Introduction

1.1 Motivation

- In an increasingly info-oriented society.
- Corporations recognize the importance of info. $\Rightarrow$ the development of corporate databases.
- Data become one vital resource$\Rightarrow$ how to manage data?

1.2 Data Management

- **Objective**: model a part of the real world accurately and efficiently so that the data are useful.
- **Abstraction** is the ability to hide detail and concentrate on common properties of a set of objects.

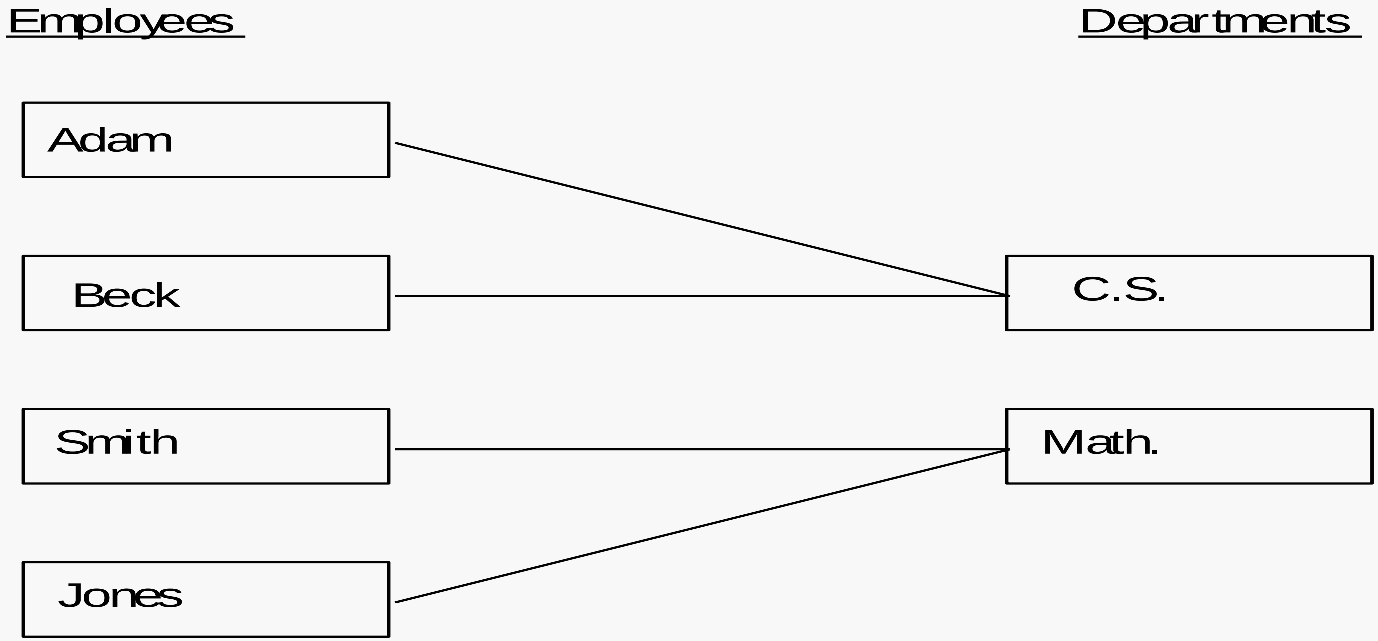
Ex:

Real World	**Model**	**Representation**
employees	EMP<name, sin, salary>	<Jones,123,21K>

- Extract relevant properties common to all employees.
- Model gives interpretation or meaning of data.
- Everyone has to understand and agree on the model.
- Also provide an acceptable model for data.

Intension and Extension of Data

- **Intension** of data is the definitional properties of data. E.g., EMPLOYEE(SIN, NAME, ADDRESS, BIRTHDATE, SALARY)
  - the abstraction of the actual data.
- **Extension** of data is the actual occurrences of data. E.g., the set of employee records.
- Apply to associations of objects.
- The intension of data (**a database schema**) describes the logical organization of data in a database and the extension of data (**a database state**) is the actual occurrence of data in the database.



Extension

Intension

1.3 Components of a Database System

1. Hardware

- Basic hardware: CPU, I/O devices.

2. Software

Application Programs

- Perform specific functions, e.g. accounting, billing, payroll ...

Utility Programs

- Facilitate utilization & maintenance of a database. E.g. loading, reorganization and analysis routines.

Operating System

- Controls the computer system resources.
- Perform I/O.

Database Management System (DBMS)

- Dictate how data are organized & accessed.
- Major functions:

(a) Define data structures.

(b) Allow data to be stored, retrieved & updated easily & efficiently.

(c) Provide wide variety of access methods.

(d) Provide multiple views of data. E.g., EMP file sorted on Emp# or on Dept.

(e) Support concurrency control, integrity, security and recovery.

(1) Concurrency Control

- Operations of different users may interleave.

	User A	**User B**
1.	Read AMT (10),	Read AMT (10),
2.	Decremented by 5,	Decremented by 3,
3.	Write AMT,	Write AMT,

(i) Read AMT (for A)
(ii) Read AMT (for B)
(iii) Decremented by 5
(iv) Decremented by 3
(v) Write AMT (for A)
(vi) Write AMT (for B)

AMT = 7 ???

(2) Security

- Protection of data against unauthorized disclosure, alternation or destruction.
  (a) Physical Security
  - protecting the physical resources from fire, flood,...
  - controlling over physical access to the system.
  (b) Against Legal Users
  - accomplished by password.
  - may not see all data - **access control** - defined when an account is assigned.

(3) Integrity

- Ensuring the data in the d.b. are accurate and meaningful.
- Data have properties:

  e.g.

  EMP(EMP#, NAME, ADDR, SALARY),

  MANAGER(EMP#, DEPT)

  SALARY < 1M;

  EMP# is unique,

  EMP.EMP#$\supseteq$MANAGER.EMP#.

(4) Recovery from failures

- E.g., disk head crash, main memory malfunction, bugs or incorrect data.
- Read Saving_Acct
  Saving_Acct = Saving_ Acct - 10
  Write Saving_ Acct
  Read Checking_Acct
  Checking_Acct = Checking_Acct +10
  Write Checking_Acct
- Must be recovered ASAP & be transparent.

3. Data

- Facts about the organization.
- Logical files - as seen by application programmers.
- Physical files - as seen by system programmers.
- Files are interrelated, e.g., IN_CHARGE(EMP, DEPT, PROJECT).

4. People

(a) Users

- Employ the d.b. system to satisfy a business need.
- Casual users or end users.

(b) Operation personnel

- Machine operators, data entry personnel etc.

(c) System development personnel

- Design, implement & maintain the system.

(d) Database Administrator (DBA).

5. Procedures

- Users - how to sign on, how to use terminal, how to input data....
- Operation staffs - how to start and stop process, how to perform backup, how to mount disks ....
- Failures - describing what to do.

1.4 DBMS Architecture

- Dynamic environment.
- Programs are not independent ⇒ high maintenance cost.
- Reduce interdependence among various components.
- **Data independence**: the degree of independence between data and application programs that use them so that either can be changed with minimal effects on the other.
- Logical changes, e.g., add/delete logical data. Physical changes, e.g., storage media & access methods.

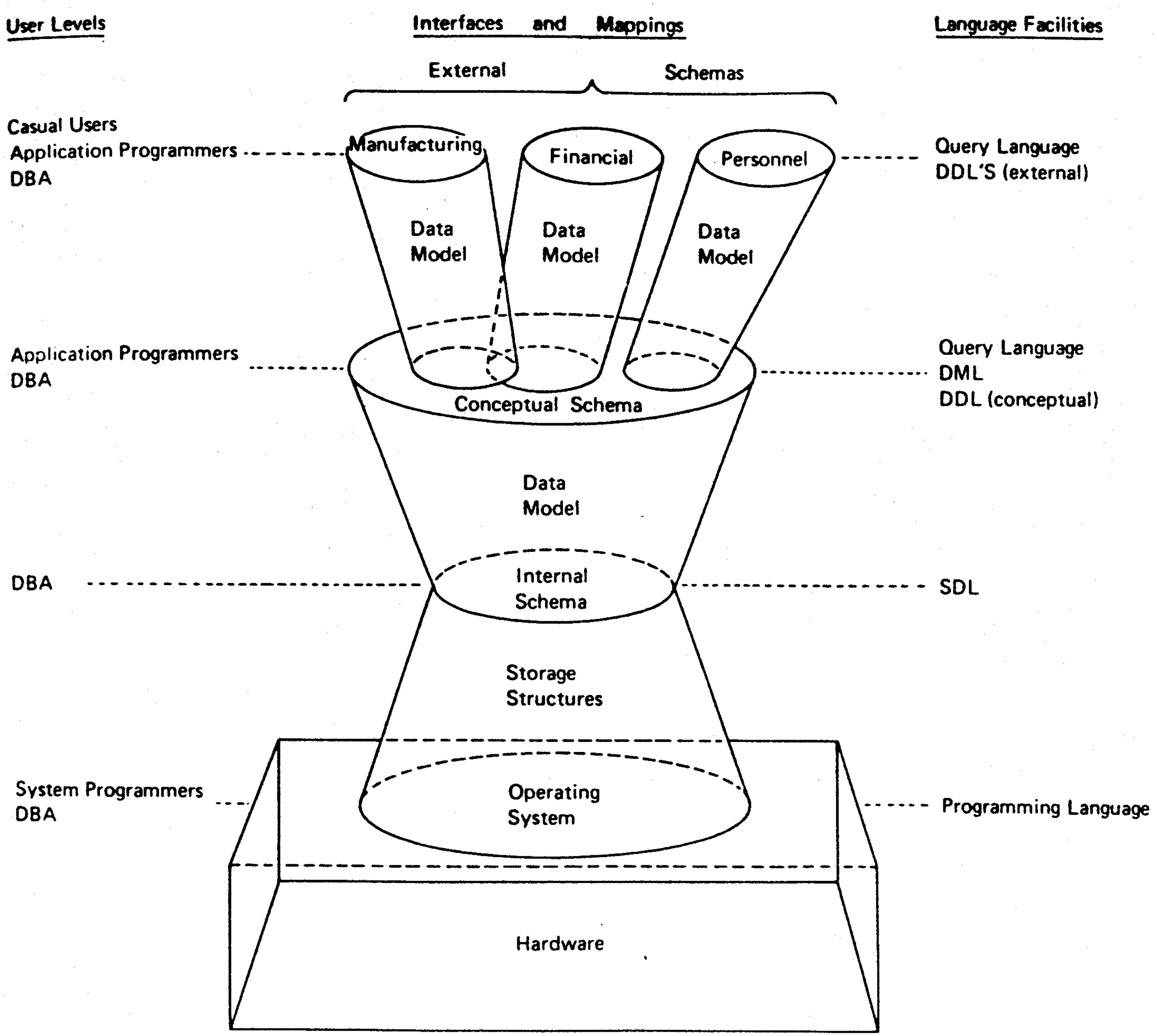


Fig. 4.5—1 A DBMS architecture.

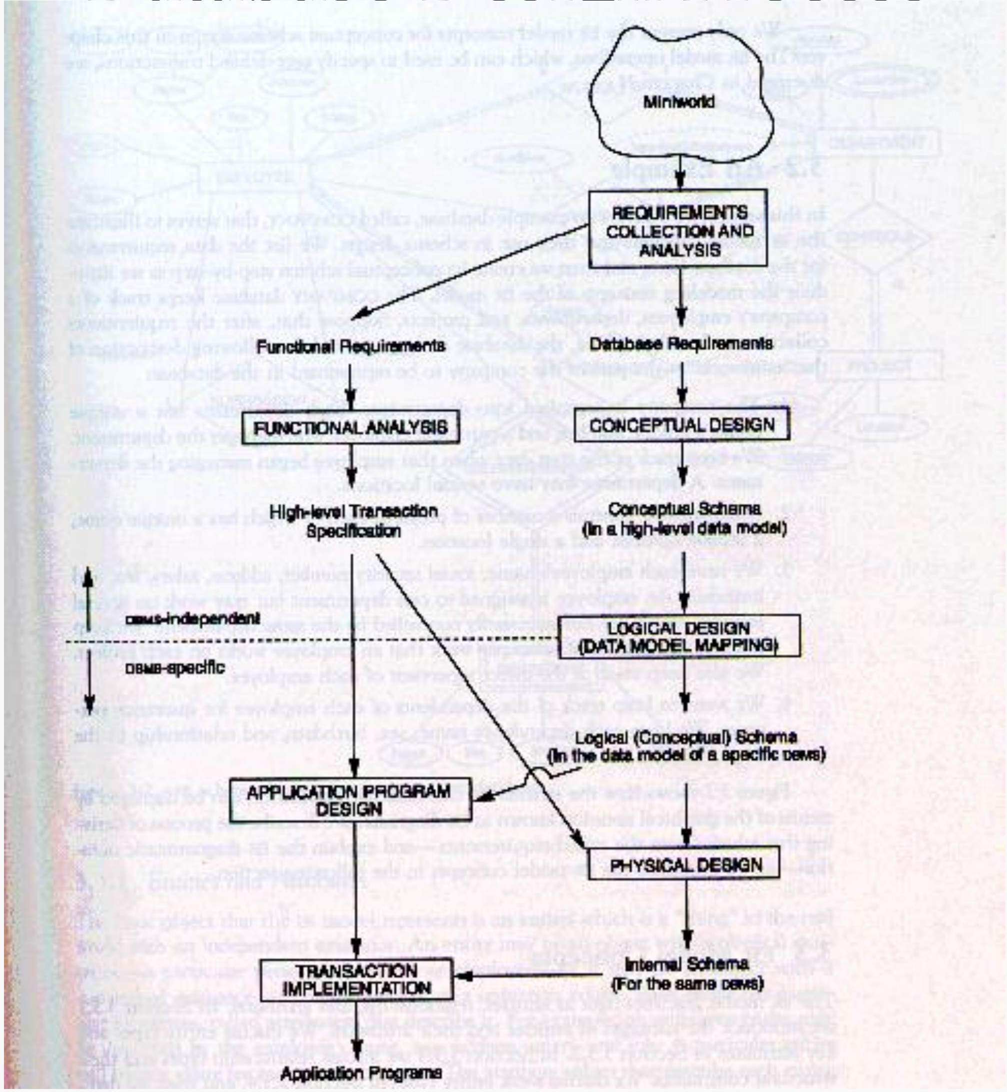
Lauguage Facilities

- **Data Definition Language (DDL)**
  - specifies conceptual schema and subschemas and the mappings between them.
- **Data dictionary**, **data directory** or **system catalog**
  - result of compilation of DDL statements in a schema; metadata.
- **Data manipulation language (DML)**
  - commands issued in a host program.
  - preprocess or compiler interprets these commands as calls to DBMS.
- **Query language**
  – a complete language for manipulating data interactively.

Data Models

- Guidelines for the logical organization of data.
- Like a programming language.
- **Structure** - ways in which data are logically organized.
- **Operations** - actions on structure.
- **Constraints** - logical restrictions on data.
  - Inherent - implied by the structure, e.g., sets of objects.
  - Explicit - integrity constraints or assertions.
- **Value-based:**
  - objects are denoted by their values or properties.
- **Object-based:**
  - allows pointers or references to other objects.

Database Design Process

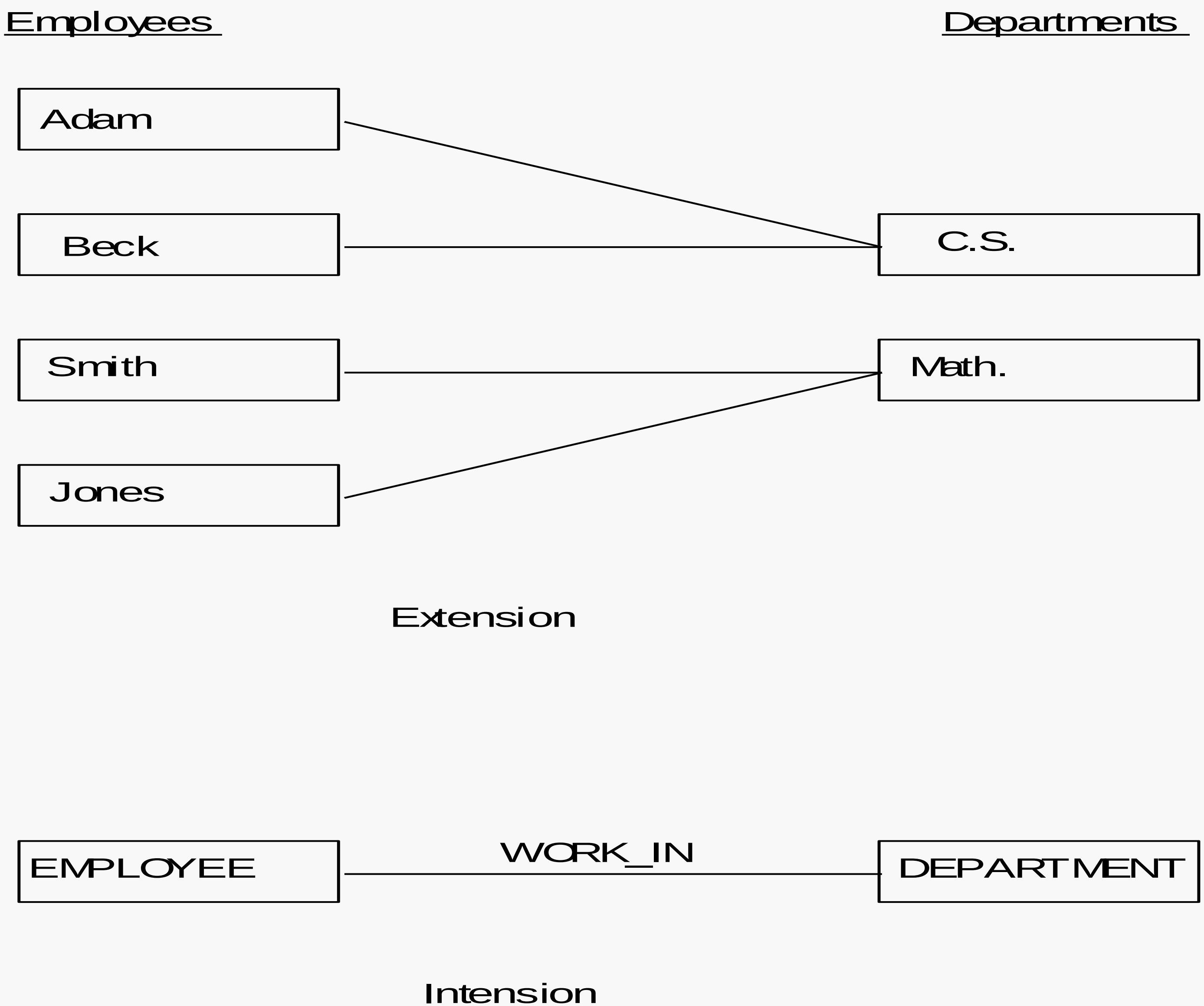


Entity-Relationship Model (ERM)

1. Structure

- Either an entity or a relationship.
- **Entity** - a thing or an object that has some properties & can be distinctly identified. E.g., accounting dept., employee Jones .... .
- **Entity set** - entities with similar properties. (Extension)
- **Entity type** - abstraction of a set of entities with the same properties. (Intension)
  E.g., DEPT(NAME, ADDR, MANAGER, TEL)
- **Attributes** (intension), an instance of an attribute is a **value**, the set of possible values is the **domain** of attribute.
- Values could be **single-valued** or **multi-valued**.
- Attributes could be **simple** or **composite**, and could be **single-valued** or **multi-valued**.

- **Relationship** - an association between two or more entities.
  E.g., **John** works in the **personnel dept**.
- **Relationship set** - a collection of similar relationships.
  E.g., John works in personnel dept.
  Smith works in accounting dept.
  Paul works in EDP dept.
  .
- **Relationship type** - abstraction of similar relationships.
  E.g., WORK_IN between EMP & DEPT.
- Relationship type can have attributes. E.g., length_of_service.
- N-ary: defined on N entity types, need not be distinct.
- Object-based data model.

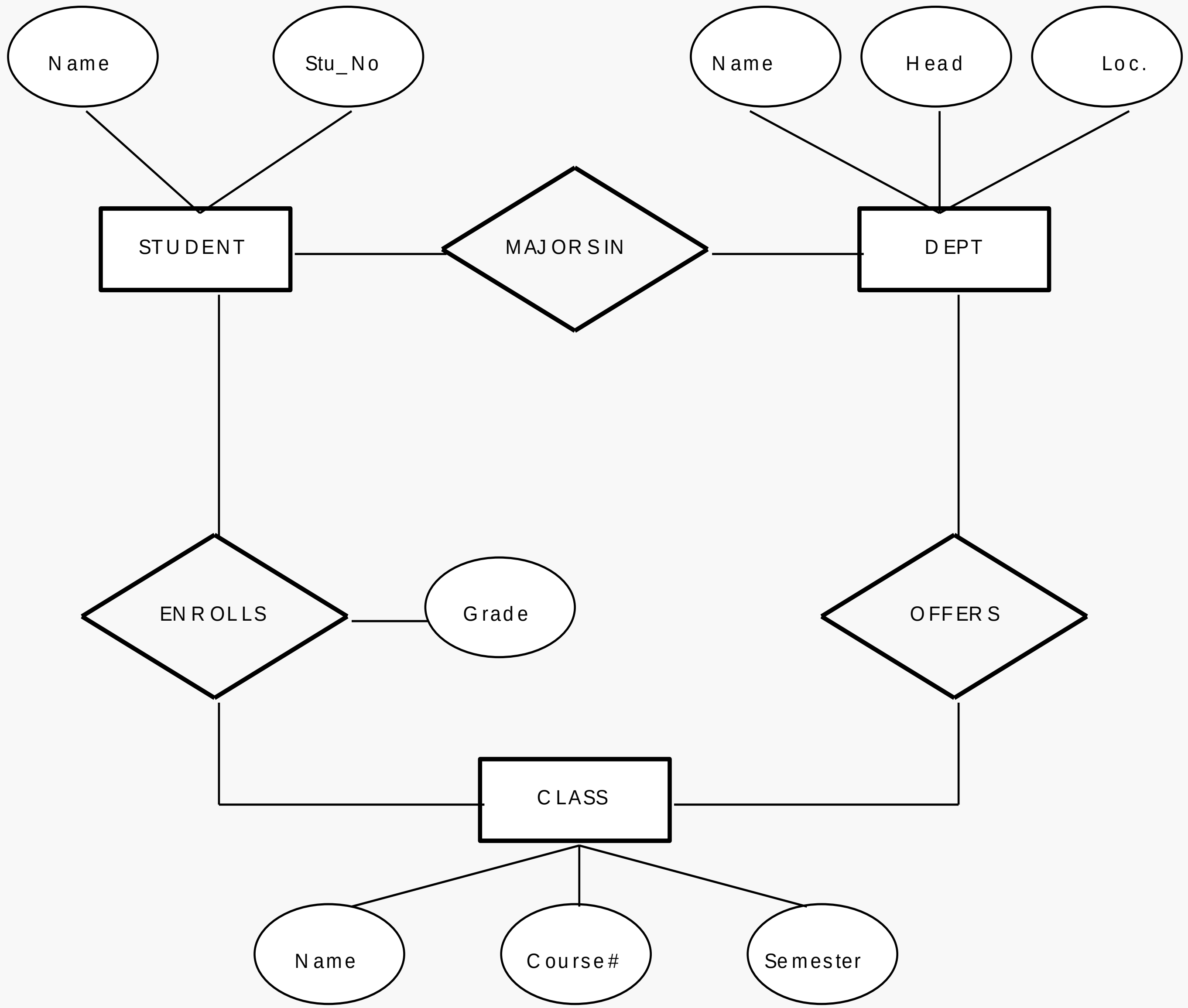

Employees
Departments
Adam
Beck
Smith
Jones
C.S.
Math.
Extension
EMPLOYEE
WORK_IN
DEPARTMENT
Intension

Entity-Relationship Diagram (ERD)

- ERD
  - labeled rectangles - entity types and attributes must be specified.
  - labeled diamonds - relationship types, linked to their entity types,
  - single or doubled labeled ellipses - single or multi-valued attributes.
  - attributes in an entity type can also be specified as a record type.

- In this university, we have many departments and each department is offering many courses. On the other hand, a course is offered by exactly one department. The properties of a department that we are interested in are its name, its chairman and the location of its general office. Every course is represented by its course# and the semester in which the course is offered. An example of a course# is "CS360" and an instance of a semester is "90W" which denotes 90 winter term. We assume each course has at most one section in each semester. Other information is its name, such as "Introduction to Theory of Computation."

Students are majoring with a department and each department can have many students majoring in it. Students can enroll in many courses and a course can be taken by many students. The grade obtained by a student is also recorded. The student information are the student name and the student number. Student numbers are assumed to be unique and are used to identify individual students.


Name
Stu_No
Name
Head
Loc.
STUDENT
MAJORS IN
DEPT
ENROLLS
Grade
OFFERS
CLASS
Name
Course#
Semester

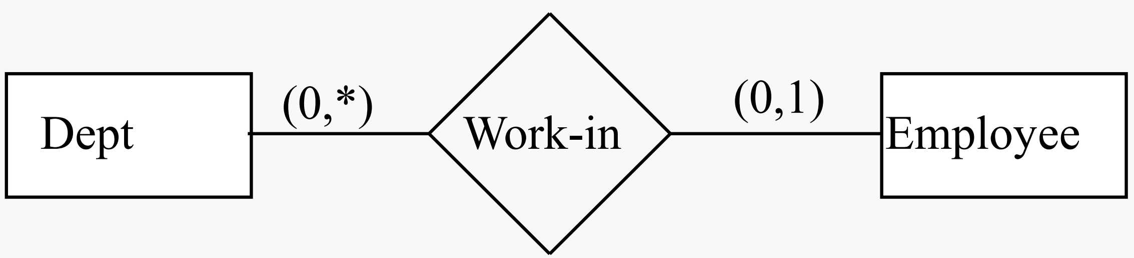
2. Constraints

2.1 Primary Keys

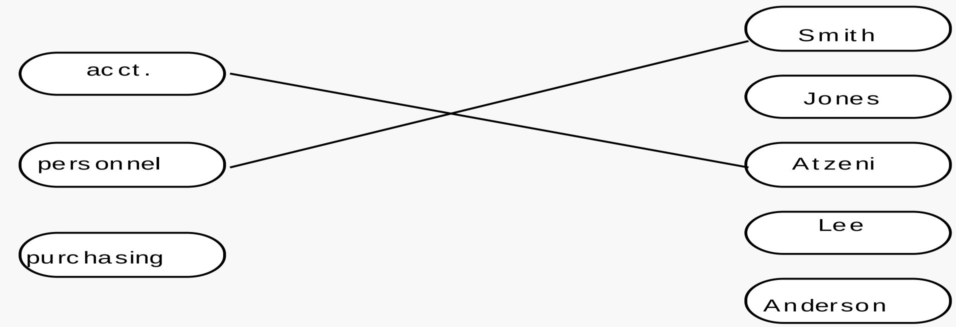
- **Candidate key** - a minimal set of attributes of an entity type whose values - uniquely identify an entity at all times
  - Optional
- **Primary key** - principal means of identifying an entity.
  - must be a candidate key
  - underlined attributes

2.2 Cardinality

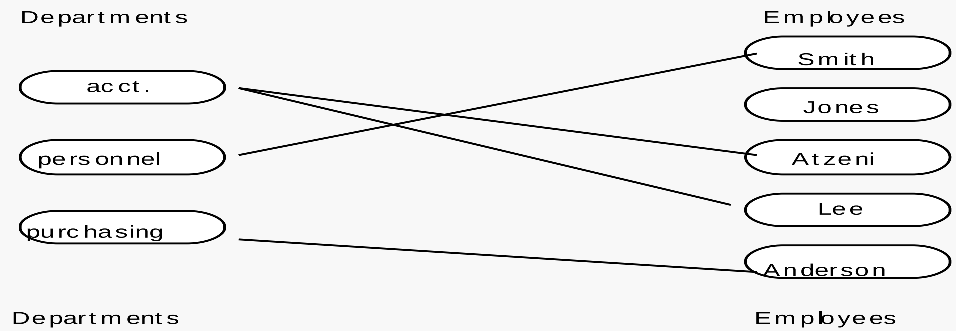
- For relationship types.
- **Cardinality** - restrictions placed on the number of relationships an entity may participate.
- Binary relationships:
  1:1 , 1:N , N:M
- (min, max): non-negative integers or *



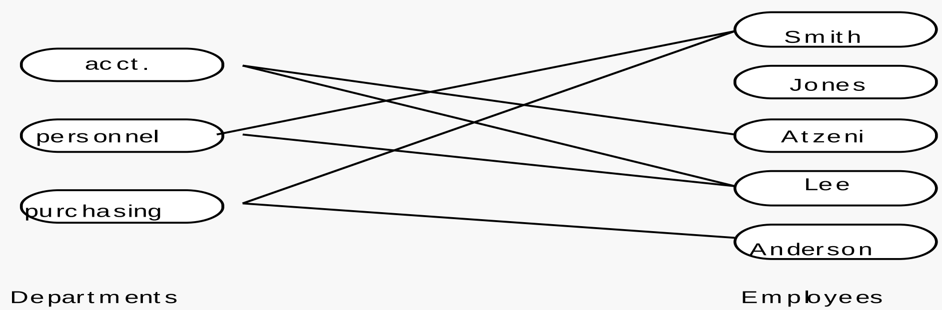
A dept has zero or more employees while an employee has zero or one dept associated with it.



1:1



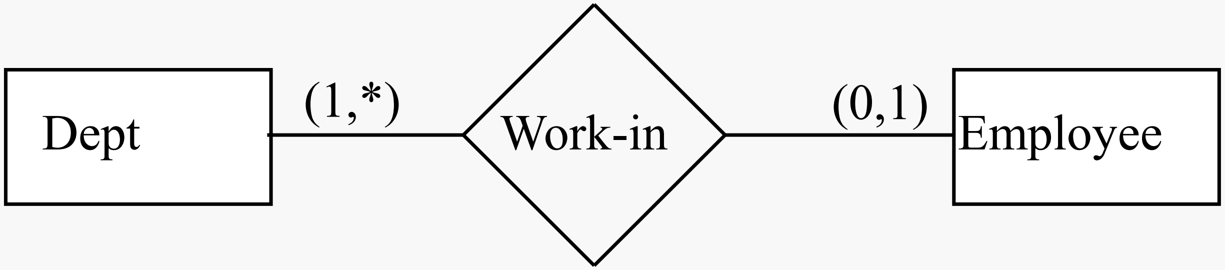
1:many

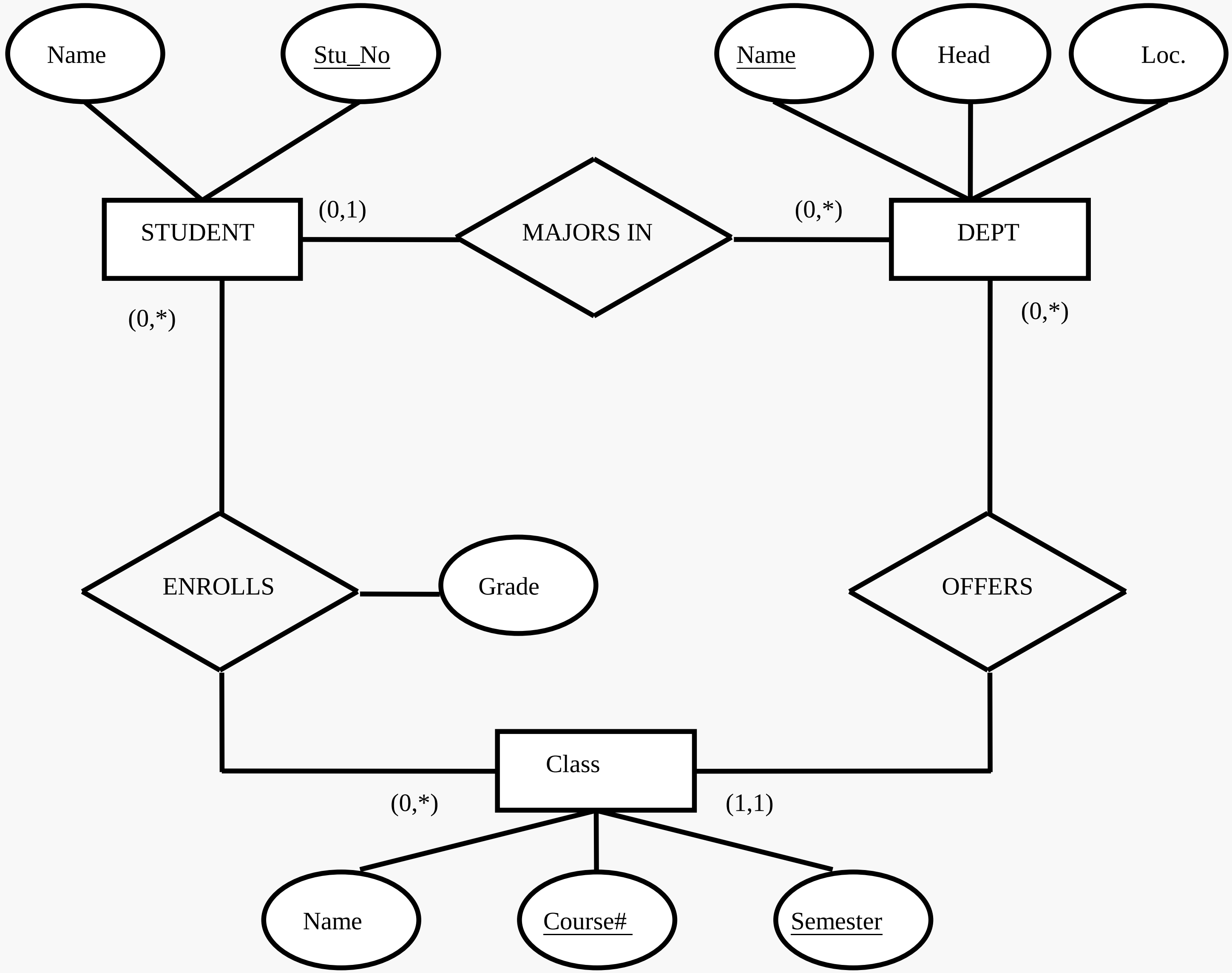


many:many

2.3 **Existence Constraints**

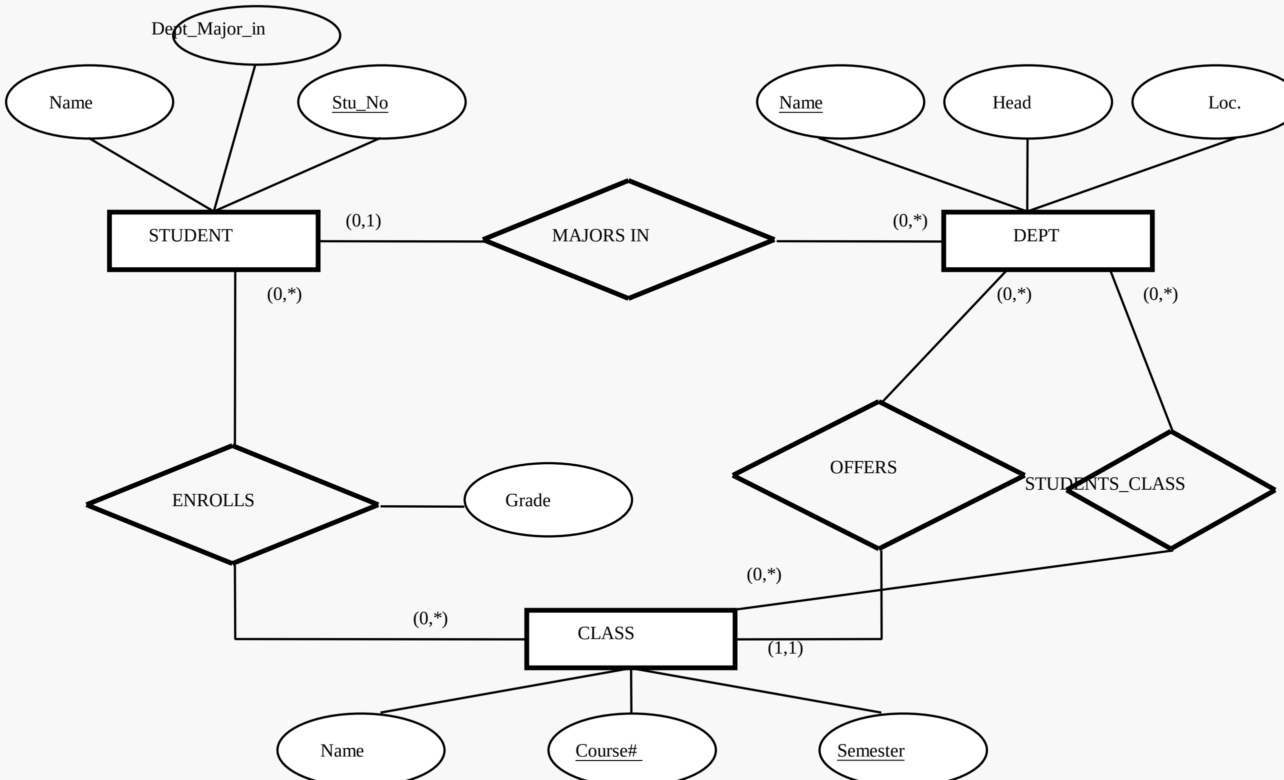
- Govern if an entity can exist independently.
- An entity type A (Dept) is **totally dependent** on an entity type B (Employee) (via a relationship type C (Work-in)) if every entity on A is **always** associated with at least one entity on B via the relationship type C.




Name
Stu_No
Name
Head
Loc.
STUDENT
(0,1)
MAJORS IN
(0,*)
DEPT
(0,*)
(0,*)
ENROLLS
Grade
OFFERS
Class
(0,*)
(1,1)
Name
Course#
Semester

Guidelines for Drawing ERDs

- A relationship or an attribute is **redundant** if its removal from an ERD does not affect the information content.
- Keys of entity types are for identification purposes.



3 Extended ERM

3.1 Generalization & Specialization

- In a university, there are academic staffs and non-academic staffs. Academic staffs consist of professors and the librarians. Academic staffs are unionized and have benefits that are different from non-academic staffs. Non-academic staffs are administrative staffs, clerical staffs and graduate students. Non-academic staffs can make contribution to a pension fund. Since contribution is optional, different staffs may have different contributions.

Librarian (Emp#, Name, Salary, Addr, Professional allowance, union due, library_name).

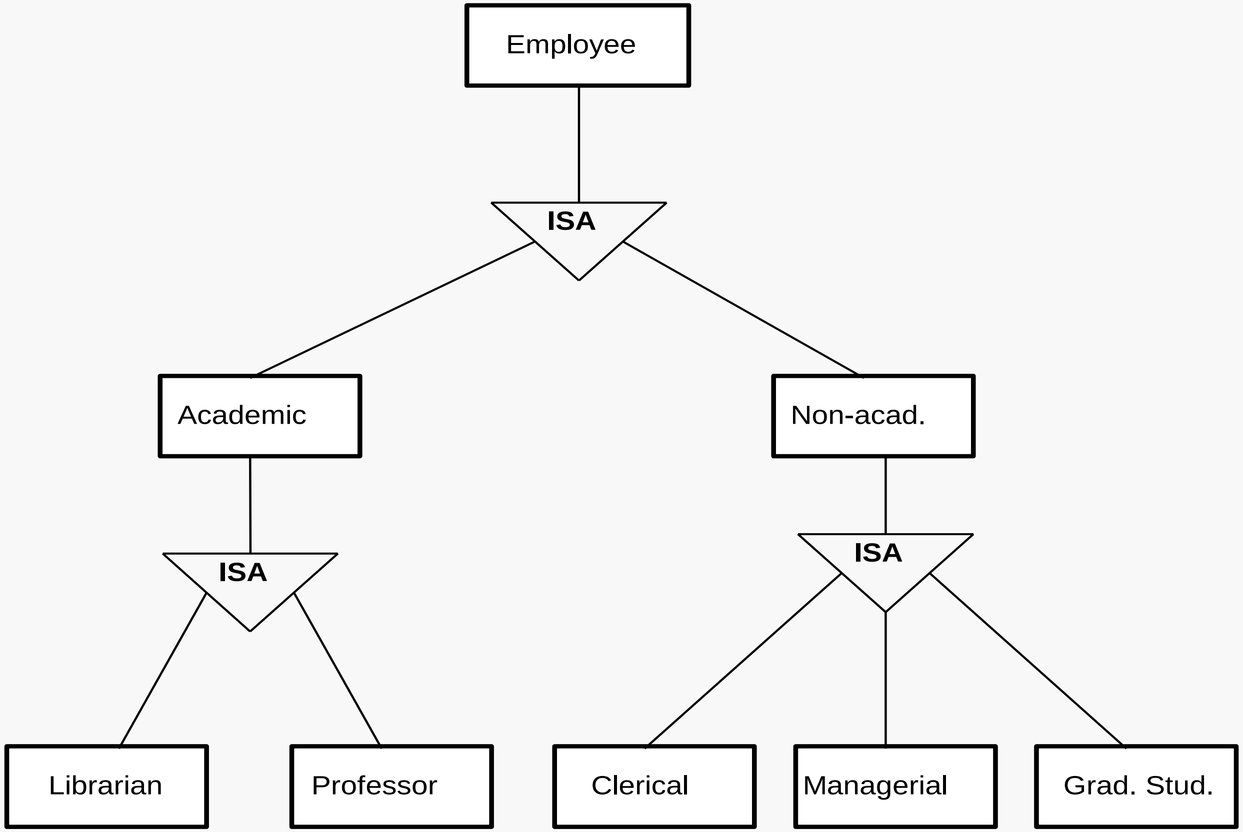
Professor (Emp#, Name, Salary, Addr, Professional allowance, union due, rank, dept).

Managerial (Emp#, Name, Salary, Addr, RRSP_contribution, Job_title, Office).

Clerical (Emp#, Name, Salary, Addr, RRSP_contribution, Job_title, Skills).

Grad.Stud. (Emp#, Name, Salary, Addr, RRSP_contribution, Job_title, Dept_work_for, Student#)

- All share some common properties or attributes.
- **Generalization** or **ISA-relationship**: two levels of entity types.

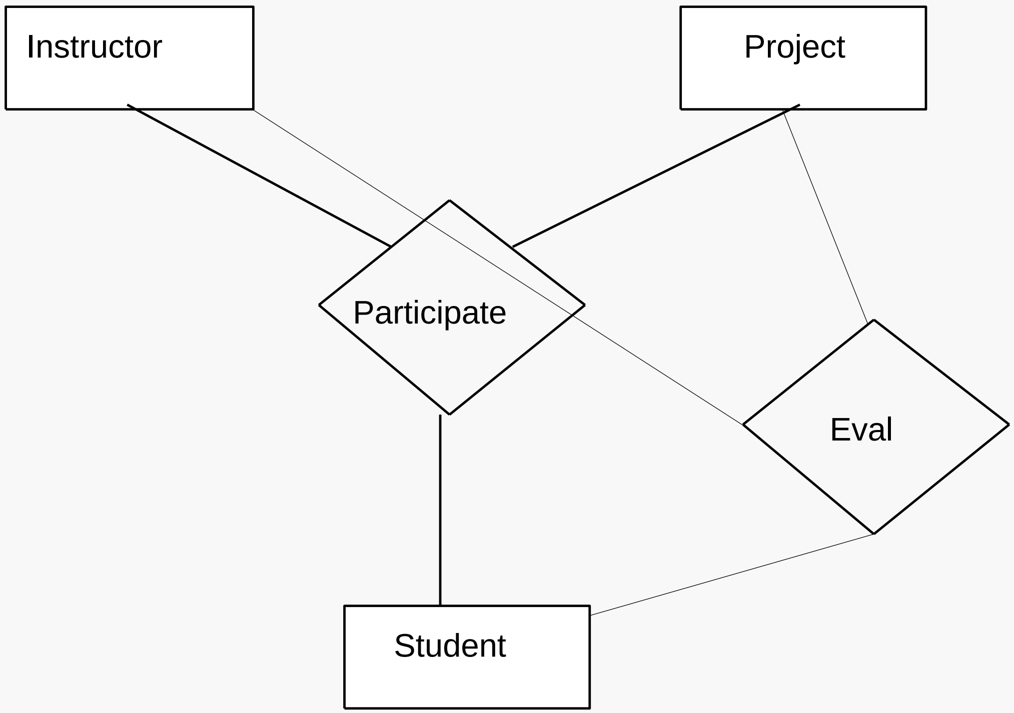


Employee(<u>Emp#</u>, Name, Salary, Address)
Academic(Professional Allowance, Union due)
Non-acad(RRSP_contribution, Job_title)
Grad.Stud.(<u>Student#</u>, Dept_work_for)
Librarian(Library_name)
Professor(Rank, Dept)
Managerial(Office)
Clerical(Skills)

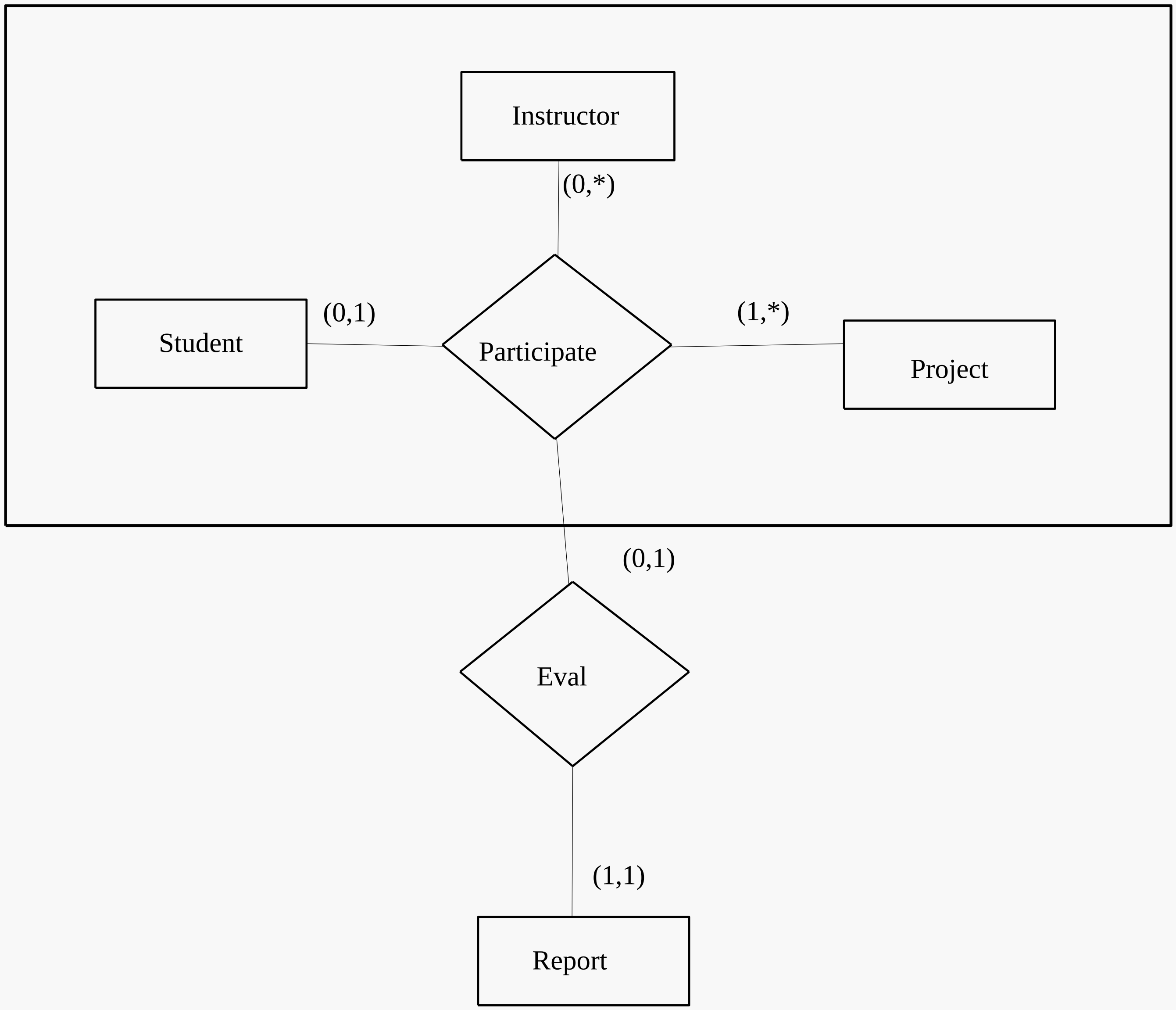
- The higher level - the **generalized** entity type, the lower-level - **constituent** entity types.
- The constituent entity types are **specializations** of the generalized entity type.
- Attributes of generalized entity type are **inherited** by its constituent entity types.
- A specialized entity **is a** generalized entity.
- Generalization relationships can be built on another generalization relationships to form a **generalization hierarchy** or **ISA-hierarchy**.

3.2 Aggregation

- Consider the ternary relationship *participate*
- Suppose we want to record evaluations of a student by a guide on a project

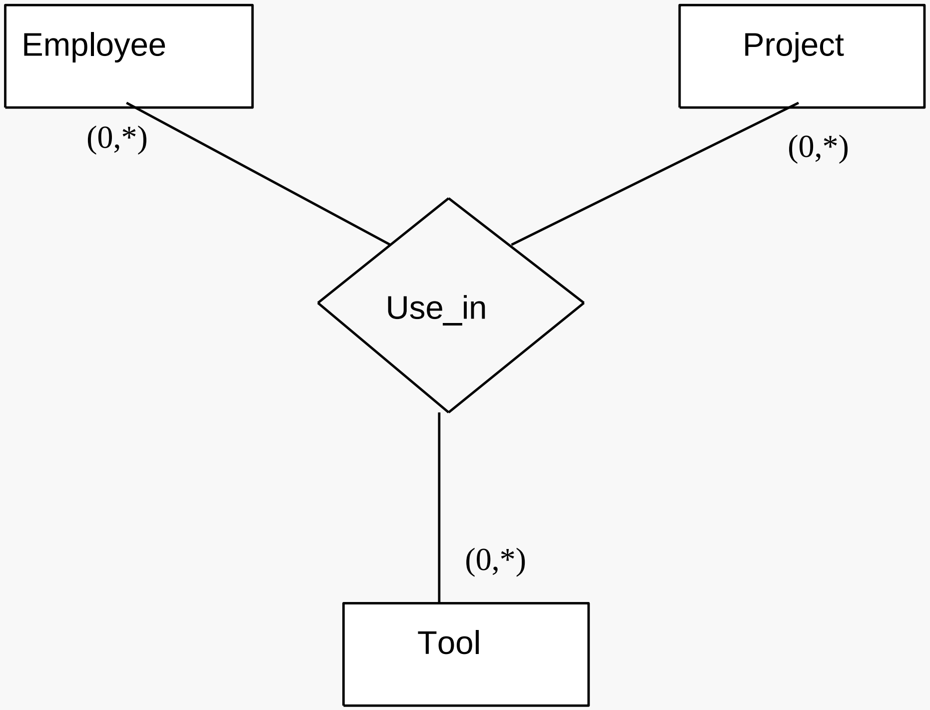


- A student is guided by a particular instructor on a particular project – view it as a high-level object
- A student, instructor, project combination may have an associated evaluation
- Without introducing redundancy and capture the dependency, the following diagram represents:

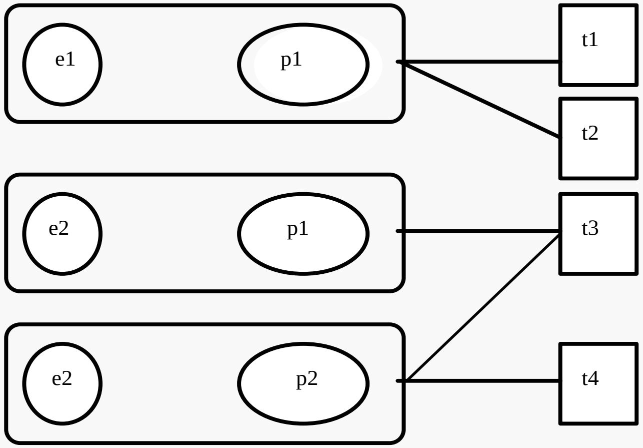

Instructor
(0,*)
Student
(0,1)
Participate
(1,*)
Project
(0,1)
Eval
(1,1)
Report

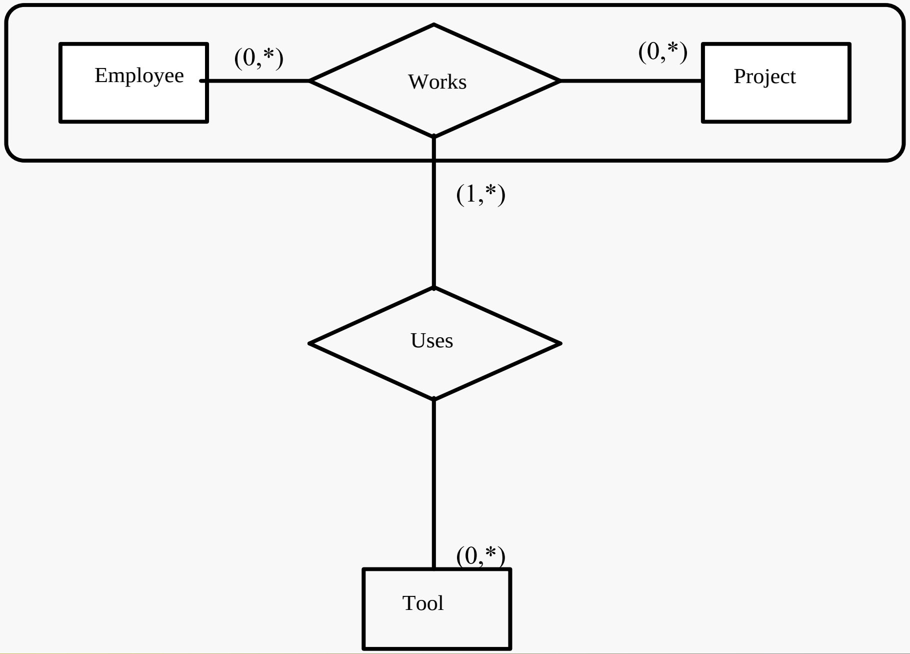
Another Example

- Suppose no restrictions on employees and projects, and a tool can be assigned to zero or more person-project combination. Consider the following two scenarios:
  - a person-project combination can use exactly one tool at any time.
  - a person-project combination can use one or more tools at any time
- Both are represented as follows:


Employee
Project
(0,*)
(0,*)
Use_in
(0,*)
Tool

- Represent the second scenario, the first is represented by changing the cardinality





Storage Systems and File Structures

Introduction

- Data are stored on some storage medium.
- Primary storage: operate on by CPU, fast but expensive and with limited capacity.
- Secondary storage: less expensive, larger capacity, but slower access time. Cannot be processed directly by the CPU; need to copy to primary storage first.

Primary Storage

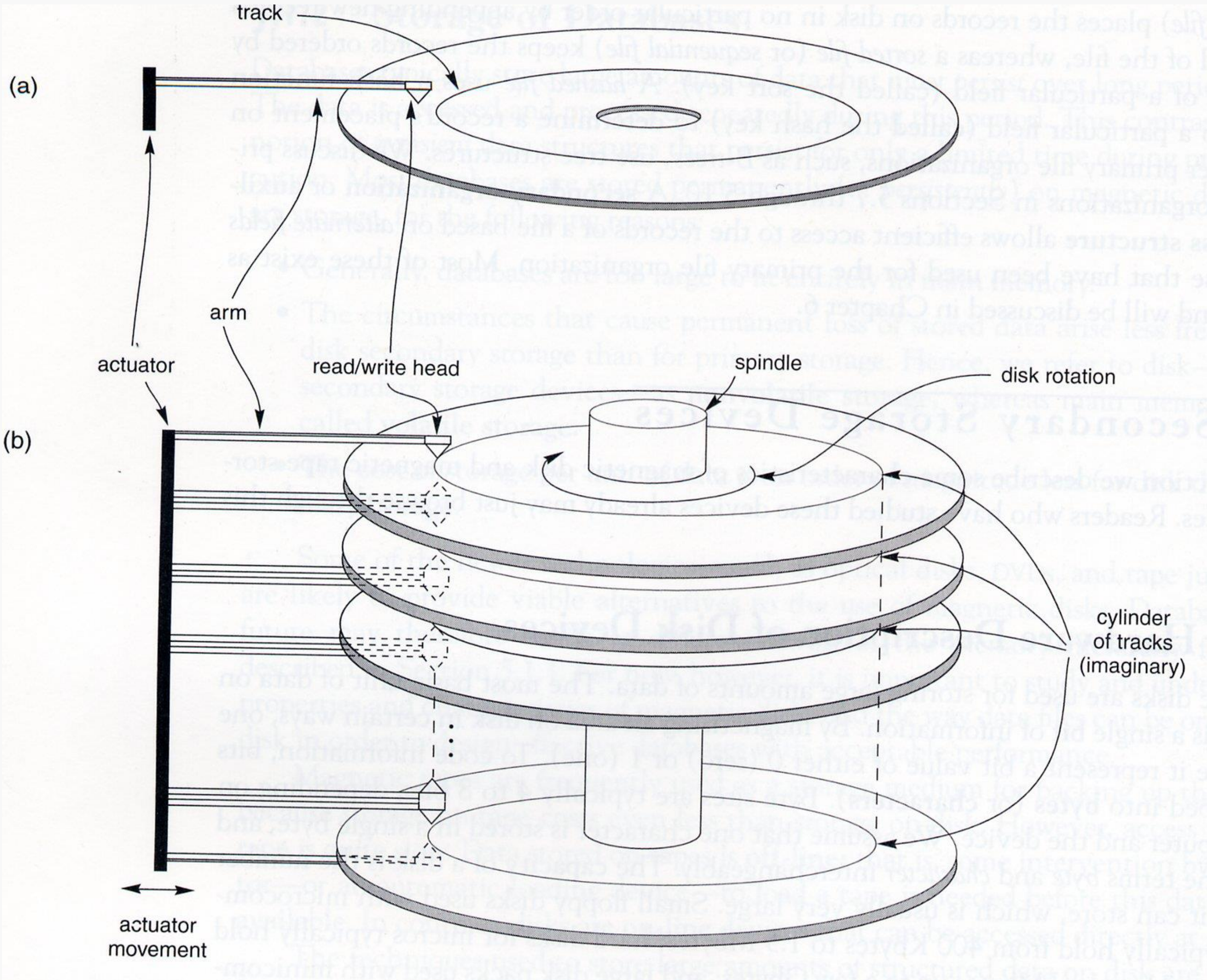
- Cache memory, dynamic random-access memory (DRAM).
- Fast but expensive.

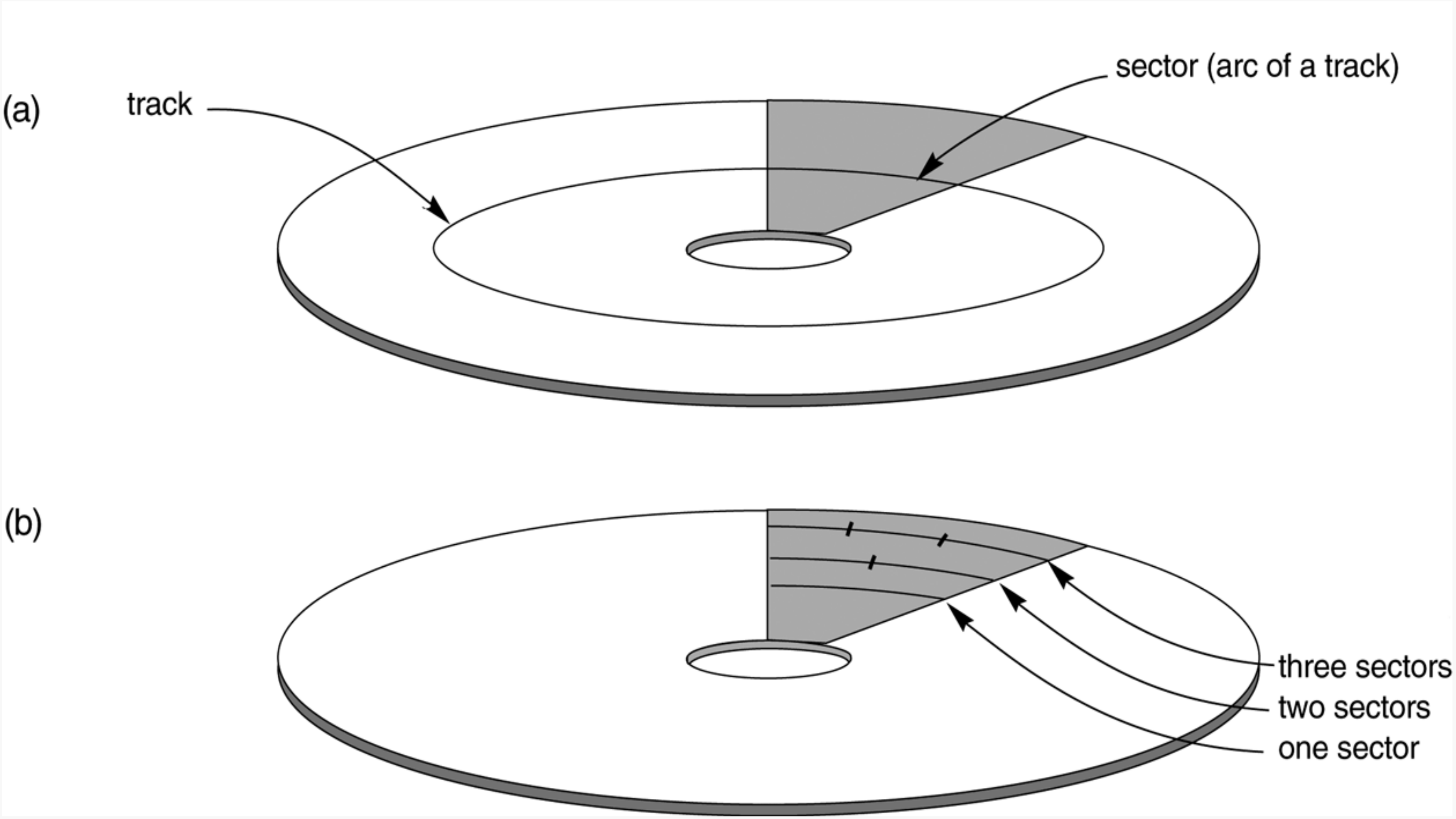
Secondary Storage

- The storage capacity is measured in gigabytes (Gbyte or 1 billion bytes), and terabytes (1000 Gbytes).
- Disk, tape, CD-ROM, flash memory.
- Databases – large amount of data and must persist over a long period of time.
- Too large to fit entirely in main memory.

Disk Storage Devices

- Disks are the most common.
- Capacity of a disk – up to terabytes.
- Data stored as magnetized areas on magnetic disk surfaces.
- A **disk pack** contains several magnetic disks connected to a rotating spindle.
- Disks are divided into concentric circular **tracks** on each disk **surface**.
- Each circle is called a *track*; each has a distinct diameter.
- Tracks with the same diameter on a disk pack form a *cylinder*.
- Each track is divided into equal size units called *blocks* or *pages*.


track
(a)
arm
actuator
read/write head
spindle
disk rotation
(b)
cylinder
of tracks
(imaginary)
actuator
movement


(a)
track
sector (arc of a track)
(b)
three sectors
two sectors
one sector

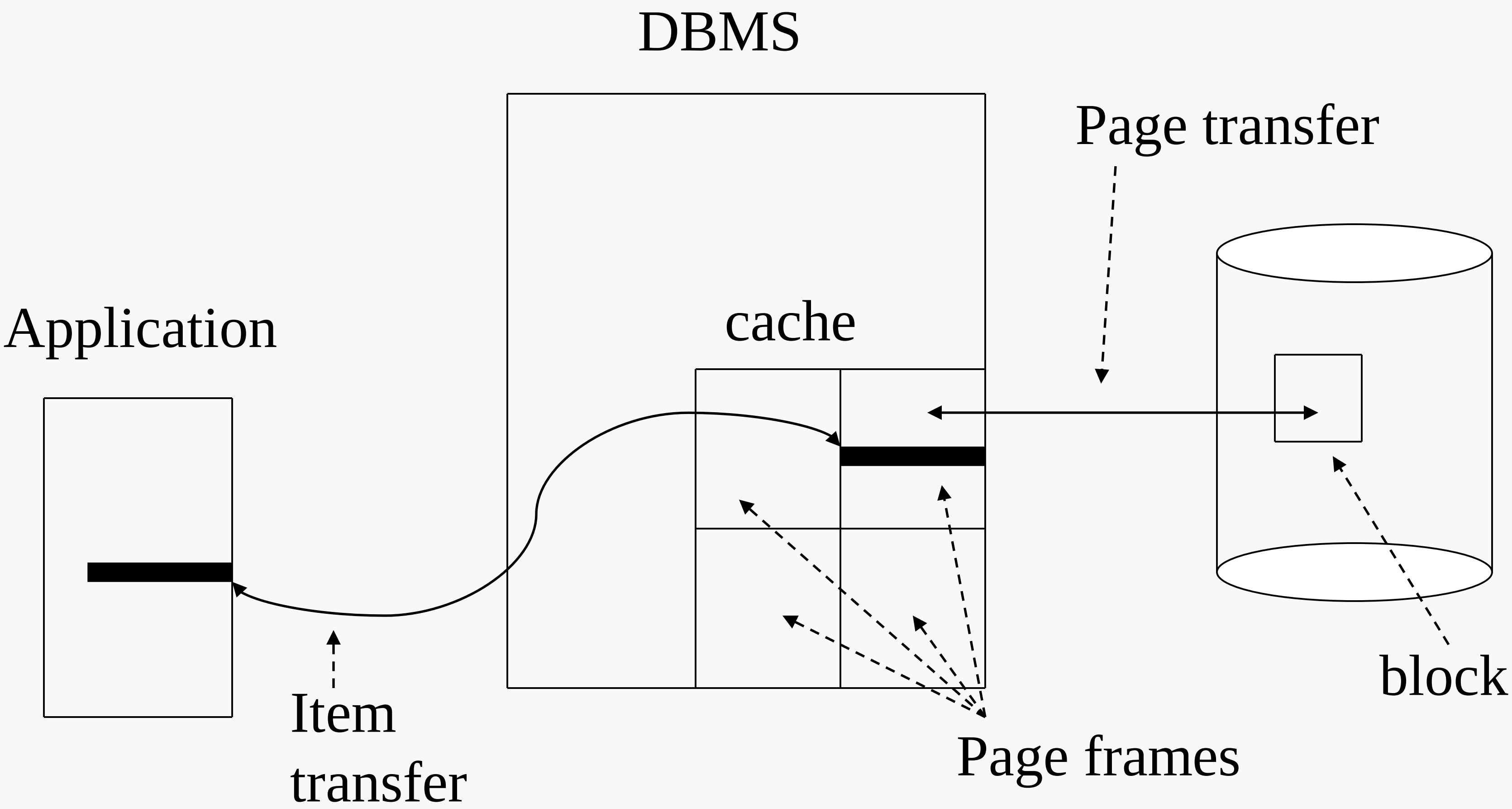
- Whole blocks are transferred between disk and main memory for processing.
- A disk pack is mounted on a disk drive which consists of read/write heads, and a motor that rotates the disks.
- Disk packs are rotated in a constant speed (3600 – 15000 rpm).
- A physical disk block (hardware) address consists of: the track number, the surface number and the block number (within a track).
- A **read-write head** moves to the track that contains the block to be transferred.

- The read/write heads are activated when position on the right tracks.
- Total time to access a block: **seek time + rotational delay** + transfer time.
- Access time: several to 30 milli-seconds.
- CPU processing time: nano-seconds.
- **I/O is the bottleneck** !!!!!

Reducing Latency and Page accesses

- Store pages containing related information close together on disk
  - *Justification*: If application accesses *x*, it will next access data related to *x* with high probability
- Keep cache of recently accessed pages in main memory
  - *Rationale*: request for page can be satisfied from cache instead of disk
  - Purge pages when cache is full
    - For example, use LRU algorithm
    - Record clean/dirty state of page (clean pages don't have to be written)

Accessing Data Through Cache



- A **file** is a *sequence* of records.
- Records are stored on disk blocks.
- A file can have **fixed-length** records or **variable-length** records.
- No record can span two blocks
- The physical disk blocks that are allocated to hold the records of a file can be *contiguous, linked, or indexed*.

Ordered Files

- Also called a **sequential** file.
- File records are kept sorted by the values of an *ordering field*.
- Insertion is expensive: records must be inserted in the correct order.
    - It is common to keep a separate unordered *overflow* file for new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its *ordering field* value.
    - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
			⋮			
	Acosta, Marc					
block 2	Adams, John					
	Adams, Robin					
			⋮			
	Akers, Jan					
block 3	Alexander, Ed					
	Alfred, Bob					
			⋮			
	Allen, Sam					
block 4	Allen, Troy					
	Anders, Keith					
			⋮			
	Anderson, Rob					
block 5	Anderson, Zach					
	Angeli, Joe					
			⋮			
	Archer, Sue					
block 6	Arnold, Mack					
	Arnold, Steven					
			⋮			
	Atkins, Timothy					
			⋮			
block n –1	Wong, James					
	Wood, Donald					
			⋮			
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
			⋮			
	Zimmer, Byron					

Indexing Structures for Files

Introduction

- Given an attribute value, retrieves all records with a given value.
- Without indexes, may require a sequential search the whole file. SLOW if the file is LARGE!!
- An index consists of extra information (a data structure) added to a file to provide faster access to data $\Rightarrow$ an index file and a data file.
- With cost!!

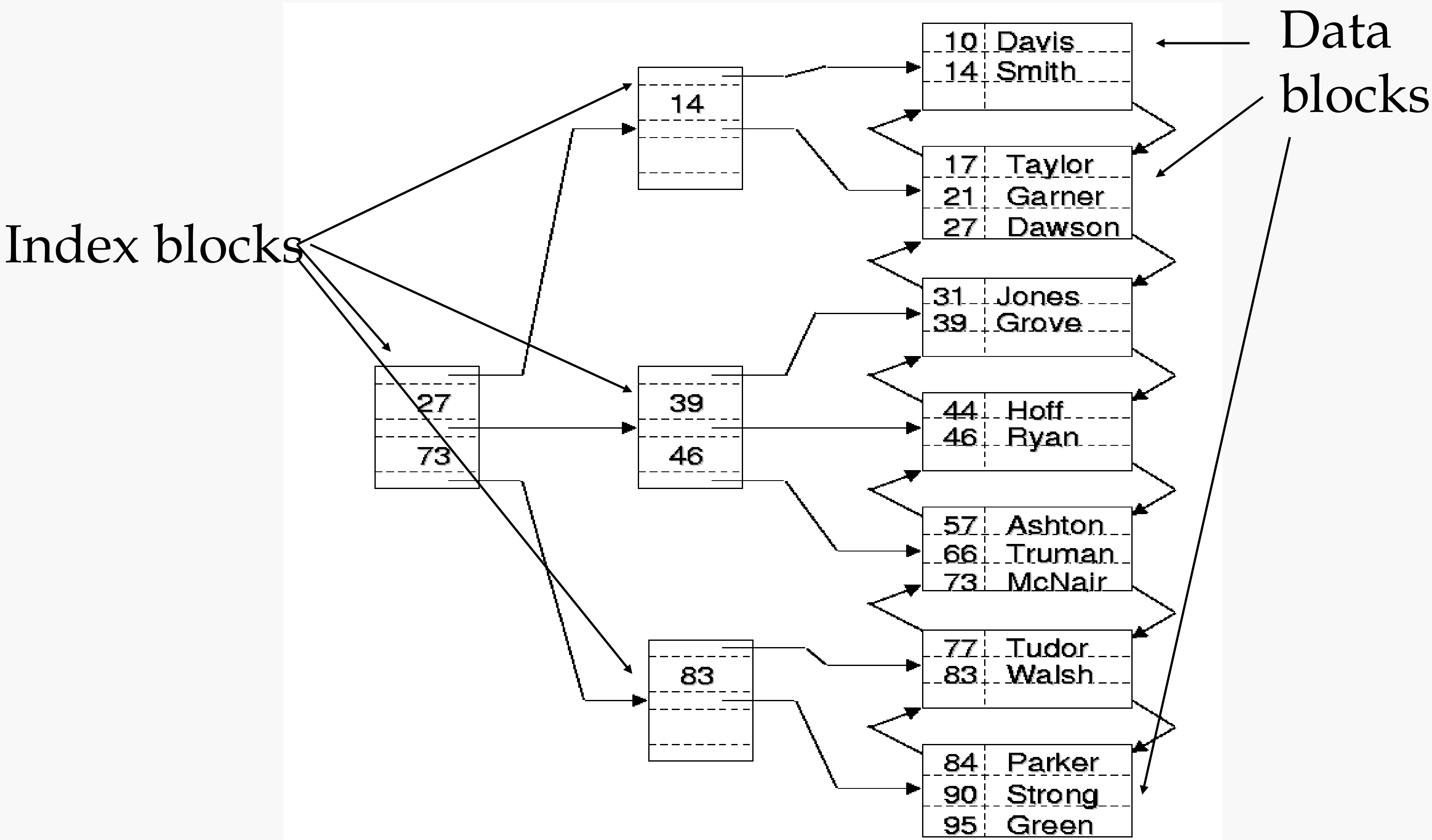
Types of Indices

- *Search key*: the set of attributes on which an index is built.
- *Primary* index: the search key is the primary key, otherwise *secondary* index.
- *Ordered* index: the search key values in the index file are ordered, otherwise *unordered*.

B^+-trees

- B^+-trees are widely-used index structures.
- B^+-trees are fully *dynamic*: they easily grow and shrink.
- B^+-trees have two parts: index blocks and data blocks.
- Assume the index attribute is the primary key.
- Entries in a leaf are sorted records. Entries in non-leaf are <*key*> and <*ptr*>, where <*ptr*> points to a page.
- Search key values in an index is in sorted order – multi-way in-order traversal

A B⁺-tree index of order 3



Definition: A B^+-tree (index) of order **m** is a m-way search tree in which

1. All leaves are on the same level.
2. With the exception for the root, every node has at least $\lfloor (m-1)/2 \rfloor$ and at most m-1 keys, and these keys are sorted in ascending order from left to right. The root can have as few as one key.
3. A node with K keys has K+1 pointers to children on the next level, which corresponds the partition induced on the key space by those K keys.

B+-tree (of order m) blocks

- Index blocks:
  - ➢each block stores a maximum of *m-1* keys and *m* pointers
  - ➢each block stores at least $\lfloor (m-1)/2 \rfloor$ keys and $\lfloor (m-1)/2 \rfloor + 1$ pointers

P_0	K_1	P_1	K_2	…	K_{m-1}	P_{m-1}

  - $K_1 < K_2 < \ldots < K_{m-1}$
- Data blocks: at least $\lfloor d/2 \rfloor$ and at most d records, where d is the maximum number of records that can be accommodated in a node. Records are sorted.
- m and d need **not** be the same.

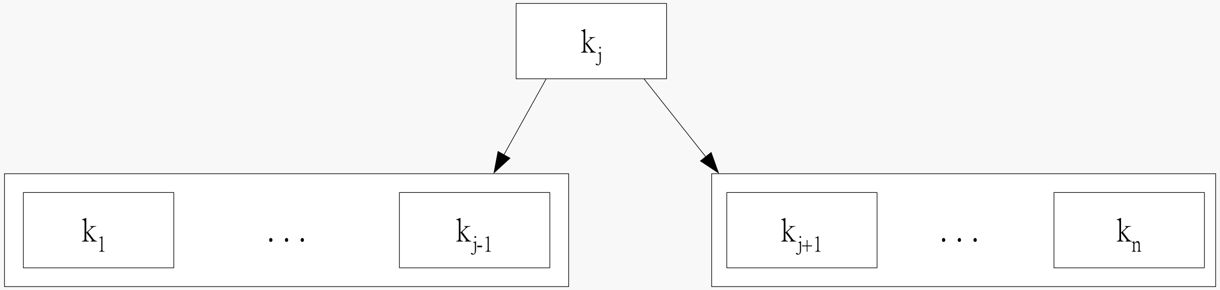
Insertion and Deletion

- An insertion into a node that is not full is quite efficient.
- If a node is full the insertion causes a split into two nodes
- Splitting may propagate to other tree levels
- A deletion is quite efficient if a node does not become less than half full
- Two nodes on the same level are **siblings** if they share a common parent key.
- If a deletion causes a node to become less than half full, it must be redistributed or merged with a sibling

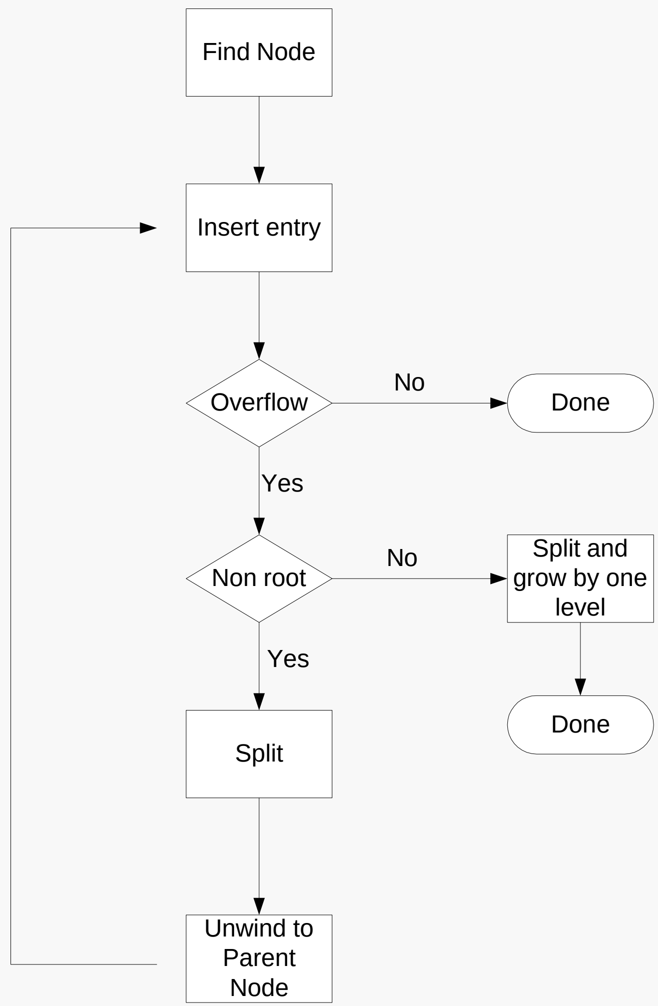
Split

Data Node: Even number of records: distribute evenly between left and right children. Odd number, give one more record to the left. All records are sorted ascendingly from left to right.
Promote the largest key value K_j of the newly created left child to the parent index node.

Index Node: Sort the entries. Even number of entry: the first half go to the left. The next key is promoted and inserted into parent. The remaining keys to the right child. Odd number: The middle key is promoted to the parent, while the remaining two halves are distributed to the left and right children, respectively.

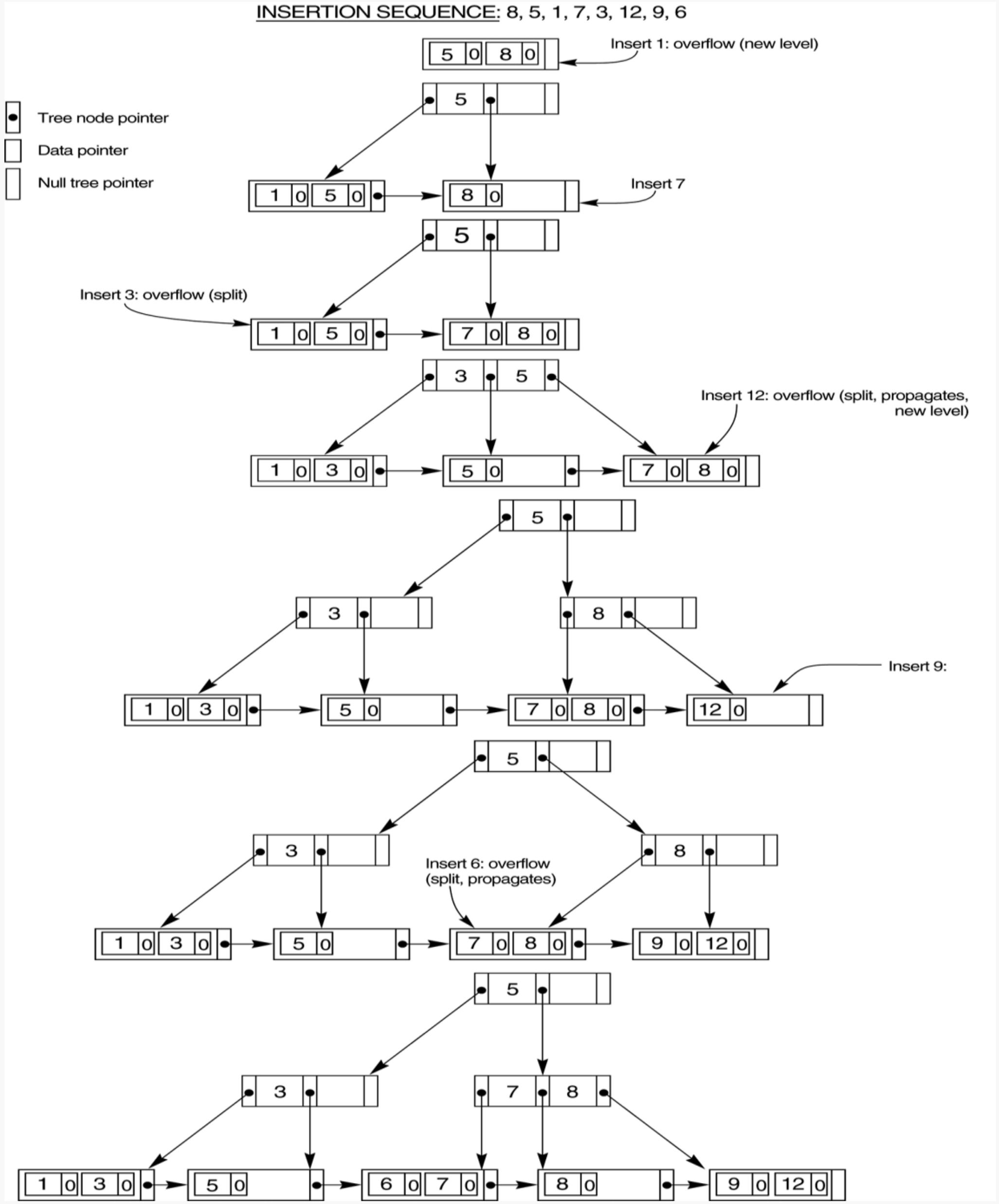


INSERTION



Find Node
Insert entry
Overflow
No
Done
Yes
Non root
No
Split and grow by one level
Done
Yes
Split
Unwind to Parent Node

An example of insertion in a B+-tree index of order 3 and with leaf capacity min=1, max=2.



Redistribution

Data Node: Redistribute the records as evenly as possible between the two nodes (siblings). Update their parent key value accordingly after redistribution.

Index Node: The redistribution must be done via the two index nodes (siblings) common parent key entry. Again redistribute the entries between the silbings as evenly as possible.

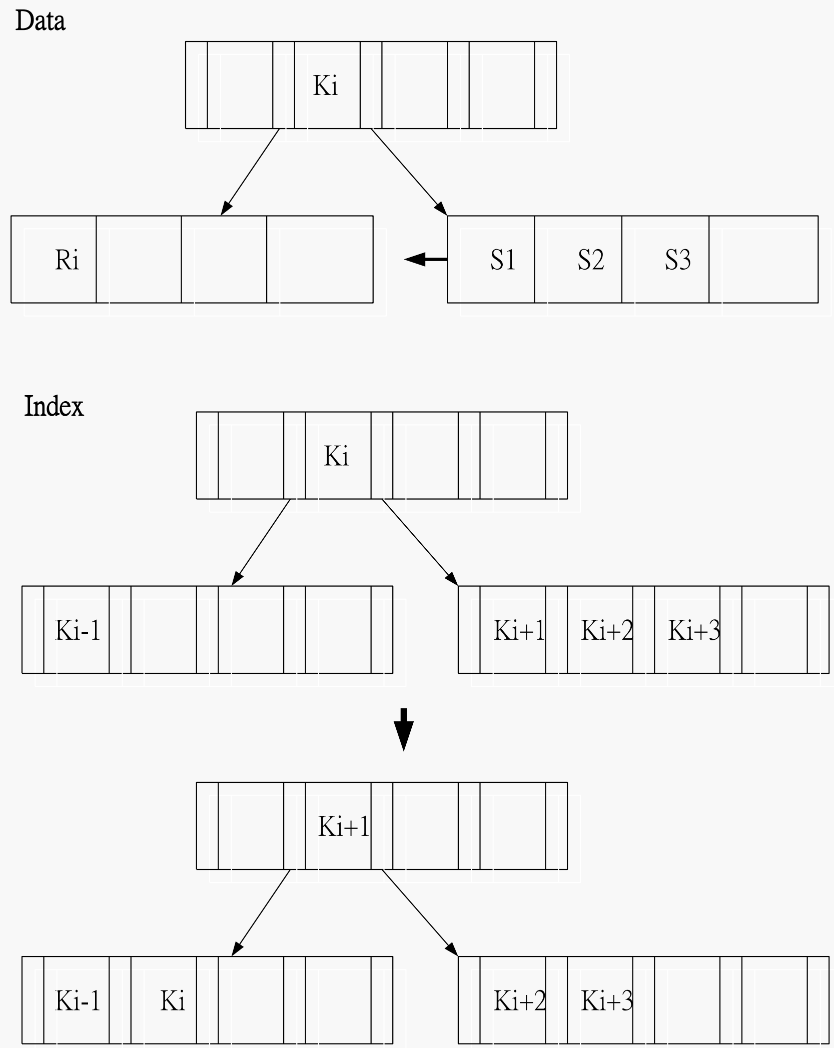
Merge

Data Node:Merge the two data blocks (siblings) into one, with the deletion of parent key recursively.

Index Node: The sibling has the minimum number while the current one has one less than minimum. Merge keys in these two siblings **and** its parent key to form a new index node, then delete merged parent key in the parent node recursively.

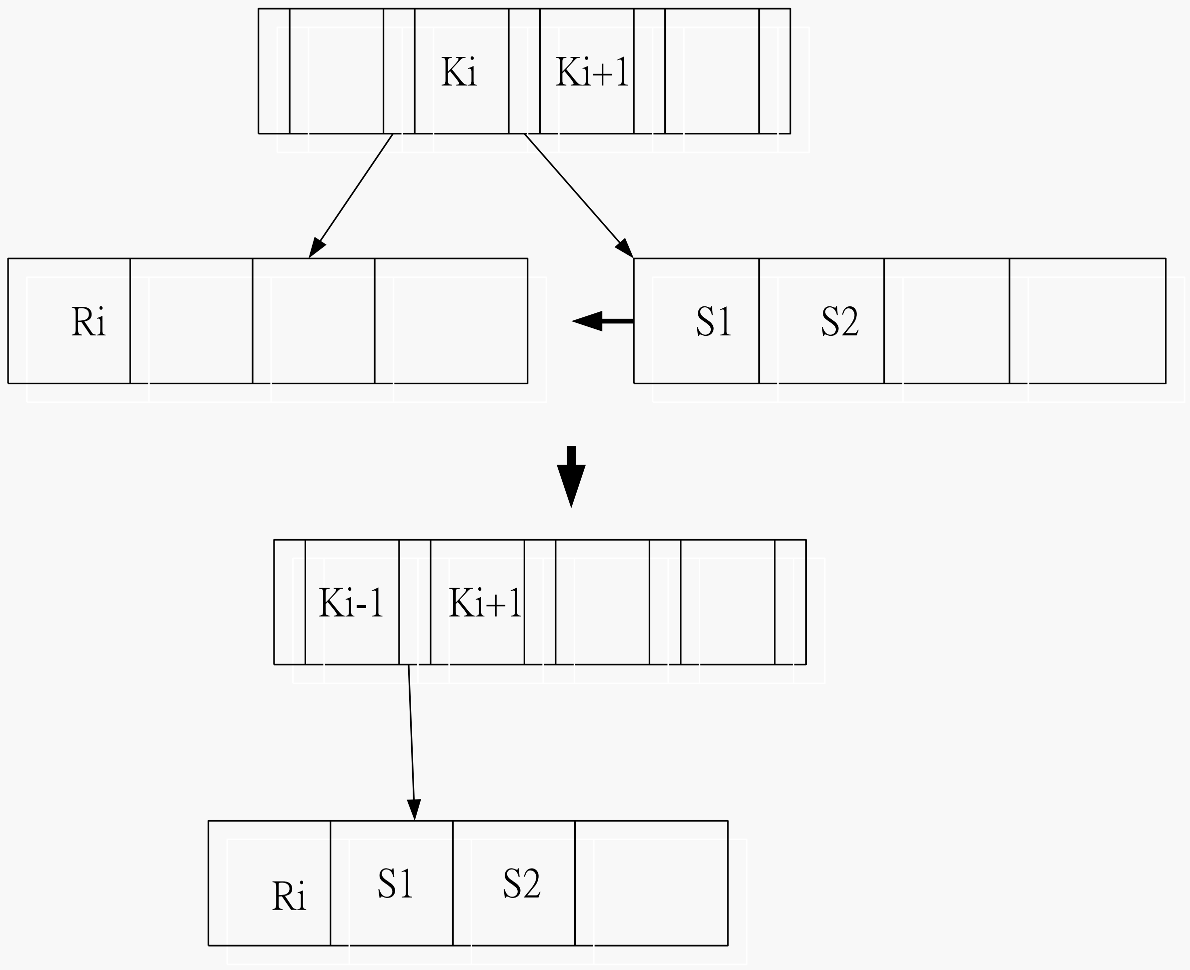
No matter the redistributed or merged nodes are index or data, they must be siblings of some parent key entry.

Redistribution

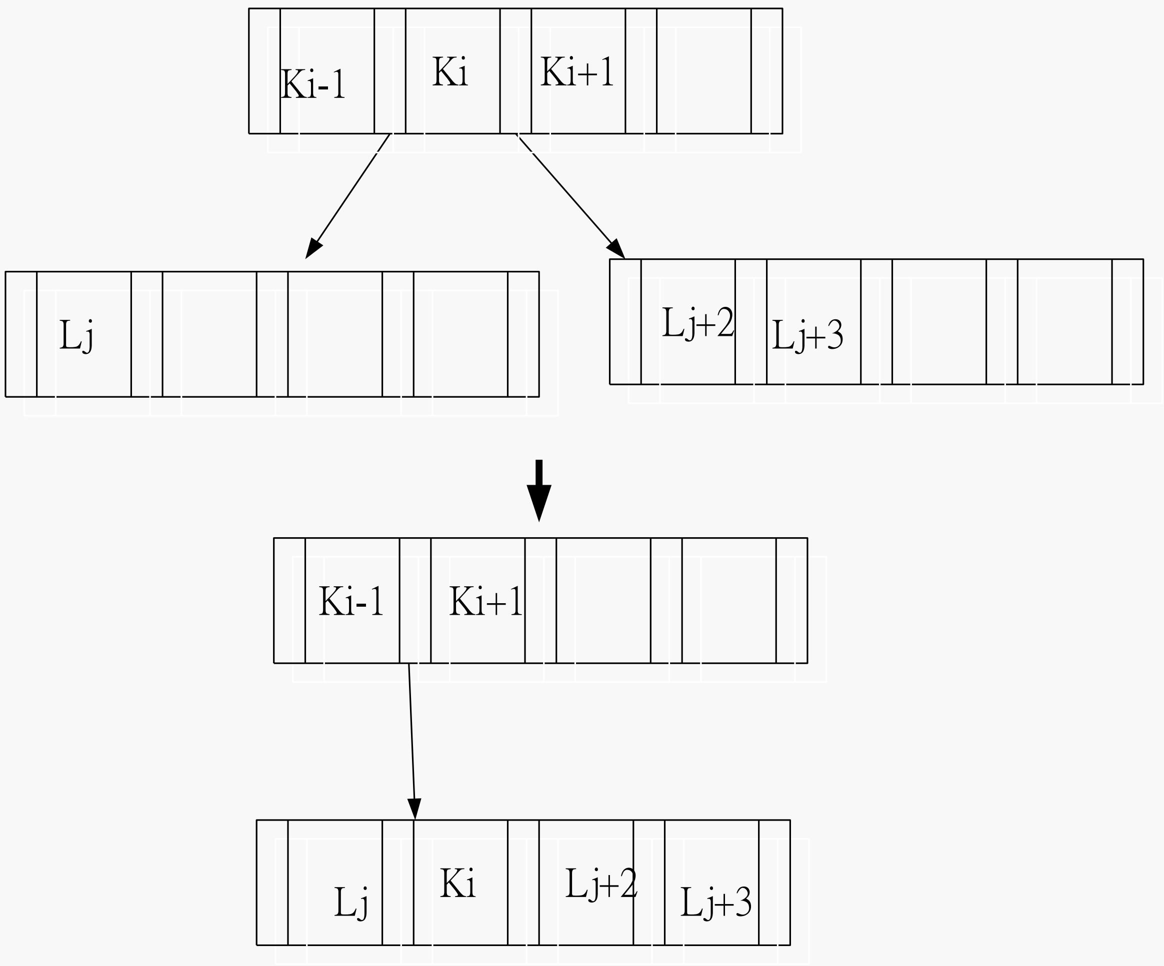


Merging

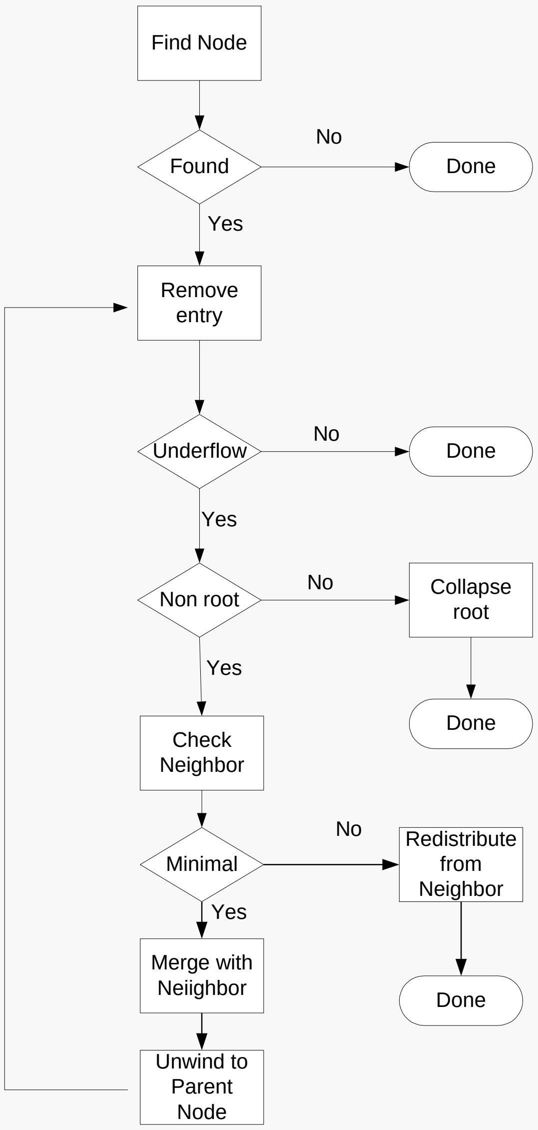
Data



Index



DELETION


Find Node
Found
No
Done
Yes
Remove entry
Underflow
No
Done
Yes
Non root
No
Collapse root
Done
Yes
Check Neighbor
Minimal
No
Redistribute from Neighbor
Yes
Done
Merge with Neiighbor
Unwind to Parent Node

An example of deletion from a B⁺-tree.

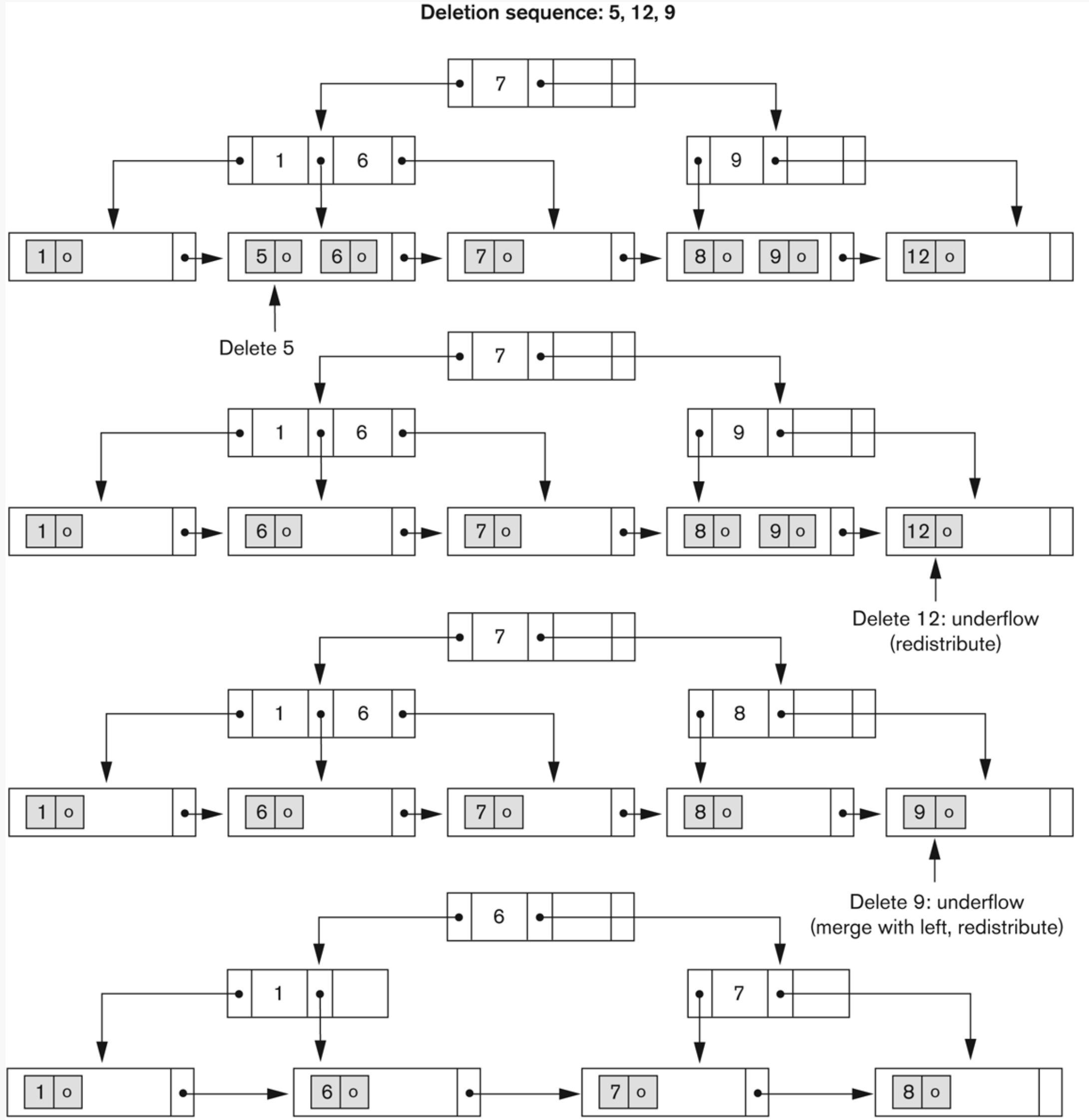
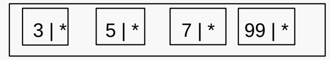


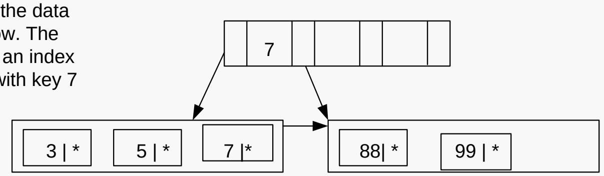
Figure 14.13
An example of deletion from a B⁺-tree.

The min and max number of records or entries in each nodes are 2 and 4, respectively. Let the insertion sequence be 5,3,7,99,88,1,6,10, 25, 98,20,77,2, 4

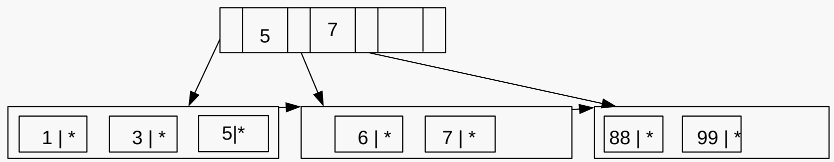
After 5, 3, 7 and 99 are inserted



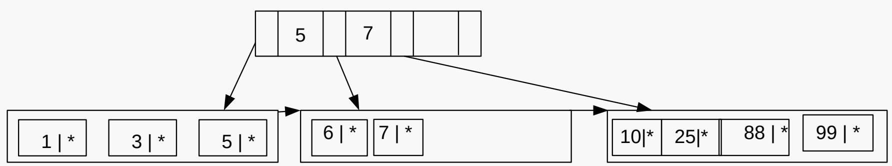
Insert 88 cause the data node to overflow. The node is split and an index node is created with key 7



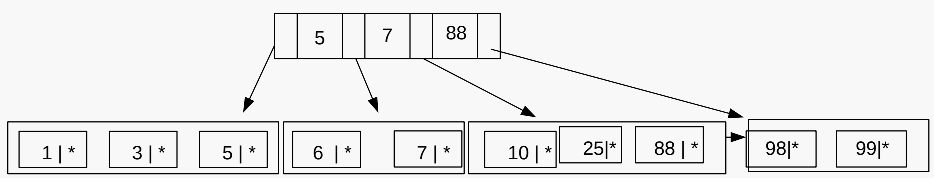
Insert 1, no problem, but insert 6 causes a data node to overflow. A split occurs



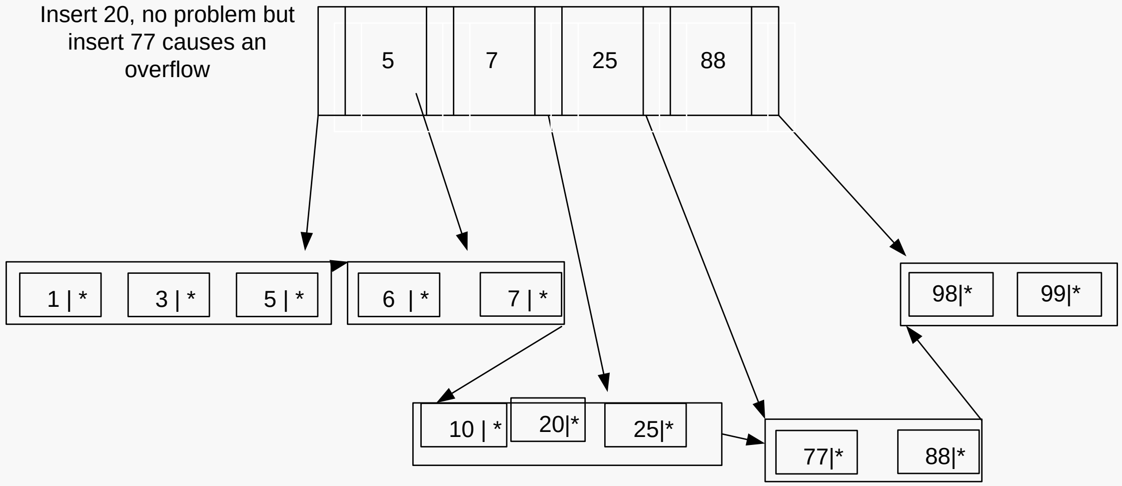
Insert 10 and 25 causes no problem



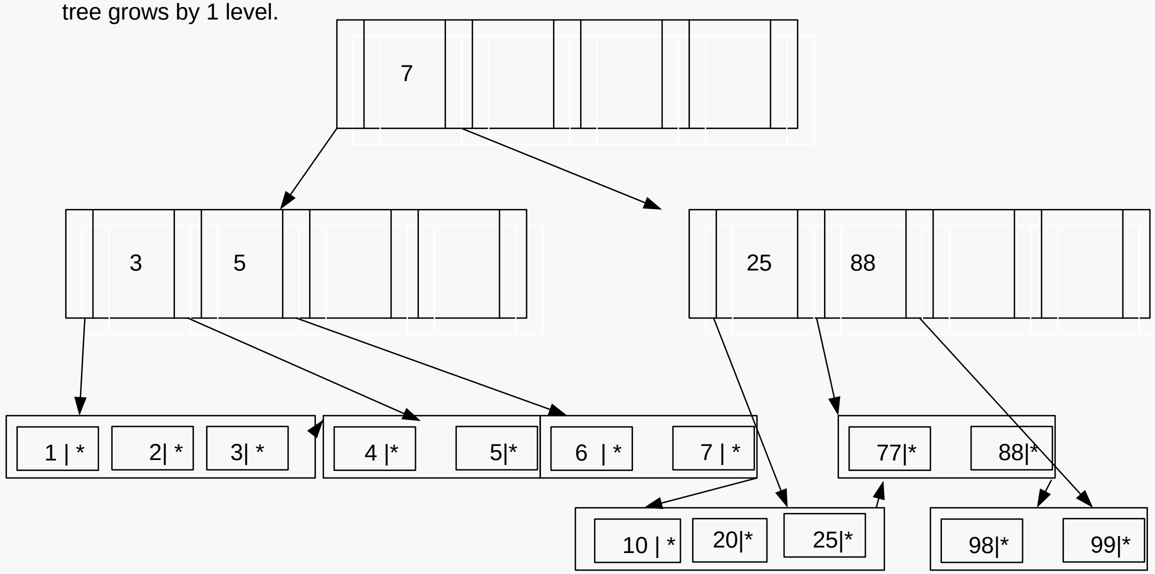
Insert 98, overflow, split node

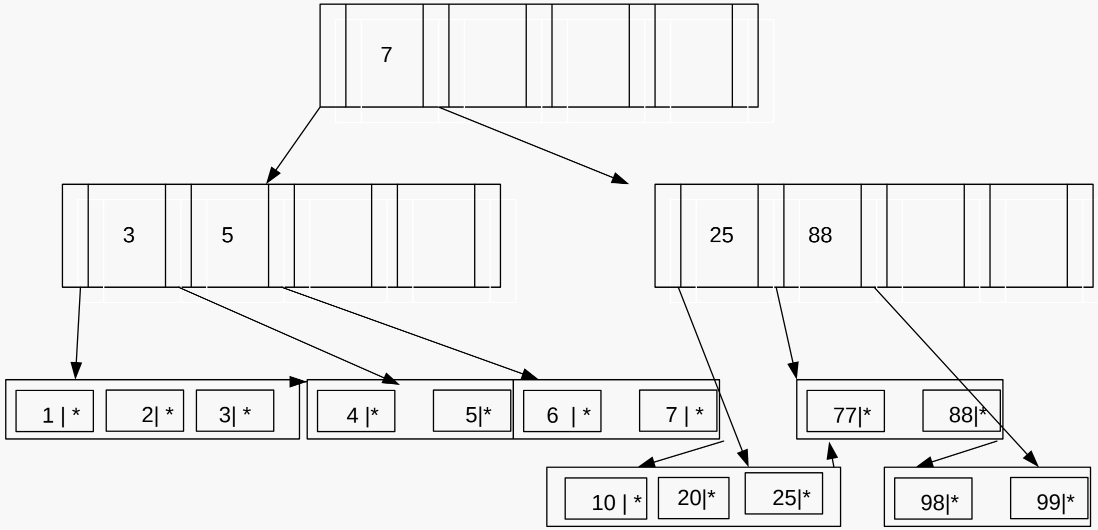


Insert 20, no problem but insert 77 causes an overflow

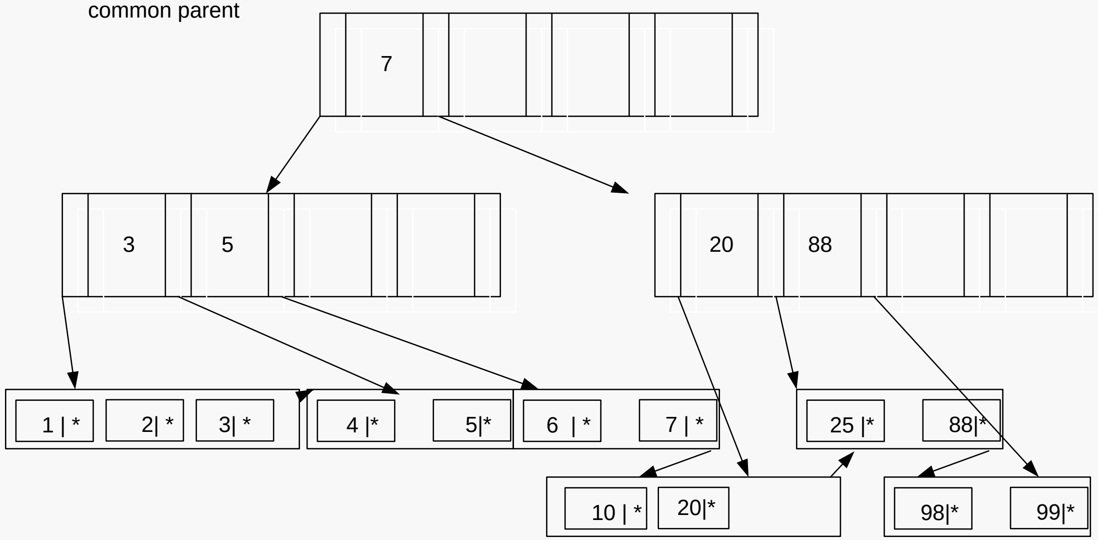


Insert 2, no problem, insert 4, overflow and split. The insertion into parent causes it to split, and the tree grows by 1 level.

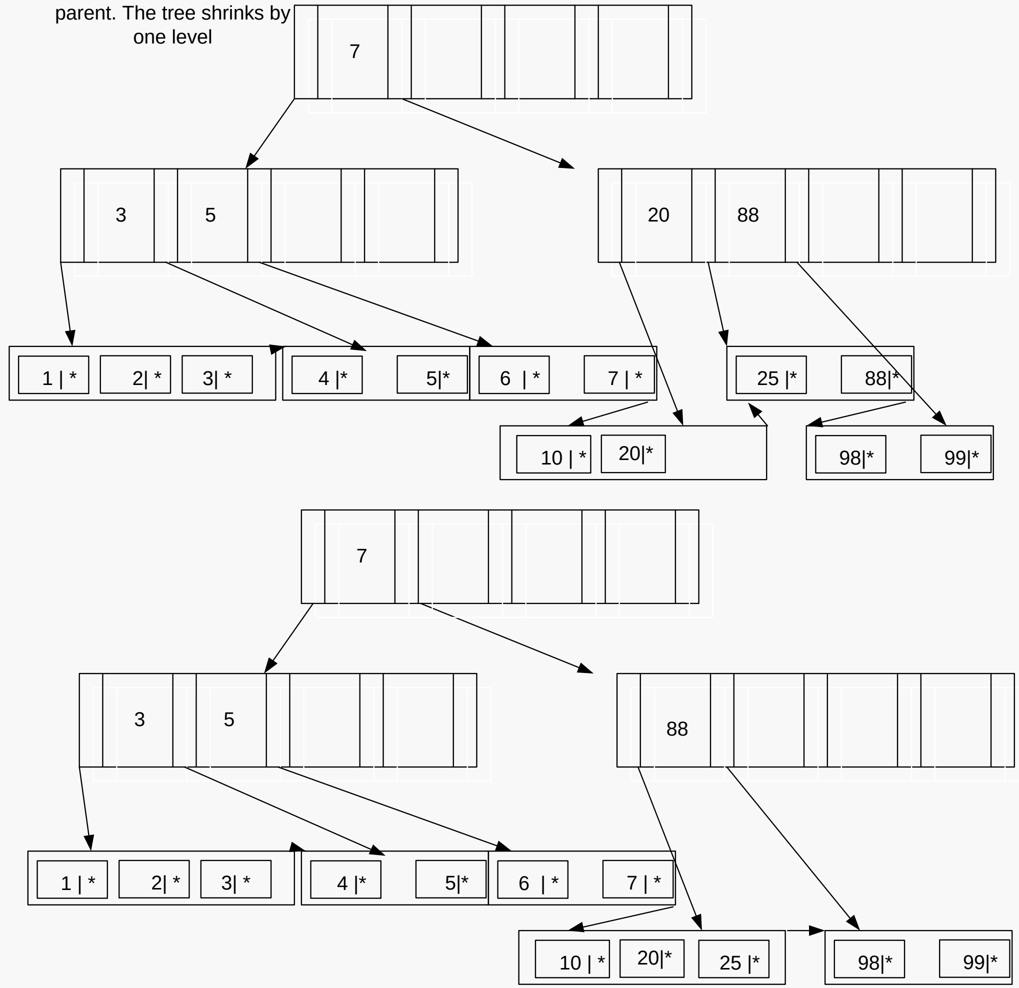


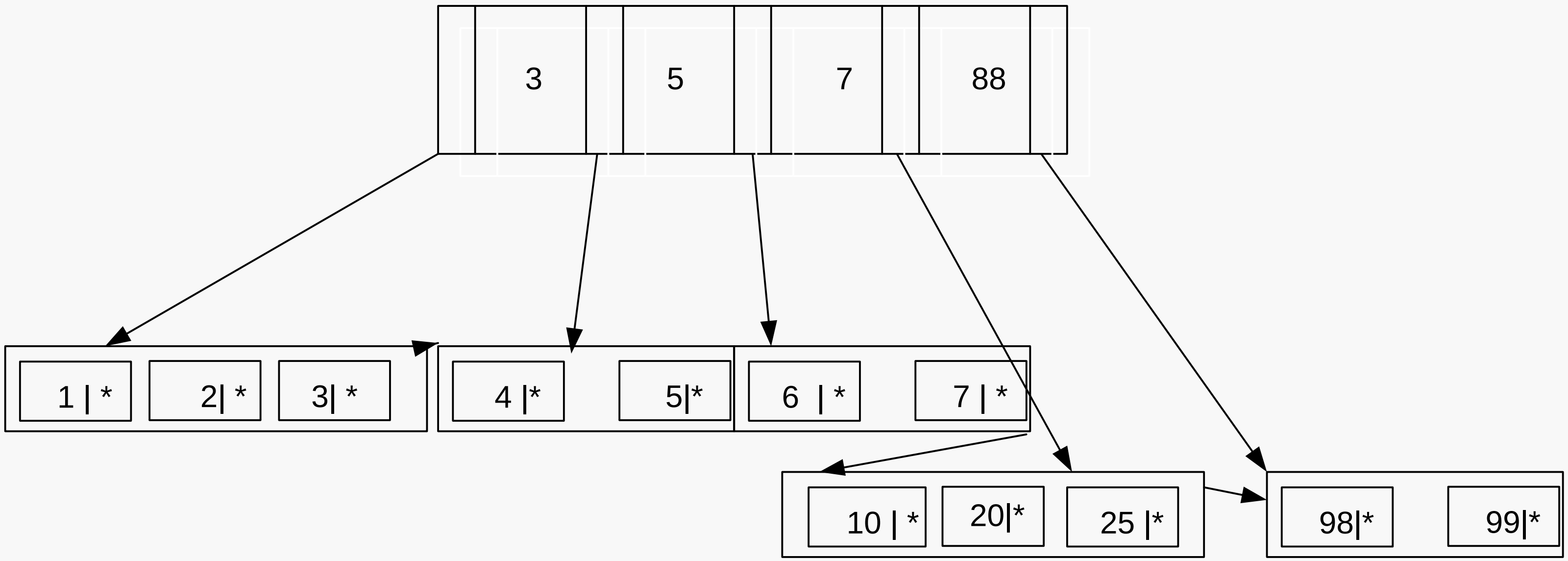


Detete 77, a data node underflows, redistribute from a sibling via the common parent



delete 88, a data node to underflows. Both siblings cannot spare a record, merge with left sibling resulting the deletion of their parent key 20. Redistribute is not possible.A merge with a sibling and their common parent. The tree shrinks by one level




3
5
7
88
1 | *
2| *
3| *
4 |*
5|*
6 | *
7 | *
10 | *
20|*
25 |*
98|*
99|*

Relational Model

Introduction

- Proposed by Ted Codd in late 60's.
- Simplicity & math. elegance.
- Commercial systems:

  IBM DB2.

  INFORMIX.

  ORACLE.

  .

  .

  .

Structure

HOSPITAL(H_CODE,	NAME,	#_OF_BED)
22	Doctor	412
13	Children	846
18	General	987

WARD(H_CODE,	W_CODE,	NAME,	#_OF_BED)
22	1	Recovery	10
13	3	Intensive Care	21
22	6	Psychiatric	30
18	3	Intensive Care	40
13	2	Maternity	24
18	4	Cardiac	32
22	4	Cardiac	20

PATIENT(REG#, NAME, H_CODE, W_CODE, SEX)

REG#	NAME	H_CODE	W_CODE	SEX
33992	Rasky, K.	13	3	M
88288	Neal, M.	18	4	F
22221	Ashby, W.	13	2	M
44777	Miller, A.	13	3	F
13556	Lista, H.	18	3	F
22677	Lee, A.	22	6	F

STAFF(H_CODE, W_CODE, EMP#, NAME, DUTY, SHIFT, SALARY)

H_CODE	W_CODE	EMP#	NAME	DUTY	SHIFT	SALARY
18	3	1009	Bell	Nurse	M	21K
22	6	2200	Scott	Intern	A	33K
18	4	3399	Smith	Intern	E	24K

DOCTOR(H_CODE, DOC#, NAME, SPECIALTY)

H_CODE	DOC#	NAME	SPECIALTY
18	60711	Ashby, W.	Pediatrics
22	58521	Miller, G.	Psychiatry
13	45355	Glass, D.	Neurology
18	76667	Lee, A.	Cardiology
13	39899	Adams, C.	Gynecology
13	66332	Best, K.	Cardiology

ATTENDING_DOCTOR(DOC#, REG#)

DOC#	REG#
45355	33992
76667	88288
66332	22221
39899	44777
66332	13556

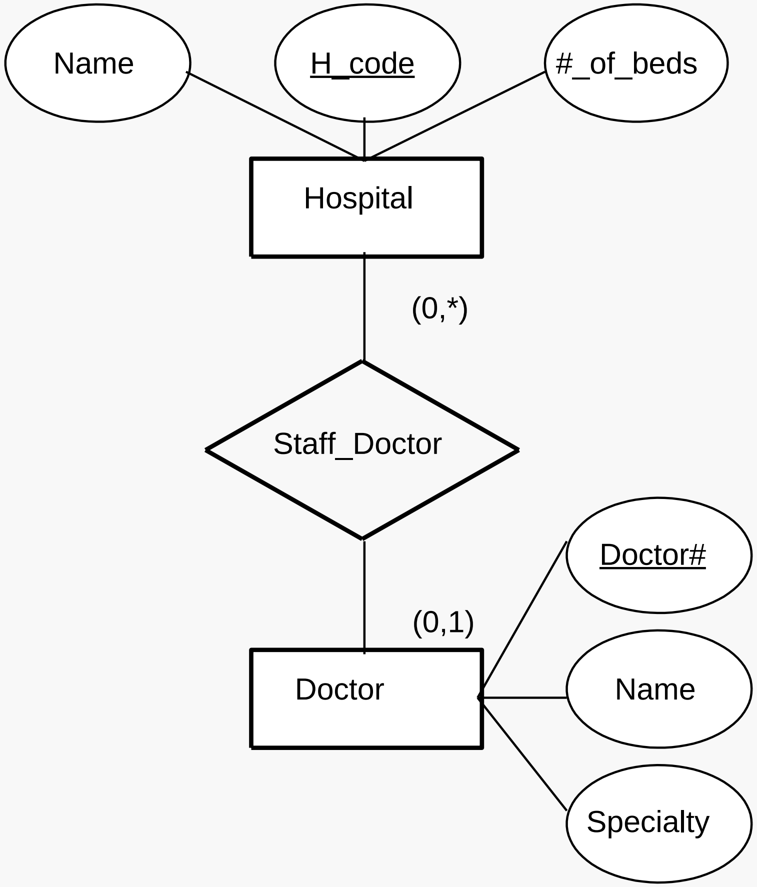
PATIENT_DIAGNOSIS(REG#, DIAGNOSIS_TYPE)

REG#	DIAGNOSIS_TYPE
88288	Cardiac Disease
44777	AIDS
33992	Cancer
13556	Cardiac Disease
22221	Cancer

- One data structuring tool - **relations**.
- $D_1 \times . . . \times D_n$ = $\{<a_1, . . . , a_n> |\ a_i \in D_i, \forall i\}$.
- r is a relation on n sets if r is a subset of $D_1 \times . . . \times D_n$, i.e., r is a set of **tuples** or **rows**.
- D_j is the j^{th} **domain** of r, r is of **degree n** or r is an **n-ary** relation.
- **Tables** are used to represent relations.
- A **relation scheme** - a relation name or a set of attributes.
- An attribute is defined on a domain of the relation. Domains are **atomic**.
- Closed World Assumption (CWA) – not currently known to be true is false; a relation is complete.

- **A relation scheme** defines the **intension** of a relation.
- Intension - a **database schema** or a **relational schema R** = $\{R_1, \ldots, R_k\}$.
- Extension - a database state $<r_1, \ldots, r_k>$.
- Only one tool - both entity and relationship types represented by relations.

Key propagation

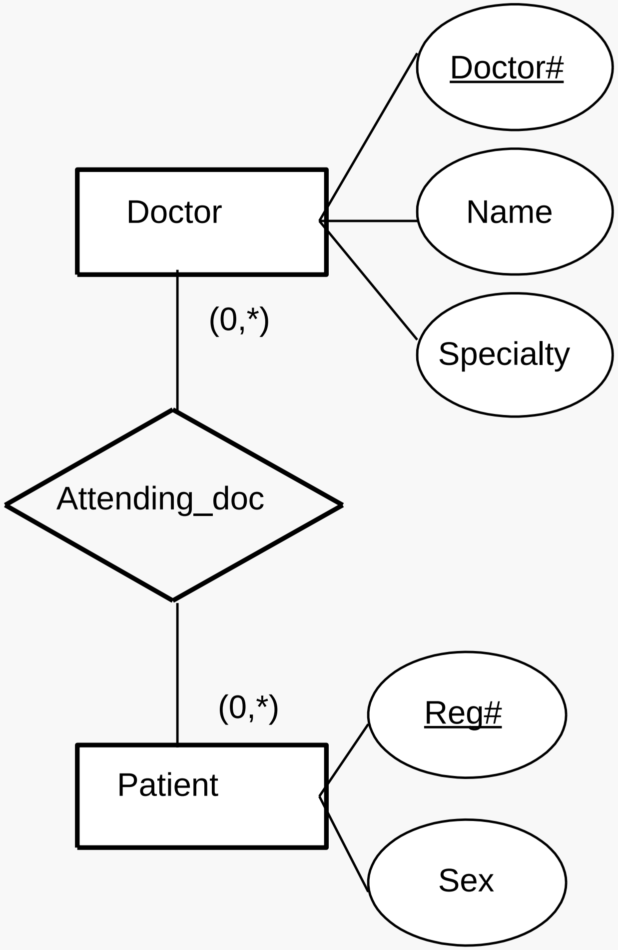


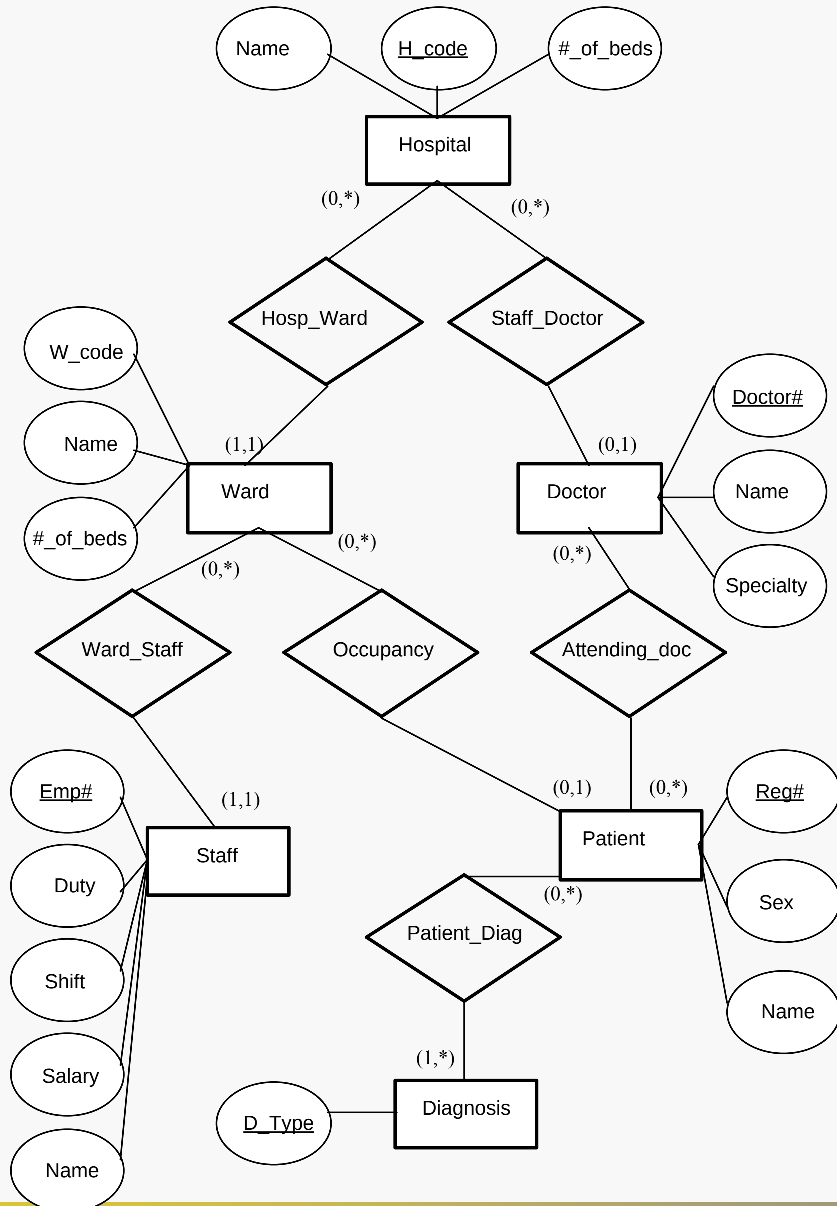
Doctor(<u>Doctor#</u>, Name, Specialty, H_code).

A new relation

- Undesirable to propagate keys for many:many relationships.
- By a separate relation

Attending_doctor(Doctor#, Reg#)




Name
H_code
#_of_beds
Hospital
(0,*)
(0,*)
Hosp_Ward
Staff_Doctor
W_code
Name
#_of_beds
(1,1)
Ward
(0,1)
Doctor
Doctor#
Name
Specialty
(0,*)
(0,*)
(0,*)
Ward_Staff
Occupancy
Attending_doc
Emp#
Duty
Shift
Salary
Name
(1,1)
Staff
(0,1)
(0,*)
Patient
Reg#
Sex
Name
(0,*)
Patient_Diag
(1,*)
D_Type
Diagnosis

Hospital (**H_code**, Name, #_of_beds)
Ward (**H_code, W_code**, Name, #_of_beds)
Doctor(**Doctor#**, Name, Specialty, H_code)
Staff(**Emp#**, Duty, Shift, Salary, Name, H_code, W_code)
Patient (**Reg#**, Sex, Name, H_code, W_code)
Attending_doctor (**Doctor#, Reg#**)
Patient_Diagnosis (**D_type, Reg#**)

Constraints

1. Inherent

- No duplicate rows are permitted.
  - at least one candidate key.
  - a *candidate key* is any minimal subset of attributes in a relation scheme, the values of which uniquely identify tuples.
  - one *primary key* - cannot be updated nor contain nulls (**entity integrity rule**).
- The ordering of the attributes and rows in the table is insignificant.

2. Explicit

- Core SQL-99 does provide
  - **domain** constraints.
  - **primary key** constraints.
  - **foreign key (referential integrity)** constraints.

Domains

- Possible attribute values can be specified
  - Using a CHECK constraint or
  - Creating a new domain
- Domain can be used in schema declarations
- Domain is a schema element

```
CREATE DOMAIN Grades CHAR (1)
   CHECK  (VALUE IN ('A', 'B', 'C', 'D', 'F'))
CREATE TABLE Transcript (
     ….,
     Grade Grades,
     …  )
```

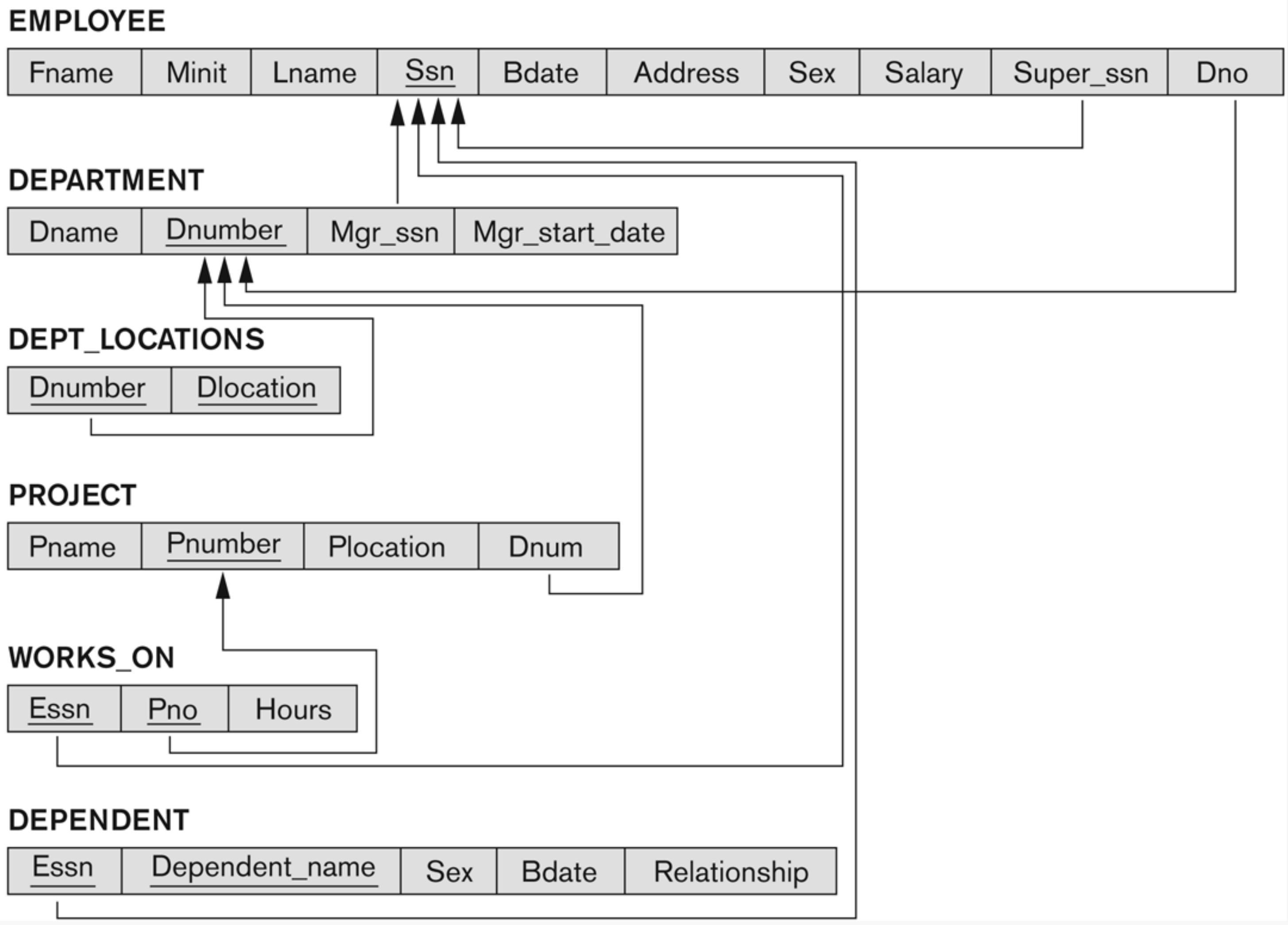
Foreign Key (Referential Integrity) Constraints

- A set of attributes FK in R_1 is a *foreign key* that references R_2 if it satisfies the following two rules:

1. The attributes FK defined on the same domains as the primary key PK of R_2.
2. A value of FK in a tuple t_1 of r_1 (R_1) either occurs as a value of PK for *some* tuple t_2 in $r_2(R_2)$ or is null.

- R1 is called the *referencing* relation while R2 is the *referenced* relation with respect to this foreign key constraint.

Figure 5.7
Referential integrity constraints displayed on the COMPANY relational database schema.



<u>**Tables**</u>

- Basic structure - (base) tables.

<u>**Create tables**</u>

- **Format** : **CREATE TABLE <*table name*> (<*column decl.*>$^+$ [,<*table_constraint*>$^+$]);**
- **<*column decl.*> := <column name> <data type> [DEFAULT <value>]**[<***col_constraint***>$^+$]
- <***col_constraint***>:= {**NOT NULL** | [**CONSTRAINT** name] **UNIQUE** | **PRIMARY KEY** | **CHECK** (*search_cond*) | **REFERENCES** <table> [(column name)] **[ON {UPDATE | DELETE}** <***effect***>]}

- ***table_constraint*:=**
  [CONSTRAINT name]
  {UNIQUE (<column name>$^+$) |
  PRIMARY KEY (<column name>$^+$) |
  FOREIGN KEY (<column name>$^+$)
  REFERENCES <table name> [ON {UPDATE |
  DELETE} <*effect*>]}
- **<*effect*> := SET NULL | NO ACTION (RESTRICT) | CASCADE | SET DEFAULT**

```
CREATE TABLE EMPLOYEE
    ( ...,
      Dno          INT          NOT NULL       DEFAULT 1,
    CONSTRAINT EMPPK
      PRIMARY KEY(Ssn),
    CONSTRAINT EMPSUPERFK
      FOREIGN KEY(Super_ssn) REFERENCES EMPLOYEE(Ssn)
                   ON DELETE SET NULL          ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
      FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                   ON DELETE SET DEFAULT       ON UPDATE CASCADE );
CREATE TABLE DEPARTMENT
    ( ...,
      Mgr_ssn      CHAR(9)      NOT NULL       DEFAULT '888665555',
      ...,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE(Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY(Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                   ON DELETE SET DEFAULT       ON UPDATE CASCADE );
CREATE TABLE DEPT_LOCATIONS
    ( ...,
    PRIMARY KEY(Dnumber, Dlocation),
    FOREIGN KEY(Dnumber) REFERENCES DEPARTMENT(Dnumber)
                   ON DELETE CASCADE           ON UPDATE CASCADE );
```

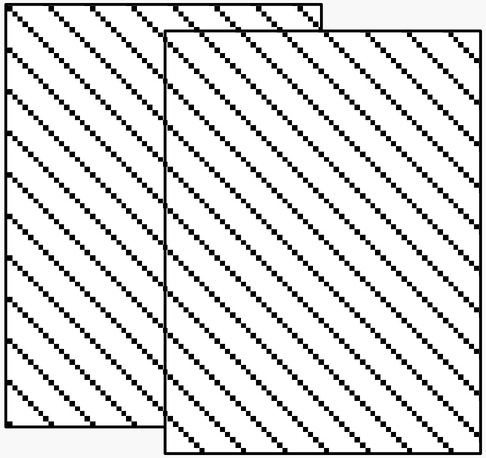
Figure 8.2
Example illustrating how default attribute values and referential integrity triggered actions are specified in SQL.

Relational Algebra

- Each operation takes relations produces another relation.
- Six basic operations: **union (∪)**, **set difference (-)**, **Cartesian product (×)**, **projection (π)**, **selection (σ) and rename (ρ).**
- Relation variables R,S,T,. . . to denote operands in an expression.

Union

- $R \cup S = \{ t \mid t \text{ in is r or t is in s} \}$.
- Same set of attributes.
- Duplicate tuples are eliminated.



Ex:R (A B C)

A	B	C
1	2	x
3	4	x
7	8	x

S(A B C)

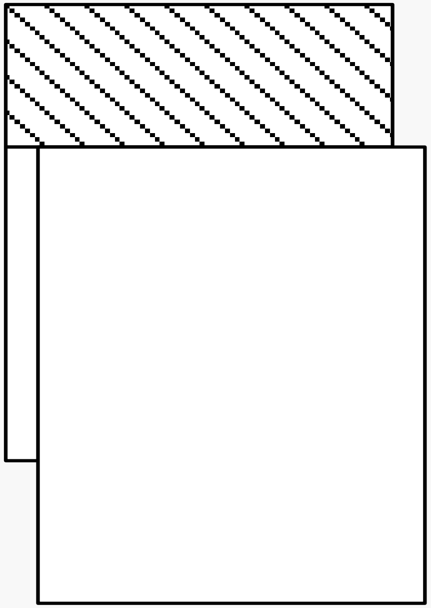
A	B	C
1	2	x
5	6	y
9	9	y

R∪S:

A	B	C
1	2	x
3	4	x
7	8	x
5	6	y
9	9	y

Set Difference

- R-S = {t | t is in r but not in s}.



Ex: R(A B C)

A	B	C
1	2	x
3	4	x
7	8	x

S(A B C)

A	B	C
1	2	x
5	6	y
9	9	y

R-S:

A	B	C
3	4	x
7	8	x

Student

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones

Instructor

FNAME	LNAME
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

Student∪Instructor

FN	LN
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Francis	Johnson
Ricardo	Brown
John	Smith

Instructor - Student

FNAME	LNAME
John	Smith
Ricardo	Browne
Francis	Johnson

Projection

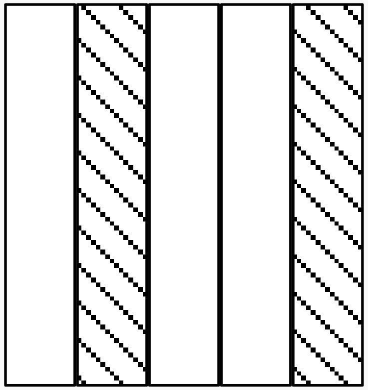
- Remove some columns and/or rearrange some of the remaining columns.
- $R[A_{i1}, \ldots, A_{im}] = \pi_{A_{i1}, \ldots, A_{im}}(R)$
  $= \{t[A_{i1} \ldots A_{im}] \mid t \text{ is in } r\}$.
- A relation defined on $\{A_{i1}, \ldots, A_{im}\}$

Ex: R(A B C)

A	B	C
1	2	x
3	4	x
7	8	x

$\pi_{C,A}(R)$:

C	A
x	1
x	3
x	7



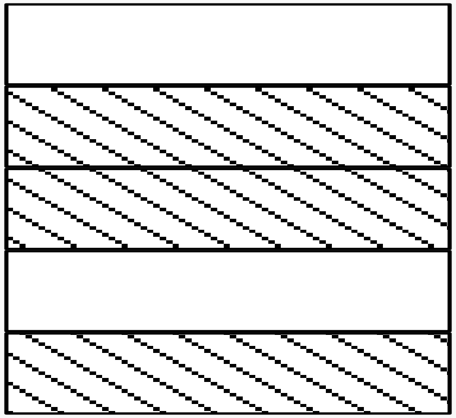
Employee

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

$\pi_{\text{LNAME,FNAME,SALARY}}(\text{EMPLOYEE})$

LNAME	FNAME	SALARY
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

Selection



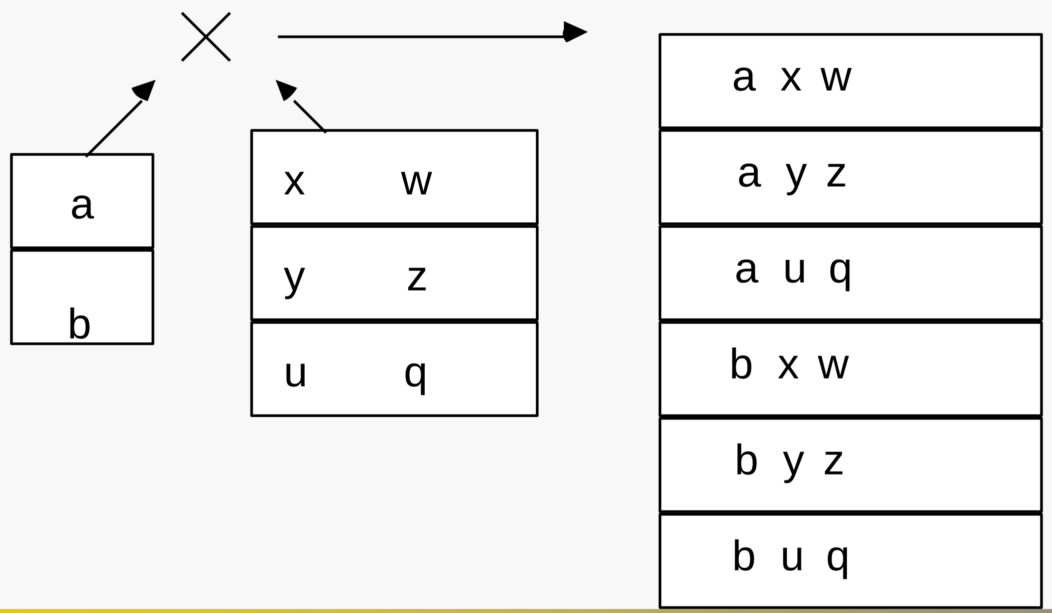
- F is expressed as Boolean combination of conditions. A **condition** can either be "A_i op value" or "A_i op A_j", where op is one of the {<, ≤, >, ≥, =, ≠}. F is connected by the logical operators "&", "|" or "not". Parentheses are also allowed to establish the precedence.
  $\sigma_F(R) = \{ t \mid t \text{ satisfies } F \text{ and } t \text{ is in } r\}$.
- A tuple t is said to **satisfy** F if substituting the corresponding values of t into F, F is evaluated to true.

$\sigma_{\text{(DNO=4 AND SALARY>25000) OR (DNO=5 AND SALARY>30000)}}$ (EMPLOYEE)

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

Cartesian Product

- Degree(R) = k1, degree(S) = k2.
- A relation defined on $\{R.A_1, \ldots, R.A_{k1}, S.B_1, \ldots, S.B_{k2}\}$.
- $R \times S = \{ t \mid t[R.A_1, \ldots, R.A_{k1}] \text{ in } r \ \& \ t[S.B_1, \ldots, S.B_{k2}] \text{ in } s\}$.
- $|r| \times |s|$ tuples.



Ex :

R(A B C)

A	B	C
a1	b1	c1
a2	b2	c2
a3	b3	c1

S(C D E)

C	D	E
c1	d1	e1
c2	d2	e2
c2	d3	e3

$R \times S$ =

R.A	R.B	R.C	S.C	S.D	S.E
a1	b1	c1	c1	d1	e1
a1	b1	c1	c2	d2	e2
a1	b1	c1	c2	d3	e3
a2	b2	c2	c1	d1	e1
.	.	.	.	.	.
.	.	.	.	.	.

Rename

- $\rho_{\text{new-name}}(R)$: rename the operand R to the new-name.
- $\rho_{S(A1,\ldots,An)}(R)$: R has n attributes B1, . . . , Bn and the result is renamed to a relation S with attributes A1 , . . ., An.

- Any variable is a relational algebra expression.
  Let E1 and E2 be relational algebra expressions, then
  (E1$\cup$ E2), (E1- E2), (E1$\times$ E2),
  σ_F(E1), π_S(E1), ρ_S(E1)
  are all relational algebra expressions.
  Finite number of applications of the above rules.
- A language that provides at least the retrieval power of relational algebra is said to be **relationally complete**.

θ-join

- $R \underset{A\theta B}{*} S := \sigma_{A\theta B}(R \times S)$,

Ex: R:

A	B	C
1	2	3
4	5	6
7	8	9

S:

D	E
3	1
6	2

$R \underset{B<D}{*} S$:

A	B	C	D	E
1	2	3	3	1
1	2	3	6	2
~~4~~	~~5~~	~~6~~	~~3~~	~~1~~
4	5	6	6	2
~~7~~	~~8~~	~~9~~	~~3~~	~~1~~
~~7~~	~~8~~	~~9~~	~~6~~	~~2~~

(Natural) Join

- Columns are named by attributes.
- A relation defined on Attr(R)∪Attr(S).
- R*S = {t | t is a tuple on Attr(R)∪Attr(S) & there exist u in r and v in s such that t[R] = u and t[S] = v}.

Ex :

R (A B C)

A	B	C
a1	b1	c1
a2	b2	c2
a3	b3	c1

S (C D E)

C	D	E
c1	d1	e1
c2	d2	e2
c2	d3	e3

R*S:

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b2	c2	d2	e2
a2	b2	c2	d3	e3
a3	b3	c1	d1	e1

PNAME	PNUMBER	PLOCATION	DNUM
ProductX	1	Bellaire	5
ProductY	2	Suqarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

Project

ESSN	PNO	HOURS
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	null

Works_on

Examples

Project * Works_on := $\sigma_{\text{Pnumber=Pno}}$ (Project×Works_on)

Pnumber=Pno

PNAME	PNUMBER	PLOCATION	DNU	ESSN	PNO	HOURS
ProductX	1	Bellaire	5	123456789	1	32.5
ProductX	1	Bellaire	5	453453453	1	20.0
.	.	.	.	.	.	.

Project * $\rho_{\text{Works_on(ESSN,PNUMBER,HOURS)}}$Works_on

PNAME	PNUMBER	PLOCATION	DNU	ESSN	HOURS
ProductX	1	Bellaire	5	123456789	32.5
ProductX	1	Bellaire	5	453453453	20.0
.	.	.	.	.	.

Intersection

- $R \cap S = \{ t \mid t \in r \;\&\; t \in s \}$.

Ex:

R:

A	B	C
a	b	c
d	a	f
c	b	d

S:

A	B	C
b	g	a
d	a	f

$R \cap S = \{<d, a, f>\}$.

Hospital (**H_code**, Name, #_of_beds)
Ward (**H_code, W_code**, Name, #_of_beds)
Doctor(**Doctor#**, Name, Specialty, H_code)
Staff(**Emp#**, Duty, Shift, Salary, Name, H_code, W_code)
Patient (**Reg#**, Sex, Name, H_code, W_code)
Attending_doctor (**Doctor#, Reg#**)
Patient_Diagnosis (**D_type, Reg#**)

Q1:Find the names of all doctors whose specialty is gynecology.

π_{NAME} ($\sigma_{SPECIALTY='Gynecology'}$ (DOCTOR)).

Q2:Find the names and salaries of all interns working in the evening shift.

$\pi_{NAME,SALARY}$ (σ $_{DUTY='Intern'\&SHIFT='E'}$(STAFF)).

Q3:List the names of those doctors whose specialty is cardiology, who are female and who are also patients. Assuming NAMEs are used to denote objects.

(π_{NAME} (σ $_{SEX='F'}$ (PATIENT))) $\cap$
(π_{NAME} (σ $_{SPECIALTY='Cardiology'}$ (DOCTOR))).

Q4:Find all hospital codes which have a doctor specialized in heart but has no cardiac ward.

$H1 = \pi_{H\_CODE} (\sigma_{SPECIALTY='Cardiology'} (DOCTOR))$.

$H2 = \pi_{H\_CODE} (\sigma_{NAME='Cardiac'} (WARD))$.

H1-H2.

Q5:Find all hospital names which have a doctor treating patients with AIDS.

DOCTORS_TREATING_AIDS_PATIENT(DOCTOR#)=

$\pi_{DOCTOR\#}$

$((\sigma_{D\_TYPE='AIDS'}(PATIENT\_DIAGNOSIS) * ATTENDING\_DOCTOR))$

π_{NAME} (HOSPITAL* (DOCTORS_TREATING_AIDS_PATIENT* $\pi_{DOCTOR\#,H\_CODE}$ (DOCTOR)))

Q6:Find doctor# whose specialty is cardiology and treats only patients with heart disease. A heart disease patient has exactly one diagnosis with type cardiac disease.

HD(DOCTOR#)= $\pi_{\text{DOCTOR\#}}$ ($\sigma_{\text{SPECIALTY='Cardiology'}}$ (ATTENDING_DOCTOR* DOCTOR))

NON_HEART_DISEASE_PATIENT(REG#) =
($\pi_{\text{REG\#}}$ (ATTENDING_DOCTOR) - $\pi_{\text{REG\#}}$ (PATIENT_DIAGNOSIS))
$\cup$
($\pi_{\text{REG\#}}$ ($\sigma_{\text{D-TYPE} \neq \text{'Cardiac Disease'}}$ (PATIENT_DIAGNOSIS)))

HD_WITH_SOME_NON_HEART_DISEASE_PATIENT(DOCTOR#)=
$\pi_{\text{DOCTOR\#}}$
(HD*ATTENDING_DOCTOR*NON_HEART_DISEASE_PATIENT)

HD - HD_WITH_SOME_NON_HEART_DISEASE_PATIENT.

Core SQL-99

- System R
  - between 1974 -79.
  - prototype.
  - database recovery management.
  - automatic concurrency control.
  - flexible authorization mechanism.
  - dynamic database definition.
- Via **SQL (Structured Query Language)**, SEQUEL or SQUARE.
- ANSI & ISO adopted SQL as the standard relational d.b. language : SQL-89, SQL92 (SQL2), SQL-99, SQL3…
- Commands can be invoked either interactively or via an application program.

Data Structures

Data Types

- Number: INTEGER, SMALLINT, NUMERIC, DECIMAL, REAL, DOUBLE, FLOAT …
- Character String: CHAR, VARCHAR…
- Bit: BIT, VARBINARY…
- Date, Time.

Table

- Just a relation except duplicates are allowed, but can be a relation.
- Interactive SQL (ISQL) - table-at-a-time operations.
- Commands embedded in application programs can be tuple-at-a-time operation.

Indexes

- If indexed, then join and selection can be performed much faster.
- But take space and time.

Null Value

- Nulls allowed.
- "Value unknown" or "value inapplicable."

Problems of Nulls

1. Keys

- Can't have null in the primary key. E.g., SIN in EMP, if nulls are allowed, then two rows are identical.

2. Evaluation of Conditions

- Ambiguity can occur.

Ex:

EMP#	NAME	AGE	SPOUSE
111	Smith	30	Nancy
222	Jones		

SELECT NAME FROM EMP WHERE SPOUSE = 'NANCY'.

- Jones' tuple evaluated to ?.
- SQL selects rows in which the result is evaluated to **true**.

Three-valued Logic

AND	T	?	F
T	T	?	F
?	?	?	F
F	F	F	F

OR	T	?	F
T	T	T	T
?	T	?	?
F	T	?	F

NOT	
T	F
?	?
F	T

Core SQL-99

- As a stand-alone interactive language or invoked from a host language like C, PL/1, COBOL or Assembler.
- Complete language - DDL & DML.
- Control, retrieval and modification.

Control (DDL)

- Relevant to application programmers.
- Three classes:

  TABLE: CREATE, ALTER, DROP.

  INDEX: CREATE, DROP.

  VIEW: CREATE, DROP (discussed later).
- Access control: GRANT and REVOKE.

Tables

- Basic structure - (base) tables.

Create tables

- **Format** : **CREATE TABLE <*table name*> (<*column decl.*>$^{+}$ [,<*table_constraint*>$^{+}$]);**
- **<*column decl.*> := <column name> <data type> [DEFAULT <value>]**[<***col_constraint***>$^{+}$]
- <***col_constraint***>:= {**NOT NULL** | [**CONSTRAINT** name]
  UNIQUE |
  PRIMARY KEY |
  CHECK (*search_cond*) | **REFERENCES** <table> [(column name)] **[ON {UPDATE | DELETE}** <***effect***>]}

- ***table_constraint*:=**
  [CONSTRAINT name]
  {UNIQUE (<column name>$^+$) |
  PRIMARY KEY (<column name>$^+$) |
  FOREIGN KEY (<column name>$^+$)
  REFERENCES <table name> [ON {UPDATE |
  DELETE} <*effect*>]
- **<*effect*> := SET NULL | NO ACTION (RESTRICT) | CASCADE | SET DEFAULT**

Ex:CREATE TABLE SUPPLIER

```
(S#   CHAR(5) NOT NULL, SNAME  CHAR(10),
STATUS SMALLINT, CITY VARCHAR(30),
CONSTRAINT PP PRIMARY KEY (S#));
```

- Create a new empty table.
- Data can be entered.

Ex: CREATE TABLE SP

```
(S#   CHAR(5) NOT NULL DEFAULT 'IBM',
P#    CHAR(6) NOT NULL, QTY INTEGER,
CONSTRAINT QTY_MAX CHECK(QTY<=1500)),
CONSTRAINT SPP PRIMARY KEY (S#, P#),
CONSTRAINT SFK FOREIGN KEY (S#) REFERENCES
SUPPLIER ON DELETE SET DEFAULT,
CONSTRAINT PFK FOREIGN KEY (P#) REFERENCES
PART ON DELETE CASCADE);
```

<u>Alter tables</u>

- **Format 1** : **ALTER TABLE <table name> ADD {<*column decl.*> |<*primary-key-def*>|<*foreign-key-def*>};**
- **Format 2** : **ALTER TABLE <table name> DROP {COLUMN <col name> | PRIMARY KEY | <foreign key name>};**

Ex: ALTER TABLE SUPPLIER ADD DISCOUNT SMALLINT;

- All existing records are expanded with nulls, but not physically changed.

Drop tables

- Can be dropped any time.
- Eliminate definition and data.
- All indexes are dropped.
- **Format** : **DROP [TABLE t | VIEW v] {CASCADE | RESTRICT};**
- *Cascade* –foreign key constraints are also dropped.
- Core SQL-99 just has the default **RESTRICT**.

Ex: DROP TABLE SUPPLIER;

Index

- Improve search performance, but with cost.
- Table could have zero or more indexes. An index is defined on exactly one table.
- Users never reference to indexes.
- Automatically maintained by system.
- **Format: CREATE [UNIQUE] INDEX <index name> ON <table name> (<column name> [<*order*>] [, <column name> [<order>]] ...);**
  <*order*> := ASC | DESC.

Ex: CREATE UNIQUE INDEX SNUM ON SP (S#);

Drop indexes

- **Format: DROP INDEX <index name>;**

Ex: DROP INDEX SNUM;

<u>Retrieval</u>

- SELECT - items desired.
- FROM - tables (context).
- WHERE - a <*predicate*>.
- <*predicate*> := conditions connected by parentheses and Boolean operators AND, OR and NOT.
- **Simpler Format: SELECT [DISTINCT] <items>**
  FROM <tables>
  [WHERE <predicate>]
  [GROUP BY <attrs> [HAVING <*predicate*>]]
  [ORDER BY <attrs>];

SUPPLIER(S#, SNAME, STATUS, CITY)

S1	DOW	20	N.Y.
S2	IBM	10	Paris
S3	Texaco	30	L.A.
S4	Shell	50	Paris

Q: Find S# and STATUS for suppliers in Paris.

```
SELECT 'SUPPLIER# =', S#, STATUS
FROM SUPPLIER
WHERE CITY='Paris';
```

Result:

SUPPLIER# =	S2	10
SUPPLIER# =	S4	50

SP (S#, P#, QTY)

S1	P1	10
S2	P1	20
S1	P2	30
S3	P2	100

Q: Find P# that are supplied by someone.

SELECT DISTINCT P#
FROM SP;

- First find the qualified P#, then DISTINCT applied.
  Result: P1
  P2

SUPPLIER(S#,	**SNAME,**	**STATUS,**	**CITY)**
S1	DOW	20	N.Y.
S2	IBM	10	Paris
S3	Texaco	30	L.A.
S4	Shell	50	Paris

Q: Get all suppliers.

```
SELECT *
FROM SUPPLIER;
```

Q: Find S# for suppliers in Paris with status ≠ 20.

```
SELECT S#
FROM SUPPLIER
WHERE CITY = 'Paris' AND
STATUS <> '20';
```

SUPPLIER(**<u>S#</u>,**	**SNAME,**	**STATUS,**	**CITY)**
S1	DOW	20	N.Y.
S2	IBM	10	Paris
S3	Texaco	30	L.A.
S4	Shell	50	Paris

Q: Find all suppliers whose names begin with the letter T.

SELECT * FROM SUPPLIER
WHERE SNAME LIKE 'T%';

Result: S3 Texaco 30 L.A.

- '%' - any sequence of n characters, where $n \geq 0$.
- '_' - any single character.

Q: Find all suppliers whose names do not contain the letter W.

SELECT * FROM SUPPLIER
WHERE SNAME NOT LIKE '%W%';

SUPPLIER(S#,	**SNAME,**	**STATUS,**	**CITY)**
S1	DOW	20	N.Y.
S2	IBM	10	Paris
S3	Texaco	30	L.A.
S4	Shell	50	Paris

Q: Get S# and STATUS for suppliers in Paris and SNAME is not null, in descending order of STATUS.

```
SELECT S#, STATUS AS ST
FROM SUPPLIER
WHERE CITY = 'Paris' AND SNAME IS NOT NULL
ORDER BY STATUS DESC;
```

- Apply S-F-W, then ORDER.

SUPPLIER(S#, SNAME, STATUS, CITY)
PART(P#, PNAME, COLOR, WEIGHT)
SP(S#, P#, QTY)
Q: Get P# for parts that weight more than 18 pounds but not supplied by S2.

```
SELECT P#
FROM PART
WHERE WEIGHT > 18
EXCEPT
SELECT P#
FROM SP
WHERE S# = 'S2';
```

- UNION, INTERSECT, EXCEPT.
- Optionally with ALL - retain duplicates in the result.

SUPPLIER(<u>S#</u>, SNAME, STATUS, CITY)
PART(<u>P#</u>, PNAME, COLOR, WEIGHT)
SP(<u>S#, P#</u>, QTY)

Q: For each part supplied, find P# and supplier's location.

```
SELECT DISTINCT P#, CITY
FROM SP, SUPPLIER
WHERE SP.S# = SUPPLIER.S#;
```

- Context is SP × SUPPLIER. Equivalently, SP and SUPPLIER are assigned with a corresponding row.
- Also known as INNER JOIN
- SELECT DISTINCT P#, CITY
  FROM SP NATURAL JOIN SUPPLIER (or INNER JOIN ON SP.S#=SUPPLIER.S#))

SUPPLIER(S#, SNAME, STATUS, CITY)

S#	SNAME	STATUS	CITY
S1	DOW	20	N.Y.
S2	IBM	10	Paris
S3	Texaco	30	L.A.
S4	Shell	50	Paris

SP(S#, P#, QTY)

S#	P#	QTY
S1	P1	10
S2	P2	30
S3	P1	20
S5	P1	20

Q: Find suppliers and their parts supplied, including those who do not supply any part.

SELECT * FROM SUPPLIER LEFT OUTER JOIN SP
ON SP.S# = SUPPLIER.S#;

S#,	SNAME,	STATUS,	CITY,	S#	P#,	QTY
S1	DOW	20	N.Y.	S1	P1	10
S2	IBM	10	Paris	S2	P2	30
S3	Texaco	30	L.A.	S3	P1	20
S4	Shell	50	Paris			

- Left, right or full outer join

SUPPLIER(S#, SNAME, STATUS, CITY)
PART(P#, PNAME, COLOR, WEIGHT)
SP(S#, P#, QTY)

Q: List SNAME located in London or Paris.

```
SELECT SNAME  FROM SUPPLIER
WHERE CITY IN ('London', 'Paris');
```

- Result of S-F-W is a set (table) ⇒ can be used in another WHERE clause.

Q: Find S# for suppliers who supply at least one red part.

```
SELECT DISTINCT S#
FROM SP WHERE P# IN
              (SELECT P# FROM PART
               WHERE COLOR = 'Red');
```

- **Subquery**.

SUPPLIER(S#, SNAME, STATUS, CITY) SP(S#, P#, QTY)

S#	SNAME	STATUS	CITY	S#	P#	QTY
S1	DOW	20	N.Y.	S1	P1	10
S2	IBM	10	Paris	S1	P2	20
S3	Texaco	30	L.A.	S2	P2	30
S4	Shell	50	Paris	S3	P1	20

Q: Find SNAME for suppliers who do not supply P1.

```
SELECT DISTINCT SNAME
FROM SUPPLIER AS S
WHERE 'P1' NOT IN
  (SELECT P#  FROM SP WHERE S# = S.S#);
```

- S# implicitly qualified by SP.
- S is a **label** or **alias** for SUPPLIER (Correlated subquery).

SUPPLIER(S#, SNAME, STATUS, CITY)
PART(P#, PNAME, COLOR, WEIGHT)
SP(S#, P#, QTY)

Q: Get P# for parts that are as least as heavy as all other parts.

```
SELECT P#
FROM PART
WHERE WEIGHT >= ALL
          (SELECT WEIGHT
           FROM PART);
```

- θ[ALL|SOME], where θ is one of <, =<, >, >=, =, <>.

SUPPLIER(S#, SNAME, STATUS, CITY)
PART(P#, PNAME, COLOR, WEIGHT)
SP(S#, P#, QTY)

- EXISTS(SELECT ...) evaluated to true if SELECT does not return an empty table.

Q: Find SNAME for suppliers who do not supply P1.

"For each supplier, print the name if there is no shipment in SP with P1 which is supplied by the supplier."

```
SELECT SNAME FROM SUPPLIER
WHERE NOT EXISTS
   (SELECT * FROM SP
   WHERE S# = SUPPLIER.S# AND P# ='P1');
```

SUPPLIER(S#, SNAME, STATUS, CITY)
PART(P#, PNAME, COLOR, WEIGHT)
SP(S#, P#, QTY)

- UNIQUE(SELECT ...) evaluated to false if SELECT return a table containing duplicates, otherwise true.

Q: Find suppliers who supply at least two distinct parts with the same quantity.

```
SELECT SNAME FROM SUPPLIER
WHERE
NOT UNIQUE
   (SELECT QTY FROM SP
   WHERE S# = SUPPLIER.S#);
```

Q: Does S1 supply P1?

```
SELECT * FROM SP
   WHERE S# = 'S1' AND P# ='P1';
```

- Empty result iff S1 does not supply P1.

Q: Find parts that are not supplied by Supplier S1.

```
SELECT * FROM PART
WHERE NOT EXISTS
   (SELECT * FROM SP
   WHERE S# = 'S1' AND P# =PART.P#);
```

- The result is empty iff S1 supplies every single part in the Part relation.

Q: Find supplier names who supply every part in the PART relation.

{Supplier.name | ∀Part∃SP(Part.P#=SP.P# & SP.S# = Supplier.S#)}

{Supplier.name | -∃Part-∃SP(Part.P#=SP.P# & SP.S# = Supplier.S#)}

```
SELECT SNAME FROM SUPPLIER
WHERE NOT EXISTS
   (SELECT * FROM PART
   WHERE NOT EXISTS
   (SELECT * FROM SP
   WHERE S# = SUPPLIER.S#  AND P# =PART.P#));
```

Built-in functions

COUNT(*): no of tuples.

COUNT(DISTINCT <attr>): no of nonduplicate values.

SUM([DISTINCT] <item>): sum of (numeric) values.

AVG([DISTINCT] <item>): average of (numeric) values.

MAX(<item>): largest value.

MIN(<item>): smallest value.

<item> := abstraction of a collection of values, e.g., TAX+PREMIUM.

- Except for COUNT(*), these functions operate on nonnull values.

SP (S#, P#, QTY)

S1	P1	10
S1	P2	20
S2	P2	30
S3	P1	20

Q:How many different suppliers currently supplying part to us.

SELECT COUNT(DISTINCT S#) FROM SP;

- Count (SELECT DISTINCT S# FROM SP).

Q: Find the total quantity of P2 supplied.

SELECT SUM(QTY) FROM SP
WHERE P#='P2';

- Arithmetic expressions are allowed in SELECT, e.g., AVG(QTY), SUM(QTY) + 5, AVG(QTY+WEIGHT).

SP (S#, P#, QTY)

S1	P1	10
S1	P2	20
S2	P2	30
S3	P1	20

Q: Find all orders that have quantity that is greater than the average quantity of all orders.

```
SELECT *
FROM SP
WHERE QTY >
    (SELECT AVG(QTY) FROM SP);
```

- A relation can be partitioned into groups according to values in some attributes.

SP(S# P# QTY)

S1	P1	10
S2	P1	5
S3	P1	8
S1	P2	7
S5	P2	2
S3	P2	6
.		
.		

Q: Find P# (in SP) and the total quantity supplied.

SELECT P#, SUM(QTY) FROM SP GROUP BY P#;

- Each item in SELECT must be a **unique** property of a group.

SP(S# P# QTY)

S#	P#	QTY
S1	P1	10
S2	P1	5
S3	P1	8
S1	P2	7
S5	P2	2
S3	P2	6
S2	P4	9
S5	P4	7

- After partitioning, groups can be qualified or disqualified using HAVING clause.

Q: List the P# and total quantity supplied for parts that are having more than 2 shipments.

SELECT P#, SUM(QTY) FROM SP
GROUP BY P# HAVING COUNT(*) > 2;

SP(S# P# QTY)

S1	P1	10
S1	P2	7
S2	P1	5
S2	P3	9
S5	P2	2
S5	P4	7

Q: List the total quantity supplied by each supplier who supplies at least one part that is not supplied by any other supplier.

```
SELECT S#, SUM(QTY) AS SUMQTY FROM SP AS
OUTER GROUP BY S# HAVING EXISTS
    (SELECT * FROM SP
    WHERE SP.S# = OUTER.S# AND SP.P# NOT IN
    (SELECT P# FROM SP WHERE SP.S#<>OUTER.S#));
```

SP(S# P# QTY)

S#	P#	QTY
S1	P1	10
S1	P2	7
S2	P1	5
S2	P3	9
S5	P2	2
S5	P4	7

Q: List the total quantity supplied by each supplier who supplies at least one part that is not supplied by any other supplier.

```
SELECT SP.S#, SUM(SP.QTY) AS SUMQTY
FROM SP, (SELECT DISTINCT S# FROM SP AS
OUTER WHERE  SP.P# NOT IN (
    SELECT P# FROM SP WHERE
    SP.S# <> OUTER.S#)) AS QUAL-SUP(S#)
WHERE QUAL-SUP.S# = SP.S# GROUP BY SP.S#
```

Select Statement:= <subquery> [ORDER BY {**<column name**> [ASC|DESC]}$^+$];

<*subquery*>:= SELECT [DISTINCT]

{* | <*expr*> [AS <alias>]} FROM <tables>

[WHERE <*predicate*>]

[GROUP BY <column name>$^+$] [HAVING <*predicate*>]

|

<*subquery*> <*binary op*> [ALL] <*subquery*>;

<*binary op*>:= UNION | INTERSECT | EXCEPT.

1. The Cartesian product of all tables in the FROM clause is formed.
2. Rows not satisfying the WHERE are eliminated.
3. The remaining rows are grouped in accordance with the GROUP BY clause.
4. Group not satisfying the HAVING are then eliminated.
5. The expressions of the SELECT clause are evaluated.
6. If the keyword DISTINCT is present, duplicate rows are now eliminated.
7. Evaluate binary operator for subqueries up to this point.
8. Finally, the set of all selected rows is sorted if an ORDER BY is present.

SUPPLIER(S#, SNAME, STATUS, CITY)
PART(P#, PNAME, COLOR, WEIGHT)
SP(S#, P#, QTY)

Modification

Insert

- **Format: INSERT INTO {<table> | <view>} [(<attrs>)] {VALUES (<data items>) | <*select statement*>};**

Q: Insert P7, name washer, weight 2 and color unknown into the PART relation.

```
INSERT INTO PART (P#, WEIGHT, PNAME) VALUES
('P7', '2', 'Washer');
```

Q: For each part supplied, get the P# and the total quantity supplied of that part, and save the result in the TEMP table.

```
INSERT INTO TEMP (P#, TOTQTY)
    SELECT P#, SUM(QTY) FROM SP GROUP BY P#;
```

SUPPLIER(S#, SNAME, STATUS, CITY)
PART(P#, PNAME, COLOR, WEIGHT)
SP(S#, P#, QTY)

Delete

- **Format: DELETE <table> [<*where clause*>];**

Q: Delete all suppliers in London from SUPPLIER.

DELETE SUPPLIER WHERE CITY = 'London';

SUPPLIER(<u>S#</u>, SNAME, STATUS, CITY)
PART(<u>P#</u>, PNAME, COLOR, WEIGHT)
SP(<u>S#, P#</u>, QTY)

<u>Update</u>

- **Format: UPDATE <table> SET <attr> = <*expr*> [, <attr> = <*expr*>] ... [<where clause>];**

Q: Change the color of P2 to yellow, increase its weight by 5 and let its PNAME to be unknown.

```
UPDATE PART
SET COLOR =  'Yellow'
    WEIGHT = WEIGHT + 5
    PNAME = NULL
WHERE P# = 'P2';
```

Views

Introduction

- An external schema in SQL - a set of tables and views.
- A view is a **virtual** table
  - derived from one or more base tables or views.
  - computed dynamically.
- For retrieval - just like a table.
  For update - may have problem.

Create views

- **Format: CREATE VIEW <view> [(<column name>$^{+}$)] AS <*subquery*>[WITH CHECK OPTION];**

Ex: CREATE VIEW PQ (P#, SUMQTY)
AS SELECT P#, SUM(QTY)
FROM SP GROUP BY P#;

Views can be updatable or read-only.

The optional WITH CHECK OPTION clause specifies that, for updatable views, inserts and updates performed through the view should not be permitted if they result in rows that would be *invisible* to the view <subquery>.

Ex: CREATE VIEW S AS SELECT * FROM SUPPLIER
WHERE STATUS <=20 WITH CHECK OPTION;

No update is allowed if changes, via this view, result in status > 20.

Drop views

- A view can be deleted any time.
- **Format: DROP VIEW <view>;**

Ex: DROP VIEW PQ;

View update problem

- Consider a view containing total quantity of each P# supplied.
  CREATE VIEW TQ (P#, SUMQTY)
  AS SELECT P#, SUM(QTY)
  FROM SP GROUP BY P#;
- If allow to update this view, cannot translate the operation into the base tables. E.g., change SUMQTY of a part from 3 to 4, add/delete a tuple.
- View update problem
  -**No** simple solution.
  -Standard imposed a number of restrictions on how a view can be defined via a <subquery>.

Interactive vs. Non-Interactive SQL

- *Interactive SQL*: SQL statements input from terminal; DBMS outputs to screen
  - Inadequate for many applications
    - SQL has very limited expressive power (not Turing-complete)
- *Non-interactive SQL*: SQL statements are included in an application program written in a host language, like C, Java, COBOL,…

Introducing SQL Into the Application

- SQL statements can be incorporated into an application program in two different ways:
  - ➢ ***Statement Level Interface*** (SLI): Application program is a mixture of host language statements and SQL statements
  - ➢ ***Call Level Interface*** (CLI): Application program is written entirely in host language
    - ❖ SQL statements are values of string variables that are passed as arguments to host language (library) procedures

Statement Level Interface

- SQL statements embedded in the application have a *special syntax* that sets them apart from host language constructs
  - ➢ e.g., EXEC SQL *SQL_statement*
- *Precompiler* scans program and translates SQL statements into calls to host language library procedures that communicate with DBMS
- *Host language compiler* then compiles program
- Embedded SQL has two forms:
  - ➢ ***Static*** SQL: Useful when SQL portion of program is known at compile time
  - ➢ ***Dynamic*** SQL: Useful when SQL portion of program *not* known at compile time. Application constructs SQL statements *at run time* as values of host language variables

Call Level Interface

- Application program written entirely in host language (no precompiler)
  - Examples: JDBC, ODBC
- SQL statements are values of string variables constructed *at run time* using host language
  - Similar to dynamic SQL
- Application uses string variables as arguments of library routines that communicate with DBMS
  - e.g. executeQuery("*SQL query statement*")

Embedded SQL

1. Introduction

- ISQL - limited expressiveness.
- Dual-mode.
- SQL statements embedded in a variety of PL's called **host** languages.
- System- and language-dependent.
- Assume C and ORACLE (Pro*C/C++).
- CREATE TABLE CUSTOMERS

```
(
CID CHAR(4) NOT NULL,
CNAME VARCHAR(13),
CITY VARCHAR(20),
DISCNT REAL
);
```

2. Static SQL

```
/* Given a city, return customer Name and Discnt*/
#include <stdio.h>
#include <ctype.h>
exec sql include sqlca;
int prompt1(char[], char[], int);
char prompt[] = "Please enter customer city: ";
main()
    {
    /*declare SQL host variables.*/
    exec sql begin declare section;
    VARCHAR cust_city[21], cust_name[14];
    float cust_discnt;
    VARCHAR user_name[20], user_pwd[10];
    exec sql end declare section;
```

```
strcpy(user_name.arr, “testdb”);
user_name.len = strlen(“testdb”);
strcpy(user_pwd.arr, “passwd”);
user_pwd.len = strlen(“passwd”);
exec sql connect :user_name identified by
:user_pwd;
exec sql whenever sqlerror stop;
exec sql whenever not found go to next;
```

```
exec sql declare c_city cursor for
        select cname, discnt
        from customers where city = :cust_city;
while((prompt1(prompt, cust_city.arr, 20))>=0) {
        cust_city.len = strlen(cust_city.arr);
        exec sql open c_city;
        while(TRUE)
            {exec sql fetch c_city into :cust_name, :cust_discnt;
            cust_name.arr[cust_name.len] = ‘\0’;
            printf(“customer name is %s and discnt
            is %5.1f\n”, cust_name.arr,cust_discnt);}
        next:exec sql close c_city;
        exec sql commit work;}
exec sql disconnect;}
```

- VARCHAR: ORACLE data type, struct {unsigned short len; unsigned char arr[21] } cust_city.
- Embedded SQL statements are prefixed by EXEC SQL.
- SQL Communication Area is a declared structure containing member variables.
- **host variables:** referenced by SQL statements, prefixed with a **colon**.
- Host variables must be defined within an embedded SQL declare section.
- EXEC SQL WHENEVER <*condition*> <*action*>;
- EXEC SQL {COMMIT | ROLLBACK} WORK: end of a transaction.

Cursor Operations

- EXEC SQL DECLARE <curson-name> [INSENSITIVE][SCROLL] CURSOR FOR embedded-SELECT-statement [ORDER BY order-item-comma-list] [FOR {READ ONLY | UPDATE OF column-name [, column-name] ... }];
- INSENSITIVE - create a copy of the rows in the result set and all accesses through the cursor will be to that copy
- If INSENSITIVE is not specified, and is not declared READ ONLY, then the current row of the base table can be updated or deleted through the cursor

- EXEC SQL OPEN <cursor-name>;

  the query associated with cursor is executed and an active set is returned.

- EXEC SQL FETCH [[row-selector] FROM] <cursor-name> INTO <host-variable> [, host-variable] . . .;

  advances cursor according to row-selector in the active set and then assign field values to host variables.

- The row-selector is one of

  FIRST

  NEXT

  PRIOR

  LAST

  ABSOLUTE *n*

  RELATIVE *n*

- The option SCROLL in the declaration of the cursor means that all forms of the FETCH statement are allowable. If SCROLL is not specified, only NEXT is allowable.
- EXEC SQL UPDATE <table-name> SET <attr>=<expr>[,..] WHERE CURRENT OF <cursor>;

  update the current row pointed at by the cursor with given values.

Updating a set of tuples

```
:
exec sql declare cursor c for …;
:
exec sql open cursor c;
while (true){
    exec sql fetch c into :var …;
    :
    exec sql update <table> set <attribute> = <expr>
    where current of c;
    exec sql commit work;}
:
```

3. Dynamic SQL

- The facility that allows you to execute SQL statements whose complete text you don't know until at run time.
- May result in reduced execution performance.
- They are stored in character strings, then **prepared** and **executed**.
- May contain dummy host variables, do not need to be declared.
- Can be executed in two ways:
  - prepared and executed in one step.
  - prepared and then executed as many times as required.

Execute Immediate
Format: EXECUTE IMMEDIATE <statement-variable>

exec sql execute immediate "create table dyn1 (col1 char(4))";

```
CREATE TABLE EMP(
    EMPNO INTEGER,
    ENAME VARCHAR(10),
    DEPNO INTEGER);
```

Prepare and Execute Statements

- Prepare: as if processed by preprocessor and DBMS as in static SQL.
- Format:
  PREPARE <statement-name> FROM <statement-variable>
- Execute with the current values, or use with a cursor.
- Format:
  EXECUTE <statement-name> [INTO var-list][USING <parameter-list>]

Retrieve all employee names, given a deptno

An Example

```
#define USERNAME "SCOTT"
#define PASSWORD "TIGER"
#include <stdio.h>
exec sql include sqlca;
exec sql begin declare section;
    char *username = USERNAME;
    char *password = PASSWORD;
    varchar sqlstmt [80];
    varchar ename[11];
    int deptno = 10;
exec sql end declare section;
main ()
{
```

```
exec sql whenever sqlerror goto error;
exec sql connect :username identified by
:password;
puts("\nConnected to Oracle. \n");
sqlstmt.len = sprintf(sqlstmt.arr, "select ename
from emp where deptno = :v1");
puts(sqlstmt.arr);
printf("   v1=%d\n", deptno);
printf("\nEmployee\n);
printf("___________\n");
exec sql prepare s from :sqlstmt;
```

/* The declare cursor statement associates a cursor with a prepare statement. The cursor name does not appear in the declare section. A single cursor name can be declared more than once. */
exec sql declare c cursor for s;
/* The open statement evaluates the active set of the prepared query using the specified input host variables, which are substituted positionally for placeholders in the prepared query. */
exec sql open c using :deptno;
exec sql whenever not found goto notfound;
/*Loop until the not found condition is detected. */

```
while(TRUE)
    {
    /* If there are more select-list fields then output
    host variables, the extra fields will not be
    returned. Specifying more output host variables
    than select-list fields results in Oracle error.*/
    exec sql fetch c into :ename;
    /*null-terminated the array before output.*/
    ename.arr[ename.len] ='\0';
    puts(ename.arr);
    }
```

```
notfound:
    exec sql close c;
    exec sql commit release;
    puts("\nHave a nice day!\n");
    exit(0);
error:
    printf("\nSQL error.\n");
    exec sql whenever sqlerror continue;
    exec sql rollback release;
    exit(1)
}
```

Dynamic SQL and Cursors

- Open format:
  OPEN <cursor-name> [<using-clause>]
- Using-clause specifies current values for dynamic parameters.

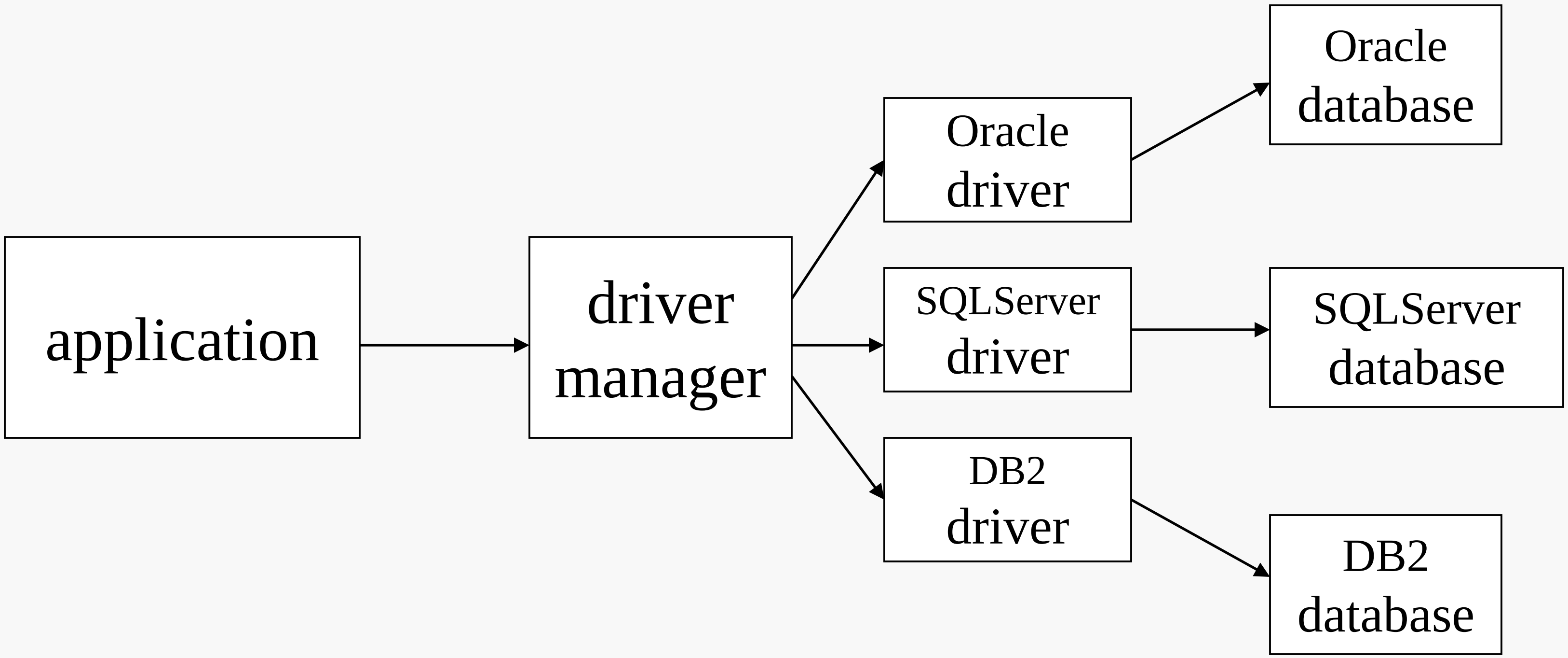
JDBC

Reference:

http://www.tutorialspoint.com/jdbc/index.htm

JDBC Database Access with Java, by Hamilton, Cattell and Fisher, Addison-Wesley.

- Call-level interface (CLI) for executing SQL from a Java program
- SQL statement is constructed at run time as the value of a Java variable (as in dynamic SQL)
- JDBC passes SQL statements to the underlying DBMS. Can be interfaced to any DBMS that has a JDBC driver
- Download the appropriate JDBC driver
- For our db2, place db2jcc4.jar and db2jcc_license_cu.jar in your classpath


application
driver manager
Oracle driver
SQLServer driver
DB2 driver
Oracle database
SQLServer database
DB2 database

An Overview

1. establish a connection with a database
2. send SQL statements
3. process the results

```
:
Class.forName(“com.ibm.db2.jcc.DB2Driver”);
Connection con = DriverManager.getConnection
    (“jdbc:db2://linux028.student.cs.uwaterloo.ca:50002
/cs348”, “login”, “password”);
Statement stmt = con.createStatement();
stmt.executeUpdate("set schema myDB");
stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE);
ResultSet rs = stmt.executeQuery(“SELECT a, b, c
    FROM Tablel”);
while (rs.next()) {
    int x = rs.getlnt(”a”);
    String s = rs.getString(“b”);
    float f = rs.getFloat(“c”);}
:
```

An Example

```
CREATE TABLE COFFEES
    (COF_NAME VARCHAR(32),
    SUP_ID INTEGER,
    PRICE FLOAT,
    SALES INTEGER,
    TOTAL INTEGER);
```

CreateCoffees.java

```
import java.sql.*;
public class CreateCoffees
    {
    public static void main(String args[])
        {
        String url =
    "jdbc:db2://linux028.student.cs.uwaterloo.ca:50002/cs348";
        Connection con;
        String createString;
        createString = "create table COFFEES " +
                "(COF_NAME VARCHAR(32), " +
                "SUP_ID INTEGER, " +
                "PRICE FLOAT, " +
                "SALES INTEGER, " +
                "TOTAL INTEGER)";
        Statement stmt;
```

```
try {
    Class.forName("COM.ibm.db2.jcc.DB2Driver");
    }
catch(java.langClassNotFoundException e)
    { System.err.print(”ClassNotFoundException: “);
    System.err.println(e.getMessage());
    System.exit(1);}
try {
    con = DriverManager.getConnection(url, “myLogin”,
    “myPassword”);
    stmt = con.createStatement();
    stmt.executeUpdate("set schema CoffeeDB");
    stmt.executeUpdate(createString);
    stmt.close();
    con.close();
    }
catch(SQLException ex) {
    System.err.println(”SQLException:” + ex.getMessage());
    System.exit(1);}}
```

InsertCoffee.java

```
import java.sql.*;
public class InsertCoffee
    {
    public static void main(String args[])
            {
            String url =
    “jdbc:db2://linux028.student.cs.uwaterloo.ca:50002/cs348”;
            Connection con;
            Statement stmt;
            String query = “select COF_NAME, PRICE from COFFEES”;
            try {
                Class.forName("COM.ibm.db2.jcc.DB2Driver");
                }
            catch(java.lang.ClassNotFoundException e)
               {
                System.err.print(”ClassNotFoundException: “);
                System.err.println(e.getMessage());
                System.exit(1);
                }
```

```
try {
    con = DriverManager.getConnection(url, “myLogin”,
    “myPassword”);
    stmt = con.createStatement();
    stmt.executeUpdate("set schema CoffeeDB");
    stmt.executeUpdate(”insert into COFFEES “ +
                    “values(‘Colombian’, 101, 7.99, 0, 0)”);
    stmt.executeUpdate(”insert into COFFEES “ +
                    “values(’French_Roast’, 49, 8.99, 0, 0)”);
    stmt.executeUpdate(”insert into COFFEES “ +
                    “values(’Espresso’, 150, 9.99, 0, 0)”);
    stmt.executeUpdate(”insert into COFFEES “ +
                    “values(’Colombian_Decaf’, 101, 8.99, 0, 0)”);
    stmt.executeUpdate(”insert into COFFEES “ +
                    “values(’French_Roast_Decaf’, 49, 9.99, 0, 0)”);
```

```
    ResultSet rs = stmt.executeQuery(query);
    System.out.println("Coffees and Prices:");
    while (rs.next())
            {
            String s = rs.getString("COF_NAME");
            float f = rs.getFloat("PRICE");
            System.out.println(s + "" + f);
            }
    stmt.close();
    con.close();
    }
catch(SQLException ex)
    {
    System.err.println("SQLException: " + ex.getMessage());
    System.exit(1);
    }
}
```

Summary

import java.sql.*; -- *import all classes in package* java.sql

Class.forName (*driver name*); // *static method of class* Class
// *register specified driver*

Connection con = DriverManager.getConnection(*Url*, *Id*, *Passwd*);

- Static method of class DriverManager; attempts to connect to DBMS
- If successful, creates a connection object, con, for managing the connection

Statement stat = con.createStatement ();

- Creates a statement object stat
- Statements have executeQuery() & executeUpdate() methods

String query =
“SELECT C.COF_NAME FROM COFFEES C” +
“WHERE C.SUPID = 1211” ;

ResultSet res = stat.executeQuery (query);

- Creates a result set object, res.
- Prepares and executes the query.
- Stores the result set produced by execution in res (analogous to opening a cursor).
- The query string can be constructed at run time (as above).
- The input parameters are plugged into the query when the string is formed (as above)

Preparing and Executing a Query

String query =
“SELECT C.COF_NAME FROM COFFEES C” +
“WHERE C.SUPID =?”;

placeholders

- PreparedStatement ps = con.prepareStatement (query);
  - Creates a prepared statement object, ps, containing the prepared statement
  - Placeholders (?) mark positions of in parameters; special API is provided to plug the actual values in positions indicated by the ?’s

```
String  SID, j;
……………
ps.setString(1, SID);     // set value of first in parameter

ResultSet  res = ps.executeQuery ( );
```

- Creates a result set object, res
- Executes the query
- Stores the result set in res
- Initializes the cursor

```
while  ( res.next ( ) )  {                          // advance the cursor
      j = res.getString (“COF_NAME”);   // fetch output value
      …process output value…
}
```

GetXXX

- JDBC offers two ways to identify the column from which a getXXX method gets a value. One way is to give the column name, as was done in the example above. The second way is to give the column index.
- String s = rs.getString(1);
- float n = rs.getFloat(2);
- getByte, getShort, getLong, getDouble, getBoolean, getDate, getTime.

Update weekly sales of coffees

```
PreparedStatement updateSales;
String updateString = “update COFFEES “ +
    “set SALES = ?, set TOTAL = TOTAL+ ? where COF_NAME
    like ?”;
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {“Colombian”, “French_Roast”, “Espresso”,
    “Colombian_Decaf”, “French_Roast_Decaf” };
int len = coffees.length;
for(int i = 0; i < len; i++)
    {
    updateSales.setlnt(1, salesForWeek[i]);
    updateSales.setlnt(2, salesForWeek[i]);
    updateSales.setString(3, coffees[i]);
    updateSales.executeUpdate();
    }
```

```
con.setAutoCommit(false);
PreparedStatement updateSales = con.prepareStatement(
"UPDATE COFFEES SET SALES = ? WHERE COF_NAME
    LIKE ?");
updateSales.setlnt(1, 50);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate();
PreparedStatement updateTotal = con. prepareStatenient(
"UPDATE COFFEES SET TOTAL = TOTAL + ? WHERE
    COF_NAME LIKE ?");
updateTotal.setlnt(1, 50);
updateTotal.setString(2, "Colombian");
updateTotal.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

Schema Analysis

- J.D. Ullman, **Principles of Database and Knowledge-based Systems**, Volume I, Computer Science Press, 1988.

- Schema analysis
  - a formal study of schema properties.
  - identify classes of good database schemas.
- Restricted to the relational model.

Ex: CAR(MODEL, #CYL, ORIGIN, TAX, LIC_FEE)

MODEL	#CYL	ORIGIN	TAX	LIC_FEE
RABBIT	4	GER	15	35
MIRAFIORI	4	ITY	18	35
ACCORD	4	JAP	20	35
CUTLASS	8	USA	0	50
MUSTANG	4	CAN	0	35
MUSTANG	6	USA	0	42
2000	6	CAN	0	42
AUDI	4	GER	15	35

MODEL #CYL→ORIGIN TAX;
ORIGIN→ TAX; #CYL→ LIC_FEE

- Logical data duplication.
- Peculiar phenomena arise - **update anomalies.**
  1. Can't insert 10 cyl. car lic_fee.
  2. Delete <Cutlass, 8, . . ., 50>.
  3. Update lic_fee for 4-cyl. cars.
- Cause: several "pieces" of info. lumped together.

Notation

- R, X, Y, Z - sets of attributes or relation schemes.
- r - a relation defined on a relation scheme R.
- s, t, u, v - tuples defined on some relation scheme.
- A, B, C, D, E - single attributes.
- ABC - a set consists of the attributes A, B and C.
- F, G, H - sets of functional dependencies.
- f, g, h - single functional dependencies.
- Let t be a tuple. Then t[X] denotes the X-components of t.

Functional Dependencies (fd's)

- Let r defined on R, r is said to **satisfy** $X \rightarrow Y$ if whenever two tuples in r agree on their X-components, then they also agree on their Y-components.

Ex:

A	B	C	D
0	2	3	4
1	2	3	5
6	7	8	9
6	7	8	10

Is $A \rightarrow B$ satisfied?

Is $B \rightarrow C$ satisfied?

Is $X \rightarrow D$ satisfied, where X is any subset of ABC.

Is $AC \rightarrow A$ satisfied?

Is $X \rightarrow Y$ satisfied, where Y is a subset of X.

- $X \rightarrow Y$ is **trivial** if Y is a subset of X.
- Suppose R(A, B, C) satisfies $\{A \rightarrow B, B \rightarrow C\}$, then R satisfies $A \rightarrow C$.
- An fd g **is logically implied** by F, denoted by $F \models g$, if whenever r satisfies F, r satisfies g.
- $\{A \rightarrow B, B \rightarrow C\} \models A \rightarrow B, B \rightarrow C, AC \rightarrow B, AB \rightarrow C, A \rightarrow C$, plus all the trivial fd's.

Ex:CAR(<u>MODEL,#CYL</u>,ORIGIN, TAX, LIC_FEE)

{#CYL→LIC_FEE} |=MODEL #CYL→LIC_FEE.

Let s and t be an arbitrary pair of tuples from the CAR relation.

s[MODEL #CYL] = t[MODEL #CYL]

By the def. of #CYL → LIC_FEE,

s[LIC_FEE] = t[LIC_FEE].

Hence MODEL #CYL→LIC_FEE.

- The **closure of a set of fd's F**, $F^+=\{X \to Y \mid F \models X \to Y\}$.

Ex: $F = \{A \to B, B \to C\}$, then $F^+ \supseteq \{A \to B, B \to C, AC \to B, AB \to C, A \to C,$ plus all the trivial fd's$\}$.

- F is a **cover** of (or **equivalent** to) G, $F \approx G$ if $F^+ = G^+$.

Ex: $F = \{A \to B, B \to C\}$, then every A-value determines a B-value and a C-value.

- The **closure of a set of attributes X**, $X^+=\{A \mid X \to A$ is in $F^+\}$.

Ex: $F = \{A \to B, C \to D\}$, then $A^+ = \{A, B\}$, $AC^+ = \{A, B, C, D\}$.

- X is a (**candidate**) **key** of R if $X \to R \in F^+$ and no proper subset of X has this property.
- Y is a **superkey** of R if Y contains a key of R.

Theorem: Given F, and XY is subset of R. The following are equivalent.

1. F $\models$ X$\rightarrow$Y.
2. Y is a subset of X^+.

Theorem: Let Y = $A_1 \ldots A_n$. F $\models$ X$\rightarrow$Y iff F $\models$ X$\rightarrow A_1$ and F $\models$ X$\rightarrow A_2$ and ... and F $\models$ X$\rightarrow A_n$.

- F is a *minimal* cover if

1. Every rhs of an fd in F is a single attribute and
2. For no X$\rightarrow$A in F such that F-{X$\rightarrow$A} $\approx$ F and
3. For no X$\rightarrow$A in F and a proper subset Z of X such that F-{X$\rightarrow$A} $\cup${Z$\rightarrow$A} $\approx$ F

Computing Closure of X

- Determine if an fd $X \rightarrow Y$ is in F^+.

Input: X and F

Output: X^+.

Method:

```
closure := X;
while (changes to closure) do
    for each Y'→Z in F do
    if Y' is a subset of closure &
    Z⊄closure then closure := closure ∪ Z ;
    end;
end;
```

Ex: closure = {MODEL, #CYL};

MODEL #CYL→ORIGIN ⇒ closure = {MODEL, #CYL,ORIGIN}.

ORIGIN→TAX ⇒ closure = {MODEL, #CYL, ORIGIN, TAX}.

#CYL →LIC_FEE ⇒ closure = CAR.

Decomposition

- CAR(MODEL, #CYL, ORIGIN, TAX, LIC_FEE)
  F={MODEL #CYL→ORIGIN TAX, ORIGIN→ TAX, #CYL→ LIC_FEE}
- Decompose into
  CARS(MODEL, #CYL, ORIGIN)
  TAXATION(ORIGIN, TAX)
  LICENSING(#CYL, LIC_FEE)
- $\{R_1, \ldots, R_k\}$ is a **decomposition** of R if each R_i is a subset of R and $\cup R_i = R$.
- A in R is **prime** if A is in X, for some key X. Otherwise A is **nonprime**.

Normal Forms

- An fd is assumed to be a 'piece' of fact.
- R is in **1NF** if all the underlying domains are atomic.
- R is in **2NF** if each nonprime attr. is fully dependent on every key of R.
- A is **fully dependent** on X if X determines A and no proper subset of X has this property.

Ex:CAR(<u>MODEL,#CYL</u>,ORIGIN,TAX,LIC_FEE)

#CYL→LIC_FEE and LIC_FEE is nonprime implies CAR is not in 2NF.

CT(<u>MODEL, #CYL</u>, ORIGIN, TAX)

LICENSING(<u>#CYL</u>, LIC_FEE), are in 2NF.

- CT(MODEL, #CYL, ORIGIN, TAX) still has problems.

1. Can't store taxation info.
2. Delete the last car manufactured in a country $\Rightarrow$ lost tax. info.
3. Update tax rate $\Rightarrow$ many tuples updated.

- R is in **3NF** if there is no nonprime attribute that is transitively dependent on a key of R.
- Let X and Y be sets of attributes s.t. $X \rightarrow Y$ holds, but $Y \rightarrow X$ does not hold. Let A be an attribute that is not in XY and for which $Y \rightarrow A$ holds. We say that A is **transitively dependent** on X.

Ex: CT(<u>MODEL, #CYL</u>, ORIGIN, TAX)

$F = \{MODEL\ \#CYL \rightarrow ORIGIN, ORIGIN \rightarrow TAX\}$.

X = MODEL #CYL, Y = ORIGIN, A=TAX $\Rightarrow$ CT is not in 3NF.

- Decompose into
  CARS(<u>MODEL, #CYL</u>, ORIGIN)
  TAXATION(<u>ORIGIN</u>, TAX).
- Both are in 3NF.
- R is in **Boyce-Codd Normal Form (BCNF)** if whenever $X \rightarrow A$ holds in R, A is not in X, then X is a superkey of R.
- $\{R_1, \ldots, R_k\}$ is in 2(3 or BC)NF w.r.t. F if each R_i is 2(3 or BC)NF w.r.t. $F^+ | R_i$, where $F^+ | R_i = \{X \rightarrow A \mid X \rightarrow A$ in F^+ and $XA \subseteq R_i\}$.

Lossless Join

- Whatever that can be represented in the original scheme can also be represented by the decomposition, i.e., I can be recovered from $<\pi_{R1}(I), \ldots, \pi_{Rk}(I)>$.

Ex: SUPPLIER(S#, CITY, STATUS)

F ={S#→CITY, CITY→STATUS)

STATUS transitively dependent on S#.

Decompose into BCNF relations.

SS(S#, STATUS), SC(STATUS, CITY)

SUPPLIER	(S#,	CITY,	STATUS)
	S1	PARIS	20
	S2	LONDON	10
	S3	N.Y.	20

SS(<u>S#</u>,	STATUS)
S1	20
S2	10
S3	20

SC(STATUS,	<u>CITY</u>)
20	PARIS
10	LONDON
20	N.Y.

SUPPLIER	(<u>S#</u>,	CITY,	STATUS)
	S1	PARIS	20
	S2	LONDON	10
	S3	N.Y.	20
	S3	PARIS	20
	S1	N.Y.	20

- Can't recover original info., get "diluted", e.g. <S1, N.Y., 20> is not in I.

- A decomposition **R** = $\{R_1, \ldots, R_k\}$ of **U** is a **lossless join (LJ) decomposition wrt F** if for every relation **r** on **U** satisfying F, **r** = $\pi_{R1}(\mathbf{r}) * \ldots * \pi_{Rk}(\mathbf{r})$. Otherwise **R** is **lossy.**
- Testing can be done efficiently (via the chase algorithm).

Theorem: R1 * R2 (or {R1, R2}) is LJ iff

$Attr(R1) \cap Attr(R2) \rightarrow R1$ is in F^+ or

$Attr(R1) \cap Attr(R2) \rightarrow R2$ is in F^+.

Algorithm **Chase**

Input: A decomposition R={R_1,…, R_k} of U and F

Output: Determine R is LJ or not

Step:

1. Construct a tableau T_R for R
2. Apply fd's to T_R until no more changes to T_R
3. Output yes exactly there is a row of distinguished variables in Chase(T_R)

For each fd $X \rightarrow A$, if two tuples agrees on X but disagree on A, do

1. If one is a dv, replace the other by the dv.
2. If both are ndv's, replace one with higher index with lower index.

Ex: R={ABC, CEF,CDE, AD} and F={$A \to C$, $C \to D$, $CE \to F$, $C \to E$}

T_R:

A	B	C	D	E	F
a	a	a	ϕ_1	ϕ_2	ϕ_3
ϕ_4	ϕ_5	a	ϕ_6	a	a
ϕ_7	ϕ_8	a	a	a	ϕ_9
a	ϕ_{10}	ϕ_{11}	a	ϕ_{12}	ϕ_{13}

CHASE(T_R):

A	B	C	D	E	F
a	a	a	a	a	a
ϕ_4	ϕ_5	a	a	a	a
ϕ_7	ϕ_8	a	a	a	a
a	ϕ_{10}	a	a	a	a

$F=\{A\to C, C\to D, CE\to F, C\to E\}$

A	B	C	D	E	F
a	a	a	ϕ_1	ϕ_2	ϕ_3
ϕ_4	ϕ_5	a	ϕ_6	a	a
ϕ_7	ϕ_8	a	a	a	ϕ_9
a	ϕ_{10}	**a**	a	ϕ_{12}	ϕ_{13}

$F=\{A\to C, C\to D, CE\to F, C\to E\}$

A	B	C	D	E	F
a	a	a	**a**	**a**	ϕ_3
ϕ_4	ϕ_5	a	**a**	a	a
ϕ_7	ϕ_8	a	a	a	ϕ_9
a	ϕ_{10}	**a**	a	**a**	ϕ_{13}

Chase(T_R):

$F=\{A\to C, C\to D, CE\to F, C\to E\}$

A	B	C	D	E	F
a	a	a	**a**	**a**	**a**
ϕ_4	ϕ_5	a	**a**	a	a
ϕ_7	ϕ_8	a	a	a	**a**
a	ϕ_{10}	**a**	a	**a**	**a**

Dependency Preserving

Ex: SUPPLIER(S#, CITY, STATUS),

F={S#→CITY, CITY→STATUS}.

FIRST: SC(<u>S#</u>, CITY), CS(<u>CITY</u>, STATUS)

SECOND: SC(<u>S#</u>, CITY), SS(<u>S#</u>, STATUS)

- Both are BCNF & LJ.
- For FIRST, enforce key dependencies is sufficient. But not for SECOND.

SC(S#,	CITY)	SS(S#,	STATUS)
S1,	PARIS	S1	20
S2	PARIS	S2	30

CITY→STATUS violated !

In general, check the fd in SC * SS.

- SECOND does not faithfully represent the original scheme.

- A decomposition $\mathbf{R} = \{R_1, \ldots, R_k\}$ is **dependency preserving (d.p.)** if $(\cup F_i)^+ = F^+$, where $F_i = F^+ | R_i = \{X \rightarrow A \mid X \rightarrow A$ is in F^+ & XA in $R_i\}$.
- In SECOND, $(\cup F_i) = \{S\# \rightarrow CITY, S\# \rightarrow STATUS\}^+ \neq F^+$.
- But FIRST does.
  $(\cup F_i) = \{S\# \rightarrow CITY, CITY \rightarrow STATUS\} \approx F$.

Ex: R(ABCDE), F= {A $\rightarrow$ B, BC $\rightarrow$ E, ED $\rightarrow$ A}.

1. List all keys for R.
2. Is R in 3NF?
3. Is R in BCNF?

1. Consider all possible subsets of attributes: CDE, ACD, BCD are the only keys.
2. R is in 3NF because all attributes are prime.
3. R is not in BCNF because A $\rightarrow$ B is nontrivial and A is not a superkey.

- There is an algorithm for generating a 3NF, LJ and d.p. decomposition

- There is an algorithm for generating a BCNF and LJ decomposition

Transaction Management

1. Introduction

1. Creating an inconsistent state.

read(A)
A=A-50
write(A)
read(B) ←System crash
B= B+50
write(B)

2. Errors of concurrent execution. The lost update problem.

T1	T2
Read(X)	
X=X-1	
	Read(X)
	X=X-1
Write(X)	
	Write(X)

The dirty read problem

<u>T1</u>	<u>T2</u>	
Read(X)		
X=X-1		
Write(X)		
	Read(X)	
	X=X-1	
	Write(X)	
Read(Y)		
Y=Y+1		← T1 aborted

The inconsistent analysis problem.

T1	T2
Read(A)	
A=A-1000	
Write(A)	
	sum=0
	Read(A)
	sum=sum+A
	Read(B)
	sum=sum+B
Read(B)	
B=B+1000	
Write(B)	

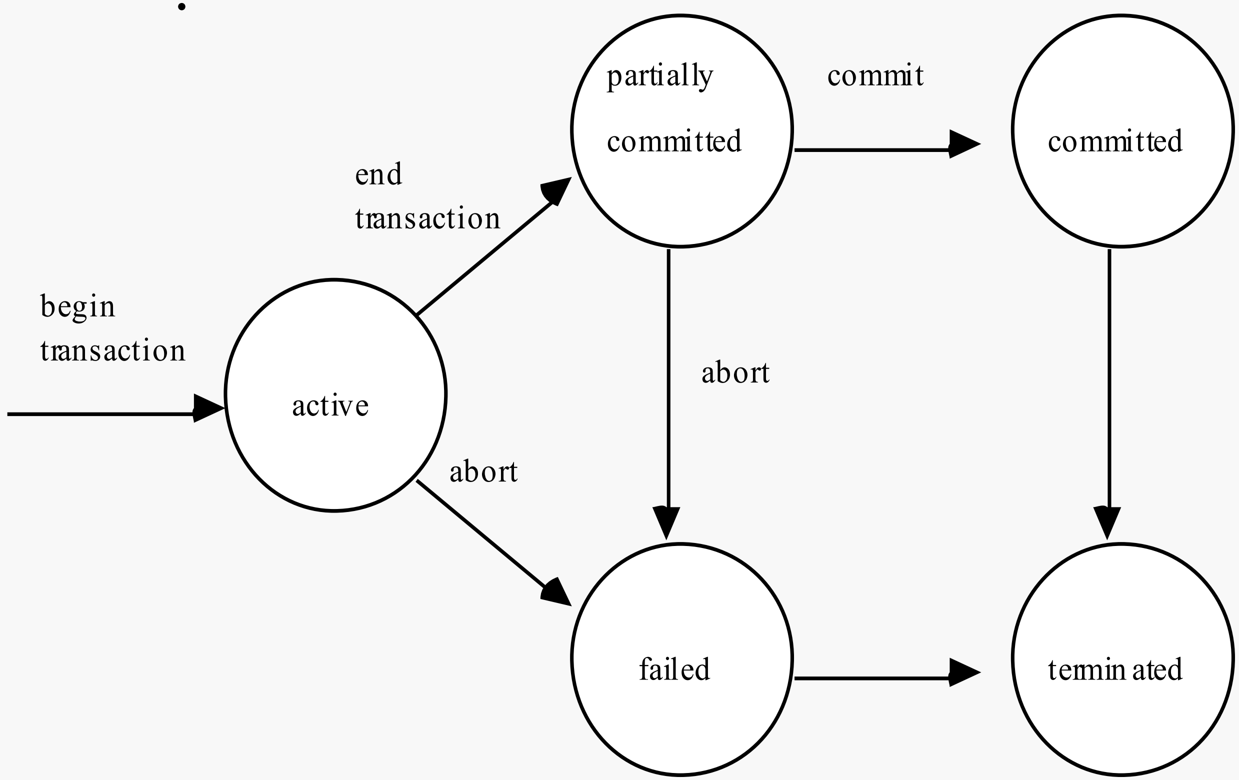
2. Transactions

- A **transaction** is a logical unit of work.

Required properties:

- **Atomicity**. A transaction is either performed in its entirety or not performed at all.
- **Consistency.** Take the database from one consistent state to another.
- **Isolation.** No interference among concurrently executing transactions.
- **Durability.** Once a transaction reaches a **committed** state, its changes on the database will never be lost.

- exec sql commit work
  exec sql rollback work
  begin_transaction
  end_transaction
  .
  .



3. Schedules

- A **schedule** S for a set of transactions T_1 , ... , T_n is a sequence of steps obtained by merging the steps in T_i's with the relative order of steps in each transaction is being preserved.

Ex: Consider the following two transactions which might be a transfer of fund from one account to another with the property that the sum A+B+C is preserved.

T1	T2
Read(A)	
A=A-10	
Write(A)	
Read(B)	
B=B+10	
Write(B)	
	Read(B)
	B=B-20
	Write(B)
	Read(C)
	C=C+20
	Write(C)

T1	T2
Read(A)	
	Read(B)
A=A-10	
	B=B-20
Write(A)	
	Write(B)
Read(B)	
	Read(C)
B=B+10	
	C=C+20
Write(B)	
	Write(C)

Cascading Rollback

T1	T2	
Read(A)		
A=A-10		
Write(A)		
	Read(A)	
	A=A-20	
	Write(A)	
Read(B)		
B=B+10		
Write(B)		← T1 aborted
	Read(C)	
	C=C+20	
	Write(C)	

- A schedule is said to **avoid cascading rollback** if every transaction in the schedule only reads items that were written by committed transactions.

Concurrency Control

1. Introduction

Ex:Reserve: READ # OF SEATS.
DECREMENT # OF SEATS.
WRITE # OF SEATS.

# OF SEATS IN D.B.	10	10	10	9	9	9
A	R.		DEC.	W.		
B		R.			DEC.	W.
# OF SEATS IN A	10	10	9	9	9	9
# OF SEATS IN B		10	10	10	9	9

- From user viewpoint - uniprogramming ⇒ in **some serial** order.

Serial and Nonserial Schedules

Ex: T1 transfers $1 from A to B while T2 transfers $2 from B to C.

T1	T2
Read(A)	
A=A-1	
Write(A)	
Read(B)	
B=B+1	
Write(B)	
	Read(B)
	B=B-2
	Write(B)
	Read(C)
	C=C+2
	Write(C)

T1	T2
	Read(B)
	B=B-2
	Write(B)
	Read(C)
	C=C+2
	Write(C)
Read(A)	
A=A-1	
Write(A)	
Read(B)	
B=B+1	
Write(B)	

- **Serial** schedules.

T1	T2
Read(A)	
	Read(B)
A=A-1	
	B=B-2
Write(A)	
	Write(B)
Read(B)	
	Read(C)
B=B+1	
	C=C+2
Write(B)	
	Write(C)

- A **nonserial** schedule, but seems to be "equivalent" to T2 then T1; a **serializable** schedule.

T1	T2
Read(A)	
	Read(B)
A=A-1	
	B=B-2
Write(A)	
Read(B)	
	Write(B)
	Read(C)
B=B+1	
	C=C+2
Write(B)	
	Write(C)

- An update is **lost**!! Not equivalent to any serial schedule; **non-serializable**.

2. Serializability

Basic Concepts

- Read and Write, Rlock and Wlock, Insert and Delete operations.
- Read and Write $\Rightarrow$ modifies an item based on its current value.
- Write $\Rightarrow$ overwrites an item.
- A schedule S is **serial** if all steps of every transaction occur **consecutively** in S.

Interpretation of Operations

	T1		T2		T1	T2
1.	Read(A)	f_1			Read(A)	
2.	Write(A)				Write(A)	
3.			Read(A)	f_2		Read(A)
4.			Write(A)		Read(B)	
5.	Read(B)	g_1				Write(A)
6.	Write(B)				Write(B)	
7.			Read(B)	g_1		Read(B)
			Write(B)			Write(B)

- Assume distinct function for each modification.
- $f_n(\ldots(f_2(f_1(A)))\ldots)$ and $g_m(\ldots(g_2(g_1(A)))\ldots)$ produces the same result precisely when $m=n$ and $f_i=g_i$, $\forall i$.

View Serializability

	T1	T2
1.	Read(A)	
2.	Write(A)	
3.		Read(A)
4.	Read(B)	
5.		Write(A)
6.	Write(B)	
7.		Read(B)
8.		Write(B)

	T1	T2
1.	Read(A)	
2.	Write(A)	
3.	Read(B)	
4.	Write(B)	
5.		Read(A)
6.		Write(A)
7.		Read(B)
8.		Write(B)

- Schedules S1 and S2 are **view equivalent** if

1. The set of transactions in S1 and S2 are the same.
2. For each data item Q, if transaction Ti reads the initial value of Q in S1, then the transaction Ti must, in S2, also read the initial value of Q.
3. For each Q, if in S1, Ti reads Q and the value of Q read by Ti was last written in step p of Tj, then the same will hold in S2.
4. For each Q, if in S1, if Ti is the last transaction that writes Q, then the same also holds in S2.

- S is **view serializable** or just **serializable** if it is view equivalent to some serial schedule.

Ex: Consider the following two schedules.

S1:

T1	T2	T3	
Read(A)			
	Write(A)		
Write(A)			
		Write(A)	f

S2:

T1	T2	T3	
Read(A)			
Write(A)			
	Write(A)		
		Write(A)	f

- Final A value: f(A)

The Read Before Write Model

- A transaction must read the item before it can be written.
- A transaction can read without write.

Ex: Transfer fund among accounts.

T1	T2	T1	T2	T1	T2
R. A		R. A		R. A	
A=A-1			R. B	A=A-1	
W. A		A=A-1			R. B
R. B			B=B-2	W. A	
B=B+1		W. A			B=B-2
W. B			W. B	R.B	
	R. B	R. B			W. B
	B=B-2		R. C	B=B+1	
	W. B	B=B+1			R. C
	R. C		C=C+2	W. B	
	C=C+2	W. B			C=C+2
	W. C		W. C		W. C
(**SERIAL**)		(**SERIALIZABLE**)		(**NONSER**.)	
A=A-1		A=A-1		A=A-1	
B=B+1-2		B=B-2+1		B=B+1	
C=C+2		C=C+2		C=C+2	

- Serializable schedule ⇒T2 then T1.

Conflicting Operations

	T1	T2
1.	Read(A)	
2.		Read(A)
3.	Write(A)	
4.	Read(B)	
5.		Write(A)
6.	Write(B)	
7.		Read(B)
8.		Write(B)

Steps 1 & 2: no conclusion.
Steps 4 & 5: no conclusion.
Steps 2 & 3: *impossible* to be equivalent to T1 then T2.
Steps 3 & 5: *impossible* to be equivalent to T2 then T1.
Steps 6 & 7: same as above.

- Two steps I and J **conflict** in a schedule S if they are operations by **different** transactions on the **same** data item, and at least one of these instructions is a **Write** operation.
- Conflicting pairs are those pairs that potentially produce different effects on a database state if their order is changed.
- A schedule in which there are two instructions I and J from transactions T1 and T2 on item Q.

1. I = Read(Q) and J = Read(Q). Non-conflicting.
2. I = Read(Q) and J = Write(Q). Conflicting.
3. I = Write(Q) and J = Read(Q). Conflicting.
4. I = Write(Q) and J = Write(Q). Conflicting.

A Serializability Test for the Read before Write Model

Input: A schedule S.

Output: Determines if S is serializable. If so, output a serial schedule.

Method:

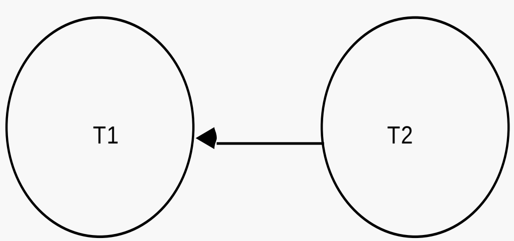
1. Create a precedence graph G(V,E).
   Nodes-transactions. Edges:
   T_i "write(Q)" before T_j "read (Q)" or T_i "read(Q)" before T_j "write (Q)", draw $T_i \rightarrow T_j$.
2. If a cylce exists, then S is not serializable. Else find an ordering s.t. T_i precedes T_j if $T_i \rightarrow T_j$.
   By topological sort. Repeat until empty: output T_i if T_i has no incoming edge, then remove T_i and all outgoing edges.
   The output is an equivalent serial schedule.

Theorem: The algorithm correctly determines if a schedule with read before write is serializable.

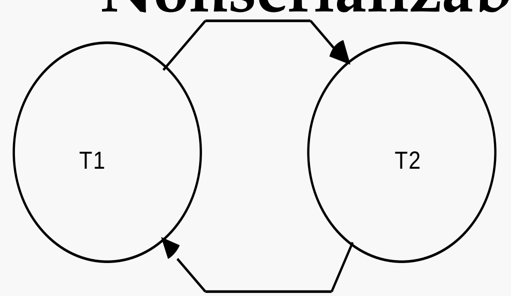
Ex:

T1	T2	T1	T2
R. A		R. A	
	R. B		R. B
W. A		W. A	
	W. B	R. B	
R. B			W. B
	R. C		R. C
W. B		W. B	
	W. C		W. C

Serializable



Nonserializable



3. Locking

A Model with Read & Write Locks

- **RLOCK(or shared lock)** - prevent others from writing a new value, many transactions can hold a RLOCK on the same item.
- **WLOCK(or exclusive lock)** - read & write, no other can RLOCK or WLOCK. Others have to wait for unlocking the item.
- Removed by UNLOCK.
- Lock manager grants lock or transaction waits.

Problems

1. Livelock (Starvation)

- **Livelock** is a situation in which a transaction Ti never be able to get hold of a lock.
- Solution: keep a FIFO lock queue.

2. Deadlock

- Deadlock occurs when each member of a set of two or more transaction S is waiting to lock an item currently held by some other member in S.

Ex: T1: WLOCK A

.

T2: WLOCK B

.

T1: WLOCK B $\Rightarrow$ wait.

.

T2: WLOCK A $\Rightarrow$ wait.

- Solutions

(a) Prevention

(i) Request all locks at once.

(ii) Assign a linear order to items. All transactions request locks in this order.

(b) Detection

By drawing a wait-for graph.

Nodes - transactions.

Edges - $T_i \rightarrow T_j$ if T_i is waiting to lock an item held by T_j.

Cycle exists iff deadlock.

Kill one or more transactions.

A Serializability Test for Read & Write Lock Model

- S_1 and S_2 are **equivalent** if

1. The set of transactions in S_1 and S_2 are the same.
2. For each data item Q, if a transaction T_i reads the initial value of Q in S_1, then the transaction T_i must, in S_2, also read the initial value of Q.
3. For each Q, if in S_1, T_i RLOCK Q or T_i WLOCK Q and the value of Q read by T_i was last written by T_j at step p, then the same will hold in S_2.
4. For each Q, S_1 and S_2 produce the same final value of Q.

- T_i: WLOCK A , ... , T_j: (R)WLOCK A $\Rightarrow T_i \rightarrow T_j$.
- T_i: RLOCK A , ... , T_j: WLOCK A $\Rightarrow T_i \rightarrow T_j$.

Input: A schedule S for RLOCK and WLOCK model.

Output: Test if S is serializable, if so, output a serial schedule.

Method:

1. A precedence graph G=(V,E). Nodes - transactions. Edges -

   (i) If T_i: RLOCK A and later T_j:WLOCK A, then $T_i \rightarrow T_j$.

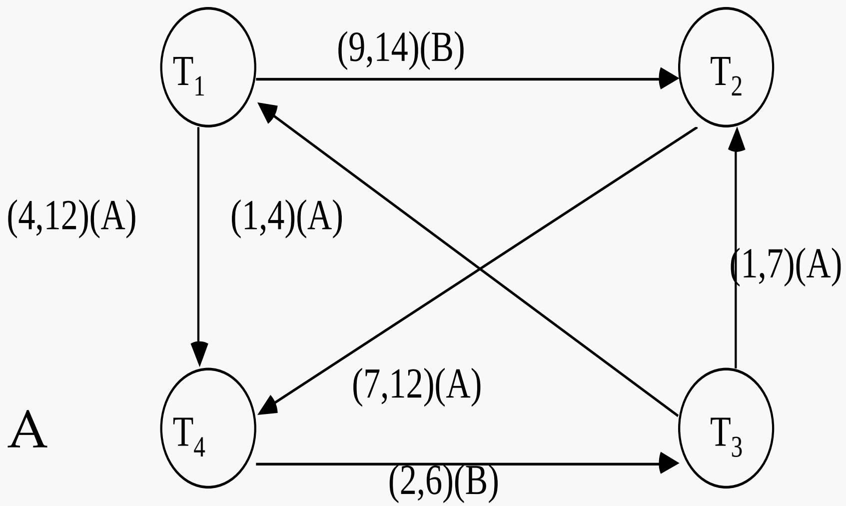
   (ii) If T_i:WLOCK A and if later T_j:RLOCK A (but before other WLOCK A) then $T_i \rightarrow T_j$.

   (iii) If T_i:WLOCK A and if later T_j:WLOCK A then $T_i \rightarrow T_j$.
2. If G has no directed cycle, produce a topological sort on G.
3. If G has a directed cycle then S is not serializable.

Theorem: The algorithm correctly determines if S is serializable.

Ex:

	T1	T2	T3	T4
(1)			WL A	
(2)				RL B
(3)			UNL A	
(4)	RL A			
(5)				UNL B
(6)			WL B	
(7)		RL A		
(8)			UNL B	
(9)	WL B			
(10)		UNL A		
(11)	UNL A			
(12)				WL A
(13)	UNL B			
(14)		RL B		
(15)				UNL A
(16)		UNL B		



Two-Phase Locking

- Simple protocol: in each transaction, all locks must precede all unlocks.
- Growing phase, shrinking phase $\Rightarrow$two phases.

Theorem: Two-phase locking protocol, in which all read- and write-locks precede all unlocks, guarantees serializability.

Isolation Levels in Relational Systems

- In the simple models, accesses are made to a *named item* (for example $r(x)$).
  - x can be locked
- In relational databases, accesses are made to *items that satisfy a predicate* (for example, a SELECT statement)
  - What should we lock?
  - What is a conflict?
- Concurrency controls in relational systems
  - Different isolations levels – some are no guarantee of serializability

Phantom Updates

Accounts(<u>name</u>, <u>type</u>, balance) Depositors(<u>name</u>, totbal)

Audit:

```
SELECT SUM (balance)
FROM Accounts
WHERE name = 'Mary';

SELECT totbal
FROM Depositors
WHERE name = 'Mary'
```

NewAccount:

```
INSERT INTO Accounts
VALUES ('Mary',3,100);

UPDATE Depositors
SET totbal = totbal + 100
WHERE name = 'Mary'
```

- RL & WL model is not adequate–insertion of tuples (phantom)
- Interleaved execution is not serializable

Assume Row Locking & Even Hold Locks Until End of Transaction

- The two SELECT statements in Audit may see inconsistent data
  - The second may see the effect of NewAccount; the first does not
- **Problem**: Audit's SELECT and NewAccount's INSERT do not commute, but the row locks held by Audit did not delay the INSERT
  - The inserted row is referred to as a *phantom*

Another Anomaly - Non-Repeatable Read

T1

SELECT SUM (balance)
FROM Accounts
WHERE name = ‘Mary’

T2

UPDATE Accounts
SET balance = 1.05 * balance
WHERE name = ‘Mary’

*does **not** introduce a phantom into predicate name=‘Mary’*

T1

SELECT SUM (balance)
FROM Accounts
WHERE name = ‘Mary’

Preventing Phantom

- *Predicate locking* prevents phantoms
  - A predicate describes a set of ***satisfying*** rows, some are in a table and some are **not**; e.g. *name* = *'Mary'*
  - Every SQL statement has an associated predicate
  - Insert a single tuple, the predicate is (A1 = v1) & (A2 = v2) & . . . &(An = vn), where Ai's are inserted attributes
  - Update – delete immediately followed by insert (write locks)
  - Two predicate locks **conflict** if one is a write and there exists a row (**not** necessarily in the table) that is contained in both
  - When executing a statement, acquire a (read or write) lock on the associated predicate

Preventing Phantoms With Predicate Locks

Audit:

SELECT SUM (balance)
FROM Accounts
WHERE name = 'Mary'

NewAccount:

INSERT INTO Accounts
VALUES ('Mary',3,100)

- Audit gets (long) read predicate lock *name='Mary'*.
- NewAccount requests a write predicate lock (*name='Mary' & type=3 & bal=100*)
  - Request denied since predicates overlap

Conflicts And Predicate Locks

Example 1

```
SELECT SUM (balance)
FROM Accounts
WHERE name = 'Mary'
```

```
DELETE
FROM Accounts
WHERE  balance < 100
```

- Statements conflict since predicates overlap and one is a write
  - There might be an account with bal < 100 and name = 'Mary'

Conflicts And Predicate Locks

Example 2

```
SELECT SUM (balance)
 FROM Accounts
 WHERE name = 'Mary'
```

```
DELETE
FROM Accounts
WHERE  name = 'John'
```

- Statements commute since predicates are disjoint.
  - There can be no rows (in or not in Accounts) that satisfy both predicates

Transaction Isolation Levels

- A transaction under 2PL won't release a lock until it commits or aborts. Scheduling that guarantees perfect serializability can be very intrusive on performance.
- Weakening the requirement of serializability⇒don't have such a strong guarantee of isolation but allow more concurrency.
- Set Transaction format:
  SET TRANSACTION
  {READ ONLY | READ WRITE}
  ISOLATION LEVEL
  {READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE};

SQL Isolation Levels

- Each SQL statement is executed atomically and is isolated from the execution of other statements
  - DBMS might be executing several SQL statements (from different transactions) concurrently
- Different transactions in the same application can execute at different levels, and their isolation levels are guaranteed
- READ UNCOMMITTED – dirty reads, non-repeatable reads, and phantoms allowed
- READ COMMITTED - dirty reads not allowed, but non-repeatable reads and phantoms allowed
- REPEATABLE READ – dirty reads, non-repeatable reads not allowed, but phantoms allowed
- SERIALIZABLE – dirty reads, non-repeatable reads, and phantoms not allowed; all schedules are serializable

Locking Implementation of SQL Isolation Levels

- SQL standard does not say *how* to implement levels
- Locking implementation is based on:
  - **Entities locked**: *rows & predicates.*
  - **Lock modes**: *read* (shared) & *write* (exclusive)
  - **Lock duration**:
    - *Short* - locks acquired in order to execute a statement are released when statement completes
    - *Long* - locks acquired in order to execute a statement are held until transaction completes
    - *Medium* – something in between (we give example later)
  - If conflict, a transaction has to wait. Could have starvation or deadlock.

Locking Implementation of SQL Isolation Levels

- Write locks are handled identically at all isolation levels:
  - Long-duration row & predicate write locks are associated with UPDATE, DELETE, and INSERT statements
- Read locks handled differently at each level:
  - READ UNCOMMITTED: no read locks
    - Hence a transaction can read a write-locked item!
    - Allows dirty reads, non-repeatable reads, and phantoms
  - READ COMMITTED: short-duration read locks on rows and on read predicate locks
    - Prevents dirty reads, but non-repeatable reads and phantoms are possible

Locking Implementation

- REPEATABLE READ: long-duration read locks on rows and short-duration on read predicate locks
  - Prevents dirty and non-repeatable reads, but phantoms are possible since read predicate locks are short-duration
- SERIALIZABLE: long-duration read lock and read *predicate* lock
  - Prevents dirty reads, non-repeatable reads, and phantoms and …
  - guarantees serializable schedules

Read Uncommitted

- Can read but request no lock on data or predicates $\Rightarrow$ can read data items *even if* the items currently have a read or write lock on it.
- Rough sums of branch balances.
- T2 at READ UNCOMMITTED

T1: *r(t1:1000)*
T1: *w(t1:900)*
T2: *r(t1:900)*
T2: *r(t2:500)*
T2: commit
T1: *r(t2:500)*
T1: *w(t2:600)*
T1: commit

Read Committed

- Except Read Uncommitted, all other cannot read uncommitted data.
- No data modified by T is changed until T commits.
- Suppose Ti with this level of isolation. Two of the three pairs of conflicting operations on rows by concurrently active transactions impossible:

(i) Ti:W(A)→Tj:R(A)

(ii) Ti:W(A)→Tj:W(A).

However, it is possible to have Ti:R(A)→Tj:W(A).

- *Nonrepeatable read*:

T1:R(A,50), T2:W(A,80), T2:Committed, T1:R(A,80), T1:Committed.

- *Scholar's lost update:*

T1:R(A,5), T2:R(A,5), T2:W(A,8), T2:Committed, T1:W(A,10), T1:Committed.

Cursor Stability

- A commonly implemented isolation level (not in the SQL standard) deals with cursor access
- Update via a cursor: **update … where current of cursor.**
- An extension of READ COMMITTED:
  - Long-duration write locks on row and predicates
  - Short-duration read locks on rows
  - Additional locks for handling cursors
- Read lock on row accessed through cursor is *medium-duration*; held until cursor is moved

Ex:

```
exec sql declare deposit cursor for select balance
    from accounts where name ='Mary' for update
    of balance;
exec sql open deposit;
(now loop through rows in deposit, and for each
    pass do)
exec sql fetch deposit into :balance;
balance = balance + 10;
exec sql update account set balance = :balance
    where current of deposit;
(end of loop)
exec sql close deposit;
exec sql commit work;
```

- To perform the same function under Read Committed isolation level, the transaction needs to be changed to the following:

  exec sql update accounts set balance = balance+10 where name ='Mary';

  exec sql commit work;

- Update holds a lock on each row for the duration of update.

Repeatable Read

- 2PL on individual tuples.
- A transaction can repeatedly read the same item and since between read, a long-duration read lock is held on the item.
- Two predicate locks are in *conflict* if the write predicate potentially changes the member of the read predicate.
- Cannot prevent phantom updates.

Phantoms Updates

Accounts(<u>name</u>, <u>type</u>, balance) Depositors(<u>name</u>, totbal)

Audit:

```
SELECT SUM (balance)
FROM Accounts
WHERE name = 'Mary';

SELECT totbal
FROM Depositors
WHERE name = 'Mary';
```

NewAccount:

```
INSERT INTO Accounts
VALUES ('Mary',3,100);

UPDATE Depositors
SET totbal = totbal + 100
WHERE name = 'Mary';
```

- If Audit run at repeatable read, won't prevent NewAccount to start executing. But ok if run at serializable.
- How about NewAccount? What if the order of two statements changed?

Summary

- Consider all other possible concurrent executing transactions, ask what properties the transaction needs to have.
- Set the transaction's isolation level to the weakest.

Crash Recovery

1. Introduction

- Many possible causes - programming errors; hardware errors; operator errors; other failures : fluctuations in the power supply, fire, or even sabotage.
- A recovery scheme for the detection of failures and the restoration of the database to a consistent state that existed prior to the occurrence of the failure.

1.1 Recovery Environment

Storage Types

- Storage systems: cache memory, main memory, tape and disk.
- **Volatile storage**. Data on this type of storage devices can be lost very easily. E.g., main and cache memory.
- **Nonvolatile storage**. Data on these devices are more reliable than volatile storage. E.g., disks, tapes.
- **Stable storage**. Data is **assumed** to be 100% secure.
- To simplify the discussion, writing a block either is completely successful or no data in the target block is changed.

Failure types

- **Transaction-local failure**. Only one transaction is affected. Perform a rollback.
- **System-wide failure, database is recoverable**. Some or all transactions are affected. E.g., system crash, power failure.
- **Media failure**. A non-recoverable damage is detected on some nonvolatile storage device. E.g., head crash, virus attack, fire, flood.

Storage Structure and Operations

- A **logical file** is a file as seen by an application programmer and consists of a set of l**ogical records**.
- The atomic unit of storage allocation and data transfer is called a **block**, or a **physical record** or a **page**.
- A **block** contains one or more logical records.
- Whenever a file is open a **buffer** is allocated. The size of the buffer is at least as large as the block size of the physical file.
- Block movements between the disk and buffer are invoked by:

1. **input**(X), transfers the block in which data item X resides to buffer.
2. **output**(X), transfers the buffer block on which X resides to the disk and replaces the appropriate block.

- An application program interacts with the database system through:

1. **read**(X, xi): assigns the value of data item X to the local variable xi.
   (i) If the block on which X resides is not in main memory, then issue **input**(X).
   (ii) Assign to xi the value of X from the buffer block.
2. **write**(X, xi), assigns the value of local variable xi to X in the buffer.
   (i) If the block for X is not in main memory, then issue **input**(X).
   (ii) Assign the value xi to X in the block for X.

An Example

T: declare local variables a, b.

read(A,a)

a:=a-50

write(A,a)

read(B,b)

b:= b+50

write(B,b)

- There is no guarantee of when the blocks are actually read in or written out.

Transactions

T: declare local variables a, b.

read(A,a)

A=1000

a:=a-50

write(A,a)

read(B,b)

B=2000, A=950

b:= b+50

System crash!!

write(B,b)

- The recovery system must ensure the **atomicity and durability of a transaction**.

2. Recovery Schemes

- First discuss techniques for failures which result in no data lost on nonvolatile storage.
- Media failure $\Rightarrow$ data stored on nonvolatile storage may be destroyed.
- Make **NO** assumption on when a failure occurs.

2.1 Log-based Techniques

- A **log** file - a sequential file maintained by system and residing on a **stable** storage.
- A change is made to the database, a record containing values of the changed item is written to the log file.

2.1.1 Incremental Log with Deferred Updates

T: declare local variables a, b.

read(A,a)	A=1000
a:=a-50	
write(A,a)	
read(B,b)	B=2000, A=950
b:= b+50	System crash!!
write(B,b)	

- A straightforward solution: the database will not be changed until we are certain that updates can be performed on the actual data.
- Recorded all changes by a transaction on the log.

- Protocol:
  (i) T starts its execution, <T **starts**> is written to the log.
  (ii) During its execution, any **write**(X, x) operation by T $\Rightarrow$ writing <T, X, *new value of* X> to the log.
  (iii) Partially commits, write <T **commits**>.
- **No** data block is output until all log records for T are output.
- **RDU:**
  Redo all writes of the committed transactions in the order in which they were written to the log.

- Consider a transaction is executing and a system failure occurs.
  Case (1): Failure occurs before all log records are written onto stable storage. The log records for the transaction are simply ignored.
  Case (2): All log records are written onto stable storage. With **RDU** algorithm, we are guaranteed with the atomicity and durability properties of the transaction.

T: **read**(A,a) A=1000

a:=a - 50

write(A,a)

read(B,b) B= 2000

b:=b+50

write(B,b)

W: **read**(C, c) C=700

c:=c-100

write(C,c)

- The log records:

<T **starts**>

<T, A, 950>

<T, B, 2050>

<W **starts**>

<T **commits**>

<W, C, 600>

<W **commits**>

T: **read**(A,a)
a:=a - 50
write(A,a)
read(B,b)
b:=b+50
write(B,b)
W: **read**(C, c)
c:=c-100
write(C,c) System crash!!

- Log file:
  <T **starts**>
  <T, A, 950>
  <T, B, 2050>
  <W **starts**>
  <T **commits**>
  <W, C, 600>
- After **RDU**, A=950, B=2050 and C=700. Atomicity preserves.

2.1.2 Incremental Log with Immediate Updates

- Apply all updates ‘immediately’ to the database and keep a log of all changes to the database state.
- Protocol:
  (i) T starts its execution, <T **starts**> is written to the log.
  (ii) During its execution, any **write**(X,x) by T is **preceded** by the writing <T, X, *old value of* X, *new value of* X> to log.
  (iii) Partially commits $\Rightarrow$ write <T **commits**> to log.

- In this scheme, before executing an output operation on a block in main memory, all log records pertaining to data on that block **must be force-output** to stable storage, if they are not already in stable storage.
- **RIU**:
  1. Undo all writes of uncommitted transactions in the reverse order in which they were written in the log.
  2. Redo all writes of committed transactions in the order in which they were written in the log.

- Show correctness:
  Case (1) <T **commits**> has been written out.
  redo(T) is invoked.
  Case (2) <T **commits**> has not been written out.
  Have to undo all changes by invoking **undo**(T).
  Algorithm **undo**(T) consults the log file and restores the value of all data items updated by T to their old values.

T: **read**(A,a) A=1000
a:=a - 50
write(A,a)
read(B,b) B= 2000
b:=b+50
write(B,b)
W: **read**(C, c) C=700
c:=c-100
write(C,c)

- If successful, log file:
  <T **starts**>
  <T, A, 1000, 950>
  <W **starts**>
  <W , C, 700, 600>
  <T, B, 2000, 2050>
  <T **commits**>
  <W **commits**>

T: **read**(A,a) A=1000
a:=a - 50
write(A,a)
read(B,b) B= 2000
b:=b+50
write(B,b)
W: **read**(C, c) C=700
c:=c-100
write(C,c) System crash!!

- Log file:

<T **starts**>
<T, A, 1000, 950>
<W **starts**>
<W , C, 700, 600>
<T, B, 2000, 2050>
<T **commits**>

- After RIU, A, B, and C are \$950, \$2050, and \$700, respectively.

2.1.3 Checkpoints

- Whenever a failure occurs $\Rightarrow$ consult the log & run all transactions once again using the information stored in the log.
- Difficulties:

1. The search process may be long.
2. Most of the transactions most likely do not need to be redone.

- **Checkpoints**. Tell the recovery scheme what changes have actually been made to the database.
- In additional to the log file, the system periodically performs checkpoints:

1. Output all log records.
2. Output all modified buffers for data blocks to nonvolatile storage.
3. Output a log record **<checkpoint L>**, where **L** is a list of active transactions.

- **<checkpoint L>** is output $\Rightarrow$ all transactions T executed up to this point with <T **commits>** have their properties preserved.

- The recovery scheme examines the log to determine two sets **C** and **U**.
  C ={T | T is a transaction that has a <T **starts**> but with <T **commits**> written after the last **<checkpoint L>**}.
  U = {T | T is a transaction that has <T **starts**> but no <T **commits**> in the log}.
- For deferred scheme.

1. Redo all writes of the committed transactions in the order in which they were written in the log.

- For immediate update scheme.

1. Undo all writes of uncommitted transactions in the reverse order in which they were written in the log.
2. Redo all writes of the committed transactions in the order in which they were written in the log.

2.2 Media Failure Recovery

- Log-based technique.
- Consider no transaction is active in the system and the database state is consistent. Make a backup copy of the consistent state onto a stable storage.
- Failure occurs some time after the backup is performed, the database can be restored to a consistent state by means of the backup copy and by consulting the log file.
- Periodically performs the following with no transaction is active.

1. Output all log records.
2. Output all buffer blocks onto the disk.
3. Dump the entire content of the database to a stable storage device.
4. Output **<dump>** to log.

- To recover: the most recent dump is used to restore the database to a previous consistent state. The log file is then consulted and all the transactions that committed after the last **<dump>** record are redone.

<u>Database Security and Authorization</u>

- Data may be misused or intentionally made inconsistent.
- **Security** refers to protection of data against unauthorized disclosure, alternation or destruction.

Database Security and the DBA

- The DBA's responsibilities include granting privileges to users who need to use the system and classifying users and data in accordance with the policy of the organization.
- DBA privileged commands include commands for granting and revoking privileges to individual accounts, users, or user groups and for performing the following types of actions:

1. Account creation.
2. Privilege granting and revocation.
3. Security level assignment: This action consists of assigning user accounts to the appropriate security classification level.

Discretionary Access Control

- Based on the granting and revoking of **privileges.**
- In the context of SQL.
- Two levels for assigning privileges:

1. *The account level:* At this level, the DBA specifies the particular privileges (such as create schema, create table, create view, backup etc.) that each account holds independently of the relations in the database.

2. *The relation* (or *table*) *level:* At this level, we can control the privilege to access each individual relation or view in the database.

- The DBA can assign a whole schema to an owner.

CREATE SCHEMA COMPANY AUTHORIZATION JSMITH;

- Each relation *R* in a database is assigned an **owner account** – the creator. The owner of a relation is given *all* privileges on that relation.
- The owner account holder can pass privileges on any of the owned relations to other users by **granting** privileges to their accounts.

Format: **GRANT** <privileges> [**ON** <objects>] **to** <users> [**WITH GRANT OPTION**];

<privileges> := <privilege>* | ALL PRIVILEGES

<privilege> := SELECT | DELETE | INSERT [col*] | UPDATE [col*] | REFERENCE [col*]

<objects> := relations, records, columns, views etc.

<users> := <id >* | PUBLIC

Format: **REVOKE [GRANT OPTION FOR]** <privileges> [**ON** <objects>] **FROM** <users> [**RESTRICT** | **CASCADE**];

An Example

- **CREATE SCHEMA** EXAMPLE **AUTHORIZATION** A1;
- *A*1 creates the two base relations EMPLOYEE and DEPARTMENT.

Employee

Name	SSN	BDate	Address	Sex	Salary	DNO

Department

DNumber	DName	MgrSsn

- *A*1 wants to grant to account *A*2 the privilege to insert and delete tuples in both of these relations. However, *A*1 does not want *A*2 to be able to propagate these privileges to additional accounts.

GRANT INSERT, DELETE **ON** EMPLOYEE, DEPARTMENT **TO** A2;

- *A*1 wants to allow account *A*3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts.

GRANT SELECT **ON** EMPLOYEE, DEPARTMENT **TO** A3 **WITH GRANT OPTION;**

- *A*3 can grant the SELECT privilege on the EMPLOYEE relation to

GRANT SELECT **ON** EMPLOYEE **TO** A4;

- *A*1 decides to revoke the SELECT privilege on the EMPLOYEE relation from *A*3

REVOKE SELECT **ON** EMPLOYEE **FROM** A3 CASCADE;

- The DBMS automatically revoke the SELECT privilege on EMPLOYEE from *A*4.

- *A*1 wants to give back to *A*3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow *A*3 to be able to propagate the privilege.

  CREATE VIEW A3EMPLOYEE **AS**
  SELECT NAME, BDATE, ADDRESS
  FROM EMPLOYEE
  WHERE DNO = 5;

  GRANT SELECT **ON** A3EMPLOYEE **TO** A3 **WITH GRANT OPTION;**

- *A*1 wants to allow *A*4 to update only the SALARY attribute of EMPLOYEE.

  GRANT UPDATE **ON** EMPLOYEE (SALARY) **TO** A4;