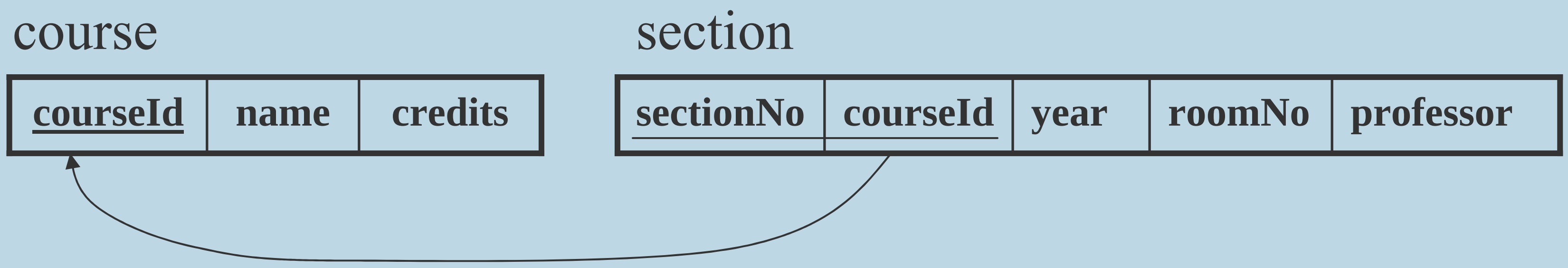
Relational representation of weak entity sets

- Let E' be a weak entity owned by a strong/weak entity E
- E' is converted to a table, say R'
- Attributes of R' will be
  - Attributes of the weak entity set E' and
  - Primary key attributes of the identifying strong entity E
  - (Or, partial key of E + primary key of the owner of E, if E is itself a weak entity)
    - These attributes will also be a foreign key in R' referring to the table corresponding to E
- Key of R' : partial key of E' + Key of E
- Multi-valued attributes are dealt separately as described later

Example



Corresponding tables are



course

<u>courseId</u>	name	credits

section

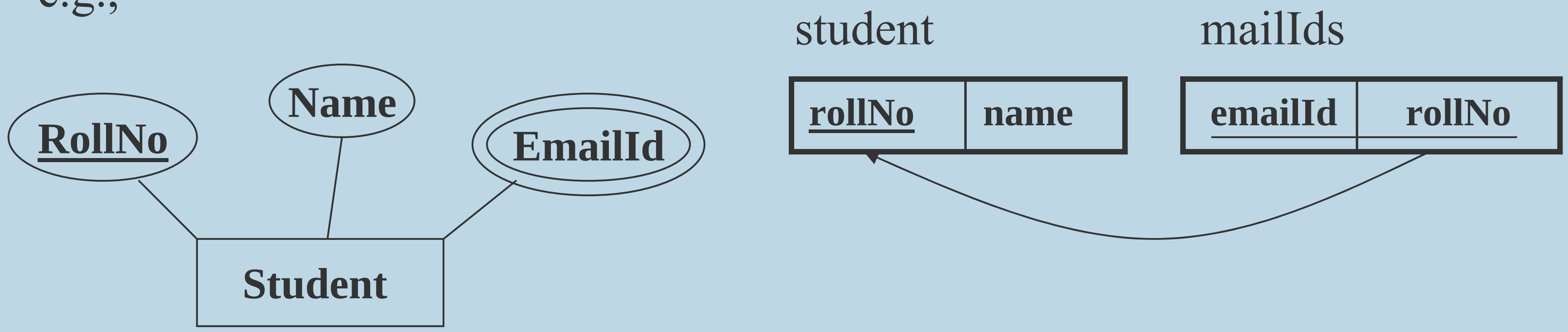
<u>sectionNo</u>	<u>courseId</u>	year	roomNo	professor

Primary key of *section* = {courseId, sectionNo}

Relational representation of multi-valued attributes

- One table for each multi-valued attribute
- One column for this attribute and
- One column for the primary key attribute of entity / relationship set for which this is an attribute.

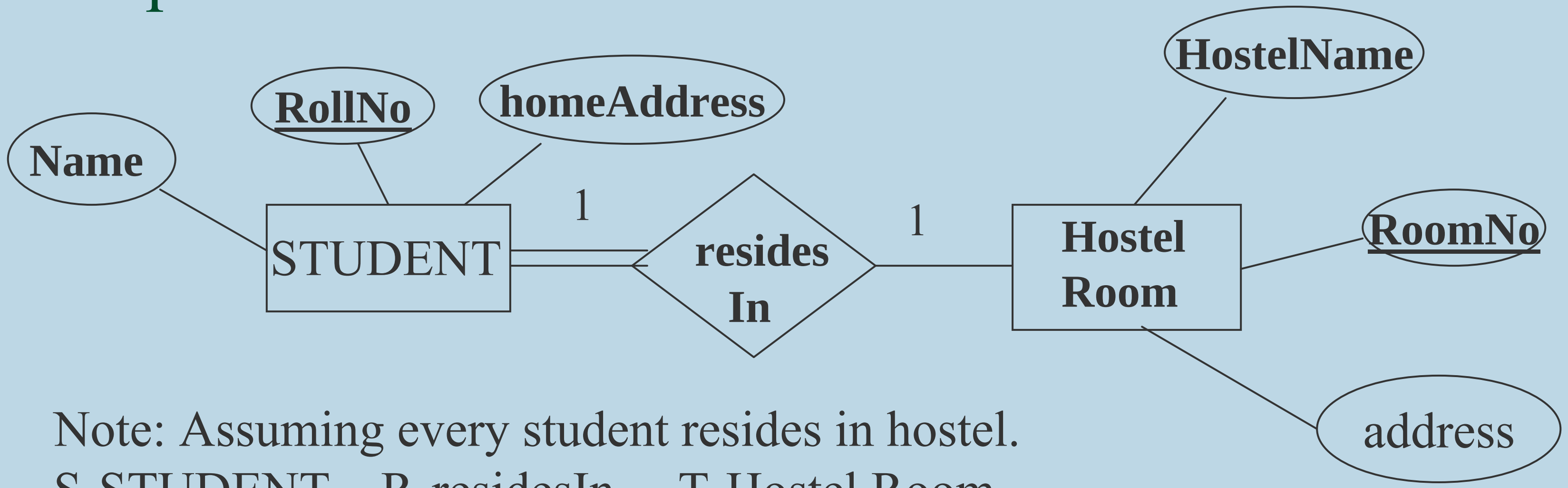
e.g.,



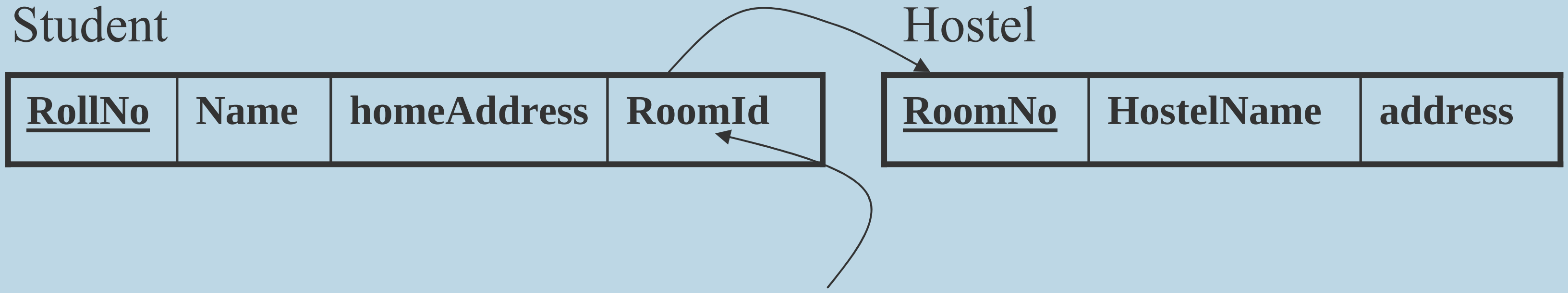
Handling Binary 1:1 Relationship

- Let S and T be entity sets in relationship R and S', T' be the tables corresponding to these entity sets
- Choose an entity set which has total participation if there is one (says, S)
- Include the primary key of T' as a foreign key in S' referring to relation T'
- Include all simple attributes (and simple components of composite attributes) of R as attributes of S'
- We can do the other way round too
  – lot of null values

Example



Note: Assuming every student resides in hostel.
S-STUDENT R-residesIn T-Hostel Room



Student

<u>RollNo</u>	Name	homeAddress	RoomId

Hostel

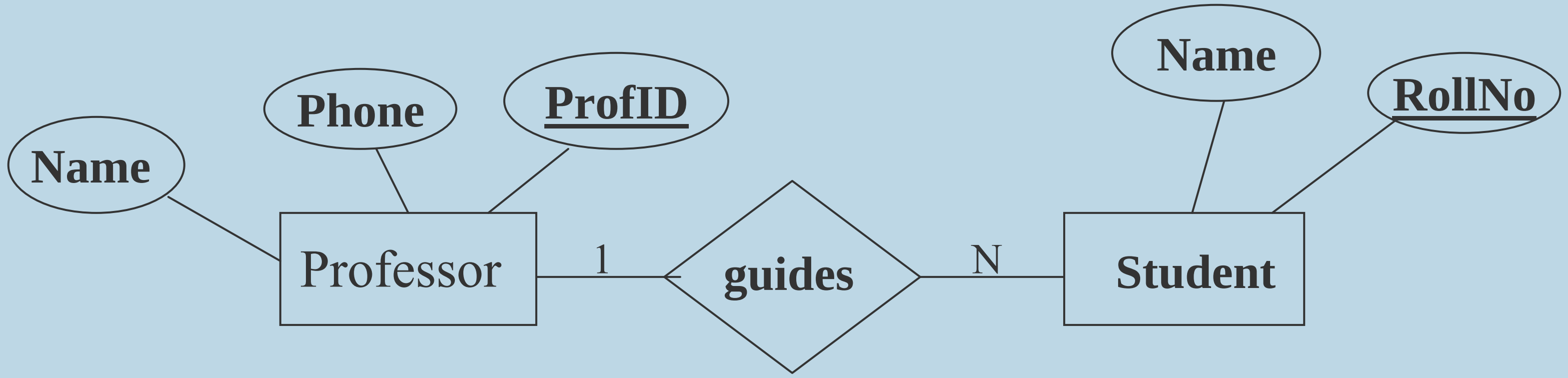
<u>RoomNo</u>	HostelName	address

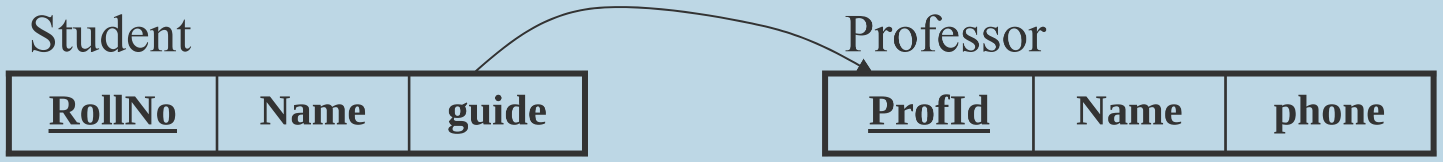
Foreign key name need
not be same as primary key
of the other relation

Handling 1:N Relationship

- Let S be the participating entity on the N-side and T the other entity. Let S' and T' be the corresponding tables.
- Include primary key of T' as foreign key in S'
- Include any simple attribute (and simple components of composite attributes) of 1:N relation type as attributes of S'

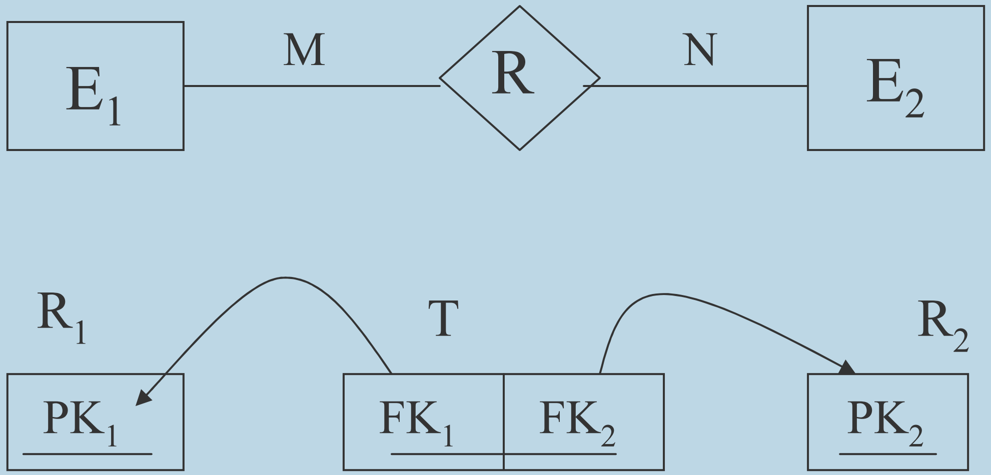
Example


Name
Phone
ProfID
Professor
1
guides
N
Student
Name
RollNo

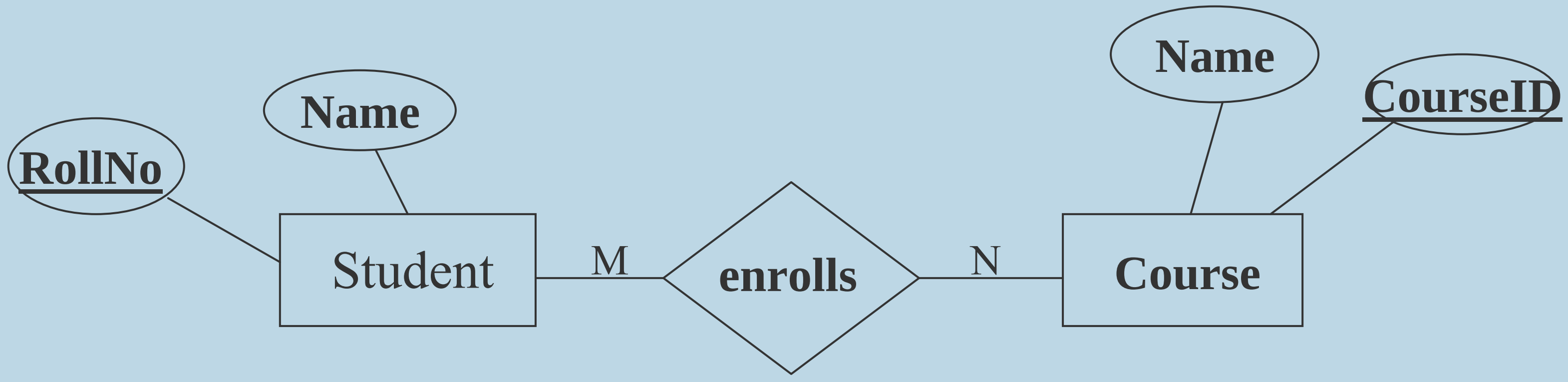

Student
RollNo
Name
guide
Professor
ProfId
Name
phone

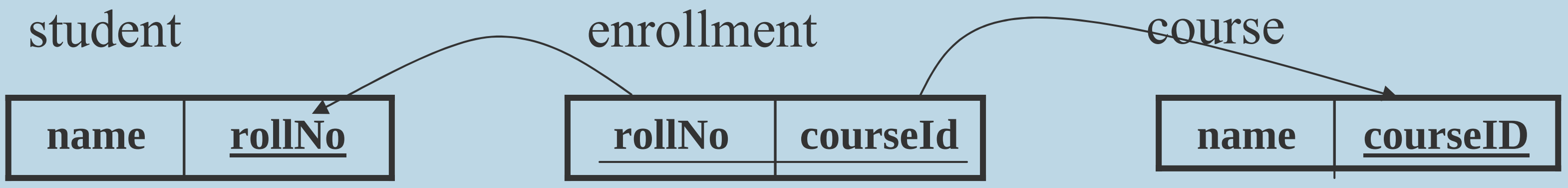
Handling M:N relationship

- Make a separate table T for this relationship R between entity sets E_1 and E_2.
  Let R_1 and R_2 be the tables corresponding to E_1 and E_2.
- Include primary key attributes of R_1 and R_2 as foreign keys in T. Their combination is the primary key in T.



Example



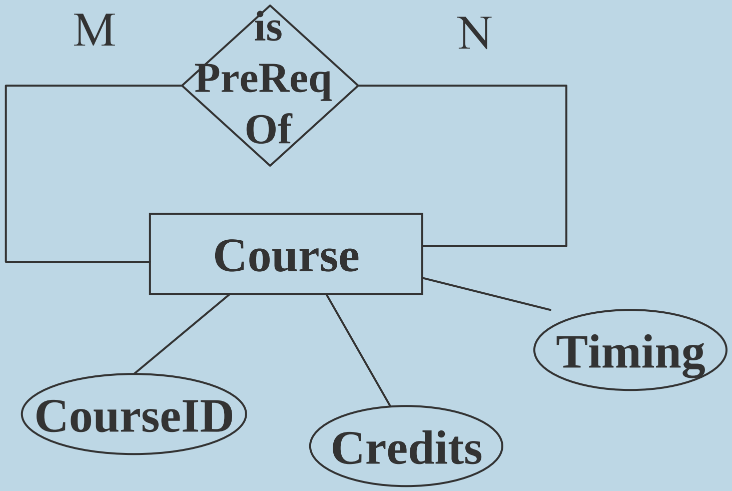


Primary key of *enrollment* table is {RollNo, CourseID}

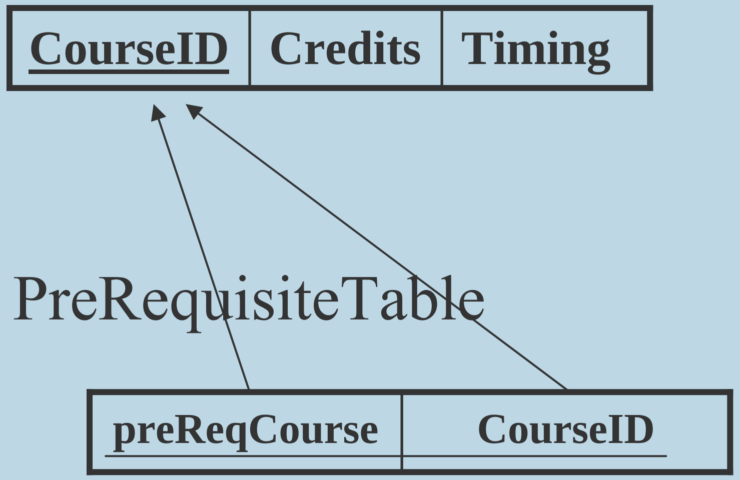
Handling Recursive relationships

- Make a table T for the participating entity set E (this might already be existing) and one table for recursive relationship R.

Example



CourseTable



<u>CourseID</u>	Credits	Timing

PreRequisiteTable

<u>preReqCourse</u>	<u>CourseID</u>

The SQL Standard

- SQL – Structured Query Language
  a 'standard' that specifies how
  - a relational schema is created
  - data is inserted / updated in the relations
  - data is queried
  - transactions are started and stopped
  - programs access data in the relations
  - and a host of other things are done

- Every relational database management system (RDBMS) is required to support / implement the SQL standard.

History of SQL

SEQUEL

- developed by IBM in early 70's
- relational query language as part of System-R project at IBM San Jose Research Lab.
- the earliest version of SQL

SQL evolution

- SQL- 86/89
- SQL- 92 - SQL2
- SQL- 99/03 - SQL3

(includes object relational features)

And the evolution continues.

Components of SQL Standard(1/2)

- *Data Definition Language* (DDL)
  Specifies constructs for schema definition, relation definition, integrity constraints, views and schema modification.

- *Data Manipulation Language* (DML)
  Specifies constructs for inserting, updating and querying the data in the relational instances (or tables).

- *Embedded SQL and Dynamic* SQL
  Specifies how SQL commands can be embedded in a high-level host language such as C, C++ or Java for programmatic access to the data.

Components of SQL Standard(2/2)

- *Transaction Control*

  Specifies how transactions can be started / stopped, how a set of concurrently executing transactions can be managed.

- *Authorization*

  Specifies how to restrict a user / set of users to access only certain parts of data, perform only certain types of queries etc.

Data Definition in SQL

Defining the schema of a relation

create table *r* (attributeDefinition-1, attributeDefinition-2,…,

name of the relation

attributeDefinition-n, [integrityConstraints-1],

[integrityConstraints-2],…,[integrityConstraints-m])

Attribute Definition –

attribute-name domain-type [NOT NULL] [DEFAULT v]

E.g.:

create table *example1* (A char(6) not null default "000000",

B int, C char (1) default "F");

Domain Types in SQL-92 (1/2)

- *Numeric data types*
  - integers of various sizes – INT, SMALLINT
  - real numbers of various precision – REAL, FLOAT, DOUBLE PRECISION
  - formatted numbers – DECIMAL (i, j) or NUMERIC (i, j)

    i – total number of digits (precision)

    j – number of digits after the decimal point (scale)
- *Character string data types*
  - fixed length – CHAR(n) – n: no. of characters
  - varying length – VARCHAR(n) – n: max.no. of characters
- *Bit string data types*
  - fixed length – BIT(n)
  - varying length – BIT VARYING(n)

Domain Types in SQL-92 (2/2)

- *Date data type*

  DATE type has 10 position format – YYYY-MM-DD

- *Time data type*

  TIME type has 8 position format – HH : MM : SS

- *Others*

  There are several more data types whose details are available in SQL reference books

Specifying Integrity Constraints in SQL

Also called Table Constraints

Included in the definition of a table

Key constraints

PRIMARY KEY (A_1,A_2,…,A_k)

specifies that {A_1,A_2,…,A_k} is the primary key of the table

UNIQUE (B_1,B_2,…,B_k)

specifies that {B_1,B_2,…,B_k} is a candidate key for the table

There can be more than one UNIQUE constraint but only one PRIMARY KEY constraint for a table.

Specifying Referential Integrity Constraints

FOREIGN KEY (A_1) REFERENCES r_2 (B_1)

- specifies that attribute A_1 of the table being defined, say r_1, is a *foreign key* referring to attribute B_1 of table r_2
- recall that this means:
  each value of column A_1 is either null or is one of the values appearing in column B_1 of r_2

Specifying What to Do if RIC Violation Occurs

RIC violation

- can occur if a referenced tuple is deleted or modified
- action can be specified for each case using qualifiers

  ON DELETE or ON UPDATE

Actions

- three possibilities can be specified

  SET NULL, SET DEFAULT, CASCADE

- these are actions to be taken on the referencing tuple
- SET NULL – foreign key attribute value to be set null
- SET DEFAULT – foreign key attribute value to be set to its default value
- CASCADE – delete the referencing tuple if the referenced tuple is deleted or update the FK attribute if the referenced tuple is updated

Table Definition Example

```
create table students (
       rollNo char(8) not null,
       name varchar(15) not null,
       degree char(5),
       year smallint,
       sex char not null,
       deptNo smallint,
       advisor char(6),
       primary key(rollNo),
       foreign key(deptNo) references
                                department(deptId)
              on delete set null on update cascade,
       foreign key(advisor) references
                                professor(empId)
              on delete set null on update cascade
 );
```

Modifying a Defined Schema

ALTER TABLE command can be used to modify a schema

Adding a new attribute

    ALTER table student ADD address varchar(30);

Deleting an attribute

- need to specify what needs to be done about views or constraints that refer to the attribute being dropped
- two possibilities
    - CASCADE – delete the views/constraints also
    - RESTRICT – do not delete the attributes if there are some views/constraints that refer to it.
- ALTER TABLE student DROP degree RESTRICT

    Similarly, an entire table definition can be deleted

Data Manipulation in SQL

Basic query syntax

select $A_1, A_2, \ldots, A_m$ ← a set of attributes from relations $R_1, \ldots, R_p$ that are required in the output table.

from $R_1, R_2, \ldots, R_p$ ← the set of tables that contain the relevant tuples to answer the query.

where θ ← a boolean predicate that specifies when a combined tuple of $R_1, \ldots, R_p$ contributes to the output.

Equivalent to:

$$\pi_{A_1, A_2, \ldots, A_n}(\sigma_\theta(R_1 \times R_2 \times \ldots\ldots \times R_p))$$

Assuming that each attribute name appears exactly once in the table.

Meaning of the Basic Query Block

- The *cross product M* of the tables in the from clause would be considered.

  Tuples in *M* that satisfy the condition θ are *selected*.

  For each such tuple, values for the attributes $A_1, A_2, \ldots, A_m$ (mentioned in the select clause) are *projected*.
- This is a conceptual description
  - in practice more efficient methods are employed for evaluation.
- The word *select* in SQL should not be confused with select operation of relational algebra.

SQL Query Result

The result of any SQL query

- a table with *select* clause attributes as column names.
- duplicate rows may be present.
  - differs from the definition of a relation.
- duplicate rows can be eliminated by specifying DISTINCT keyword in the *select* clause, if necessary.

```
SELECT DISTINCT name
FROM student
```

- duplicate rows are essential while computing aggregate functions (average, sum etc).
- removing duplicate rows involves additional effort and is done only when necessary.

Example Relational Scheme

student (rollNo, name, degree, year, sex, deptNo, advisor)

department (deptId, name, hod, phone)

professor (empId, name, sex, startYear, deptNo, phone)

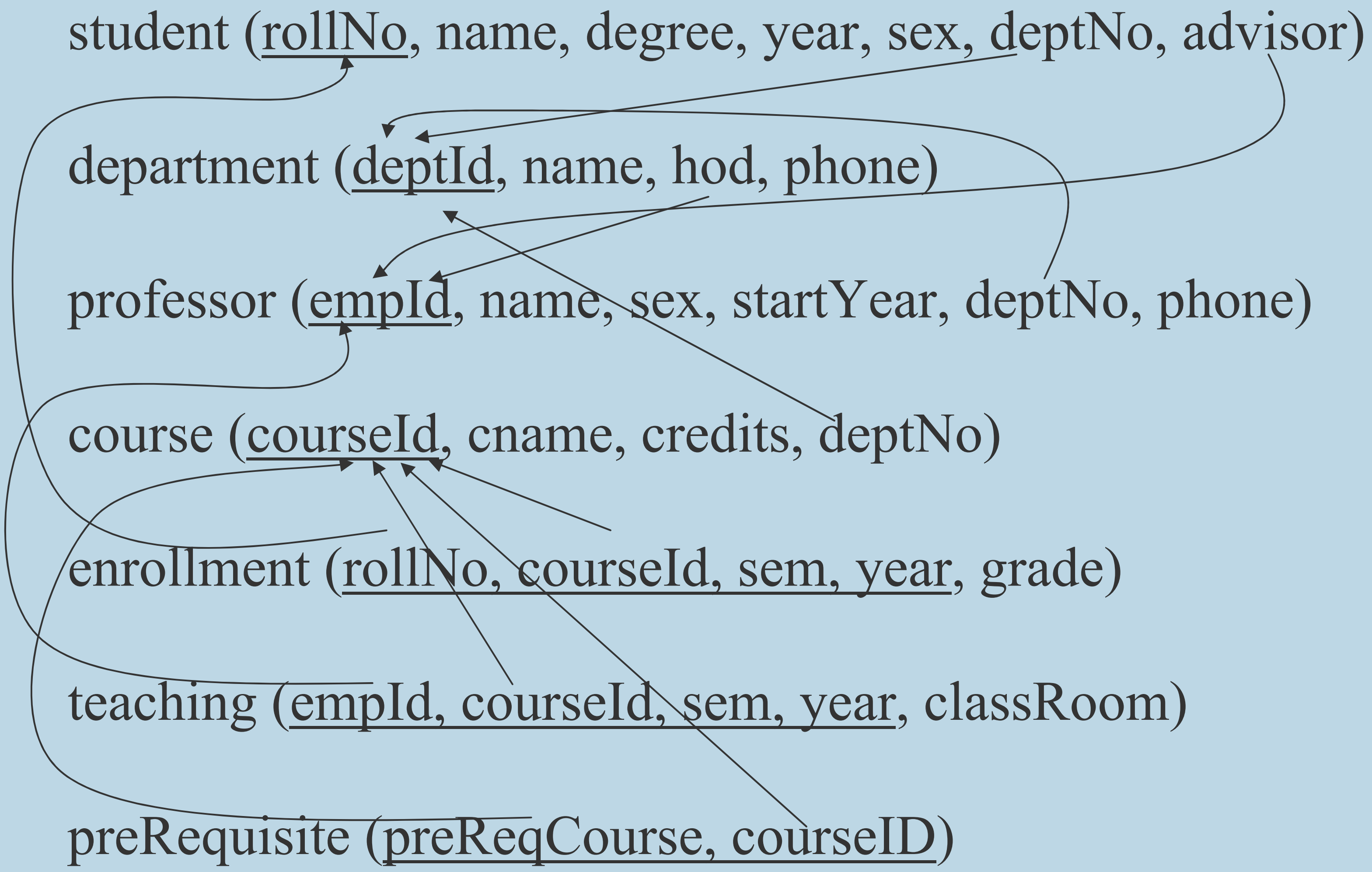
course (courseId, cname, credits, deptNo)

enrollment (rollNo, courseId, sem, year, grade)

teaching (empId, courseId, sem, year, classRoom)

preReq (preCourseId, courseId)

Example Relational Scheme with RIC's shown



student (rollNo, name, degree, year, sex, deptNo, advisor)

department (deptId, name, hod, phone)

professor (empId, name, sex, startYear, deptNo, phone)

course (courseId, cname, credits, deptNo)

enrollment (rollNo, courseId, sem, year, grade)

teaching (empId, courseId, sem, year, classRoom)

preRequisite (preReqCourse, courseID)

Example Queries Involving a Single Table

Get the rollNo, name of all women students in the dept no. 5.

```
select rollNo, name
from student
where sex = ‘F’ and deptNo = ‘5’;
```

Get the employee Id, name and phone number of professors in the CS dept (deptNo = 3) who have joined after 1999.

```
select empId, name, phone
from professor
where deptNo = 3 and startYear > 1999;
```

Examples Involving Two or More Relations (1/2)

Get the rollNo, name of students in the CSE dept (deptNo = 3)along with their advisor’s name and phone number.

```
select rollNo, s.name, f.name as advisorName,
               phone as advisorPhone
from student as s, professor as f
where s.advisor = f.empId and
             s.deptNo = ‘3’;
```

attribute renaming in the output

table aliases are required if an attribute name appears in more than one table.
Also when *same* relation appears twice in the from clause.

table aliases are used to disambiguate the common attributes

Examples Involving Two or More Relations (2/2)

Get the names, employee ID’s, phone numbers of professors in CSE dept who joined before 1995.

```
select empId, f.name, phone
from professor as f, department as d
where f.deptNo = d.deptId and
      d.name = ‘CSE’ and
      f.startYear < 1995
```

Nested Queries or Subqueries

While dealing with certain complex queries

- beneficial to specify part of the computation as a separate query & make use of its result to formulate the main query.
- such queries – nested / subqueries.

Using subqueries

- makes the main query easy to understand / formulate
- sometimes makes it more efficient also
    - sub query result can be computed once and used many times.
        - not the case with all subqueries.

Nested Query Example

Get the rollNo, name of students who have a lady professor as their advisor.

IN Operator: One of the ways of making use of the subquery result

Subquery computes the empId's of lady professors

```
select s.rollNo, s.name
from student s
where s.advisor IN
        (select empId
         from professor
         where sex = 'F');
```

NOT IN can be used in the above query to get details of students who don't have a lady professor as their advisor.

Set Comparison Operators

SQL supports several operators to deal with subquery results or in general with collection of tuples.

Combination of { =, <, ≤, ≥, >, < > } with keywords { ANY, ALL } can be used as set comparison operators.

`Get the empId, name of the senior-most Professor(s):`

```
select p.empId, p.name
from professors p
where p.startYear <= ALL ( select distinct startYear
                           from professor );
```

Semantics of Set Comparison Operators

op is one of $<, \leq, >, \geq, =, <>$

- *v op* ANY *S*
  true if for some member *x* of *S*, *v* op *x* is true
  false if for no member *x* of *S*, *v* op *x* is true

S is a subquery

- *v op* ALL *S*
  true if for every member *x* of *S*, *v* op *x* is true
  false if for some member *x* of *S*, *v* op *x* is not true

- IN is equivalent to = ANY
  NOT IN is equivalent to < > ALL

- *v* is normally a single attribute, but while using IN or NOT IN it can be a tuple of attributes

Correlated Nested Queries

If the nested query result is <u>independent</u> of the current tuple being examined in the outer query,

nested query is called *uncorrelated,*

otherwise, nested query is called *correlated.*

Uncorrelated nested query

- nested query needs to be computed only once.

Correlated nested query

- nested query needs to be re-computed for each row examined in the outer query.

Example of a Correlated Subquery

Get the roll number and name of students whose gender is same as their advisor’s.

```
select s.rollNo, s.name
from student s
where s.sex = ALL ( select f.sex
                    from professor f
                    where f.empId = s.advisor );
```

EXISTS Operator

Using *EXISTS*, we can check if a subquery result is non-empty
EXISTS (S) is true if *S* has at least one tuple / member
is false if *S* contain no tuples

```
Get the employee Id and name of professors
who advise at least one women student.
```

```
select f.empId, f.name
from professors f
where EXISTS ( select s.rollNo
               from student s
               where s.advisor = f.empId and
                     s.sex = 'F' );
```

a correlated subquery

SQL does not have an operator for universal quantification.

NOT EXISTS Operator

Obtain the department Id and name of departments that do not offer any 4 credit courses.

```
select d.deptId, d.name
from department d
where NOT EXISTS ( select courseId
                   from course c
                   where c.deptNo = d.deptId and
                         c.credits = ‘4’ );
```

Queries with *existentially* quantified predicates can be easily specified using *EXISTS* operator.

Queries with *universally* quantified predicates can only be specified after translating them to use *existential* quantifiers.

Example Involving Universal Quantifier

`Obtain the department Id and name of departments whose courses are all 3-credit courses.`

Equivalently, obtain the department Id and name of departments that do not offer a single course that is not 3-credit course

```
select d.deptNo, d.name
from department d
where NOT EXISTS ( select c.courseId
                   from course c
                   where c.deptNo = d.deptId and
                         c.credits ≠ 3);
```

Missing *where* Clause

If the *where* clause in an SQL query is not specified, it is treated as the where condition is true for all tuple combinations.

- Essentially no filtering is done on the cross product of from clause tables.

```
Get the name and contact phone of all Departments.

      select name,phone
      from department
```

Union, Intersection and Difference Operations

- In SQL, using operators *UNION, INTERSECT* and *EXCEPT*, one can perform set *union, intersection* and *difference* respectively.

- Results of these operators are sets –
  i.e duplicates are automatically removed.

- Operands need to be union compatible and also have *same* attributes in the *same* order.

Example using UNION

Obtain the roll numbers of students who are currently enrolled for either CS230 or CS232 courses.

```
(SELECT rollNo
 FROM enrollment
 WHERE courseId = ‘CS230’ and
             sem = odd and year = 2005 ) UNION
(SELECT rollNo
 FROM enrollment
 WHERE courseId = ‘CS232’ and
            sem = odd and year = 2005 );
```

Equivalent to:

```
(SELECT rollNo
FROM enrollment
WHERE (courseId = ‘CS230’ or courseID = ‘CS232’)
   and sem = odd and year = 2005 )
```

Example using INTERSECTION

Obtain the roll numbers of students who are currently enrolled for both CS230 and CS232 Courses.

```
select rollNo
from enrollment
where courseId = ‘CS230’ and
      sem = odd and
      year = 2005
INTERSECT
select rollNo
from enrollment
where courseId = ‘CS232’ and
      sem = odd and year = 2005;
```

Example using EXCEPT

Obtain the roll numbers of students who are currently not enrolled for CS230 course.

```
(SELECT rollNo
FROM enrollment
WHERE sem = odd and year = 2005 )
EXCEPT
        (SELECT rollNo
        FROM enrollment
        WHERE courseId = ‘CS230’ and
        sem = odd and year = 2005);
```

Aggregation of Data

Data analysis

- need for computing aggregate values for data
- total value, average value etc

Aggregate functions in SQL

- five aggregate function are provided in SQL
- AVG, SUM, COUNT, MAX, MIN
- can be applied to any column of a table
- can be used in the *select* clause of SQL queries

Aggregate functions

Optional keyword

- AVG ([DISTINCT]A):
  computes the average of (distinct) values in column A
- SUM ([DISTINCT]A):
  computes the sum of (distinct) values in column A
- COUNT ([DISTINCT]A):
  computes the number of (distinct) values in column A or no. of tuples in result
- MAX (A): computes the maximum of values in column A
- MIN (A): computes the minimum of values in column A

Examples involving aggregate functions (1/2)

Suppose data about Gate in a particular year is available in a table with schema

gateMarks(<u>regNo</u>,name,sex,branch,city,state,marks)

Obtain the number of students who have taken GATE in CS and their average marks

```
Select count(regNo) as CsTotal avg(marks) as CsAvg
from gateMarks
where branch = ‘CS’
```

Output

CStotal	CSavg

Get the maximum, minimum and average marks obtained by Students from the city of Hyderabad

```
Select max(marks), min(marks), avg(marks)
from gateMarks
where city = ‘Hyderabad’;
```

Examples involving aggregate functions (2/2)

Get the names of students who obtained the maximum marks in the branch of EC

```
Select name, max(marks)
from gateMarks
where branch = ‘EC’
```

Will not work

Only aggregate functions can be specified here. It does not make sense to include normal attributes ! (unless they are grouping attributes – to be seen later)

```
Select regNo, name, marks
from gateMarks
where branch = ‘EC’ and marks =
                   (select max(marks)
                    from gateMarks
                    where branch = ‘EC’);
```

Correct way of specifying the query

Date Aggregation and Grouping

Grouping

- Partition the set of tuples in a relation into groups based on certain criteria and compute aggregate functions for each group
- All tuples that agree on a set of attributes (i.e have the same value for each of these attributes) are put into a group

Called the grouping attributes

- The specified aggregate functions are computed for each group
- Each group contributes one tuple to the output
- All the grouping attributes *must* also appear in the select clause
  - the result tuple of the group is listed along with the values of the grouping attributes of the group

Examples involving grouping(1/2)

Determine the maximum of the GATE CS marks obtained by students in each city, for all cities.

```
Select city, max(marks) as maxMarks
from gateMarks
where branch = 'CS'
group by city;
```

Grouping attributes must appear in the select clause

Grouping attribute

Result:

City	maxMarks
Hyderabad	87
Chennai	84
Mysore	90
Bangalore	82

Examples involving grouping(2/2)

In the University database, for each department, obtain the name, deptId and the total number of four credit courses offered by the department

```
Select deptId, name, count(*) as totalCourses
from department, course
where deptId = deptNo and credits = 4
group by deptId, name;
```

Having clause

After performing grouping, is it possible to report information about only a subset of the groups ?

- Yes, with the help of *having clause* which is always used in conjunction with Group By clause

```
Report the total enrollment in each course in the 2nd
semester of 2004; include only the courses with a
minimum enrollment of 10.
```

```
Select courseId, count(rollNo) as Enrollment
from enrollment
where sem = 2 and year = 2004
group by courseId
having count(rollNo) ≥ 10;
```

Where clause versus Having clause

- Where clause
  - Performs tests on rows and eliminates rows not satisfying the specified condition
  - Performed *before* any grouping of rows is done
- Having clause
  - Always performed *after* grouping
  - Performs tests on groups and eliminates groups not satisfying the specified condition
  - Tests can only involve grouping attributes and aggregate functions

```
Select courseId, count(rollNo) as Enrollment
from enrollment
where sem = 2 and year = 2004
group by courseId
having count(rollNo) ≥ 10;
```

String Operators in SQL

- Specify strings by enclosing them in single quotes
  e.g., ‘Chennai’

Common operations on strings –

- pattern matching – using ‘LIKE’ comparison operator

Specify patterns using special characters –

- character ‘%’ (percent) matches any Substring
  e.g., ‘Jam%’ matches any string starting with “Jam”
- character ‘_’ (underscore) matches any single character
  e.g., (a) ‘_ _ press’ matches with any string ending
  with “press”, with any two characters before that.
  (b) ‘_ _ _ _’ matches any string with exactly four
  characters

Using the ‘LIKE’ operator

Obtain roll numbers and names of all students whose names end with ‘Mohan’

```
Select rollNo, name
from student
where name like ‘%Mohan’;
```

- Patterns are case sensitive.
- Special characters (percent, underscore) can be included in patterns using an escape character ‘\’ (backslash)

Join Operation

In SQL, usually joining of tuples from different relations is specified in ‘where’ clause

```
Get the names of professors working in CSE dept.
```

```
    Select f.name
    from professor as f, department as d
    where f.deptNo = d.deptId and
          d.name = ‘CSE’;
```

The above query specifies joining of professor and department relations on condition *f.deptNo = d.deptId and d.name = ‘CSE’*

Explicit Specification of Joining in ‘From’ Clause

```
select f.name
 from (professor as f join department as d on
         f.deptNo = d.deptId)
 where d.name = ‘CSE’;
```

Join types:

1. inner join (default):

   from (r_1 inner join r_2 on <predicate>)

   use of just ‘join’ is equivalent to ‘inner join’

2. left outer join:

   from (r_1 left outer join r_2 on <predicate>)

3. right outer join:

   from (r_1 right outer join r_2 on <predicate>)

4. full outer join:

   from (r_1 full outer join r_2 on <predicate>)

Natural join

The adjective ‘natural’ can be used with any of the join types to specify natural join.

FROM (r_1 NATURAL <join type> r_2 [USING <attr. list>])

- natural join by default considers all common attributes
- a subset of common attributes can be specified in an optional using <attr. list> phrase

REMARKS

- Specifying join operation explicitly goes against the spirit of declarative style of query specification
- But the queries may be easier to understand
- The feature is to be used judiciously

Views

- Views provide virtual relations which contain data spread across different tables. Used by applications.
  - simplified query formulations
  - data hiding
  - logical data independence
- Once created, a view is always kept *up-to-date* by the RDBMS
- View is not part of conceptual schema
  - created to give a user group, concerned with a certain aspect of the information system, their *view* of the system
- Storage
  - Views need not be stored as permanent tables
  - They can be created on-the-fly whenever needed
  - They can also be *materialized*
- Tables involved in the view definition – base tables

Creating Views

CREATE VIEW v AS <query expr>

creates a view 'v', with structure and data defined by the outcome of the query expression

Create a view which contains name, employee Id and phone number of professors who joined CSE dept in or after the year 2000.

name of the view

```
        create view profAft2K as
        (Select f.name, empId, phone
         from professor as f, department as d
         where f.depNo = d.deptId and
               d.name = 'CSE' and
               f.startYear >= 2000);
```

If the details of a new CSE professor are entered into *professor* table, the above view gets updated automatically

Queries on Views

Once created a view can be used in queries just like any other table.

e.g. `Obtain names of professors in CSE dept, who joined after 2000 and whose name starts with ‘Ram’`

```
select name
from profAft2K
where name like ‘Ram%’;
```

The definition of the view is stored in DBMS, and executed to create the temporary table (view), when encountered in query

Operations on Views

- Querying is allowed

- Update operations are usually restricted

  because – updates on a view may modify many base tables

  – there may not be a unique way of updating the base tables to reflect the update on view

  – view may contain some aggregate values

  – ambiguity where primary key of a base table is not included in view definition.

Restrictions on Updating Views

- Updates on views defined on joining of more than one table are not allowed
- For example, updates on the following view are not allowed

```
create a view Professor_Dept with professor
ID, department Name and department phone

create view profDept(profId,DeptName,DPhone) as
     (select f.empId, d.name, d.phone
      from professor f, department d
      where f.depNo = d.depId);
```

Restrictions on Updating Views

- Updates on views defined with ‘group by’ clause and aggregate functions is not permitted, as a tuple in view will not have a corresponding tuple in base relation.

- For example, updates on the following view are not allowed

```
Create a view deptNumCourses which contains the
number of courses offered by a dept.
```

```
 create view deptNumCourses(deptNo,numCourses)
   as select deptNo, count(*)
      from course
      group by deptNo;
```

Restrictions on Updating Views

- Updates on views which do not include Primary Key of base table, are also not permitted
- For example, updates on the following view are not allowed

```
Create a view StudentPhone with Student name and
phone number.
```

```
  create view StudentPhone (sname,sphone) as
      (select name, phone
       from student);
```

View StudentPhone does not include Primary key of the base table.

Allowed Updates on Views

Updates to views are allowed only if

- defined on single base table
- not defined using ‘group by’ clause and aggregate functions
- include Primary Key of base table

Inserting data into a table

- Specify a tuple(or tuples) to be inserted

```
 INSERT INTO student VALUES
('CSO5D014','Mohan','PhD',2005,'M',3,'FCS008'),
('CSO5S031','Madhav','MS',2005,'M',4,'FCE009');
```

- Specify the result of query to be inserted

  INSERT INTO r_1 SELECT … FROM … WHERE …

- Specify that a sub-tuple be inserted

```
  INSERT INTO student(rollNo, name, sex)
  VALUES (CS05M022, 'Rajasri', 'F'),
         (CS05B033, 'Kalyan', 'M');
```

  - *the attributes that can be NULL or have declared default values can be left-out to be updated later*

Deleting rows from a table

- Deletion of tuples is possible ; deleting only part of a tuple is not possible
- Deletion of tuples can be done *only from one* relation at a time
- Deleting a tuple might trigger further deletions due to *referentially triggered actions* specified as part of RIC's
- Generic form: *`delete from r where <predicate>;`*

```
Delete tuples from professor relation with start year
as 1982.
```

```
        delete from professor
       where startYear = 1982;
```

- If 'where' clause is not specified, then all the tuples of that relation are deleted (Be careful !)

A Remark on Deletion

- The where predicate is evaluated for each of the tuples in the relation to mark them as qualified for deletion *before* any tuple is actually deleted from the relation
- Note that the result may be different if tuples are deleted as and when we find that they satisfy the where condition!
- An example:

  Delete all tuples of students that scored the least marks in the CS branch:

```
DELETE
FROM gateMarks
WHERE branch = “CS” and
          marks = ANY ( SELECT MIN(marks)
                        FROM gateMarks
                        WHERE branch = “CS”)
```

Updating tuples in a relation

```
update r
set <<attr = newValue> list>
where <predicates>;
```

Change phone number of all professors working in CSE dept to “94445 22605”

```
update professors
set phone = ‘9444422605’
where deptNo = (select deptId
                from department
                where name = ‘CSE’);
```

If ‘where’ clause is not specified, values for the specified attributes in all tuples is changed.

Miscellaneous features in SQL (1/3)

- Ordering of result tuples can be done using ‘order by’ clause
  e.g., `List the names of professors who joined after 1980, in alphabetic order.`

```
select name
from professor
where startYear > 1980
order by name;
```

- Use of ‘null’ to test for a null value, if the attribute can take null
  e.g., `Obtain roll numbers of students who don’t have phone numbers`

```
select rollNo
from student
where phoneNumber is null;
```

Miscellaneous features in SQL (2/3)

- Use of ‘between and’ to test the range of a value
  e.g., Obtain names of professors who have joined between 1980 and 1990

```
select name
from professor
where startYear between 1980 and 1990;
```

- Change the column name in result relation
  e.g.,

```
select name as studentName, rollNo as studentNo
from student;
```

Miscellaneous features in SQL (3/3)

- Use of ‘distinct’ key word in ‘select’ clause to determine duplicate tuples in result.

  Obtain all distinct branches of study for students

  ```
  select distinct d.name
  from student as s, department as d
  where s.deptNo = d.deptId;
  ```

- Use of asterisk (*) to retrieve all the attribute values of selected tuples.

  Obtain details of professors along with their department details.

  ```
  select *
  from professor as f, department as d
  where f.deptNo = d.deptId;
  ```

Application Development Process

Host language (HL) – the high-level programming language in which the application is developed (e.g., C, C++, Java etc.)

Database access – using *embedded* SQL is one approach
- SQL statements are interspersed in HL program.

Data transfer –
takes place through specially declared HL variables

Mismatch between HL data types and SQL data types
- SQL 92 standard specifies the corresponding SQL types for many HLs.

Declaring Variables

Variables that need to be used in SQL statements are declared in a special section as follows:

```
EXEC SQL BEGIN DECLARE SECTION
      char rollNo[9];         // HL is C language
      char studName[20], degree[6];
      int year; char sex;
      int deptNo; char advisor[9];
EXEC SQL END DECLARE SECTION
```

Note that schema for student relation is

student(<u>rollNo</u>, name, degree, year, sex, deptNo, advisor)

Use in SQL statements: variable name is prefixed with a colon(:)

e.g., :ROLLNO in an SQL statement refers to rollNo variable

Handling Error Conditions

The HL program needs to know if an SQL statement has executed successfully or otherwise

Special variable called SQLSTATE is used for this purpose

- SQLSTATE is set to appropriate value by the RDBMS run-time after executing each SQL statement
- non-zero values indicate errors in execution
  - different values indicate different types of error situations

SQLSTATE variable <u>must</u> be declared in the HL program and HL program needs to check for error situations and handle them appropriately.

Embedding SQL statements

Suppose we collect data through user interface into variables
rollNo, studName, degree, year, sex, deptNo, advisor

A row in student table can be inserted –

```
EXEC SQL INSERT INTO STUDENT
    VALUES (:rollNo,:studName,:degree,
            :year,:sex,:deptNo,:advisor);
```

Impedance mismatch and cursors

- Occurs because, HL languages do not support set-of-records as supported by SQL
- A‘cursor’ is a mechanism which allows us to retrieve one row at a time from the result of a query
- We can declare a cursor on any SQL query
- Once declared, we use open, fetch, move and close commands to work with cursors
- We usually need a cursor when embedded statement is a SELECT query
- INSERT, DELETE and UPDATE don’t need a cursor.

Embedded SQL (1/2)

We don’t need a cursor if the query results in a single row.

e.g.,
```
EXEC SQL SELECT s.name, s.sex
         INTO :name, :sex
         FROM student s
         WHERE s.rollNo = :rollNo;
```

- Result row values name and phone are assigned to HL variables :name and :phone, using ‘INTO’ clause
- Cursor is not required as the result always contains only one row (*rollNo* is a key for *student* relation)

Embedded SQL (2/2)

- If the result contains more than one row, cursor declaration is needed

e.g.,
```
select s.name, s.degree
from student s
where s.sex = ‘F’;
```

- Query results in a collection of rows
- HL program has to deal with set of records.
- The use of ‘INTO’ will not work here
- We can solve this problem by using a ‘cursor’.

Declaring a cursor on a query

Cursor name

```
declare studInfo cursor for
     select name, degree
     from student
     where sex = 'F';
```

- Command OPEN studInfo; opens the cursor and makes it point to first record
- To read current row of values into HL variables, we use the command FETCH studInfo INTO :name, :degree;
- After executing FETCH statement cursor is pointed to next row by default
- Cursor movement can be optionally controlled by the programmer
- After reading all records we close the cursor using the CLOSE studInfo command.

Dynamic SQL

- Useful for applications to generate and run SQL statements, based on user inputs
- Queries may not be known in advance

e.g.,
```
char sqlstring [ ] = {“select * from student”};
EXEC SQL PREPARE runQ FROM sqlstring;
EXEC SQL EXECUTE runQ;
```

- ‘Sqlstring’ is a ‘C’ variable that holds user submitted query
- ‘runQ’ is an SQL variable that holds the SQL statements.

Connecting to Database from HL

ODBC (Open Database Connectivity) and
JDBC (Java Database Connectivity)

- accessing database and data is through an API
- many DBMSs can be accessed
- no restriction on number of connections
- appropriate drivers are required
- steps in accessing data from a HL program
    - select the data source
    - load the appropriate driver dynamically
    - establish the connection
    - work with database
    - close the connection.

File Organization and Indexing

The data of a RDB is ultimately stored in disk files

Disk space management:

Should Operating System services be used ?

Should RDBMS manage the disk space by itself ?

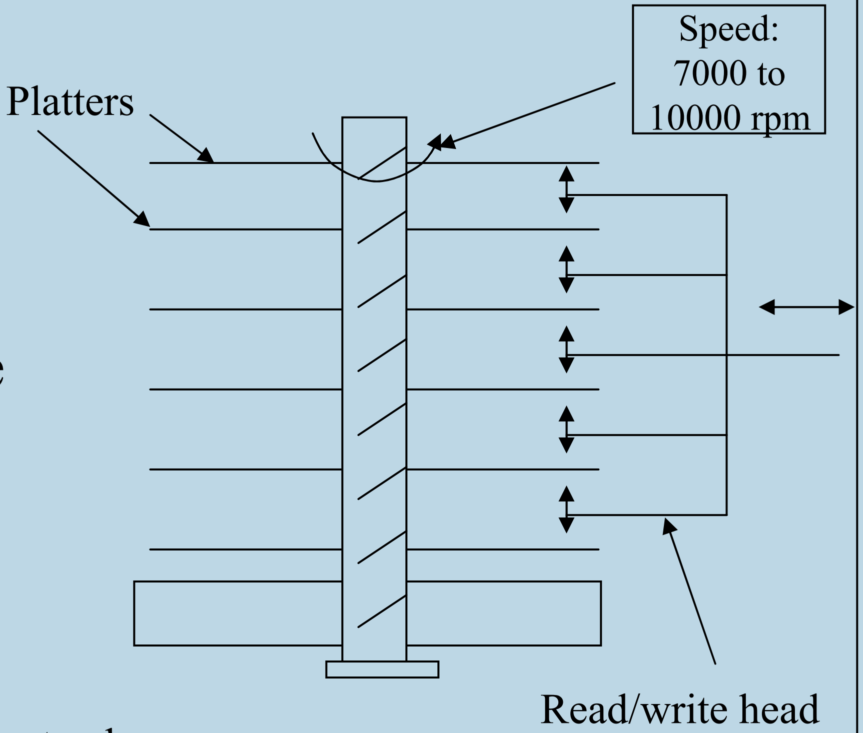
2nd option is preferred as RDBMS requires complete control over when a block or page in main memory buffer is written to the disk.

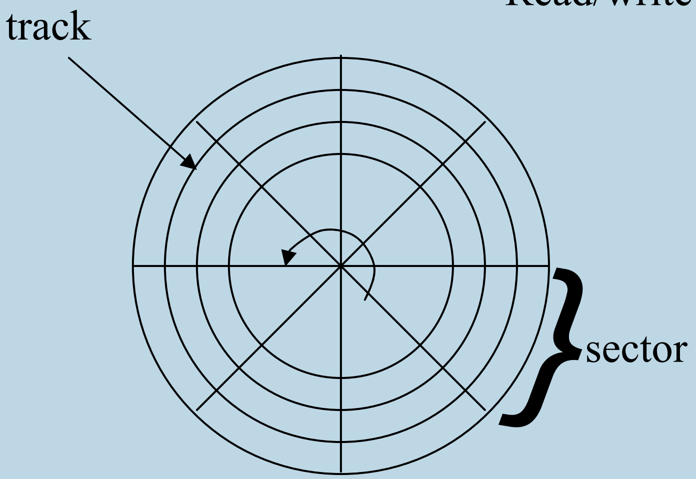
This is important for recovering data when system crash occurs

Structure of Disks

Disk

- several platters stacked on a rotating spindle
- one read / write head per surface for fast access
- platter has several tracks
  - ~10,000 per inch
- each track - several sectors
- each sector - blocks
- unit of data transfer - block
- cylinder i - track i on all platters





Data Transfer from Disk

Address of a block: Surface No, Cylinder No, Block No

Data transfer:

Move the r/w head to the appropriate track

- time needed - seek time – ~ 12 to 14 ms

Wait for the appropriate block to come under r/w head

- time needed - rotational delay - ~3 to 4ms (avg)

Access time: Seek time + rotational delay

Blocks on the same cylinder - roughly close to each other
- access time-wise

- cylinder i, cylinder (i + 1), cylinder (i + 2) etc.

Data Records and Files

Fixed length record type: each field is of fixed length

- in a file of these type of records, the record number can be used to locate a specific record
- the number of records, the length of each field are available in file header

Variable length record type:

- arise due to missing fields, repeating fields, variable length fields
- special separator symbols are used to indicate the field boundaries and record boundaries
- the number of records, the separator symbols used are recorded in the file header

Packing Records into Blocks

Record length much less than block size

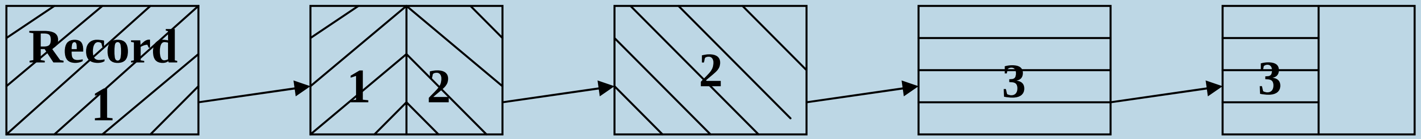
- The usual case
- Blocking factor b = $\lfloor B/r \rfloor$ B - block size (bytes)

  r - record length (bytes)

  - maximum no. of records that can be stored in a block

Record length greater than block size

- spanned organization is used



File blocks:

sequence of blocks containing all the records of the file

Mapping File Blocks onto the Disk Blocks

Contiguous allocation

- Consecutive file blocks are stored in consecutive disk blocks
- Pros: File scanning can be done fast using double buffering

  Cons: Expanding the file by including a new block in the middle of the sequence - difficult

Linked allocation

- each file block is assigned to some disk block
- each disk block has a pointer to next block of the sequence
- file expansion is easy; but scanning is slow

Mixed allocation

Operations on Files

Insertion of a new record: may involve searching for appropriate location for the new record

Deletion of a record: locating a record –may involve search; delete the record –may involve movement of other records

Update a record field/fields: equivalent to delete and insert

Search for a record: given value of a key field / non-key field

Range search: given range values for a key / non-key field

How successfully we can carry out these operations depends on the organization of the file and the availability of indexes

Primary File Organization

The logical policy / method used for placing records into file blocks

Example: *Student* file - organized to have students records sorted in increasing order of the "rollNo" values

Goal: To ensure that operations performed frequently on the file execute fast

- conflicting demands may be there
- example: on student file, access based on rollNo and also access based on name may both be frequent
- we choose to make rollNo access fast
- For making name access fast, additional access structures are needed.
  - more details later

Different File Organization Methods

We will discuss Heap files, Sorted files and Hashed files

Heap file:

Records are appended to the file as they are inserted

Simplest organization

Insertion - Read the last file block, append the record and write back the block - easy

Locating a record given values for any attribute

- requires scanning the entire file – very costly

Heap files are often used only along with other access structures.

Sorted files / Sequential files (1/2)

Ordering field: The field whose values are used for sorting the
records in the data file

Ordering key field: An ordering field that is also a key

Sorted file / Sequential file:

Data file whose records are arranged such that the values of the ordering field are in ascending order

Locating a record given the value X of the ordering field:

- Binary search can be performed
    - Address of the nth file block can be obtained from the file header
    - O(log N) disk accesses to get the required block- efficient

Range search is also efficient

Sorted files / Sequential files (2/2)

Inserting a new record:

- Ordering gets affected
  - costly as all blocks following the block in which insertion is performed may have to be modified

- Hence not done directly in the file
  - all inserted records are kept in an auxiliary file
  - periodically file is reorganized - auxiliary file and main file are merged
  - locating record
    - carried out first on auxiliary file and then the main file.

Deleting a record

- deletion markers are used.

Hashed Files

Very useful file organization, if quick access to the data record is needed given the value of a single attribute.

Hashing field: The attribute on which quick access is needed and on which hashing is performed

Data file: organized as a buckets with numbers 0,1, …, (M − 1) (bucket - a block or a few *consecutive* blocks)

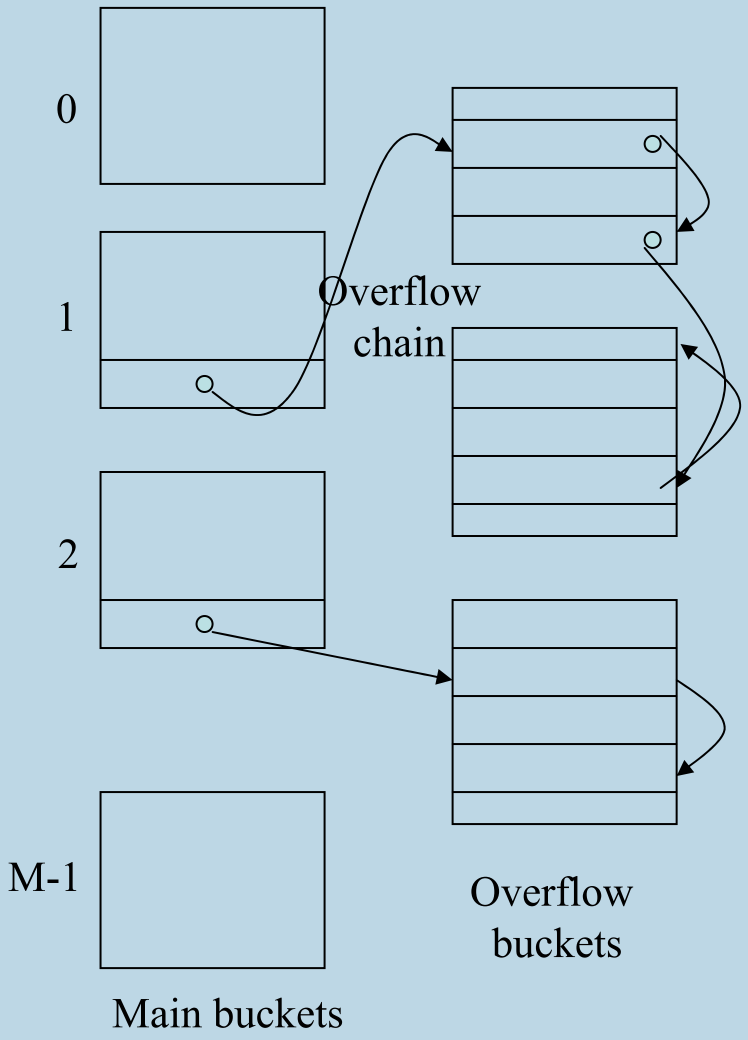
Hash function *h*: maps the values from the domain of the hashing attribute to bucket numbers

Inserting Records into a Hashed File

Insertion: for the given record R, apply *h* on the value of hashing attribute to get the bucket number *r*.

If there is space in bucket *r*, place R there else place R in the overflow chain of bucket *r*.

The overflow chains of all the buckets are maintained in the overflow buckets.

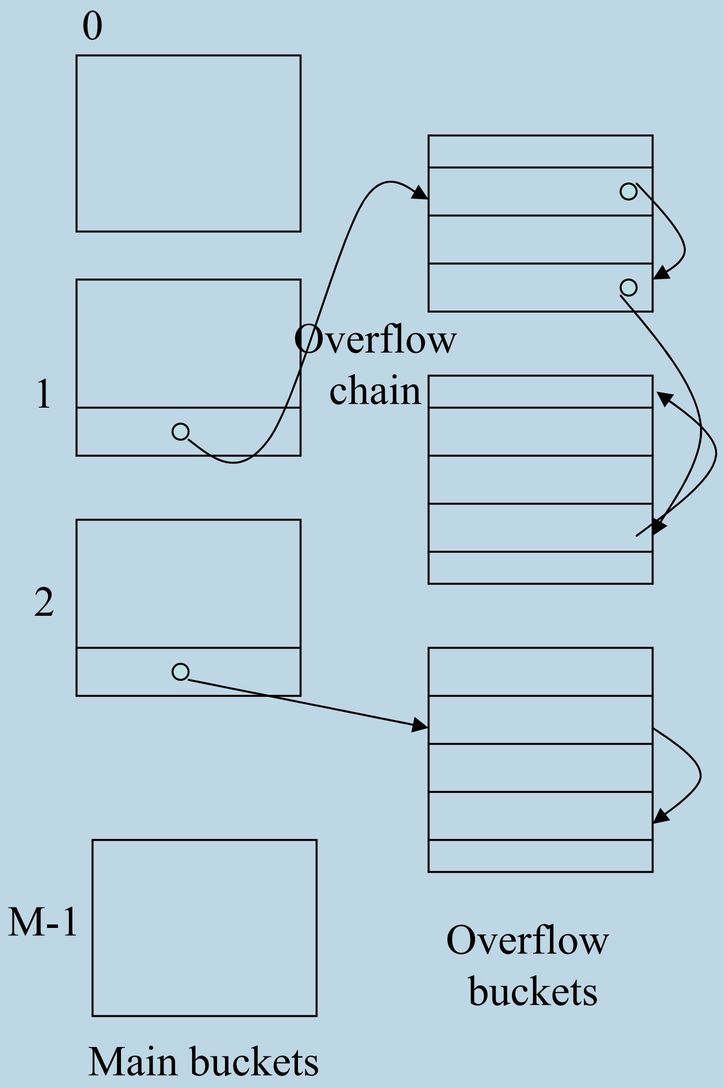


Deleting Records from a Hashed File

Deletion: Locate the record R to be deleted by applying *h*.

Remove R from its bucket/overflow chain. If possible, bring a record from the overflow chain into the bucket

Search: Given the hash filed value k, compute *r* = *h*(k). Get the bucket *r* and search for the record. If not found, search the overflow chain of bucket *r*.



Performance of Static Hashing

Static hashing:

- The hashing method discussed so far
- The number of main buckets is <u>fixed</u>

Locating a record given the value of the hashing attribute most often – one block access

Capacity of the hash file C = r * M records (r - no. of records per bucket, M - no. of main buckets)

Disadvantage with static hashing:

If actual records in the file is much less than C

- wastage of disk space

If actual records in the file is much more than C

- long overflow chains – degraded performance

Hashing for Dynamic File Organization

Dynamic files

- files where record insertions and deletion take place frequently
- the file keeps growing and also shrinking

Hashing for dynamic file organization

- Bucket numbers are integers
- The binary representation of bucket numbers
  - Exploited cleverly to devise dynamic hashing schemes
  - Two schemes
    - Extendible hashing
    - Linear hashing

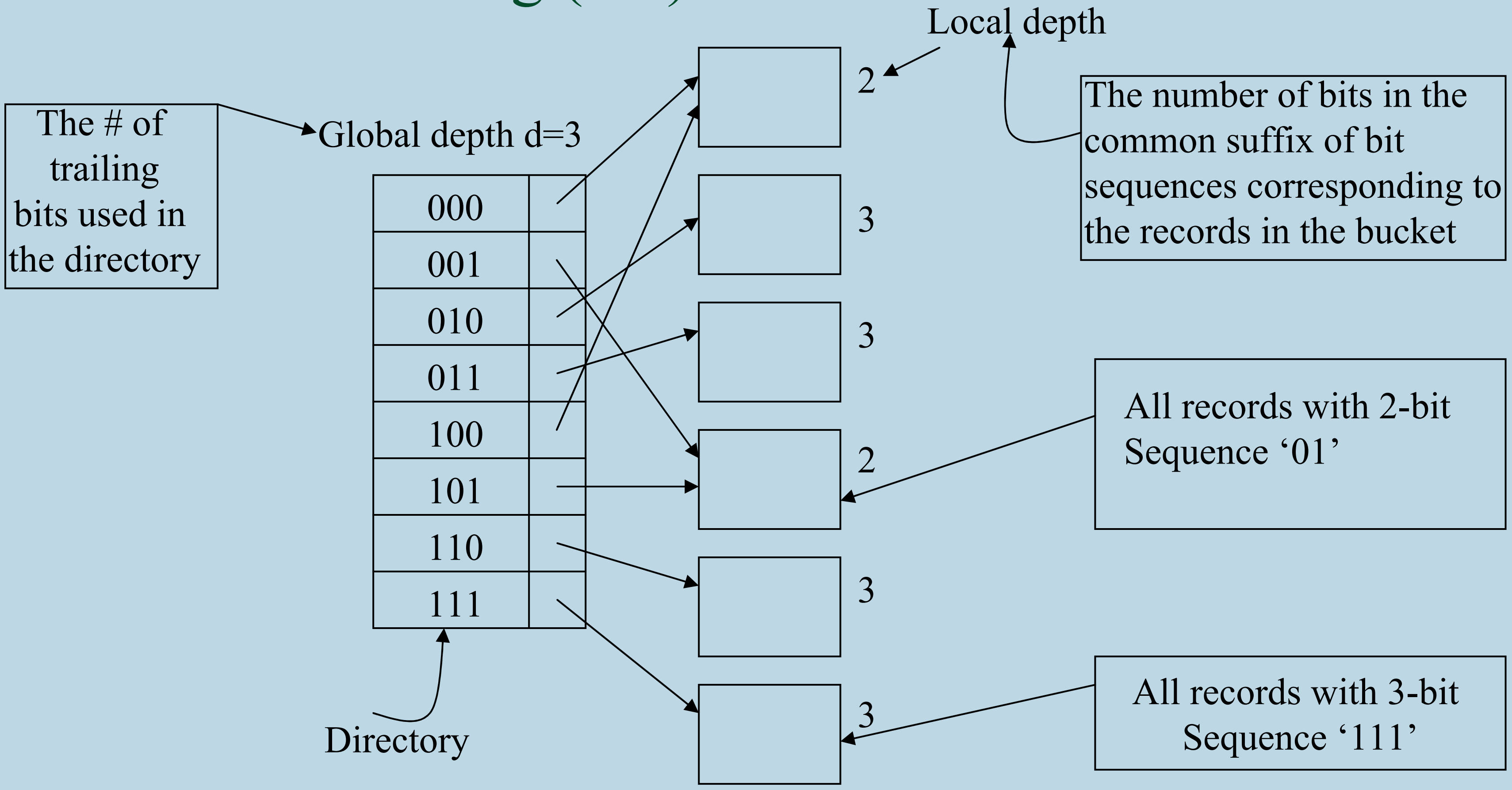
Extendible Hashing (1/2)

The *k*-bit sequence corresponding to a record R:

Apply hashing function to the value of the hashing field of R to get the bucket number *r*

Convert *r* into its binary representation to get the bit sequence
Take the *trailing k* bits

Extendible Hashing (2/2)



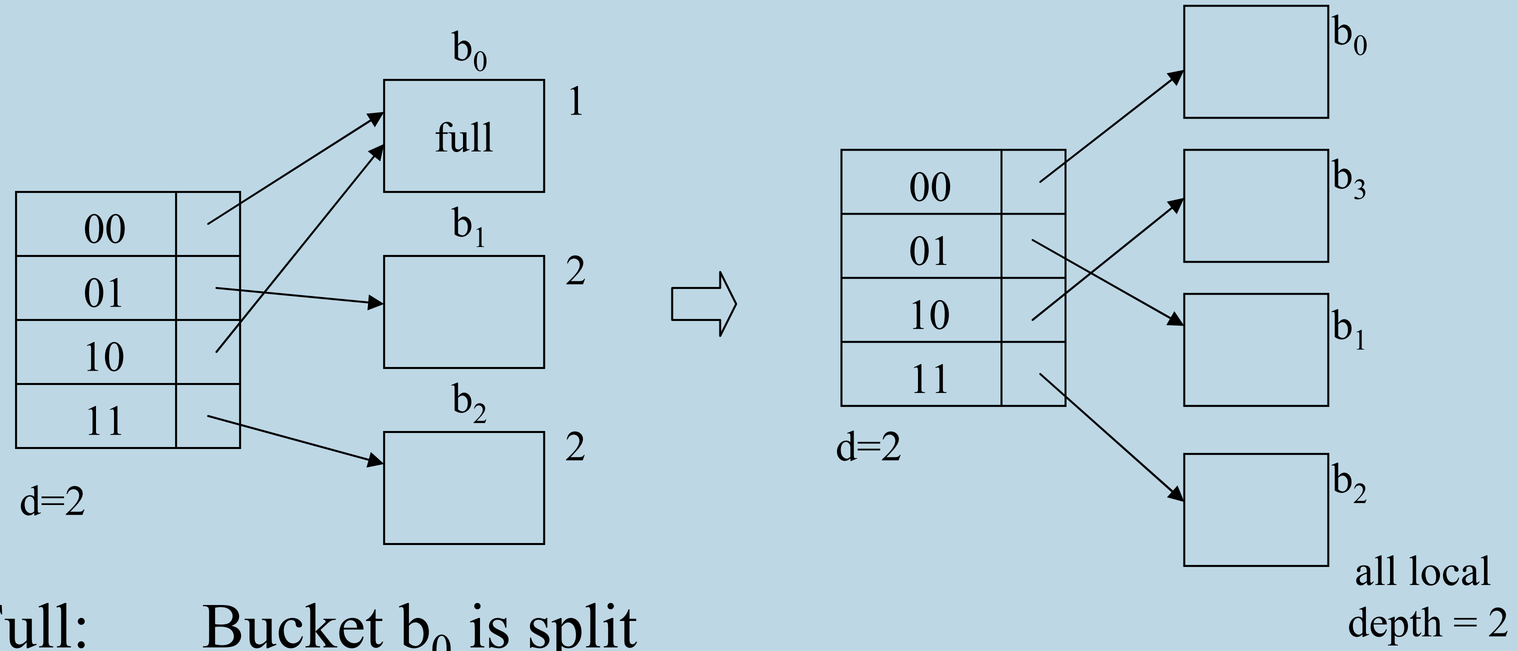
Local depth
The # of trailing bits used in the directory
Global depth d=3
000
001
010
011
100
101
110
111
Directory
2
3
3
2
3
3
The number of bits in the common suffix of bit sequences corresponding to the records in the bucket
All records with 2-bit Sequence '01'
All records with 3-bit Sequence '111'

Locating a record

Match the d-bit sequence with an entry in the directory and go to the corresponding bucket to find the record

Insertion in Extendible Hashing Scheme (1/2)

2 - bit sequence for the record to be inserted: 00



b_0 Full: Bucket b_0 is split

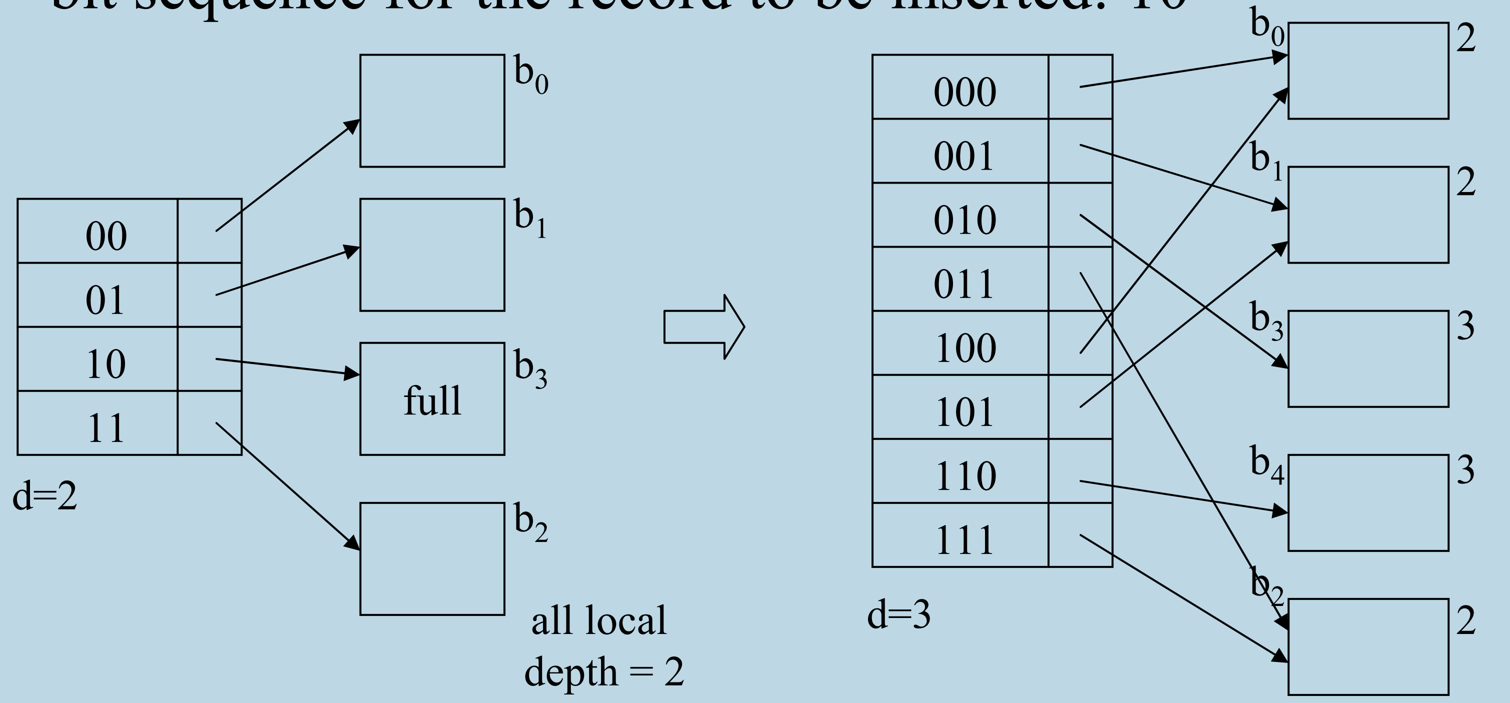
All records whose 2-bit sequence is ‘10’ are sent to a new bucket b_3. Others are retained in b_0

Directory is modified.

b_0 Not full: New record is placed in b_0. No changes in the directory.

Insertion in Extendible Hashing Scheme (2/2)

2 - bit sequence for the record to be inserted: 10

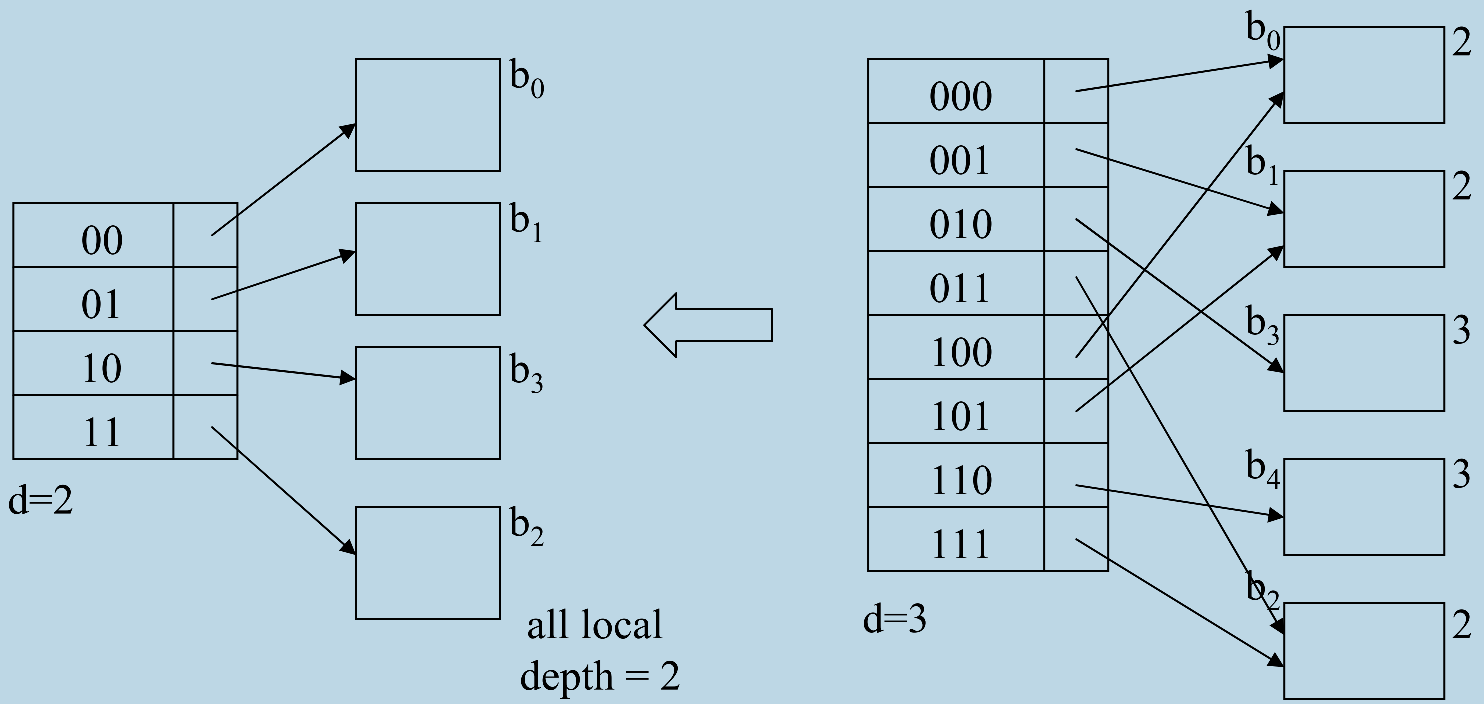


b_3 not full: new record placed in b_3. No changes.

b_3 full : b_3 is split, directory is doubled, all records with 3-bit sequence 110 sent to b_4. Others in b_3.

In general, if the local depth of the bucket to be split is equal to the global depth, directory is doubled

Deletion in Extendible Hashing Scheme



Matching pair of data buckets:

k-bit sequences have a common k-1 bit suffix, e.g, b_3 & b_4

Due to deletions, if a pair of matching data buckets

-- become less than half full – *try* to merge them into one bucket

If the local depth of all buckets is one less than the global depth

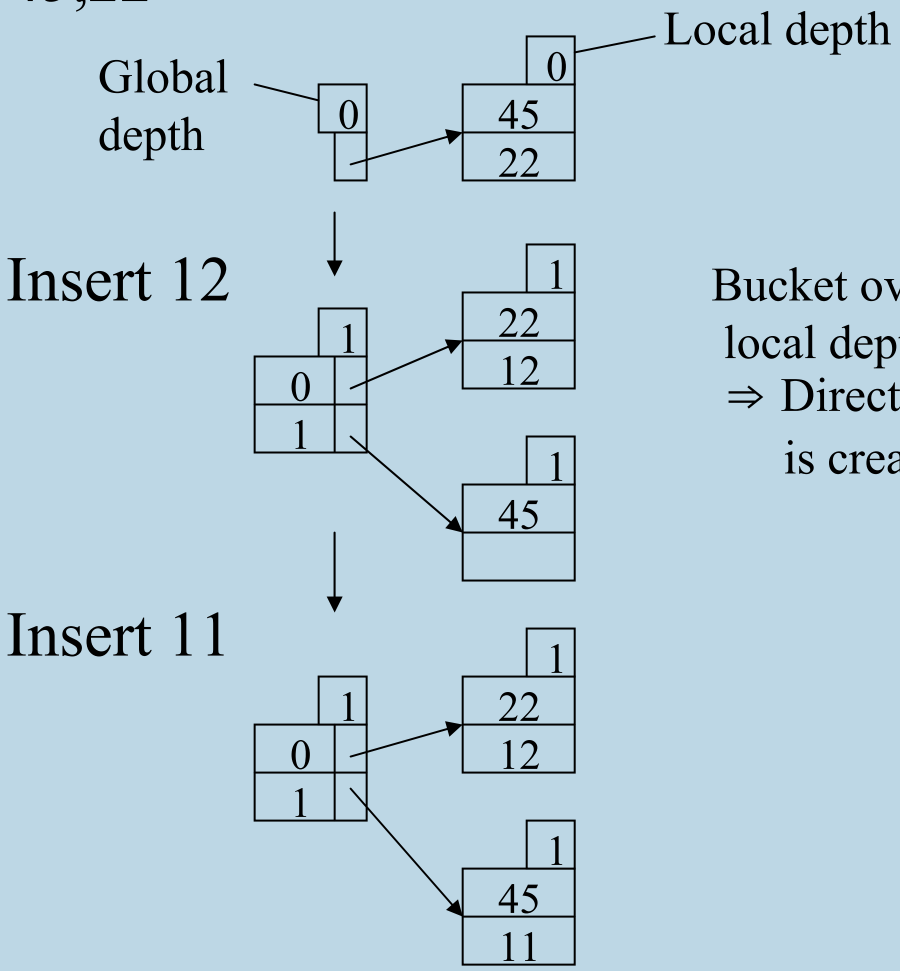
-- reduce the directory to half its size

Extendible Hashing Example

Bucket capacity – 2 Initial buckets = 1

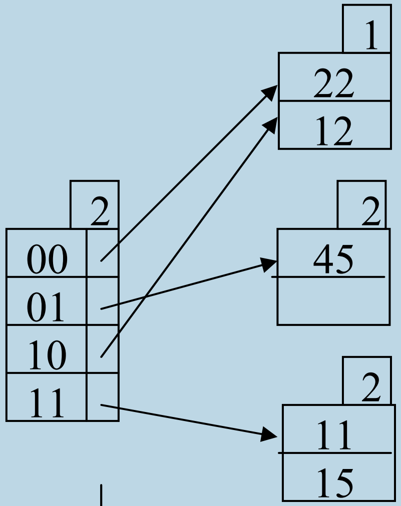
Insert 45,22

45	101101
22	10110
12	1100
11	1011



Bucket overflows
local depth = global depth
⇒ Directory doubles and split image is created

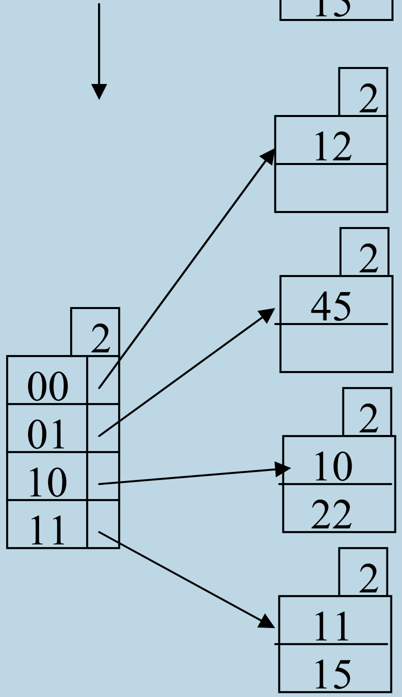
Insert 15



Overflow occurs.
Global depth = local depth
Directory doubles and split occurs

45	101101
22	10110
12	1100
11	1011
15	1111
10	1010

Insert 10



Overflows occurs.
Since local depth < global depth
Split image is created
Directory is not doubled

Linear Hashing

Does not require a separate directory structure

Uses a family of hash functions $h_0, h_1, h_2, \ldots.$

- the range of h_i is double the range of h_{i-1}

- $h_i(x) = x \bmod 2^i M$

  M - the initial no. of buckets

  (Assume that the hashing field is an integer)

Initial hash functions

$h_0(x) = x \bmod M$

$h_1(x) = x \bmod 2M$

Insertion (1/3)

Initially the structure has M main buckets (0 ,…, M-1) and a few overflow buckets

To insert a record with hash field value x, place the record in bucket $h_o(x)$

When the *first* overflow in <u>any</u> bucket occurs:

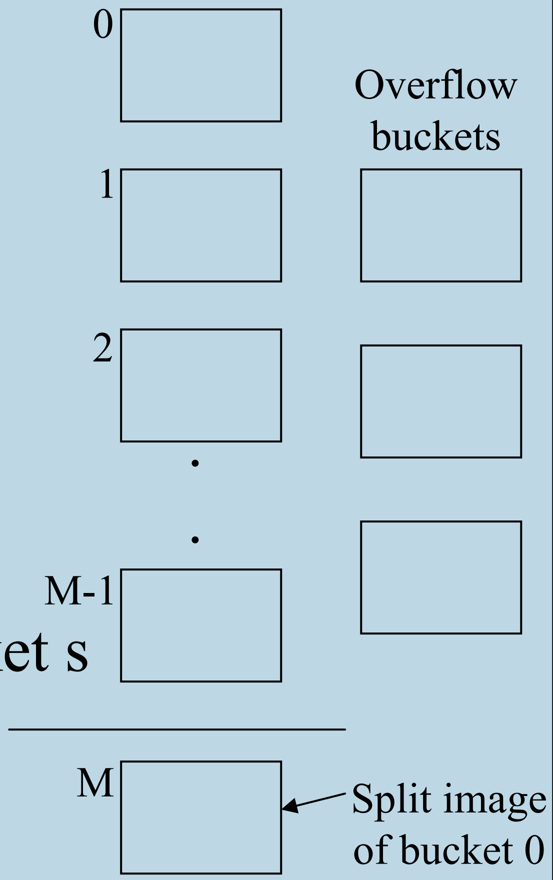
Say, overflow occurred in bucket s

Insert the record in the overflow chain of bucket s

Create a new bucket M

Split the *bucket 0* by using h_1

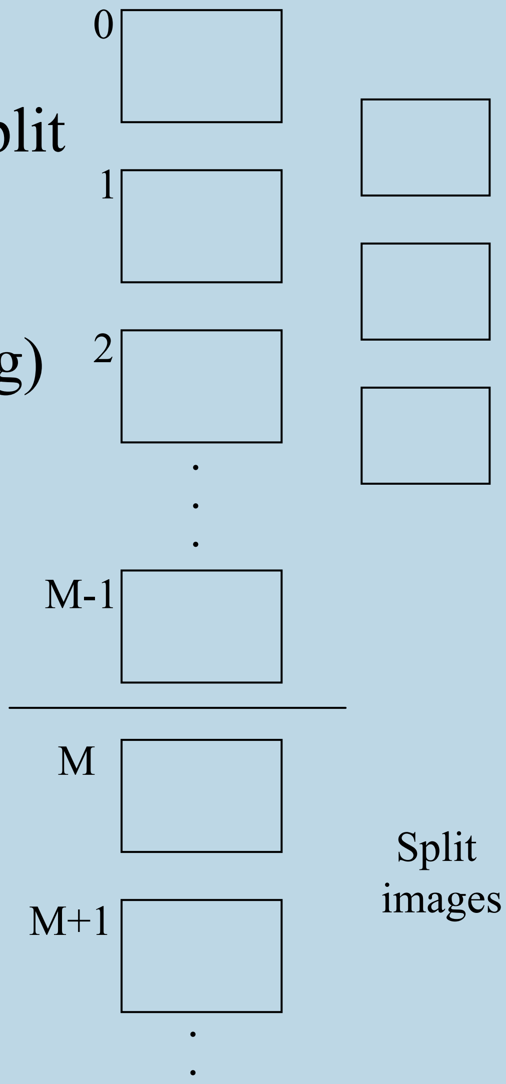
Some records stay in bucket 0 and some go to bucket M.



Insertion (2/3)

On first overflow,
  irrespective of where it occurs, bucket 0 is split
On subsequent overflows
  buckets 1, 2, 3, … are split in that order
  (This why the scheme is called linear hashing)
N: the next bucket to be split
After M overflows,
  all the original M buckets are split.
  We switch to hash functions h_1, h_2
    and set N = 0.

$$\begin{matrix} h_o \\ h_1 \end{matrix} \longrightarrow \begin{matrix} h_1 \\ h_2 \end{matrix} \longrightarrow \quad \dots \quad \begin{matrix} h_i \\ h_{i+1} \end{matrix} \longrightarrow \quad \dots$$



Nature of Hash Functions

$h_i(x) = x \bmod 2^i M$. Let $M' = 2^i M$

- Note that if $h_i(x) = k$ then $x = M'r + k$, $k < M'$

  and $h_{i+1}(x) = (M'r + k) \bmod 2M' = k$ or $M' + k$

Since,
r – even – $(M'2s + k) \bmod 2M' = k$
r – odd – $(M'(2s + 1) + k) \bmod 2M' = M' + k$

M'– the current number of original buckets.

Insertion (3/3)

Say the hash functions in use are h_i, h_{i+1}

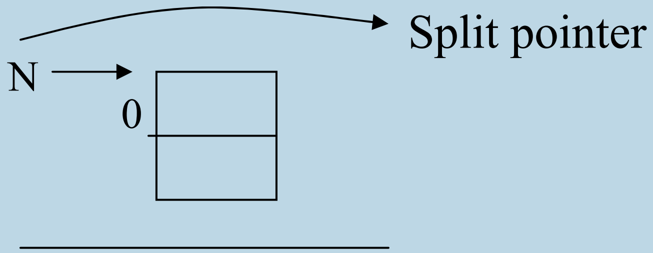
To insert record with hash field value x,

```
    Compute $h_i(x)$
    if $h_i(x) < N$, the original bucket is already split
        place the record in bucket $h_{i+1}(x)$
    else place the record in bucket $h_i(x)$
```

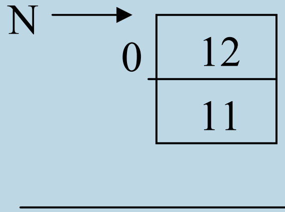
Linear Hashing Example

Initial Buckets = 1   Bucket capacity = 2 records

Hash functions
$h_0 = x \bmod 1$
$h_1 = x \bmod 2$

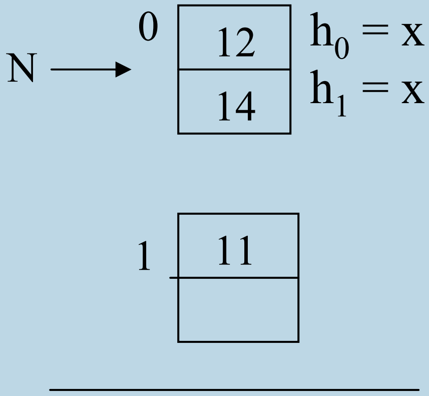


Insert 12, 11



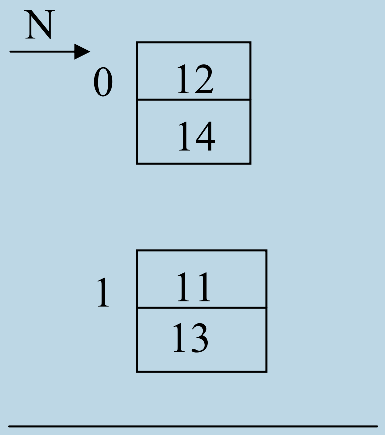
Insert 14

B_0 overflows
Bucket pointed by N is split
Hash functions are changed



$h_0 = x \bmod 2$
$h_1 = x \bmod 4$

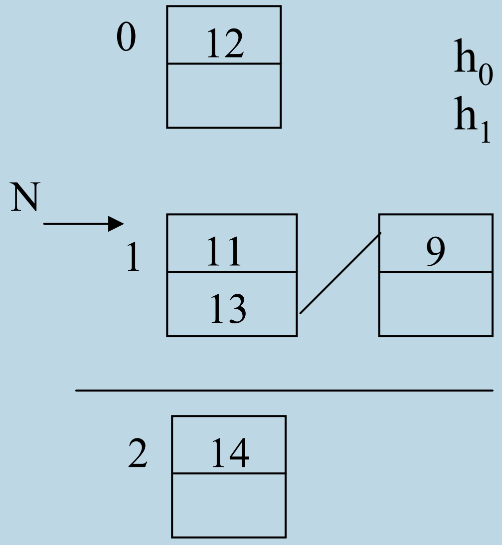
Insert 13



Insert 9

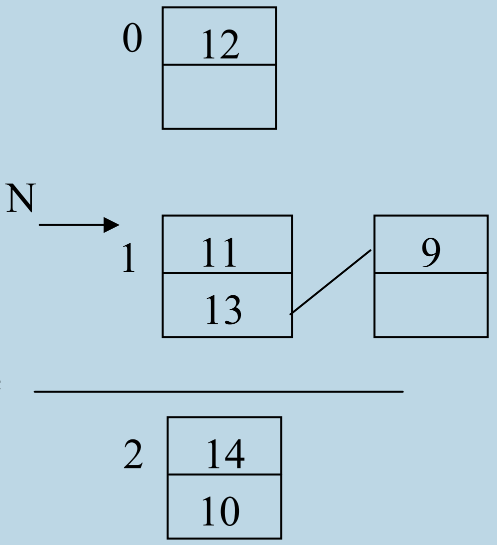
$\xrightarrow{\hspace{4cm}}$

B_1 overflows
B_0 is split using h_1
and split image
is created



$h_0 = x \bmod 2$
$h_1 = x \bmod 4$

Insert 10

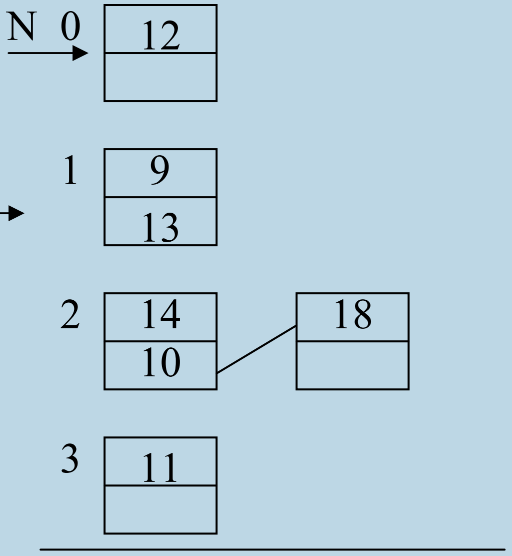


h1 is
applied here

Insert 18

$\xrightarrow{\hspace{4cm}}$

overflow at B_2
split B_1
$h_0 = x \bmod 4$
$h_1 = x \bmod 8$



Index Structures

Index: A disk data structure

– enables efficient retrieval of a record given the value (s) of certain attributes

– indexing attributes

Primary Index:

Index built on *ordering key* field of a file

Clustering Index:

Index built on *ordering non-key* field of a file

Secondary Index:

Index built on any *non-ordering* field of a file

Primary Index

Can be built on ordered / sorted files

Index attribute – ordering key field (OKF)

Index Entry:

value of OKF for the <u>first record</u> of a block B_j	disk address of B_j

Index file: ordered file (sorted on OKF)

size-no. of blocks in the data file

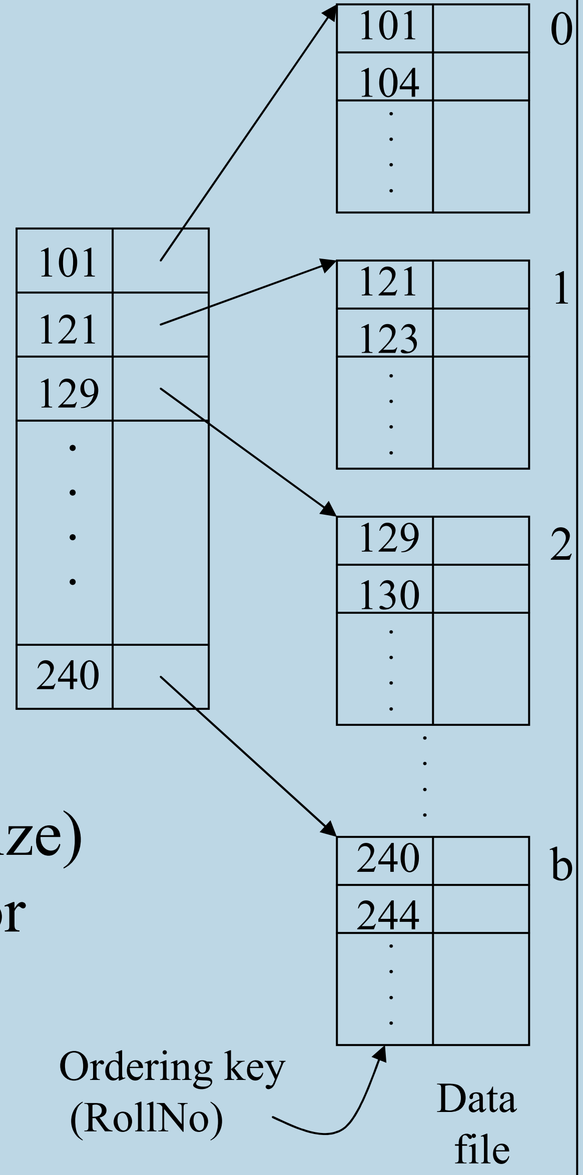
Index file blocking factor $BF_i = \lfloor B/(V+P) \rfloor$

(B-block size, V-OKF size, P-block pointer size)

- generally more than data file blocking factor

No of Index file blocks $b_i = \lceil b/BF_i \rceil$

(b - no. of data file blocks)



Record Access Using Primary Index

Given Ordering key field (OKF) value: x

Carry out binary search on the index file

m – value of OKF for the first record in the *middle block* k of the index file

$x < m$: do binary search on blocks $0 - (k - 1)$ of index file

$x \geq m$: if there is an index entry in block k with OKF value x,
use the corresponding block pointer,
get the data file block and
search for the data record with OKF value x
else do binary search on blocks $k + 1, \ldots, b_i$ of index file

Maximum block accesses required: $\lceil \log_2 b_i \rceil$

An Example

Data file:

No. of blocks b = 9500

Block size B = 4KB

OKF length V = 15 bytes

Block pointer length p = 6 bytes

Index file

No. of records r_i = 9500

Size of entry V + P = 21 bytes

Blocking factor $BF_i = \lfloor 4096/21 \rfloor = 195$

No. of blocks $b_i = \lceil r_i/BF_i \rceil = 49$

Max No. of block accesses for getting record using the primary index | $1 + \lceil \log_2 b_i \rceil = 7$

Max No. of block accesses for getting record without using primary index | $\lceil \log_2 b \rceil = 14$

Making the Index Multi-level

Index file – itself an ordered file

– another level of index can be built

Multilevel Index –

Successive levels of indices are built till the last level has one block

height – no. of levels

block accesses: height + 1

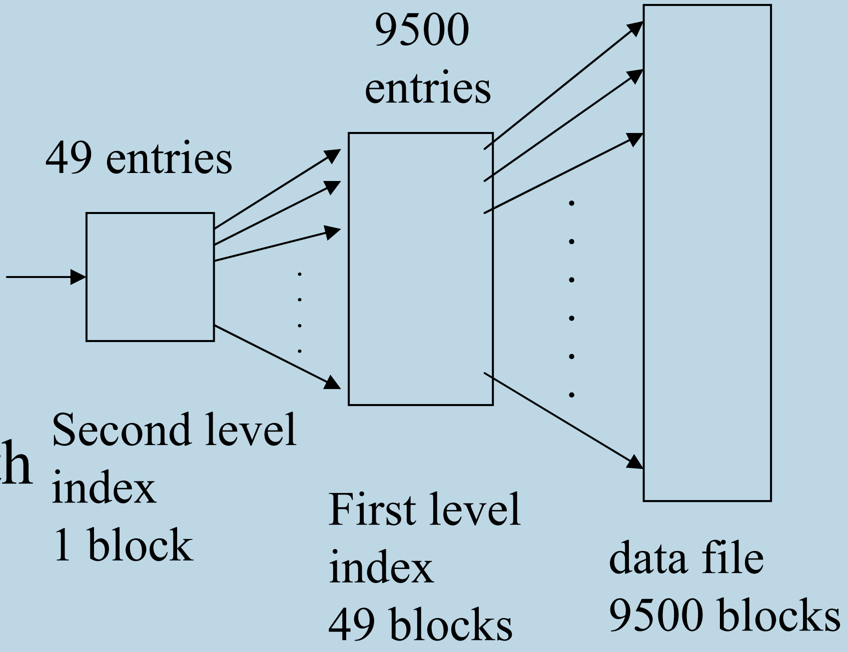
(no binary search required)

For the example data file:

No of block accesses required with

multi-level primary index: 3

without any index: 14



Range Search, Insertion and Deletion

Range search on the ordering key field:

Get records with OKF value between x_1 and x_2 (inclusive)

Use the index to locate the record with OKF value x_1 and read succeeding records till OKF value exceeds x_2.

Very efficient

Insertion: Data file – keep 25% of space in each block free

-- to take care of future insertions

index doesn't get changed

-- or use overflow chains for blocks that overflow

Deletion: Handle using deletion markers so that index doesn’t get affected

Basically, avoid changes to index

Clustering Index

Built on ordered files where ordering field is *not a key*
Index attribute: ordering field (OF)

Index entry:

Distinct value V_i of the OF	address of the first block that has a record with OF value V_i

Index file: Ordered file (sorted on OF)
size – no. of distinct values of OF

Secondary Index

Built on any non-ordering field (NOF) of a data file.

Case I: NOF is also a key (Secondary key)

value of the NOF V_i	pointer to the record with V_i as the NOF value

Case II: NOF is not a key: two options

(1)

value of the NOF V_i	pointer(s) to the record(s) with V_i as the NOF value

(2)

value of the NOF V_i	pointer to a block that has pointer(s) to the record(s) with V_i as the NOF value

Remarks:

(1) index entry – variable length record

(2) index entry – fixed length – One more level of indirection

Secondary Index (key)

Can be built on ordered and also other type of files

Index attribute: non-ordering key field

Index entry:

value of the NOF V_i	pointer to the *record* with V_i as the NOF value

Index file: ordered file (sorted on NOF values)

No. of entries – same as the no. of *records* in the data file

Index file blocking factor $Bf_i = \lfloor B/(V+P_r) \rfloor$

(B: block size, V: length of the NOF,

P_r: length of a record pointer)

Index file blocks = $\lceil r/Bf_i \rceil$

(r – no. of records in the data file)

An Example

Data file:

No. of records r = 90,000 Block size B = 4KB

Record length R = 100 bytes BF = $\lfloor 4096/100 \rfloor = 40$,

$b = \lceil 90000/40 \rceil = 2250$

NOF length V = 15 bytes length of a record pointer $P_r = 7$ bytes

Index file :

No. of records $r_i = 90,000$ record length = $V + P_r = 22$ bytes

$BF_i = \lfloor 4096/22 \rfloor = 186$ No. of blocks $b_i = \lceil 90000/186 \rceil = 484$

Max no. of block accesses to get a record using the secondary index $1 + \lceil \log_2 b_i \rceil = 10$

Avg no. of block accesses to get a record without using the secondary index $b/2 = 1125$

A very significant improvement

Multi-level Secondary Indexes

Secondary indexes can also be converted to multi-level indexes

First level index

– as many entries as there are records in the data file

First level index is an ordered file

so, in the second level index, the number of entries will be equal to the number of *blocks* in the first level index rather than the number of *records*

Similarly in other higher levels

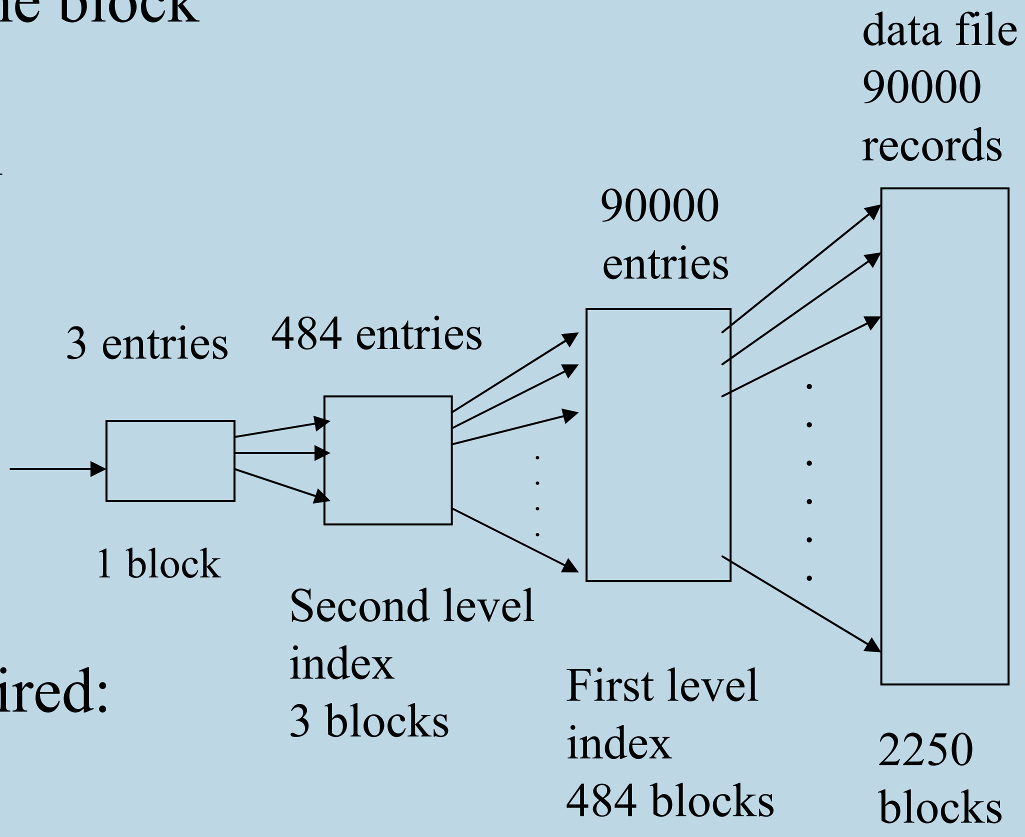
Making the Secondary Index Multi-level

Multilevel Index –

Successive levels of indices are built till the last level has one block

height – no. of levels

block accesses: height + 1



For the example data file:

No of block accesses required:

- multi-level index: 4
- single level index: 10

Index Sequential Access Method (ISAM) Files

ISAM files –

Ordered files with a multilevel primary/clustering index

Insertions:

Handled using overflow chains at data file blocks

Deletions:

Handled using deletion markers

Most suitable for files that are relatively static

If the files are dynamic, we need to go for dynamic multi-level index structures based on B^+- trees

B^+- trees

- Balanced search trees
  - all leaves are at the same level

- Leaf node entries point to the actual data records
  - all leaf nodes are linked up as a list

- Internal node entries carry only index information
  - In B-trees, internal nodes carry data records also
  - The fan-out in B-trees is less

- Makes sure that blocks are always at least half filled

- Supports both random and sequential access of records

Order

Order (m) of an Internal Node

- Order of an internal node is the maximum number of tree pointers held in it.
- Maximum of (m-1) keys can be present in an internal node

Order (m_{leaf}) of a Leaf Node

- Order of a leaf node is the maximum number of record pointers held in it. It is equal to the number of keys in a leaf node.

Internal Nodes

An internal node of a B^+- tree of order m:

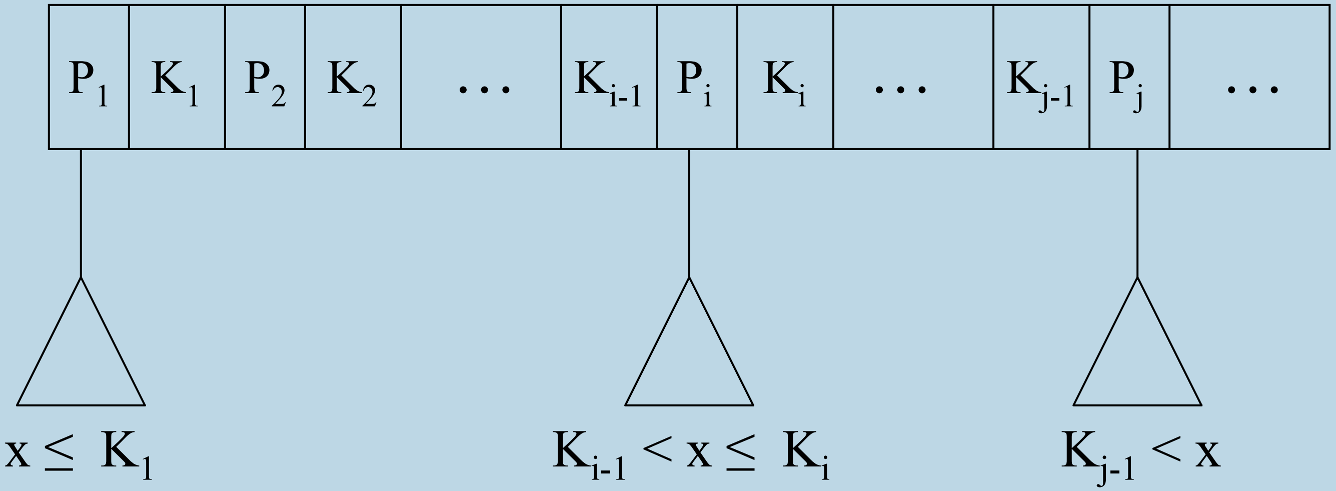
- It contains at least $\left\lceil \frac{m}{2} \right\rceil$ pointers, except when it is the root node

- It contains at most m pointers.

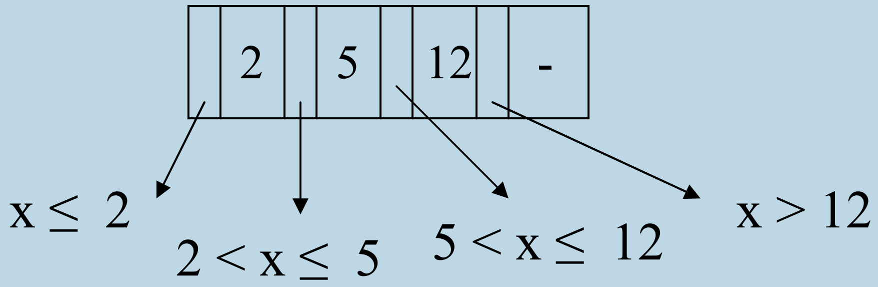
- If it has $P_1, P_2, \ldots, P_j$ pointers with $K_1 < K_2 < K_3 \ldots < K_{j-1}$ as keys, where $\left\lceil \frac{m}{2} \right\rceil \leq j \leq m$, then
  - P_1 points to the subtree with records having key value $x \leq K_1$
  - P_i $(1 < i < j)$ points to the subtree with records having key value x such that $K_{i-1} < x \leq K_i$
  - P_j points to records with key value $x > K_{j-1}$

Internal Node Structure

$$\left\lceil \frac{m}{2} \right\rceil \le j \le m$$



Example

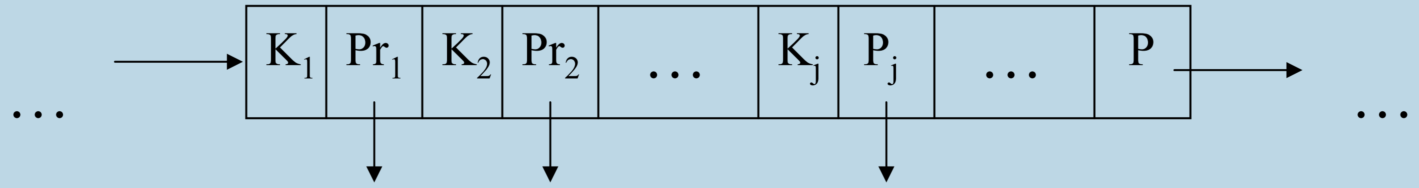


Leaf Node Structure

Structure of leaf node of B^+- of order m_{leaf} :

- It contains one block pointer P to point to next leaf node
- At least $\left\lceil \frac{m_{leaf}}{2} \right\rceil$ record pointers and $\left\lceil \frac{m_{leaf}}{2} \right\rceil$ key values
- At most m_{leaf} record pointers and key values
- If a node has keys $K_1 < K_2 < \ldots < K_j$ with $Pr_1, Pr_2 \ldots Pr_j$ as record pointers and P as block pointer, then

Pr_i points to record with K_i as the search field value, $1 \le i \le j$
P points to next leaf block



Order Calculation

Block size: B, Size of Indexing field: V
Size of block pointer: P, Size of record pointer: P_r

Order of Internal node (m):
As there can be at most m block pointers and (m-1) keys

$$(m*P) + ((m-1) * V) \leq B$$

m can be calculated by solving the above equation.

Order of leaf node:
As there can be at most m_{leaf} record pointers and keys with one block pointer in a leaf node,
m_{leaf} can be calculated by solving

$$(m_{leaf} * (P_r + V)) + P \leq B$$

Example Order Calculation

Given $B = 512$ bytes $V = 8$ bytes
$P = 6$ bytes $P_r = 7$ bytes. Then

Internal node order $m = ?$

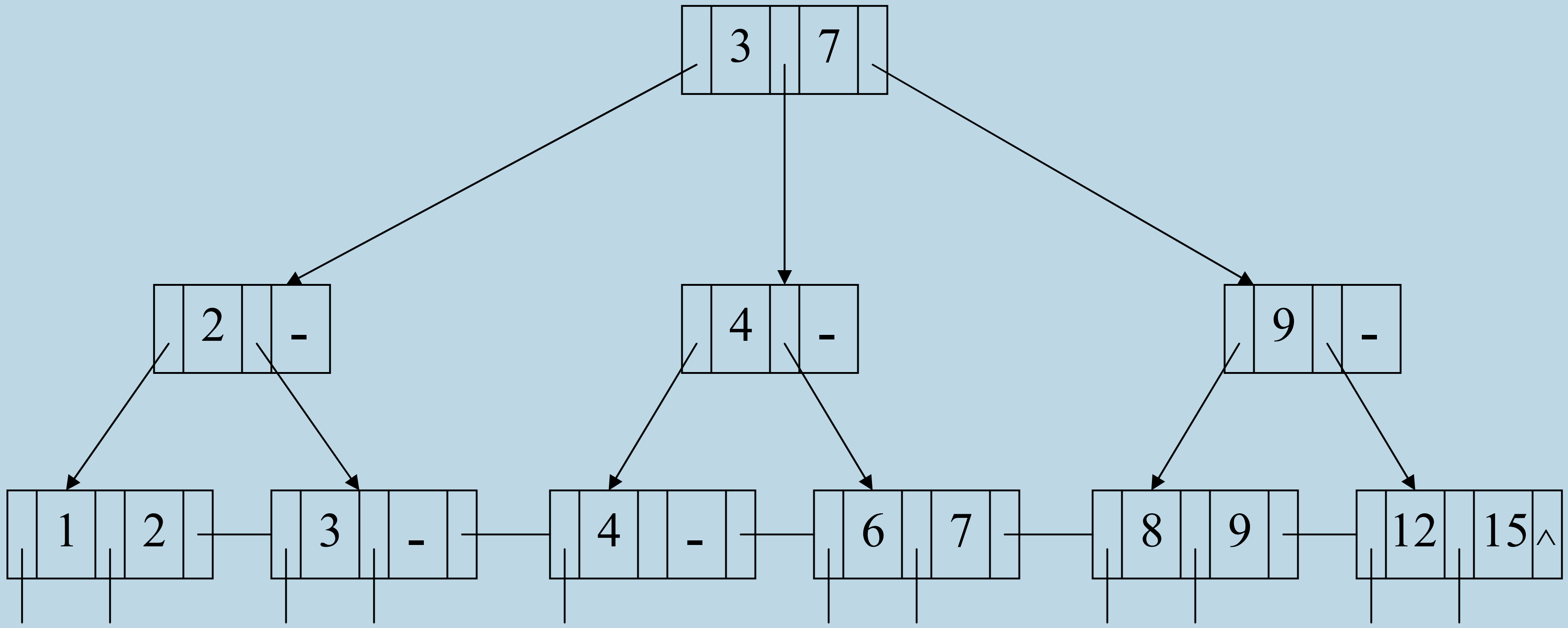
$$m * P + ((m-1) * V) \leq B$$
$$m * 6 + ((m-1) * 8) \leq 512$$
$$14m \leq 520$$
$$m \leq 37$$

Leaf order $m_{leaf} = ?$

$$m_{leaf} (P_r + V) + P \leq 512$$
$$m_{leaf} (7 + 8) + 6 \leq 512$$
$$15m_{leaf} \leq 506$$
$$m_{leaf} \leq 33$$

Example B^+- tree

$m = 3 \;\; m_{leaf} = 2$



Insertion into B^+- trees

1. Every node is inserted at leaf level

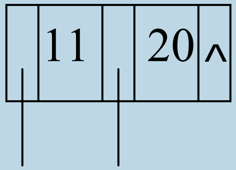
- If leaf node overflows, then
  - Node is split at $j = \left\lceil \frac{(m_{leaf} + 1)}{2} \right\rceil$
  - First j entries are kept in original node
  - Entities from j+1 are moved to new node
  - j^{th} key value is *replicated* in the parent of the leaf.
- If Internal node overflows
  - Node is split at $j = \left\lfloor \frac{(m + 1)}{2} \right\rfloor$
  - Values and pointers up to P_j are kept in original node
  - j^{th} key value is *moved* to parent of the internal node
  - P_{j+1} to the rest of entries are moved to new node.

Example of Insertions

$m = 3 \quad m_{leaf} = 2$

Insert 20, 11

1



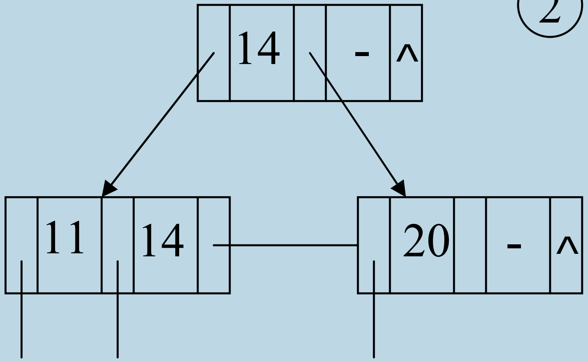
Insert 14

$\longrightarrow$

Overflow. leaf is split at $j = \left\lceil \frac{(m_{leaf} + 1)}{2} \right\rceil = 2$

14 is replicated to upper level

2

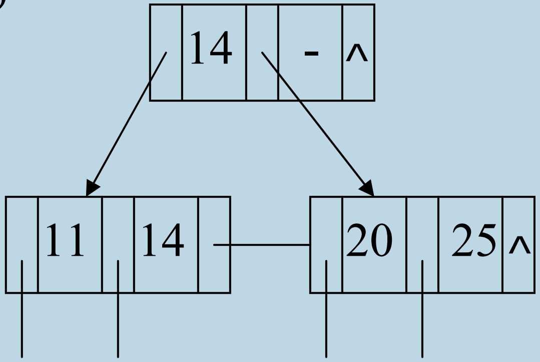


Insert 25

$\longrightarrow$

Inserted at leaf level

3

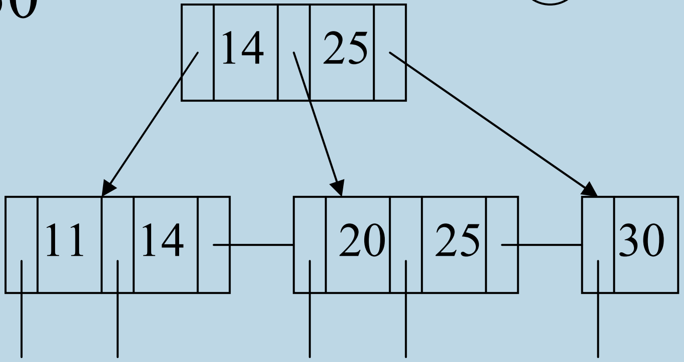


Insert 30

$\longrightarrow$

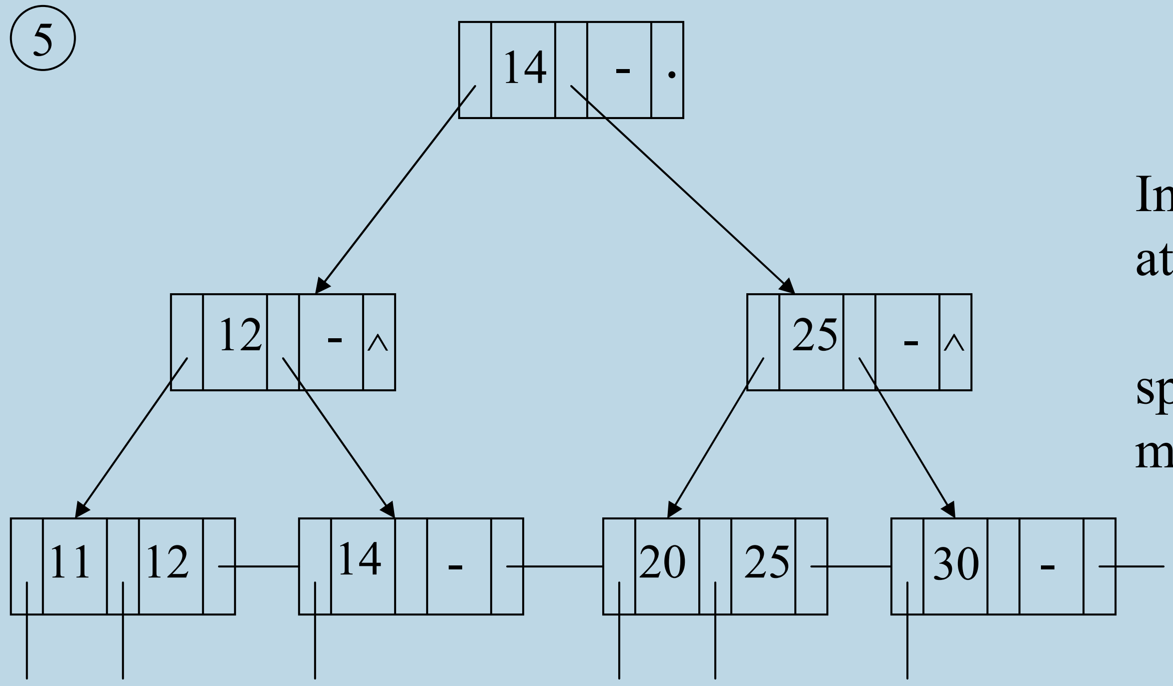
Overflow. split at 25. 25 is moved up

4



Insert 12 Overflow at leaf level.

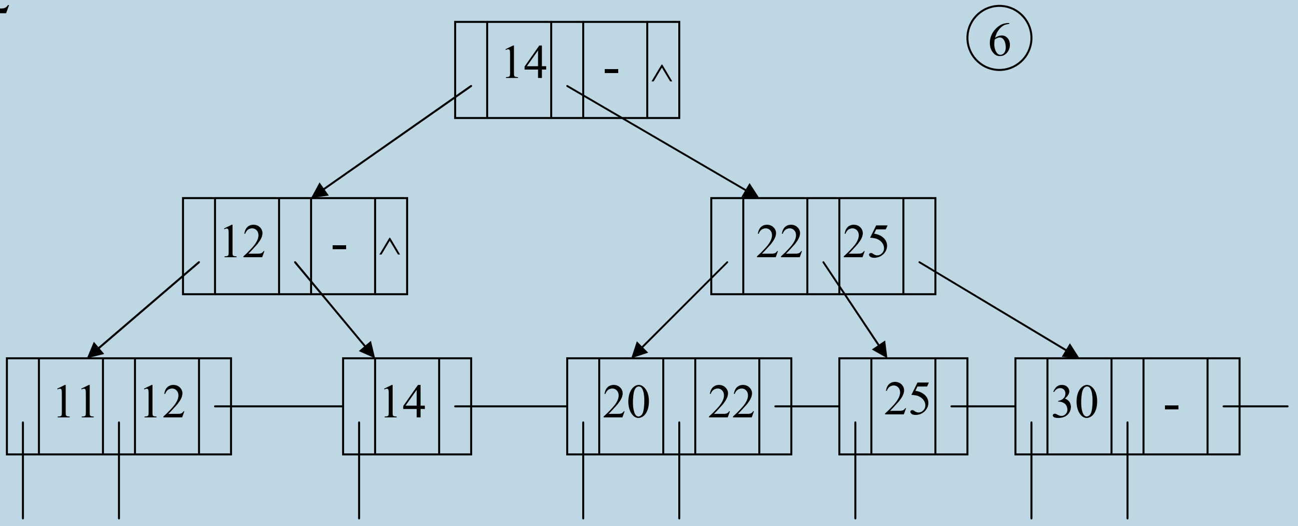
- Split at leaf level,
- Triggers overflow at internal node
- Split occurs at internal node



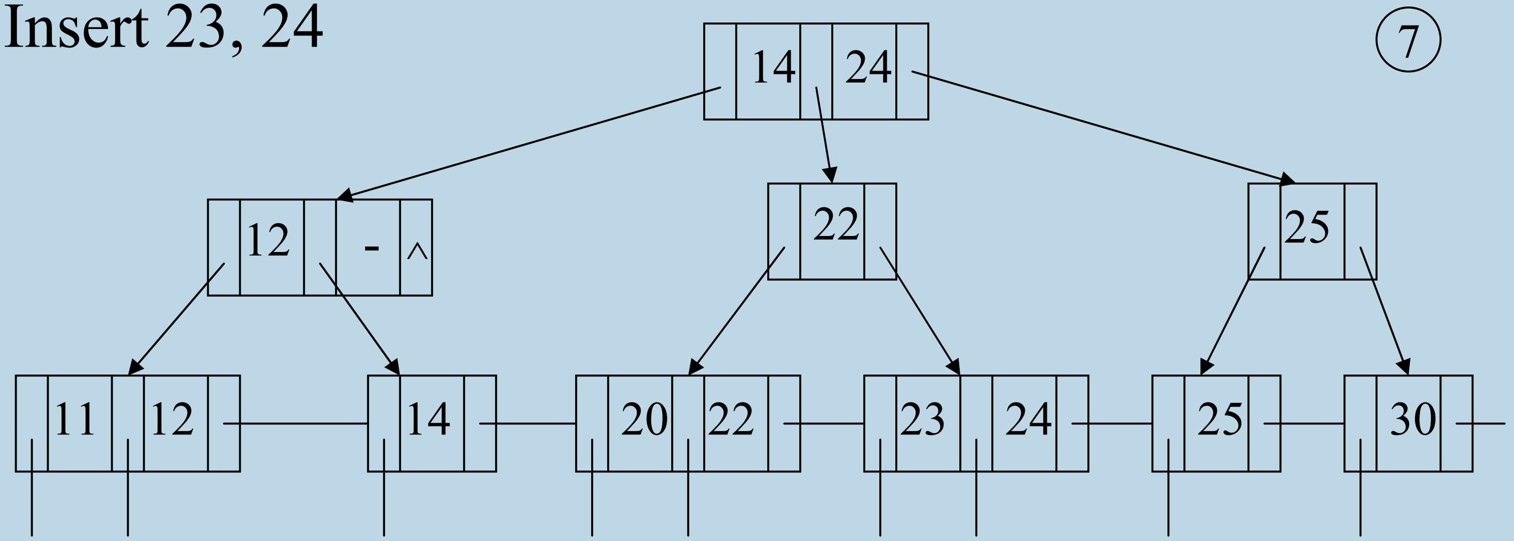
Internal node split at j = $\left\lceil \frac{m}{2} \right\rceil$

split at 14 and 14 is moved up

Insert 22



Insert 23, 24



Deletion in B^+- trees

- Delete the entry from the leaf node

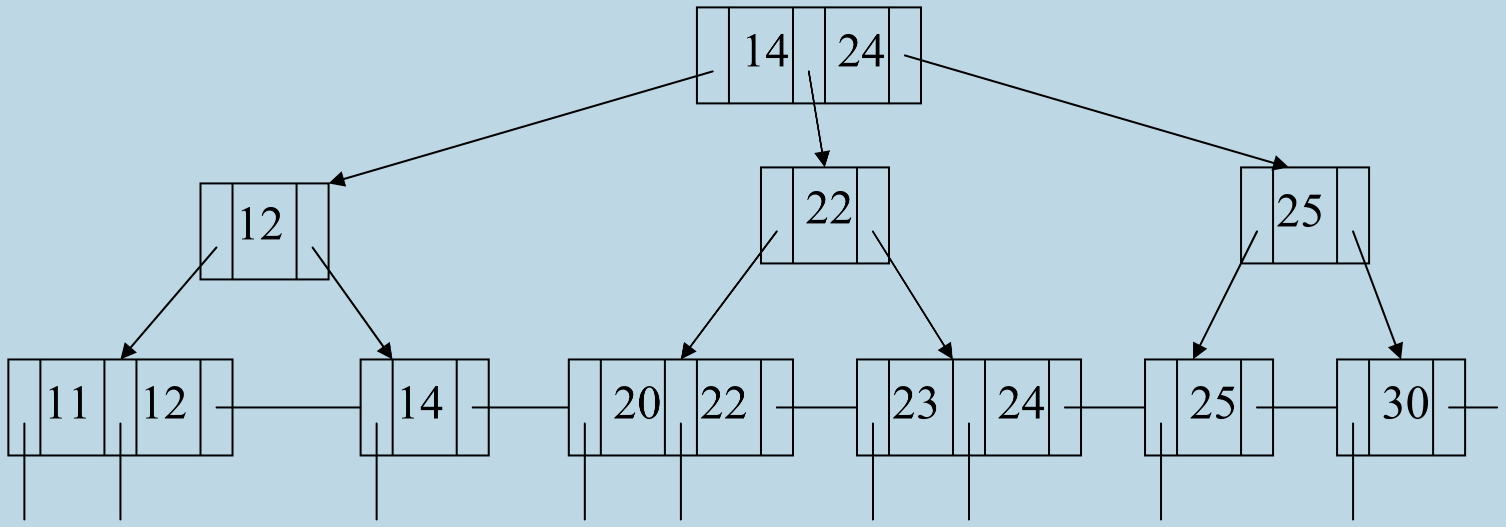
- Delete the entry if it is present in Internal node and replace with the entry to its left in that position.

- If underflow occurs after deletion
  - Distribute the entries from left sibling

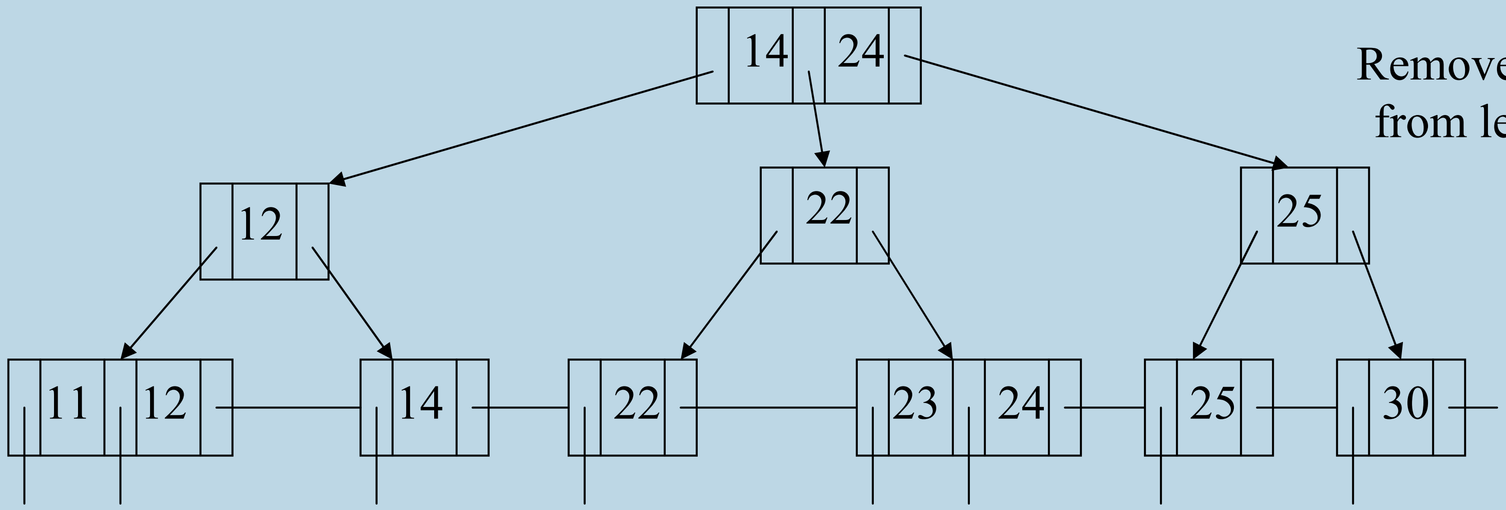
  if not possible – Distribute the entries from right sibling

  if not possible – Merge the node with left and right sibling

Example

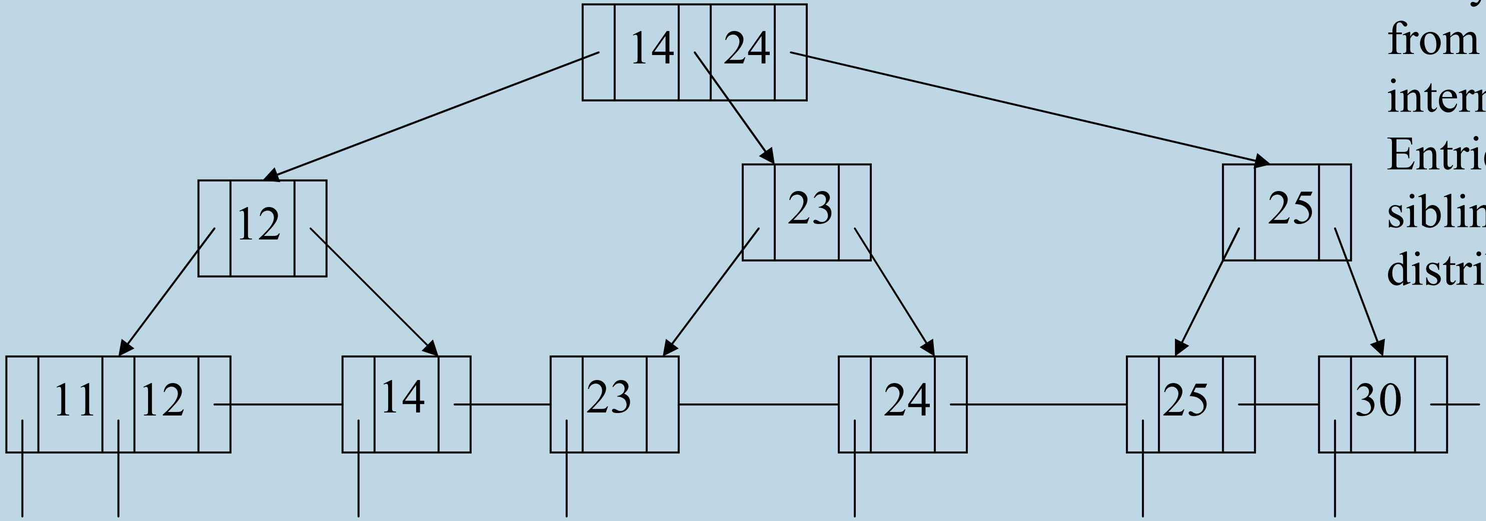


Delete 20



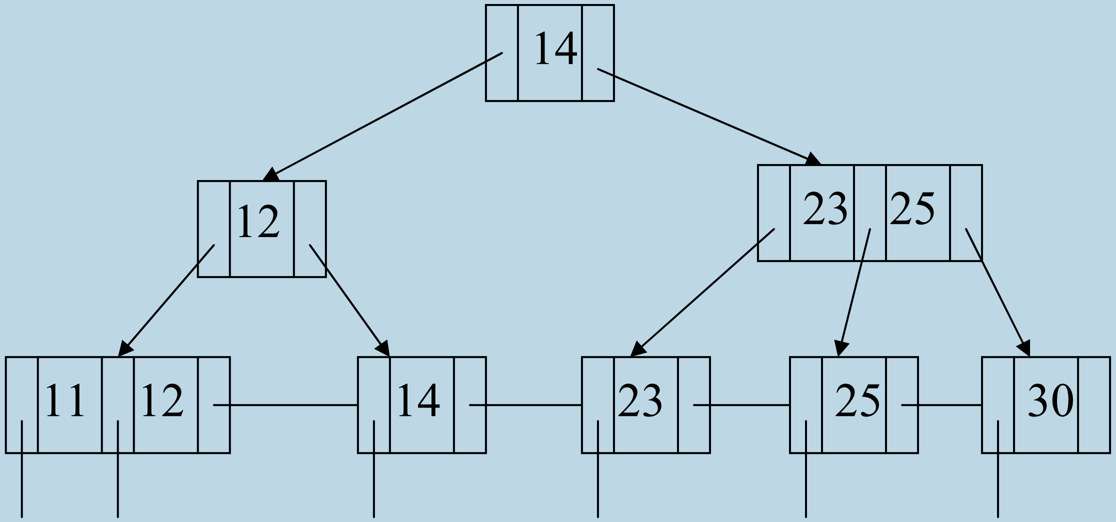
Removed entry from leaf here

Delete 22

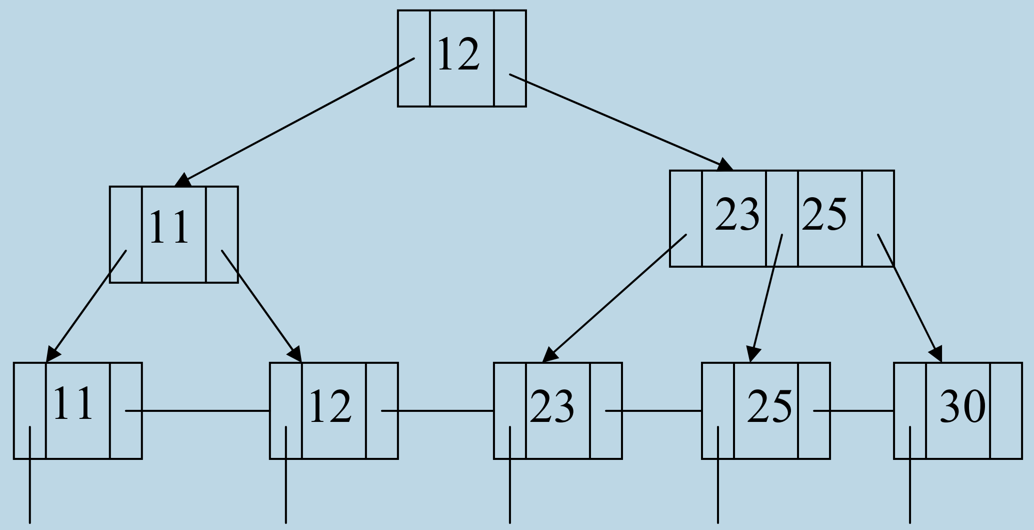


Entry 22 is removed from leaf and internal node
Entries from right sibling are distributed to left

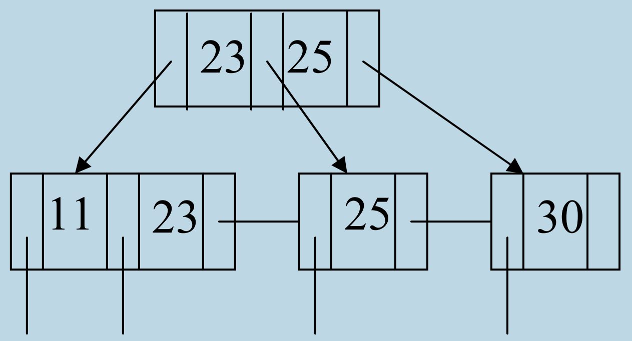
Delete 24



Delete 14



Delete 12



Level drop has occurred

Advantages of B^+- trees:

1) Any record can be fetched in equal number of disk accesses.

2) Range queries can be performed easily as leaves are linked up

3) Height of the tree is less as only keys are used for indexing

4) Supports both random and sequential access.

Disadvantages of B^+- trees:

Insert and delete operations are complicated

Root node becomes a *hotspot*

Database Design and Normal Forms

Database Design

- coming up with a 'good' schema is very important

How do we characterize the "goodness" of a schema ?
If two or more alternative schemas are available
  how do we compare them ?
What are the problems with "bad" schema designs ?

Normal Forms:
  Each normal form specifies certain conditions
  If the conditions are satisfied by the schema
    certain kind of problems are avoided

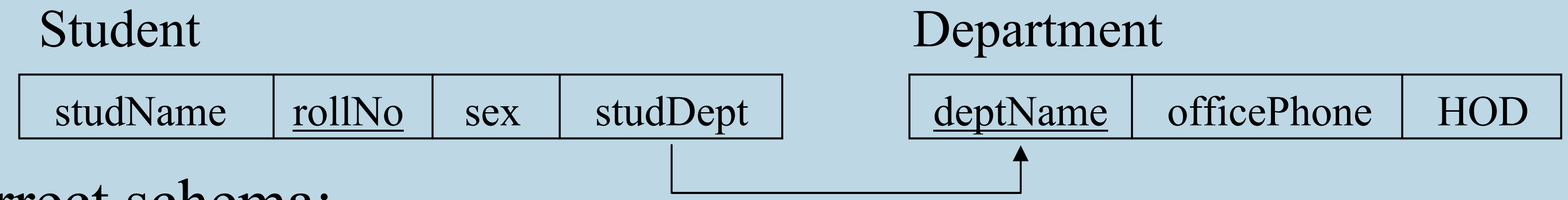
Details follow….

An Example

student relation with attributes: studName, rollNo, sex, studDept
department relation with attributes: deptName, officePhone, hod

Several students belong to a department.
studDept gives the name of the student's department.

Correct schema:



Student

studName	rollNo	sex	studDept

Department

deptName	officePhone	HOD

Incorrect schema:

Student Dept

studName	rollNo	sex	deptName	officePhone	HOD

What are the problems that arise ?

Problems with bad schema

Redundant storage of data:

Office Phone & HOD info - stored redundantly

- once with each student that belongs to the department
- wastage of disk space

A program that updates Office Phone of a department

- must change it at several places
  - more running time
  - error - prone

Transactions running on a database

- must take as short time as possible to increase transaction throughput

Update Anomalies

Another kind of problem with bad schema

Insertion anomaly:

No way of inserting info about a new department unless
we also enter details of a (dummy) student in the department

Deletion anomaly:

If all students of a certain department leave
and we delete their tuples,
information about the department itself is lost

Update Anomaly:

Updating officePhone of a department

- value in several tuples needs to be changed
- if a tuple is missed - inconsistency in data

Normal Forms

First Normal Form (1NF) - included in the definition of a relation

Second Normal Form (2NF)

Third Normal Form (3NF)

Boyce-Codd Normal Form (BCNF)

} defined in terms of functional dependencies

Fourth Normal Form (4NF) - defined using multivalued dependencies

Fifth Normal Form (5NF) or Project Join Normal Form (PJNF) defined using join dependencies

Functional Dependencies

A functional dependency (FD) $X \rightarrow Y$

(read as X *determines* Y) ($X \subseteq R$, $Y \subseteq R$)

is said to hold on a schema R if

in any instance r on R,

if two tuples t_1, t_2 ($t_1 \neq t_2$, $t_1 \in r$, $t_2 \in r$)

agree on X i.e. $t_1[X] = t_2[X]$

then they also agree on Y i.e. $t_1[Y] = t_2[Y]$

Note: If $K \subset R$ is a key for R then for any $A \in R$,

$$K \rightarrow A$$

holds because the above if …..then condition is vacuously true

Functional Dependencies – Examples

Consider the schema:

Student (studName, <u>rollNo</u>, sex, dept, hostelName, roomNo)

Since rollNo is a key, rollNo → {studName, sex, dept, hostelName, roomNo}

Suppose that each student is given a hostel room exclusively, then

hostelName, roomNo → rollNo

Suppose boys and girls are accommodated in separate hostels, then

hostelName → sex

FDs are additional constraints that can be specified by designers

Trivial / Non-Trivial FDs

An FD $X \rightarrow Y$ where $Y \subseteq X$

- called a *trivial* FD, it always holds good

An FD $X \rightarrow Y$ where $Y \nsubseteq X$

- *non-trivial* FD

An FD $X \rightarrow Y$ where $X \cap Y = \phi$

- *completely non-trivial* FD

Deriving new FDs

Given that a set of FDs F holds on R
    we can infer that a certain new FD must also hold on R

For instance,
    given that $X \rightarrow Y$, $Y \rightarrow Z$ hold on R
    we can infer that $X \rightarrow Z$ must also hold

How to systematically obtain all such new FDs ?

Unless *all* FDs are known, a relation schema is not fully specified

Entailment relation

We say that a set of FDs $F \models \{ X \rightarrow Y\}$

(read as F *entails* $X \rightarrow Y$ or

F *logically implies* $X \rightarrow Y$)

if in every instance r of R on which FDs F hold,

FD $X \rightarrow Y$ also holds.

Armstrong came up with several inference rules

for deriving new FDs from a given set of FDs

We define $F^+ = \{X \rightarrow Y \mid F \models X \rightarrow Y\}$

F^+: Closure of F

Armstrong's Inference Rules (1/2)

1. Reflexive rule

   $F \models \{X \rightarrow Y \mid Y \subseteq X\}$ for any X. Trivial FDs

2. Augmentation rule

   $\{X \rightarrow Y\} \models \{XZ \rightarrow YZ\}, Z \subseteq R.$ Here XZ denotes $X \cup Z$

3. Transitive rule

   $\{X \rightarrow Y, Y \rightarrow Z\} \models \{X \rightarrow Z\}$

4. Decomposition or Projective rule

   $\{X \rightarrow YZ\} \models \{X \rightarrow Y\}$

5. Union or Additive rule

   $\{X \rightarrow Y, X \rightarrow Z\} \models \{X \rightarrow YZ\}$

6. Pseudo transitive rule

   $\{X \rightarrow Y, WY \rightarrow Z\} \models \{WX \rightarrow Z\}$

Armstrong's Inference Rules (2/2)

Rules 4, 5, 6 are not really necessary.

For instance, Rule 5: $\{X \rightarrow Y, X \rightarrow Z\} \models \{X \rightarrow YZ\}$ can be proved using 1, 2, 3 alone

$$\left.\begin{array}{ll} 1) & X \rightarrow Y \\ 2) & X \rightarrow Z \end{array}\right\} \text{given}$$

3) $X \rightarrow XY$ Augmentation rule on 1
4) $XY \rightarrow ZY$ Augmentation rule on 2
5) $X \rightarrow ZY$ Transitive rule on 3, 4.

Similarly, 4, 6 can be shown to be unnecessary.
But it is useful to have 4, 5, 6 as <u>short-cut</u> rules

Sound and Complete Inference Rules

Armstrong showed that

Rules (1), (2) and (3) are sound and complete.

These are called Armstrong's Axioms (AA)

Soundness:

Every new FD $X \rightarrow Y$ derived from a given set of FDs F using Armstrong's Axioms is such that $F \vDash \{X \rightarrow Y\}$

Completeness:

Any FD $X \rightarrow Y$ logically implied by F (i.e. $F \vDash \{X \rightarrow Y\}$) can be derived from F using Armstrong's Axioms

Proving Soundness

Suppose $X \rightarrow Y$ is derived from F using AA in some *n* steps.

If each step is correct then overall deduction would be correct.

Single step: Apply Rule (1) or (2) or (3)

Rule (1) – obviously results in correct FDs

Rule (2) – $\{X \rightarrow Y\} \models \{XZ \rightarrow YZ\}, Z \subseteq R$

Suppose $t_1, t_2 \in r$ agree on XZ

$\Rightarrow t_1, t_2$ agree on X

$\Rightarrow t_1, t_2$ agree on Y (since $X \rightarrow Y$ holds on r)

$\Rightarrow t_1, t_2$ agree as YZ

Hence Rule (2) gives rise to correct FDs

Rule (3) – $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$

Suppose $t_1, t_2 \in r$ agree on X

$\Rightarrow t_1, t_2$ agree on Y (since $X \rightarrow Y$ holds)

$\Rightarrow t_1, t_2$ agree on Z (since $Y \rightarrow Z$ holds)

Proving Completeness of Armstrong's Axioms (1/4)

Define X^+_F (closure of X wrt F)
$= \{A \mid X \rightarrow A$ can be derived from F using AA$\}$, $A \in R$

Claim1:
$X \rightarrow Y$ can be derived from F using AA iff $Y \subseteq X^+$

(If) Let $Y = \{A_1, A_2, \ldots, A_n\}$. $Y \subseteq X^+$
$\Rightarrow X \rightarrow A_i$ can be derived from F using AA $(1 \leq i \leq n)$
By union rule, it follows that $X \rightarrow Y$ can be derived from F.

(Only If) $X \rightarrow Y$ can be derived from F using AA
By projective rule $X \rightarrow A_i$ $(1 \leq i \leq n)$
Thus by definition of X^+, $A_i \in X^+$
$\Rightarrow Y \subseteq X^+$

Completeness of Armstrong's Axioms (2/4)

Completeness:

$(F \models \{X \rightarrow Y\}) \Rightarrow X \rightarrow Y$ follows from F using AA

We will prove the contrapositive:

$X \rightarrow Y$ can't be derived from F using AA

$\Rightarrow F \not\models \{X \rightarrow Y\}$

$\Rightarrow \exists$ a relation instance r on R st all the FDs of F hold on r but $X \rightarrow Y$ doesn't hold.

Consider the relation instance r with just two tuples:

$$\begin{array}{ll} & \overbrace{1\ 1\ 1 \ldots 1}^{X^+ \text{ attributes}}\ \overbrace{1\ 1\ 1 \ldots 1}^{\text{Other attributes}} \\ r: & \\ & 1\ 1\ 1 \ldots 1\ 0\ 0\ 0 \ldots 0 \end{array}$$

Completeness Proof (3/4)

Claim 2: All FDs of F are satisfied by r

Suppose not. Let $W \rightarrow Z$ in F be an FD not satisfied by r

Then $W \subseteq X^+$ and $Z \nsubseteq X^+$

Let $A \in Z - X^+$

Now, $X \rightarrow W$ follows from F using AA as $W \subseteq X^+$ (claim 1)

$X \rightarrow Z$ follows from F using AA by transitive rule

$Z \rightarrow A$ follows from F using AA by reflexive rule as $A \in Z$

$X \rightarrow A$ follows from F using AA by transitive rule

By definition of closures, A must belong to X^+

- a contradiction.

Hence the claim.

$$\begin{array}{ll} r: & 1\ 1\ 1 \ldots 1 \quad 1\ 1\ 1 \ldots 1 \\ & \underbrace{1\ 1\ 1 \ldots 1}_{X^+} \quad \underbrace{0\ 0\ 0 \ldots 0}_{R - X^+} \end{array}$$

Completeness Proof (4/4)

Claim 3: $X \rightarrow Y$ is not satisfied by r

Suppose not

Because of the structure of r, $Y \subseteq X^+$

$\Rightarrow X \rightarrow Y$ can be derived from F using AA

contradicting the assumption about $X \rightarrow Y$

Hence the claim

Thus, whenever $X \rightarrow Y$ doesn't follow from F using AA,

F doesn't logically imply $X \rightarrow Y$

Armstrong's Axioms are complete.

Consequence of Completeness of AA

$X^+ = \{A \mid X \rightarrow A$ follows from F using AA$\}$

$\quad = \{A \mid F \models X \rightarrow A\}$

Similarly

$F^+ = \{X \rightarrow Y \mid F \models X \rightarrow Y\}$

$\quad = \{X \rightarrow Y \mid X \rightarrow Y$ follows from F using AA$\}$

Computing closures

The size of F^+ can sometimes be exponential in the size of F.

For instance, $F = \{A \rightarrow B_1, A \rightarrow B_2, \ldots.., A \rightarrow B_n\}$

$F^+ = \{A \rightarrow X\}$ where $X \subseteq \{B_1, B_2, \ldots, B_n\}$.

Thus $|F^+| = 2^n$

Computing F^+: computationally expensive

Fortunately, checking if $X \rightarrow Y \in F^+$

can be done by checking if $Y \subseteq X^+_F$

Computing attribute closure (X^+_F) is easier

Computing X^{+}_{F}

We compute a sequence of sets $X_0, X_1, \ldots$ as follows:

$X_0 := X$; // X is the given set of attributes
$X_{i+1} := X_i \cup \{A \mid \text{there is a FD } Y \rightarrow Z \text{ in } F \text{ and } A \in Z \text{ and } Y \subseteq X_i\}$

Since $X_0 \subseteq X_1 \subseteq X_2 \subseteq \ldots \subseteq X_i \subseteq X_{i+1} \subseteq \ldots \subseteq R$
and R is finite,
There is an integer i st $X_i = X_{i+1} = X_{i+2} = \ldots$
and X^{+}_{F} is equal to X_i.

Normal Forms – 2NF

Full functional dependency:

An FD $X \rightarrow A$ for which there is no proper subset Y of X such that $Y \rightarrow A$

(A is said to be fully functionally dependent on X)

2NF: A relation schema R is in 2NF if

every non-prime attribute is fully functionally dependent on any key of R

prime attribute: A attribute that is part of some key

non-prime attribute: An attribute that is not part of any key

Example

1) Book (authorName, title, authorAffiliation, ISBN, publisher, pubYear)

Keys: (authorName, title), ISBN

Not in 2NF as authorName → authorAffiliation

(authorAffiliation is not fully functionally dependent on the first key)

2) Student (rollNo, name, dept, sex, hostelName, roomNo, admitYear)

Keys: rollNo, (hostelName, roomNo)

Not in 2NF as hostelName → sex

student (rollNo, name, dept, hostelName, roomNo, admitYear)

hostelDetail (hostelName, sex)

- There are both in 2NF

Transitive Dependencies

Transitive dependency:

An FD $X \rightarrow Y$ in a relation schema R for which there is a set of attributes $Z \subseteq R$ such that

$X \rightarrow Z$ and $Z \rightarrow Y$ and Z is not a subset of any key of R

Ex: student (rollNo, name, dept, hostelName, roomNo, headDept)

Keys: rollNo, (hostelName, roomNo)

rollNo $\rightarrow$ dept; dept $\rightarrow$ headDept hold

So, rollNo $\rightarrow$ headDept a transitive dependency

Head of the dept of dept D is stored redundantly in every tuple where D appears.

Relation is in 2NF but redundancy still exists.

Normal Forms – 3NF

Relation schema R is in 3NF if it is in 2NF and no non-prime attribute of R is transitively dependent on any key of R

student (rollNo, name, dept, hostelname, roomNo, headDept)
is not in 3NF

Decompose: student (<u>rollNo</u>, name, dept, <u>hostelName, roomNo</u>)
deptInfo (<u>dept</u>, headDept)
both in 3NF

Redundancy in data storage - removed

Another definition of 3NF

Relation schema R is in 3NF if for any nontrivial FD $X \rightarrow A$ either (i) X is a superkey or (ii) A is prime.

Suppose some R violates the above definition
$\Rightarrow$ There is an FD $X \rightarrow A$ for which both (i) and (ii) are false
$\Rightarrow$ X is not a superkey and A is non-prime attribute

Two cases arise:

1) X is contained in a key – A is not fully functionally dependent on this key
   - violation of 2NF condition
2) X is not contained in a key
   $K \rightarrow X$, $X \rightarrow A$ is a case of transitive dependency
   (K – any key of R)

Motivating example for BCNF

gradeInfo (rollNo, studName, course, grade)

Suppose the following FDs hold:

1) rollNo, course → grade
2) studName, course → grade
3) rollNo → studName
4) studName → rollNo

Keys:
(rollNo, course)
(studName, course)

For 1,2 lhs is a key. For 3,4 rhs is prime
So gradeInfo is in 3NF

But studName is stored redundantly along with every course being done by the student

Boyce - Codd Normal Form (BCNF)

Relation schema R is in BCNF if for every nontrivial
FD X → A, X is a <u>*superkey*</u> of R.

In gradeInfo, FDs 3, 4 are nontrivial but lhs is not a superkey
So, gradeInfo is not in BCNF

Decompose:

gradeInfo (<u>rollNo, course</u>, grade)
studInfo (<u>rollNo</u>, <u>studName</u>)

Redundancy allowed by 3NF is disallowed by BCNF

BCNF is stricter than 3NF
3NF is stricter than 2NF

Decomposition of a relation schema

If R doesn't satisfy a particular normal form,
we decompose R into smaller schemas

What's a decomposition?

$R = (A_1, A_2, \ldots, A_n)$

$D = (R_1, R_2, \ldots, R_k)$ st $R_i \subseteq R$ and $R = R_1 \cup R_2 \cup \ldots \cup R_k$

(R_i's need not be disjoint)

Replacing R by $R_1, R_2, \ldots, R_k$ – process of decomposing R

Ex: gradeInfo (rollNo, studName, course, grade)

R_1: gradeInfo (<u>rollNo, course</u>, grade)

R_2: studInfo (<u>rollNo</u>, studName)

Desirable Properties of Decompositions

Not all decomposition of a schema are useful

We require two properties to be satisfied

(i) Lossless join property
- the information in an instance r of R must be preserved in the instances $r_1, r_2, \ldots, r_k$ where $r_i = \pi_{R_i}(r)$

(ii) Dependency preserving property
- if a set F of dependencies hold on R it should be possible to enforce F by enforcing appropriate dependencies on each r_i

Lossless join property

F – set of FDs that hold on R

R – decomposed into $R_1, R_2, \ldots, R_k$

Decomposition is <u>*lossless*</u> wrt F if

for every relation instance r on R satisfying F,

$$r = \pi_{R_1}(r) * \pi_{R_2}(r) * \ldots * \pi_{R_k}(r)$$

Lossless joins are also called non-additive joins

$R = (A, B, C)$; $R_1 = (A, B)$; $R_2 = (B, C)$

r:

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_1	c_3

r_1:

A	B
a_1	b_1
a_2	b_2
a_3	b_1

r_2:

B	C
b_1	c_1
b_2	c_2
b_1	c_3

$r_1 * r_2$:

A	B	C
a_1	b_1	c_1
a_1	b_1	c_3
a_2	b_2	c_2
a_3	b_1	c_1
a_3	b_1	c_3

Original info is distorted

Spurious tuples

Lossy join

Dependency Preserving Decompositions

Decomposition D = $(R_1, R_2, \ldots, R_k)$ of schema R *preserves* a set of dependencies F if

$$(\pi_{R_1}(F) \cup \pi_{R_2}(F) \cup \ldots \cup \pi_{R_k}(F))^+ = F^+$$

Here, $\pi_{R_i}(F) = \{ (X \rightarrow Y) \in F^+ \mid X \subseteq R_i, Y \subseteq R_i \}$
(called projection of F onto R_i)

Informally, any FD that logically follows from F must also logically follow from the union of projections of F onto R_i's
Then, D is called dependency preserving.

An example

Schema R = (A, B, C)
FDs F = $\{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$

Decomposition D = $(R_1 = \{A, B\}, R_2 = \{B, C\})$
$\pi_{R_1}(F) = \{A \rightarrow B, B \rightarrow A\}$
$\pi_{R_2}(F) = \{B \rightarrow C, C \rightarrow B\}$

$$\begin{aligned}(\pi_{R_1}(F) \cup \pi_{R_2}(F))^+ = \{&A \rightarrow B, B \rightarrow A,\\ &B \rightarrow C, C \rightarrow B,\\ &A \rightarrow C, C \rightarrow A\} = F^+\end{aligned}$$

Hence Dependency preserving

Testing for lossless decomposition property(1/6)

R – given schema with attributes $A_1, A_2, \ldots, A_n$
F – given set of FDs
D – $\{R_1, R_2, \ldots, R_m\}$ given decomposition of R

Is D a lossless decomposition?

Create an $m \times n$ matrix S with columns labeled as $A_1, A_2, \ldots, A_n$
    and rows labeled as $R_1, R_2, \ldots, R_m$

Initialize the matrix as follows:
    set $S(i,j)$ as symbol b_{ij} for all i,j.
    if A_j is in the scheme R_i, then set $S(i,j)$ as symbol a_j , for all i,j

Testing for lossless decomposition property(2/6)

After *S* is initialized, we carry out the following process on it:

```
repeat
    for each functional dependency $U \rightarrow V$ in $F$ do
        for all rows in $S$ which agree on $U$-attributes do
            make the symbols in each $V$- attribute column
              the same in all the rows as follows:
                if any of the rows has an "a" symbol for the column
                    set the other rows to the same "a" symbol in the column
                else // if no "a" symbol exists in any of the rows
                    choose one of the "b" symbols that appears
                    in one of the rows for the $V$-attribute and
                    set the other rows to that "b" symbol in the column
until no changes to S
```

At the end, if there exists a row with all "*a*" symbols then D is lossless otherwise D is a lossy decomposition

Testing for lossless decomposition property(3/6)

R = (rollNo, name, advisor, advisorDept, course, grade)

FD's = { rollNo → name; rollNo → advisor; advisor → advisorDept
rollNo, course → grade}

D : { R_1 = (rollNo, name, advisor), R_2 = (advisor, advisorDept),
R_3 = (rollNo, course, grade) }

Matrix S : (Initial values)

	rollNo	name	advisor	advisor Dept	course	grade
R_1	a_1	a_2	a_3	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	b_{25}	b_{26}
R_3	a_1	b_{32}	b_{33}	b_{34}	a_5	a_6

Testing for lossless decomposition property(4/6)

R = (rollNo, name, advisor, advisorDept, course, grade)

FD's = { rollNo → name; rollNo → advisor; advisor → advisorDept
rollNo, course → grade}

D : { R_1 = (rollNo, name, advisor), R_2 = (advisor, advisorDept),
R_3 = (rollNo, course, grade) }

Matrix S : (After enforcing rollNo → name & rollNo → advisor)

	rollNo	name	advisor	advisor Dept	course	grade
R_1	a_1	a_2	a_3	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	b_{25}	b_{26}
R_3	a_1	$\cancel{b_{32}}a_2$	$\cancel{b_{33}}a_3$	b_{34}	a_5	a_6

Testing for lossless decomposition property(5/6)

R = (rollNo, name, advisor, advisorDept, course, grade)

FD's = { rollNo → name; rollNo → advisor; advisor → advisorDept
rollNo, course → grade}

D : { R_1 = (rollNo, name, advisor), R_2 = (advisor, advisorDept),
R_3 = (rollNo, course, grade) }

Matrix S : (After enforcing advisor → advisorDept)

	rollNo	name	advisor	advisor Dept	course	grade
R_1	a_1	a_2	a_3	$\cancel{b_{14}} a_4$	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	b_{25}	b_{26}
R_3	a_1	$\cancel{b_{32}} a_2$	$\cancel{b_{33}} a_3$	$\cancel{b_{34}} a_4$	a_5	a_6

No more changes. Third row with all *a* symbols. So a lossless join.

Testing for lossless decomposition property(6/6)

R – given schema. F – given set of FDs

The decomposition of R into R_1, R_2 is lossless wrt F if and only if either $R_1 \cap R_2 \rightarrow (R_1 - R_2)$ belongs to F^+ or

$R_1 \cap R_2 \rightarrow (R_2 - R_1)$ belongs to F^+

Eg. gradeInfo (rollNo, studName, course, grade)

with FDs = {rollNo, course → grade; studName, course → grade;

rollNo → studName; studName → rollNo}

decomposed into

grades (rollNo, course, grade) and studInfo (rollNo, studName)

is lossless because

rollNo → studName

A property of lossless joins

D_1: $(R_1, R_2, \ldots, R_K)$ lossless decomposition of R wrt F

D_2: $(R_{i1}, R_{i2}, \ldots, R_{ip})$ lossless decomposition of R_i wrt $F_i = \pi_{R_i}(F)$

Then
$D = (R_1, R_2, \ldots, R_{i-1}, R_{i1}, R_{i2}, \ldots, R_{ip}, R_{i+1}, \ldots, R_k)$ is a
lossless decomposition of R wrt F

This property is useful in the algorithm for BCNF decomposition

Algorithm for BCNF decomposition

R – given schema. F – given set of FDs

```
D = {R}   // initial decomposition
while there is a relation schema $R_i$ in D that is not in BCNF do
{ let $X \rightarrow A$ be the FD in $R_i$ violating BCNF;
    Replace $R_i$ by $R_{i1} = R_i - \{A\}$ and $R_{i2} = X \cup \{A\}$ in D;
}
```

Decomposition of R_i is lossless as

$$R_{i1} \cap R_{i2} = X,\ R_{i2} - R_{i1} = A \text{ and } X \rightarrow A$$

Result: a lossless decomposition of R into BCNF relations

Dependencies may not be preserved (1/2)

Consider the schema: townInfo ($\overset{S}{\text{stateName}}$, $\overset{T}{\text{townName}}$, $\overset{D}{\text{distName}}$)
with the FDs F: $ST \rightarrow D$ (town names are unique within a state)
$D \rightarrow S$

Keys: ST, DT. – all attributes are prime
– relation in 3NF

Relation is not in BCNF as $D \rightarrow S$ and D is not a key

Decomposition given by algorithm: R_1: TD R_2: DS

Not dependency preserving as π_{R_1} (F) = trivial dependencies
$\pi_{R_2}(F) = \{D \rightarrow S\}$

Union of these doesn't imply $ST \rightarrow D$
$ST \rightarrow D$ can't be enforced unless we perform a join.

Dependencies may not be preserved (2/2)

Consider the schema: R (A, B, C)

with the FDs F: $AB \rightarrow C$ and $C \rightarrow B$

Keys: AB, AC – relation in 3NF (all attributes are prime)

– Relation is not in BCNF as $C \rightarrow B$ and C is not a key

Decomposition given by algorithm: R_1: CB R_2: AC

Not dependency preserving as $\pi_{R_1}(F)$ = trivial dependencies

$\pi_{R_2}(F) = \{C \rightarrow B\}$

Union of these doesn't imply $AB \rightarrow C$

All possible decompositions: {AB, BC}, {BA, AC}, {AC, CB}

Only the last one is lossless!

Lossless and dependency-preserving decomposition doesn't exist.

Equivalent Dependency Sets

F, G – two sets of FDs on schema R

F is said to <u>cover</u> G if $G \subseteq F^+$ (equivalently $G^+ \subseteq F^+$)

F is equivalent to G if $F^+ = G^+$ (or, F covers G and G covers F)

Note: To check if F covers G,

it's enough to show that for each FD $X \rightarrow Y$ in G, $Y \subseteq X^+_F$

Canonical covers or Minimal covers

It is of interest to reduce a set of FDs F into a “standard” form F′ such that F′ is equivalent to F.

We define that a set of FDs F is in ‘*minimal form*’ if

(i) the rhs of any FD of F is a single attribute

(ii) there are no redundant FDs in F

that is, there is no FD $X \to A$ in F

s.t $(F - \{X \to A\})$ is equivalent to F

(iii) there are no redundant attributes on the lhs of any FD in F

that is, there is no FD $X \to A$ in F s.t there is $Z \subset X$ for which

$F - \{X \to A\} \cup \{Z \to A\}$ is equivalent to F

Minimal Covers

useful in obtaining a lossless, dependency-preserving decomposition of a scheme R into 3NF relation schemas

Algorithm for computing a minimal cover

R – given Schema or set of attributes; F – given set of fd's on R

```
Step 1:  $G := F$

Step 2:  Replace every fd of the form $X \rightarrow A_1A_2A_3\ldots A_k$ in G
         by $X \rightarrow A_1$; $X \rightarrow A_2$; $X \rightarrow A_3$; … ; $X \rightarrow A_k$

Step 3: For each fd $X \rightarrow A$ in G do
            for each B in X do
                if $A \in (X - B)^+$ wrt G then
                    replace $X \rightarrow A$ by $(X - B) \rightarrow A$

Step 4: For each fd $X \rightarrow A$ in G do
            if $(G - \{X \rightarrow A\})^+ = G^+$ then
                replace G by $G - \{X \rightarrow A\}$
```

3NF decomposition algorithm

R – given Schema; F – given set of fd's on R in *minimal form*

Use BCNF algorithm to get a lossless decomposition $D = (R_1, R_2, \ldots, R_k)$
Note: each R_i is already in 3NF (it is in BCNF in fact!)

Algorithm: Let G be the set of fd's not preserved in D

```
For each fd $Z \rightarrow A$ that is in G
Add relation scheme $S = (B_1, B_2, \ldots, B_s, A)$ to D. // $Z = \{B_1, B_2, \ldots, B_s\}$
```

As $Z \rightarrow A$ is in F which is a minimal cover,
there is no proper subset X of Z s.t $X \rightarrow A$. So Z is a key for S!
Any other fd $X \rightarrow C$ on S is such that C is in $\{B_1, B_2, \ldots, B_s\}$.
Such fd's do not violate 3NF because each B_j's is prime a attribute!
Thus any scheme S added to D as above is in 3NF.

D continues to be lossless even when we add new schemas to it!

Multi-valued Dependencies (MVDs)

studCourseEmail(<u>rollNo,courseNo,emailAddr</u>)

a student enrolls for several courses and has several email addresses

rollNo $\rightarrow\rightarrow$ courseNo (read as rollNo *multi-determines* courseNo)

If (CS05B007, CS370, shyam@gmail.com)
(CS05B007, CS376, shyam@yahoo.com) appear in the data then

(CS05B007, CS376, shyam@gmail.com)
(CS05B007, CS370, shyam@yahoo.com)

should also appear for, otherwise, it implies that having gmail address has something to with doing course CS370 !!

By symmetry, rollNo $\rightarrow\rightarrow$ emailAddr

More about MVDs

Consider studCourseGrade(<u>rollNo,courseNo</u>,grade)

Note that rollNo →→ courseNo *does not* hold here even though courseNo is a multi-valued attribute of student

If (CS05B007, CS370, A)

(CS05B007, CS376, B) appear in the data then

(CS05B007, CS376, A)

(CS05B007, CS370, B) will not appear !!

Attribute ‘grade’ depends on (rollNo,courseNo)

MVD’s arise when two unrelated multi-valued attributes of an entity are sought to be represented together.

More about MVDs

Consider

studCourseAdvisor(<u>rollNo,courseNo</u>,advisor)

Note that rollNo →→ courseNo *holds* here

If (CS05B007, CS370, Dr Ravi)
(CS05B007, CS376, Dr Ravi)

appear in the data then swapping courseNo values gives rise to existing tuples only.

But, since rollNo → advisor and (rollNo, courseNo) is the key, this gets caught in checking for 2NF itself.

Alternative definition of MVDs

Consider R($\underline{X,Y,Z}$)

Suppose that X →→ Y and by symmetry X →→ Z

Then, decomposition D = (XY, XZ) should be lossless

That is, for any instance *r* on R, $r = \pi_{XY}(r) * \pi_{XZ}(r)$

MVDs and 4NF

An MVD X →→ Y on scheme R is called *trivial* if either Y ⊆ X or R = X ∪ Y. Otherwise, it is called *nontrivial.*

4NF: A relation R is in 4NF if it is in BCNF and for every nontrivial MVD X →→ A, X must be a superkey of R.

studCourseEmail(<u>rollNo,courseNo,emailAddr</u>)

is not in 4NF as

rollNo →→ courseNo and

rollNo →→ emailAddr

are both nontrivial and rollNo is not a superkey for the relation